CHAPTER 9

Deep Learning with TensorFlow

2017 was a special year for *deep learning*. In addition to the great experimental results obtained thanks to the algorithms developed, deep learning has seen its glory in the release of many frameworks with which to develop numerous projects. Some of you will certainly already know this branch of machine learning, others you have certainly heard someone mention it. Given the great importance that deep learning is taking in data processing and analysis techniques, I found it important to add this new chapter in the second edition of this book.

In this chapter you can have an introductory overview of the world of deep learning, and the artificial neural networks on which its techniques are based. Furthermore, among the new Python frameworks for deep learning, you will use *TensorFlow,* which is proving to be an excellent tool for research and development of deep learning analysis techniques. With this library you will see how to develop different models of neural networks that are the basis of deep learning.

Artificial Intelligence, Machine Learning, and Deep Learning

For anyone dealing with the world of data analysis, these three terms are ultimately very common on the web, in text, and on seminars related to the subject. But what is the relationship between them? And what do they really consist of?

In this section you will see the detailed definitions of these three terms. You will discover how in recent decades, the need to create more and more elaborate algorithms, and to be able to make predictions and classify data more and more efficiently, has led to

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_9

machine learning. Then you will discover how, thanks to new technological innovations, and in particular to the computing power achieved by the GPU, deep learning techniques have been developed based on neural networks.

Artificial intelligence

The term *artificial intelligence* was first used by John McCarthy in 1956, at a time full of great hopes and enthusiasm for the technology world. They were at the dawn of electronics and computers as large as whole rooms that could do a few simple calculations, but they did so efficiently and quickly compared to humans that they already glimpsed possible future developments of electronic intelligence.

But without going into the world of science fiction, the current definition best suited to artificial intelligence, often referred to as AI, could be summarized briefly with the following sentence:

> *Automatic processing on a computer capable of performing operations that would seem to be exclusively relevant to human intelligence.*

Hence the concept of artificial intelligence is a variable concept that varies with the progress of the machines themselves and with the concept of "exclusive human relevance". While in the 60s and 70s we saw artificial intelligence as the ability of computers to perform calculations and find mathematical solutions of complex problems "of exclusive relevance of great scientists," in the 80s and 90s it matured in the ability to assess risks, resources, and making decisions. In the year 2000, with the continuous growth of computer computing potential, the possibility of these systems to learn with machine learning was added to the definition.

Finally, in the last few years, the concept of artificial intelligence has focused on visual and auditory recognition operations, which until recently were thought of as "exclusive human relevance".

These operations include:

- Image recognition
- Object detection
- Object segmentation
- Language translation

- Natural language understanding
- Speech recognition

These are the problems still under study thanks to the deep learning techniques.

Machine Learning Is a Branch of Artificial Intelligence

In the previous chapter you saw machine learning in detail, with many examples of the different techniques for classifying or predicting data.

Machine learning (ML), with all its techniques and algorithms, is a large branch of artificial intelligence. In fact, you refer to it, while remaining within the ambit of artificial intelligence when you use systems that are able to learn (learning systems) to solve various problems that shortly before had been "considered exclusive to humans".

Deep Learning Is a Branch of Machine Learning

Within the machine learning techniques, a further subclass can be defined, called *deep learning*. You saw in Chapter 8 that machine learning uses systems that can learn, and this can be done through features inside the system (often parameters of a fixed model) that are modified in response to input data intended for learning (training set).

Deep learning techniques take a step forward. In fact, deep learning systems are structured so as not to have these intrinsic characteristics in the model, but these characteristics are extracted and detected by the system automatically as a result of learning itself. Among these systems that can do this, we refer in particular to *artificial neural networks*.

The Relationship Between Artificial Intelligence, Machine Learning, and Deep Learning

To sum up, in this section you have seen that machine learning and deep learning are actually subclasses of artificial intelligence. Figure 9-1 shows a schematization of classes in this relationship.

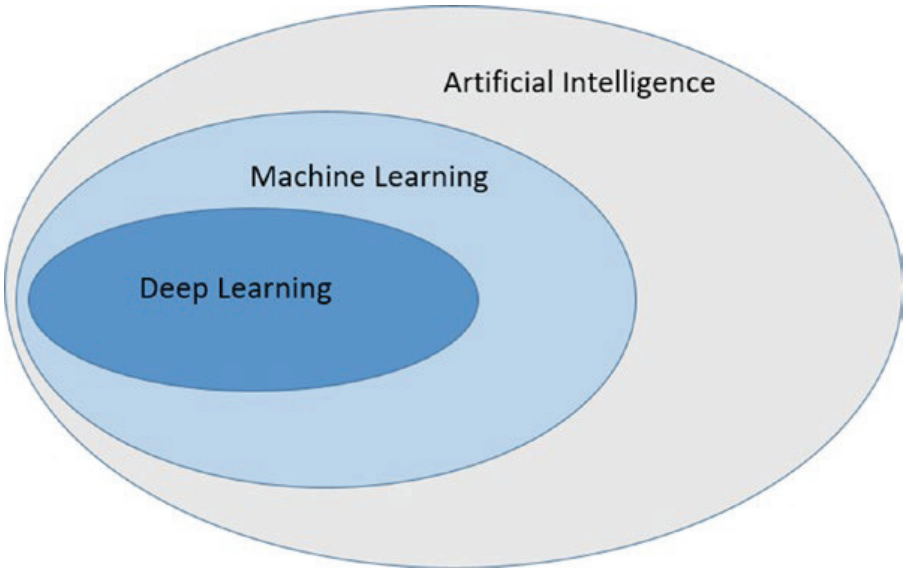

Figure 9-1. *Schematization of the relationship between artificial intelligence, machine learning, and deep learning*

Deep Learning

In this section, you will learn about some significant factors that led to the development of deep learning and read why only in these last years have there been so many steps forward.

Neural Networks and GPUs

In the previous section, you learned that in the field of artificial intelligence, deep learning has become popular only in the last few years precisely to solve problems of visual and auditory recognition.

In the context of deep learning, a lot of calculation techniques and algorithms have been developed in recent years, making the most of the potential of the Python language. But the theory behind deep learning actually dates back many years. In fact, the concept of the neural network was introduced in 1943, and the first theoretical studies on artificial neural networks and their applications were developed in the 60s.

The fact is that only in recent years the neural networks, with the related deep learning techniques that use them, have proved useful to solve many problems of artificial intelligence. This is due to the fact that only now are there technologies that can be implemented in a useful and efficient way.

In fact, at the application level, deep learning requires very complex mathematical operations that require millions or even billions of parameters. The CPUs of the 90s, even if powerful, were not able to perform these kinds of operations in efficient times. Even today the calculation with the CPUs, although considerably improved, requires long processing times. This inefficiency is due to the particular architecture of the CPUs, which have been designed to efficiently perform mathematical operations that are not those required by neural networks.

But a new kind of hardware has developed in recent decades, the *Graphics Processing Unit (GPU),* thanks to the enormous commercial drive of the videogames market. In fact this type of processor has been designed to manage more and more efficient vector calculations, such as multiplications between matrices, which is necessary for 3D reality simulations and rendering.

Thanks to this technological innovation, many deep learning techniques have been realized. In fact, to realize the neural networks and their learning, the tensors (multidimensional matrices) are used, carrying out many mathematical operations. It is precisely this kind of work that GPUs are able to do more efficiently. Thanks to their contribution, the processing speed of deep learning is increased by several orders of magnitude (days instead of months).

Data Availability: Open Data Source, Internet of Things, and Big Data

Another very important factor affecting the development of deep learning is the huge amount of data that can be accessed. In fact, the data are the fundamental ingredient for the functioning of neural networks, both for the learning phase and for their verification phase.

Thanks to the spread of the Internet all over the world, now everyone can access and produce data. While a few years ago only a few organizations were providing data for analysis, today, thanks to the IoT (Internet of Things), many sensors and devices acquire data and make them available on networks. Not only that, even social networks and search engines (like Facebook, Google, and so on) can collect huge amounts of data, analyzing in real time millions of users connected to their services (called *Big Data*).

So today a lot of data related to the problems we want to solve with the deep learning techniques, are easily available not only for a fee, but also in free form (open data source).

Python

Another factor that contributed to the great success and diffusion of deep learning techniques was the Python programming language.

In the past, planning neural network systems was very complex. The only language able to carry out this task was C ++, a very complex language, difficult to use and known only to a few specialists. Moreover, in order to work with the GPU (necessary for this type of calculation), it was necessary to know CUDA (Compute Unified Device Architecture), the hardware development architecture of NVIDIA graphics cards with all their technical specifications.

Today, thanks to Python, the programming of neural networks and deep learning techniques has become high level. In fact, programmers no longer have to think about the architecture and the technical specifications of the graphics card (GPU), but can focus exclusively on the part related to deep learning. Moreover the characteristics of the Python language enable programmers to develop simple and intuitive code. You have already tried this with machine learning in the previous chapter, and the same applies to deep learning.

Deep Learning Python Frameworks

Over the past two years many developer organizations and communities have been developing Python frameworks that are greatly simplifying the calculation and application of deep learning techniques. There is a lot of excitement about it, and many of these libraries perform the same operations almost competitively, but each of them is based on different internal mechanisms. We will see which in the next few years will be more successful or not.

Among these frameworks available today for free, it is worth mentioning some that are gaining some success.

- *TensorFlow* is an open source library for numerical calculation that bases its use on data flow graphs. These are graphs where the nodes represent the mathematical operations and the edges represent tensors (multidimensional data arrays). Its architecture is very flexible and can distribute the calculations both on multiple CPUs and on multiple GPUs.

- *Caffe2* is a framework developed to provide an easy and simple way to work on deep learning. It allows you to test model and algorithm calculations using the power of GPUs in the cloud.
- *PyTorch* is a scientific framework completely based on the use of GPUs. It works in a highly efficient way and was recently developed and is still not well consolidated. It is still proving a powerful tool for scientific research.
- *Theano* is the most used Python library in the scientific field for the development, definition, and evaluation of mathematical expressions and physical models. Unfortunately, the development team announced that new versions will no longer be released. However, it remains a reference framework thanks to the number of programs developed with this library, both in literature and on the web.

Artificial Neural Networks

Artificial neural networks are a fundamental element for deep learning and their use is the basis of many, if not almost all, deep learning techniques. In fact, these systems are able to learn, thanks to their particular structure that refers to the biological neural circuits.

In this section, you will see in more detail what artificial neural networks are and how they are structured.

How Artificial Neural Networks Are Structured

Artificial neural networks are complex structures created by connecting simple basic components that are repeated within the structure. Depending on the number of these basic components and the type of connections, more and more complex networks will be formed, with different architectures, each of which will present peculiar characteristics regarding the ability to learn and solve different problems of deep learning.

Figure 9-2 shows an example of how a generic artificial neural network is structured.

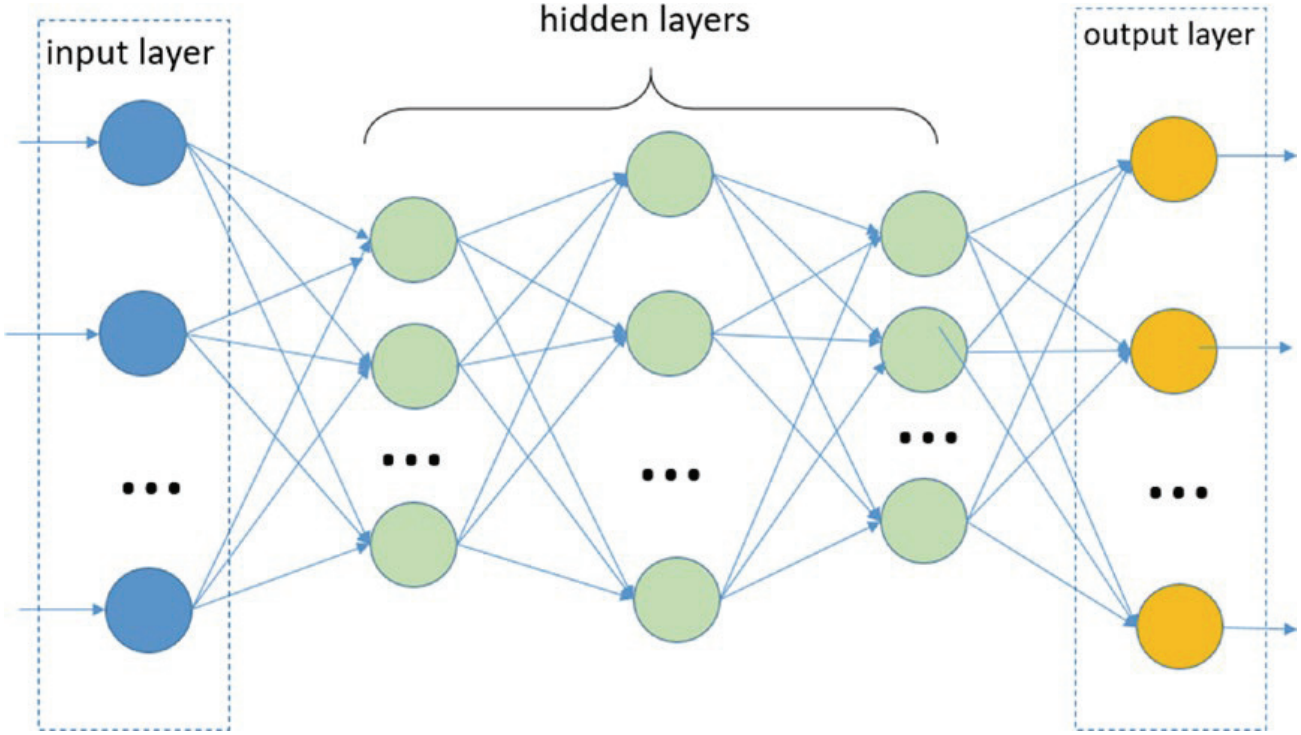

Figure 9-2. *A schematization of how a generic artificial neural network is structured*

The basic units are called *nodes* (the colored circles shown in Figure 9-2), which in the biological model simulate the functioning of a neuron within a neural network. These artificial neurons perform very simple operations, similar to the biological counterparts. They are activated when the total sum of the input signals they receive exceeds an activation threshold.

These nodes can transmit signals between them by means of connections, called *edges*, which simulate the functioning of biological synapses (the blue arrows shown in Figure 9-2). Through these edges, the signals sent by a neuron pass to the next one, behaving as a filter. That is, an edge converts the output message from a neuron, into an inhibitory or excitant signal, decreasing or increasing its intensity, according to pre-established rules (a different *weight* is generally applied for each edge).

The neural network has a certain number of nodes used to receive the input signal from the outside (see Figure 9-2). This first group of nodes is usually represented in a column at the far left end of the neural network schema. This group of nodes represents the first layer of the neural network (*input layer*). Depending on the input signals received, some (or all) of these neurons will be activated by processing the received signal and transmitting the result as output to another group of neurons, through edges.

This second group is in an intermediate position in the neural network, and is called the *hidden layer*. This is because the neurons of this group do not communicate with the outside neither in input nor in output and are therefore hidden. As you can see in Figure 9-2, each of these neurons has lots of incoming edges, often with all the neurons of the previous layer. Even these hidden neurons will be activated whether the total incoming signal will exceed a certain threshold. If affirmative, they will process the signal and transmit it to another group of neurons (in the right direction of the scheme shown in Figure 9-2). This group can be another hidden layer or the *output layer,* that is, the last layer that will send the results directly to the outside.

So in general we will have a flow of data that will enter the neural network (from left to right), and that will be processed in a more or less complex way depending on the structure, and will produce an output result.

The behavior, capabilities, and efficiency of a neural network will depend exclusively on how the nodes are connected and the total number of layers and neurons assigned to each of them. All these factors define the *neural network architecture.*

Single Layer Perceptron (SLP)

The *Single Layer Perceptron (SLP)* is the simplest model of neural network and was designed by Frank Rosenblatt in 1958. Its architecture is represented in Figure 9-3.

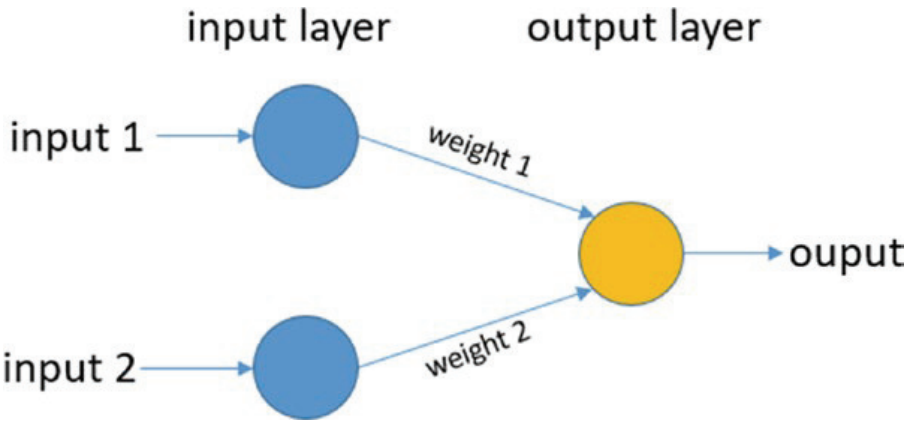

Figure 9-3. *The Single Layer Perceptron (SLP) architecture*

The Single Layer Perceptron (SLP) structure is very simple; it is a two-layer neural network, without hidden layers, in which a number of input neurons send signals to an output neuron through different connections, each with its own weight. Figure 9-4 shows in more detail the inner workings of this type of neural network.

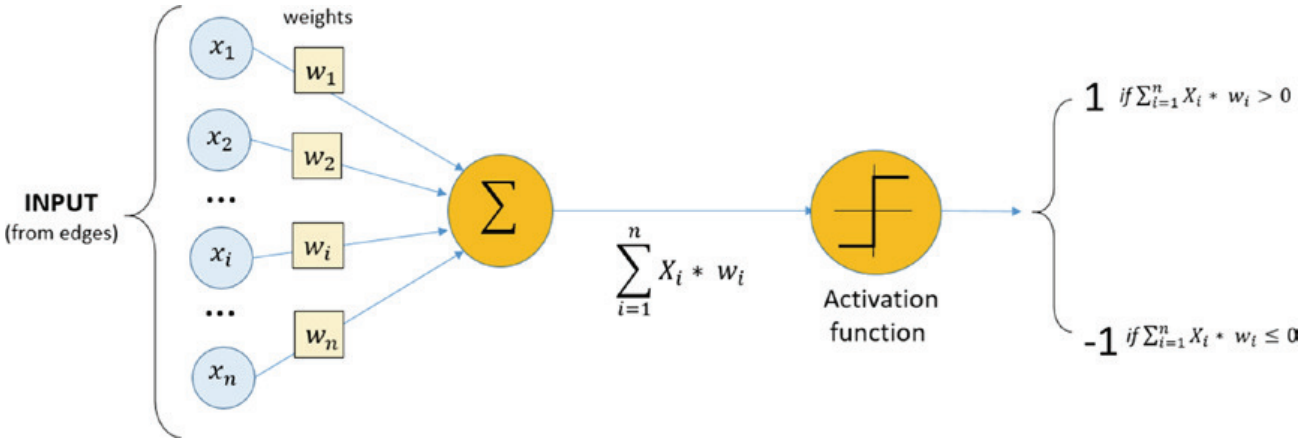

Figure 9-4. *A more detailed Single Layer Perceptron (SLP) representation with the internal operation expressed mathematically*

The edges of this structure are represented in this mathematic model by means of a weight vector consisting of the local memory of the neuron.

$$W = (w1, w2, \ldots\ldots, wn)$$

The output neuron receives an input vector signals `xi` each coming from a different neuron.

$$X = (x1, x2, \ldots\ldots, xn)$$

Then it processes the input signals via a weighed sum.

$$\sum_{i=0}^{n} w_i x_i = w_1x_1 + w_2x_2 + \ldots + w_nx_n = s$$

The total signal `s` is that perceived by the output neuron. If the signal exceeds the activation threshold of the neuron, it will activate, sending 1 as a value, otherwise it will remain inactive, sending -1.

$$\text{Output} = \begin{cases} 1, & \text{if } s > 0 \\ -1, & \text{otherwise} \end{cases}$$

This is the simplest *activation function* (see function A shown Figure 9-5), but you can also use other more complex ones, such as the sigmoid (see function D shown in Figure 9-5).

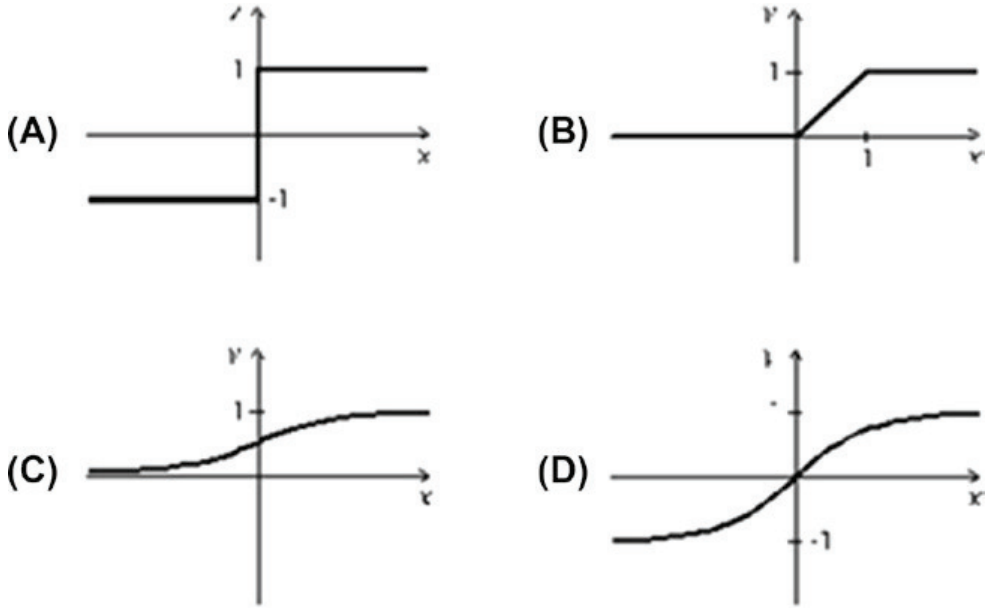

Figure 9-5. *Some examples of activation functions*

Now that you've seen the structure of the SLP neural network, you can switch to see how they can learn.

The learning procedure of a neural network, called the *learning phase,* works iteratively. That is, a predetermined number of cycles of operation of the neural network are carried out, in each of which the weights of the `wi` synapses are slightly modified. Each learning cycle is called an *epoch*. In order to carry out the learning you will have to use appropriate input data, called the *training sets* (you have already used them in depth in the Chapter 8).

In the training sets, for each input value, the expected output value is obtained. By comparing the output values produced by the neural network with the expected ones you can analyze the differences and modify the weight values, and you can also reduce them. In practice this is done by minimizing a *cost function (loss)* that is specific of the problem of deep learning. In fact the weights of the different connections will be modified for each epoch in order to minimize the cost (*loss*).

In conclusion, supervised learning is applied to neural networks.

At the end of the learning phase, you will pass to the *evaluation phase*, in which the learned SLP perceptron must analyze another set of inputs (test set) whose results are also known here. By evaluating the differences between the values obtained and those expected, the degree of ability of the neural network to solve the problem of deep learning will be known. Often the percentage of cases guessed with the wrong ones is used to indicate this value, and it is called *accuracy*.

Multi Layer Perceptron (MLP)

A more complex and efficient architecture is the *Multi Layer Perceptron (MLP)*. In this structure, there are one or more hidden layers interposed between the input layer and the output layer. The architecture is represented in Figure 9-6.

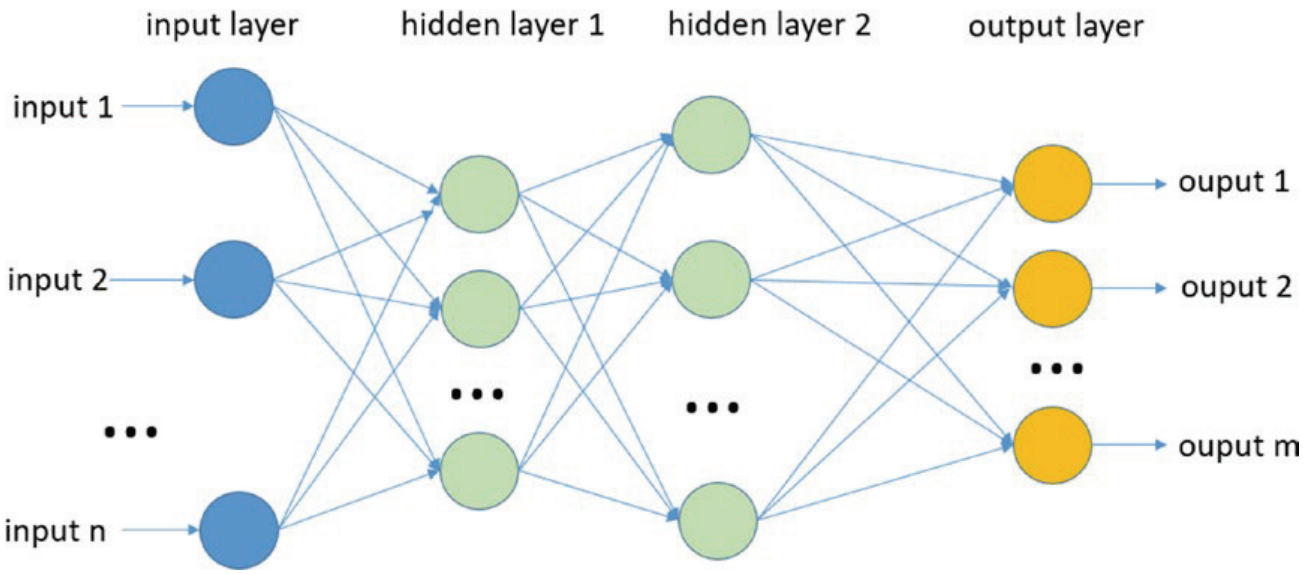

Figure 9-6. *The Multi Layer Perceptron (MLP) architecture*

At the end of the learning phase, you will pass to the *evaluation phase,* in which the learned SLP perceptron must analyze another set of inputs (test set) whose results are also known here. By evaluating the differences between the values obtained and those expected, the degree of ability of the neural network to solve the problem of deep learning will be known. Often, the percentage of cases guessed with the wrong ones is used to indicate this value, and it is called accuracy.

Although more complex, the models of MLP neural networks are based primarily on the same concepts as the models of the SLP neural networks. Even in MLPs, weights are assigned to each connection. These weights must be minimized based on the evaluation of a training set, much like the SLPs. Here, too, each node must process all incoming signals through an activation function, even if this time the presence of several hidden layers, will make the neural network able to learn more, *adapting more effectively* to the type of problem deep learning is trying to solve.

On the other hand, from a practical point of view, the greater complexity of this system requires more complex algorithms both for the learning phase and for the evaluation phase. One of these is the *back propagation algorithm*, used to effectively modify the weights of the various connections to minimize the cost function, in order to quickly and progressively converge the output values with the expected ones.

Other algorithms are used specifically for the minimization phase of the cost (or error) function and are generally referred to as *gradient descent* techniques.

The study and detailed analysis of these algorithms is outside the scope of this text, which has only an introductory function of the argument, with the goal of trying to keep the topic of deep learning as simple and clear as possible. If you are so inclined, I suggest you go deeper into the subject both in various books and on the Internet.

Correspondence Between Artificial and Biological Neural Networks

So far you have seen how deep learning uses basic structures, called artificial neural networks, to simulate the functioning of the human brain, particularly in the way it processes information.

There is also a real correspondence between the two systems at the highest reading level. In fact, you've just seen that neural networks have structures based on layers of neurons. The first layer processes the incoming signal, then passes it to the next layer, which in turn processes it and so on, until it reaches a final result. For each layer of neurons, incoming information is processed in a certain way, generating *different levels of representation* of the same information.

In fact, the whole operation of elaboration of an artificial neural network is nothing more than the transformation of information to ever more abstract levels.

This functioning is identical to what happens in the cerebral cortex. For example, when the eye receives an image, the image signal passes through various processing stages (such as the layers of the neural network), in which, for example, the contours of the figures are first detected (edge detection), then the geometric shape (form perception), and then to the recognition of the nature of the object with its name. Therefore, there has been a transformation at different levels of conceptuality of an incoming information, passing from an image, to lines, to geometrical figures, to arrive at a word.

TensorFlow

In a previous section of this chapter you saw that there are several frameworks in Python that allow you to develop projects for deep learning. One of these is *TensorFlow*. In this section you learn know in detail about this framework, including how it works and how it is used to realize neural networks for deep learning.

TensorFlow: Google's Framework

TensorFlow (https://www.tensorflow.org) is a library developed by the Google Brain Team, a group of Machine Learning Intelligence, a research organization headed by Google.

The purpose of this library is to have an excellent tool in the field of research for machine learning and deep learning.

The first version of TensorFlow was released by Google in February 2017, and in a year and a half, many update versions have been released, in which the potential, stability, and usability of this library are greatly increased. This is mainly thanks to the large number of users among professionals and researchers who are fully using this framework. At the present time, TensorFlow is already a consolidated deep learning framework, rich in documentation, tutorials, and projects available on the Internet.

In addition to the main package, there are many other libraries that have been released over time, including:

- *TensorBoard*—A kit that allows the visualization of internal graphs to TensorFlow (https://github.com/tensorflow/tensorboard).
- *TensorFlow Fold*—Produces beautiful dynamic calculation charts (https://github.com/tensorflow/fold)
- *TensorFlow Transform*—Created and managed input data pipelines (https://github.com/tensorflow/transform)

TensorFlow: Data Flow Graph

TensorFlow (https://www.tensorflow.org) is a library developed by the Google Brain Team, a group of Machine Learning Intelligence, a research organization headed by Google.

TensorFlow is based entirely on the structuring and use of graphs and on the flow of data through it, exploiting them in such a way as to make mathematical calculations.

The graph created internally to the TensorFlow runtime system is called *Data Flow Graph* and it is structured in runtime according to the mathematical model that is the basis of the calculation you want to perform. In fact, Tensor Flow allows you to define any mathematical model through a series of instructions implemented in the code. TensorFlow will take care of translating that model into the Data Flow Graph internally.

So when you go to model your deep learning neural network, it will be translated into a Data Flow Graph. Given the great similarity between the structure of neural networks and the mathematical representation of graphs, it is easy to understand why this library is excellent for developing deep learning projects.

But TensorFlow is not limited to deep learning and can be used to represent artificial neural networks. Many other methods of calculation and analysis can be implemented with this library, since any physical system can be represented with a mathematical model. In fact, this library can also be used to implement other machine learning techniques, and for the study of complex physical systems through the calculation of partial differentials, etc.

The nodes of the Data Flow Graph represent mathematical operations, while the edges of the graph represent tensors (multidimensional data arrays). The name TensorFlow derives from the fact that these tensors represent the flow of data through graphs, which can be used to model artificial neural networks.

Start Programming with TensorFlow

Now that you have seen in general what the TensorFlow framework consists of, you can start working with this library. In this section, you will see how to install this framework, how to define and use tensors within a model, and how to access the internal Data Flow Graph through sessions.

Installing TensorFlow

Before starting work, you need to install this library on your computer.

On Ubuntu Linux (version 16 or more recent) system, you can use pip to install the package:

```
pip3 install tensorflow
```

On Windows systems, you can use Anaconda to install the package:

```
conda install tensorflow
```

TensorFlow is a fairly recent framework and unfortunately it is not present on some Linux distributions and in the versions of a few years ago. So in these cases the installation of TensorFlow must be done manually, following the indications suggested on the official TensorFlow website (https://www.tensorflow.org/install/install_linux).

For those who have Anaconda (including Linux and OS) as a system for distributing Python packages on their computers, TensorFlow installation is much simpler.

Programming with the IPython QtConsole

Once TensorFlow is installed, you can start programming with this library. In the examples in this chapter, we use IPython, but you can do the same things by opening a normal Python session (or if you prefer, by using Jupyter Notebook). Via the terminal, open an IPython session by entering the following command line.

```
jupyter qtconsole
```

After opening an IPython session, import the library:

```
In [ ]: import tensorflow as tf
```

Note Remember that to enter multiple commands on different lines, you must use Ctrl+Enter. To execute the commands, press Enter only.

The Model and Sessions in TensorFlow

Before starting to program it is important to understand the internal operation of TensorFlow, including how it interprets the commands in Python and how it is executed internally. TensorFlow works through the concept of *model* and *sessions*, which define the structure of a program with a certain sequence of commands.

At the base of any TensorFlow project there is a model that includes a whole series of variables to be taken into consideration and that will define the system. Variables can be defined directly or parameterized through mathematical expressions on constants.

```
In [ ]: c = tf.constant(2,name='c')
   ...: x = tf.Variable(3,name='x')
   ...: y = tf.Variable(c*x,name='y')
   ...:
```

But now if you try to see the internal value of y (you expect a value of 6) with the `print()` function, you will see that it will give you the object and not the value.

```
In [3]: print(x)
   ...: print(y)
   ...:
<tf.Variable 'x:0' shape=() dtype=int32_ref>
<tf.Variable 'y:0' shape=() dtype=int32_ref>
```

In fact, you have defined variables belonging to the TensorFlow Data Flow Graph, that is a graph with nodes and connections that represent your mathematical model. You will see later how to access these values via sessions.

As far as the variables directly involved in the calculation of the deep learning method are concerned, *placeholders* are used, i.e. those tensors directly involved in the flow of data and in the processing of each single neuron.

Placeholders allow you to build the graph corresponding to the neural network, and to create operations inside without absolutely knowing the data to be calculated. In fact, you can build the structure of the graph (and also the neural network).

In practical cases, given a training set consisting of the value to be analyzed x (a tensor) and an expected value y (a tensor), you will define two placeholders xey, i.e., two tensors that will contain the values processed by the data for the whole neural network.

For example, define two placeholders that contain integers with the `tf.placeholder()` function.

```
In [ ]: X = tf.placeholder("int32")
   ...: X = tf.placeholder("int32")
   ...:
```

Once you have defined all the variables involved, i.e., you have defined the mathematical model at the base of the system, you need to perform the appropriate processing and initialize the whole model with the `tf.global_variables_initializer()` function.

```
In [ ]: model = tf.global_variables_initializer()
```

Now that you have a model initialized and loaded into memory, you need to start doing the calculations, but to do that you need to communicate with the TensorFlow runtime system. For this purpose a TensorFlow session is created, during which you can launch a series of commands to interact with the underlying graph corresponding to the model you have created.

You can create a new session with the `tf.Session()` constructor.

Within a session, you can perform the calculations and receive the values of the variables obtained as results, i.e., you can check the status of the graph during processing.

You have already seen that the operation of TensorFlow is based on the creation of an internal graph structure, in which the nodes are able to perform processing on the flow of data inside tensors that follow the connections of the graph.

So when you start a session, in practice you do nothing but instantiate this graph.

A session has two main methods:

- `session.extend()` allows you to make changes to the graph during the calculation, such as adding new nodes or connections.
- `session.run()` launches the execution of the graph and allows you to obtain the results in output.

Since several operations are carried out within the same session, it is preferred to use the construct `with`: with all calls to methods inherent to it.

In this simple case, you simply want to see the values of the variables defined in the model and print them on the terminal.

```
In [ ]: with tf.Session() as session:
   ...: session.run(model)
   ...: print(session.run(y))
   ...:
6
```

As you can see within the session, you can access the values of the Data Flow Graph, including the `y` variable that you previously defined.

Tensors

The basic element of the TensorFlow library is the *tensor*. In fact, the data that follow the flow within the Data Flow Graph are tensors (see Figure 9-7).

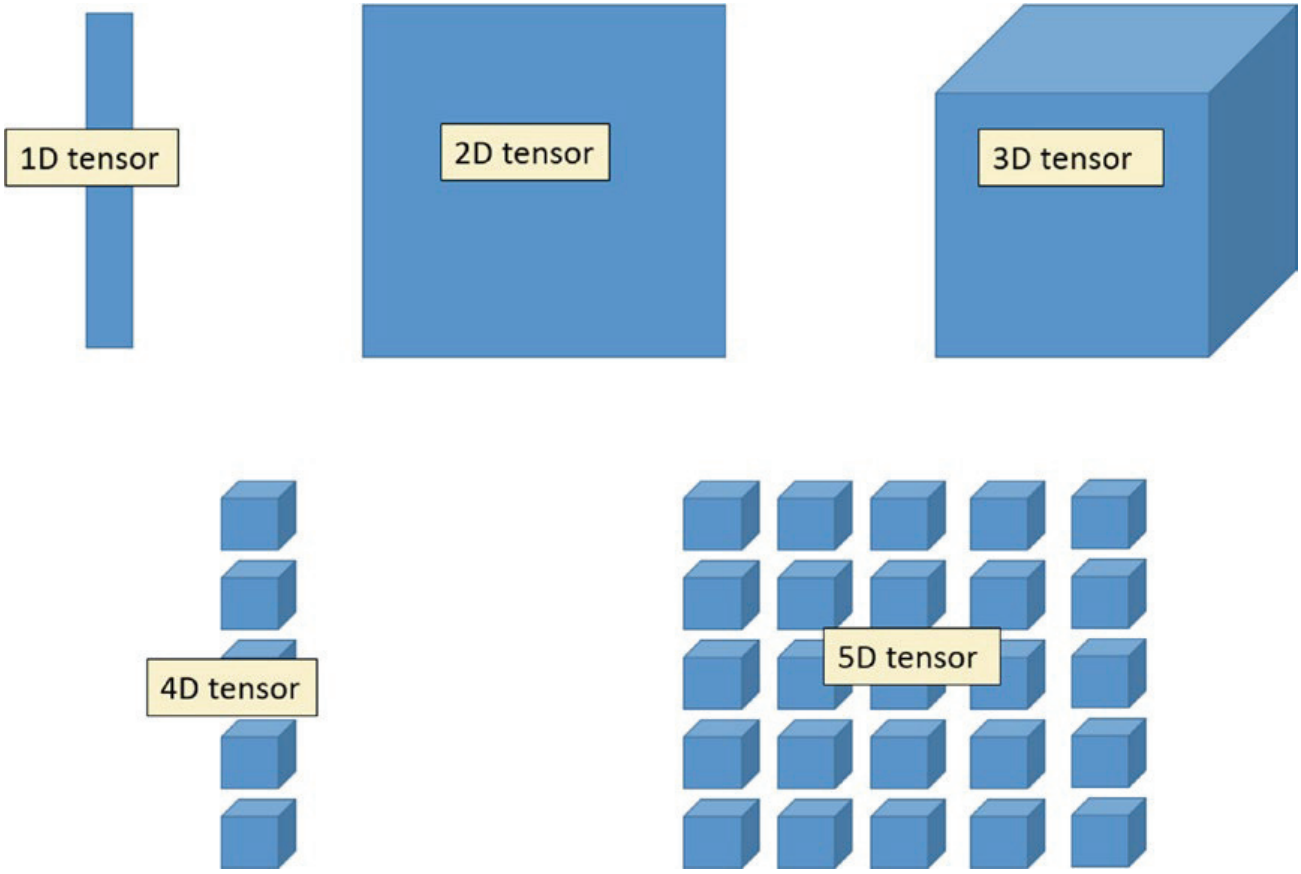

Figure 9-7. *Some representations of the tensors according to the different dimensions*

A tensor is identified by three parameters:

- `rank`—Dimension of the tensor (a matrix has rank 2, a vector has rank 1)
- **shape**—Number of rows and columns (e.g. (3.3) is a 3x3 matrix)
- **type**—Type of tensor elements.

```
type of tensor elements and columns (eg (3.3) is a 3x3 matrix)has rank 2, a
vector has rank 1)s ic
```

Tensors are nothing more than multidimensional arrays. In previous chapters, you saw how easy it is to get them thanks to the NumPy library. So you can start by defining one with this library.

```
In [ ]: import numpy as np
   ...: t = np.arange(9).reshape((3,3))
```

```
   ...: print(t)
   ...:
[[0 1 2]
[3 4 5]
[6 7 8]]
```

Now you can convert this multidimensional array into a TensorFlow tensor very easily thanks to the `tf.convert_to_tensor()` function, which takes two parameters. The first parameter is the array `t` that you want to convert and the second is the type of data you want to convert it to, in this case `int32`.

```
In [ ]: tensor = tf.convert_to_tensor(t, dtype=tf.int32)
```

If you want to see the content of the sensor now, you have to create a TensorFlow session and run it, printing the result on the terminal thanks to the `print()` function.

```
In [ ]: with tf.Session() as sess:
   ...: print(sess.run(tensor))
   ...:
[[0 1 2]
[3 4 5]
[6 7 8]]
```

As you can see, you have a tensor containing the same values and the same dimensions as the multidimensional array defined with NumPy. This approach is very useful for calculating deep learning, since many input values are in the form of NumPy arrays.

But tensors can be built directly from TensorFlow, without using the NumPy library. There are a number of functions that make it possible to enhance the tensors quickly and easily.

For example, if you want to initialize a tensor with all the values zero, you can use the `tf.zeros()` method.

```
In [10]: t0 = tf.zeros((3,3),'float64')

In [11]: with tf.Session() as session:
    ...: print(session.run(t0))
    ...:
[[ 0. 0. 0.]
```

```
[ 0. 0. 0.]
[ 0. 0. 0.]]
```

Likewise, if you want a tensor with all values of 1, you use the `tf.ones()` method.

```
In [12]: t1 = tf.ones((3,3),'float64')

In [13]: with tf.Session() as session:
    ...: print(session.run(t1))
    ...:
[[ 1. 1. 1.]
[ 1. 1. 1.]
[ 1. 1. 1.]]
```

Finally, it is also possible to create a tensor containing random values, which follow a uniform distribution (all the values within a range are equally likely to exit), thanks to the `tf.random_uniform()` function.

For example, if you want a 3x3 tensor with float values between 0 and 1, you can write:

```
In [ ]: tensorrand = tf.random_uniform((3, 3), minval=0, maxval=1,
dtype=tf.float32)

In [ ]: with tf.Session() as session:
    ...: print(session.run(tensorrand))
    ...:
[[ 0.63391674     0.38456023 0.13723993]
[ 0.7398864      0.44730318 0.95689237]
[ 0.48043406     0.96536028 0.40439832]]
```

But it can often be useful to create a tensor containing values that follow a normal distribution with a choice of mean and standard deviation. You can do this with the `tf.random_normal()` function.

For example, if you want to create a tensor of 3x3 size with mean 0 and standard deviation of 3, you will write:

```
In [ ]: norm = tf.random_normal((3, 3), mean=0, stddev=3)

In [ ]: with tf.Session() as session:
    ...: print(session.run(norm))
```

```
    ...:
[[-1.51012492      2.52284908    1.10865617]
[-5.08502769       1.92598009   -4.25456524]
[ 4.85962772      -6.69154644    5.32387066]]
```

Operation on Tensors

Once the tensors have been defined, it will be necessary to carry out operations on them. Most mathematical calculations on tensors are based on the sum and multiplication between tensors.

Define two tensors, `t1` and `t2`, that you will use to perform the operations between tensors.

```
In [ ]: t1 = tf.random_uniform((3, 3), minval=0, maxval=1, dtype=tf.
        float32)
    ...: t2 = tf.random_uniform((3, 3), minval=0, maxval=1, dtype=tf.
         float32)
    ...:
```

```
In [ ]: with tf.Session() as sess:
    ...: print('t1 = ',sess.run(t1))
    ...: print('t2 = ',sess.run(t2))
    ...:
t1 = [[ 0.22056699  0.15718663   0.11314452]
[ 0.43978345        0.27561605   0.41013181]
[ 0.58318019        0.3019532    0.04094303]]
t2 = [[ 0.16032183  0.32963789   0.30250323]
[ 0.02322233        0.79547286   0.01091838]
[ 0.63182664        0.64371264   0.06646919]]
```

Now to sum these two tensors, you can use the `tf.add()` function. To perform multiplication, you use the `tf.matmul()` function.

```
In [ ]: sum = tf.add(t1,t2)
    ...: mul = tf.matmul(t1,t2)
    ...:
```

```
In [ ]: with tf.Session() as sess:
    ...: print('sum =', sess.run(sum))
    ...: print('mul =', sess.run(mul))
    ...:
sum = [[ 0.78942883       0.73469722    1.0990597 ]
[ 0.42483664        0.62457812   0.98524892]
[ 1.30883813        0.75967956   0.19211888]]
mul = [[ 0.26865649       0.43188229    0.98241472]
[ 0.13723138        0.25498611   0.49761111]
[ 0.32352239        0.48217845   0.80896515]]
```

Another very common operation with tensors is the calculation of the determinant. TensorFlow provides the `tf.matrix_determinant()` method for this purpose:

```
In [ ]: det = tf.matrix_determinant(t1)
    ...: with tf.Session() as sess:
    ...: print('det =', sess.run(det))
    ...:
det = 0.101594
```

With these basic operations, you can implement many mathematical expressions that use tensors.

Single Layer Perceptron with TensorFlow

To better understand how to develop neural networks with TensorFlow, you will begin to implement a single layer Perceptron (SLP) neural network that is as simple as possible. You will use the tools made available in the TensorFlow library and by using the concepts you have learned about during the chapter and gradually introducing new ones.

During the course of this section, you will see the general practice of building a neural network. Thanks to a step-by-step procedure you can become familiar with the different commands used. Then, in the next section, you will use them to create a Perceptron Multi Layer neural network.

In both cases you will work with simple but complete examples of each part, so as not to add too many technical and complex details, but focus on the central part that involves the implementation of neural networks with TensorFlow.

Before Starting

Before starting, reopen a session with IPython by starting a new kernel. Once the session is open it imports all the necessary modules:

```
In [ ]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...: import tensorflow as tf
   ...:
```

Data To Be Analyzed

For the examples that you will consider in this chapter, you will use a series of data that you used in the machine learning chapter, in particular in the section about the Support Vector Machines (SVMs) technique.

The set of data that you will study is a set of 11 points distributed in a Cartesian axis divided into two classes of membership. The first six belong to the first class, the other five to the second. The coordinates (x, y) of the points are contained within a numpy `inputX` array, while the class to which they belong is indicated in `inputY`. This is a list of two-element arrays, with an element for each class they belong to. The value 1 in the first or second element indicates the class to which it belongs.

If the element has value [1.0], it will belong to the first class. If it has value [0,1], it belongs to the second class. The fact that they are float values is due to the optimization calculation of deep learning. You will see later that the test results of the neural networks will be floating numbers, indicating the probability that an element belongs to the first or second class.

Suppose, for example, that the neural network will give you the result of an element that will have the following values:

```
[0.910, 0.090]
```

This result will mean that the neural network considers that the element under analysis belongs to 91% to the first class and to 9% to the second class. You will see this in practice at the end of the section, but it was important to explain the concept to better understand the purpose of some values.

So based on the values taken from the example of SVMs in the machine learning chapter, you can define the following values.

```
In [2]: #Training set
   ...: inputX = np.arr
ay([[1.,3.],[1.,2.],[1.,1.5],[1.5,2.],[2.,3.],[2.5,1.5]
,[2.,1.],[3.,1.],[3.,2.],[3.5,1.],[3.5,3.]])
   ...: inputY = [[1.,0.]]*6+ [[0.,1.]]*5
   ...: print(inputX)
   ...: print(inputY)
   ...:
[[ 1. 3. ]
[ 1. 2. ]
[ 1. 1.5]
[ 1.5 2. ]
[ 2. 3. ]
[ 2.5 1.5]
[ 2. 1. ]
[ 3. 1. ]
[ 3. 2. ]
[ 3.5 1. ]
[ 3.5 3. ]]
[[1.0, 0.0], [1.0, 0.0], [1.0, 0.0], [1.0, 0.0], [1.0, 0.0],
 [1.0, 0.0], [0.0, 1.0], [0.0, 1.0], [0.0, 1.0], [0.0, 1.0], [0.0, 1.0]]
```

To better see how these points are arranged spatially and which classes they belong to, there is no better approach than to plot everything with matplotlib.

```
In [3]: yc = [0]*6 + [1]*5
   ...: print(yc)
...: import matplotlib.pyplot as plt
   ...: %matplotlib inline
   ...: plt.scatter(inputX[:,0],inputX[:,1],c=yc, s=50, alpha=0.9)
   ...: plt.show()
   ...:
[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

You will get the graph in Figure 9-8 as a result.

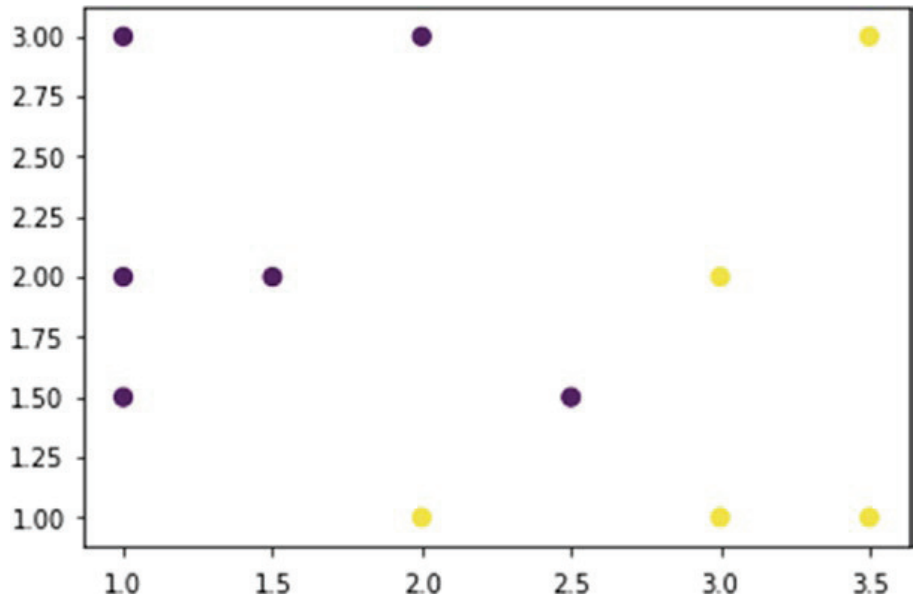

Figure 9-8. *The training set is a set of Cartesian points divided into two classes of membership (yellow and dark blue)*

To help in the graphic representation (as shown in Figure 9-8) of the color assignment, the `inputY` array has been replaced with `yc` array.

As you can see, the two classes are easily identifiable in two opposite regions. The first region covers the upper-left part, the second region covers the lower-right part. All this would seem to be simply subdivided by an imaginary diagonal line, but to make the system more complex, there is an exception with the point number 6 that is internal to the other points.

It will be interesting to see how and if the neural networks that we implement will be able to correctly assign the class to points of this kind.

The SLP Model Definition

For the examples that you will consider in this chapter, you will use a series of data that you have already used in the machine learning chapter, in particular in the section about the Support Vector Machines (SVMs) technique.

If you want to do a deep learning analysis, the first thing to do is define the neural network model you want to implement. So you will already have in mind the structure to be implemented, how many neurons and layers and compounds (in this case only one), the weight of the connections, and the cost function to be applied.

Following the TensorFlow practice, you can start by defining a series of parameters necessary to characterize the execution of the calculations during the learning phase. The *learning rate* is a parameter that regulates the learning speed of each neuron. This parameter is very important and plays a very important role in regulating the efficiency of a neural network during the learning phase. Establishing the optimal a priori value of the learning rate is impossible, because it depends very much on the structure of the neural network and on the particular type of data to be analyzed. It is therefore necessary to adjust this value through different learning tests, choosing the value that guarantees the best accuracy.

You can start with a generic value of 0.01, assigning this value to the `learning_rate` parameter.

```
In [ ]: learning_rate = 0.01
```

Another parameter to be defined is `training_epochs`. This defines how many epochs (learning cycles) will be applied to the neural network for the learning phase.

```
In [ ]: training_epochs = 2000
```

During program execution, it will be necessary in some way to monitor the progress of learning and this can be done by printing values on the terminal. You can decide how many epochs you will have to display a printout with the results, and insert them into the `display_step` parameter. A reasonable value is every 50 or 100 steps.

```
In [ ]: display_step = 50
```

To make the implemented code reusable, it is necessary to add parameters that specify the number of elements that make up the training set, and how many batches must be divided. In this case you have a small training set of only 11 items. So you can use them all in one batch.

```
In [ ]: n_samples = 11
   ...: batch_size = 11
   ...: total_batch = int(n_samples/batch_size)
   ...:
```

Finally, you can add two more parameters that describe the size and number of classes to which the incoming data belongs.

```
In [ ]: n_input = 2 # size data input (# size of each element of x)
   ...: n_classes = 2 # n of classes
   ...:
```

Now that you have defined the parameters of the method, let's move on to building the neural network. First, define the inputs and outputs of the neural network through the use of *placeholders*.

```
In [ ]: # tf Graph input
   ...: x = tf.placeholder("float", [None, n_input])
   ...: y = tf.placeholder("float", [None, n_classes])
   ...:
```

Then you have just *implicitly* defined an SLP neural network with two neurons in the input layer and two neurons in the output layer (see Figure 9-9), defining an input placeholder `x` with two values and a placeholder of output `y` with two values. *Explicitly*, you have instead defined two tensors, the tensor `x` that will contain the values of the input coordinates, and a tensor `y` that will contain the probabilities of belonging to the two classes of each element.

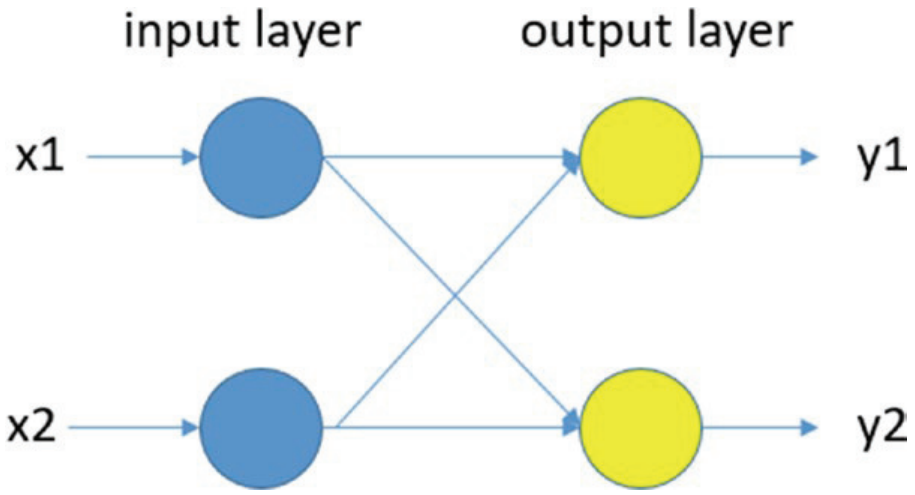

Figure 9-9. *The Single Layer Perceptron model used in this example*

But this will be much more visible in the following example, when dealing with MLP neural networks. Now that you have defined the placeholders, occupied with the weights and the bias, which, as you saw, are used to define the connections of the neural network. These tensors `W` and `b` are defined as variables by the constructor `Variable()` and initialized to all zero values with `tf.zeros()`.

```
In [ ]: # Set model weights
    ...: W = tf.Variable(tf.zeros([n_input, n_classes]))
    ...: b = tf.Variable(tf.zeros([n_classes]))
    ...:
```

The variables weight and bias you have just defined will be used to define the evidence x * W + b, which characterizes the neural network in mathematical form. The `tf.matmul()` function performs a multiplication between tensors x * W, while the `tf.add()` function adds to the result the value of bias `b`.

```
In [ ]: evidence = tf.add(tf.matmul(x, W), b)
```

From the value of the evidence, you can directly calculate the probabilities of the output values with the `tf.nn.softmax()` function.

```
In [ ]: y_ = tf.nn.softmax(evidence)
```

The `tf.nn.softmax()` function performs two steps:

- It calculates the evidence that a certain Cartesian entry point `xi` belongs to a particular class.
- It converts the evidence into probability of belonging to each of the two possible classes and returns it as `y_`.

Continuing with the construction of the model, now you must think about establishing the rules for the minimization of these parameters and you do so by defining the cost (or loss). In this phase you can choose many functions; one of the most common is the mean squared error loss.

```
In [ ]: cost = tf.reduce_sum(tf.pow(y-y_,2))/ (2 * n_samples)
```

But you can use any other function that you think is more convenient. Once the cost (or loss) function has been defined, an algorithm must be established to perform the minimization at each learning cycle (optimization). You can use the `tf.train.GradientDescentOptimizer()` function as an optimizer that bases its operation on the Gradient Descent algorithm.

```
In [ ]: optimizer =tf.train.GradientDescentOptimizer(learning_
rate=learning_rate).minimize(cost)
```

With the definition of the cost optimization method (minimization), you have completed the definition of the neural network model. Now you are ready to begin to implement its learning phase.

Learning Phase

Before starting, define two lists that will serve as a container for the results obtained during the learning phase. In `avg_set` you will enter all the cost values for each epoch (learning cycle), while in `epoch_set` you will enter the relative epoch number. These data will be useful at the end to visualize the cost trend during the learning phase of the neural network, which will be very useful for understanding the efficiency of the chosen learning method for the neural network.

```
In [ ]: avg_set = []
    ...: epoch_set=[]
    ...:
```

Then before starting the session, you need to initialize all the variables with the function you've seen before, `tf.global_variables_initializer()`.

```
In [ ]: init = tf.global_variables_initializer()
```

Now you are ready to start the session (do not press Enter at the end; you must enter other commands within the session).

```
In [ ]: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
```

You have already seen that every learning step is called an epoch. It is possible to intervene within each epoch, thanks to a `for` loop that scans all the values of `training_epochs`.

Within this cycle for each epoch, you will optimize using the `sess.run` (optimizer) command. Furthermore, every 50 epochs, the condition `if% display_step == 0` will be satisfied. Then you will extract the cost value via `sess.run(cost)` and insert it in the `c` variable that you will use for printing on the terminal as the `print()` argument that stores the `avg_set` list, using the `append()` function. In the end, when the `for` loop has been completed, you will print a message on the terminal informing you of the end of the learning phase. (Do not press Enter, as you still have to add other commands ...)

```
In [ ]: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
    ...:     for i in range(training_epochs):
```

```
...:         sess.run(optimizer, feed_dict = {x: inputX, y: inputY})
...:         if i % display_step == 0:
...:             c = sess.run(cost, feed_dict = {x: inputX, y: inputY})
...:             print("Epoch:", '%04d' % (i), "cost=", "{:.9f}".
                 format(c))
...:             avg_set.append(c)
...:             epoch_set.append(i + 1)
...:
...:     print("Training phase finished")
```

Now that the learning phase is over, it is useful to print a summary table on the terminal that shows you the trend of the cost during it. You can do this thanks to the values contained in the `avg_set` and `epoch_set` lists that you filled during the learning process.

Always in the session add these last lines of code, at the end of which you can finally press Enter and start the session by executing the learning phase.

```
In [ ]: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
    ...:     for i in range(training_epochs):
    ...:         sess.run(optimizer, feed_dict = {x: inputX, y: inputY})
    ...:         if i % display_step == 0:
    ...:             c = sess.run(cost, feed_dict = {x: inputX, y: inputY})
    ...:             print("Epoch:", '%04d' % (i), "cost=", "{:.9f}".
                     format(c))
    ...:             avg_set.append(c)
    ...:             epoch_set.append(i + 1)
    ...:
    ...:     print("Training phase finished")
    ...:
    ...:     training_cost = sess.run(cost, feed_dict = {x: inputX, y:
             inputY})
    ...:     print("Training cost =", training_cost, "\nW=", sess.run(W),
             "\nb=", sess.run(b))
    ...:     last_result = sess.run(y_, feed_dict = {x:inputX})
    ...:     print("Last result =",last_result)
    ...:
```

When the session with the learning phase of the neural network is finished, you get the following results.

```
Epoch: 0000 cost= 0.249360308
Epoch: 0050 cost= 0.221041128
Epoch: 0100 cost= 0.198898271
Epoch: 0150 cost= 0.181669712
Epoch: 0200 cost= 0.168204829
Epoch: 0250 cost= 0.157555178
Epoch: 0300 cost= 0.149002746
Epoch: 0350 cost= 0.142023861
Epoch: 0400 cost= 0.136240512
Epoch: 0450 cost= 0.131378993
Epoch: 0500 cost= 0.127239138
Epoch: 0550 cost= 0.123672642
Epoch: 0600 cost= 0.120568059
Epoch: 0650 cost= 0.117840447
Epoch: 0700 cost= 0.115424201
Epoch: 0750 cost= 0.113267884
Epoch: 0800 cost= 0.111330733
Epoch: 0850 cost= 0.109580085
Epoch: 0900 cost= 0.107989430
Epoch: 0950 cost= 0.106537104
Epoch: 1000 cost= 0.105205171
Epoch: 1050 cost= 0.103978693
Epoch: 1100 cost= 0.102845162
Epoch: 1150 cost= 0.101793952
Epoch: 1200 cost= 0.100816071
Epoch: 1250 cost= 0.099903718
Epoch: 1300 cost= 0.099050261
Epoch: 1350 cost= 0.098249927
Epoch: 1400 cost= 0.097497642
Epoch: 1450 cost= 0.096789025
Epoch: 1500 cost= 0.096120209
Epoch: 1550 cost= 0.095487759
Epoch: 1600 cost= 0.094888613
```

```
Epoch: 1650 cost= 0.094320126
Epoch: 1700 cost= 0.093779817
Epoch: 1750 cost= 0.093265578
Epoch: 1800 cost= 0.092775457
Epoch: 1850 cost= 0.092307687
Epoch: 1900 cost= 0.091860712
Epoch: 1950 cost= 0.091433071
Training phase finished
Training cost = 0.0910315
W= [[-0.70927787 0.70927781]
[ 0.62999243 -0.62999237]]
b= [ 0.34513065 -0.34513068]
Last result = [[ 0.95485419 0.04514586]
[ 0.85713255 0.14286745]
[ 0.76163834 0.23836163]
[ 0.74694741 0.25305259]
[ 0.83659446 0.16340555]
[ 0.27564839 0.72435158]
[ 0.29175714 0.70824283]
[ 0.090675 0.909325 ]
[ 0.26010245 0.73989749]
[ 0.04676624 0.95323378]
[ 0.37878013 0.62121987]]
```

As you can see, the cost is gradually improving during the epoch, up to a value of 0.168. Then it is interesting to see the values of the W weights and the bias of the neural network. These values represent the parameters of the model, i.e., the neural network instructed to analyze this type of data and to carry out this type of classification.

These parameters are very important, because once they are obtained and knowing the structure of the neural network used, it will be possible to reuse them anywhere without repeating the learning phase. Do not consider this example that takes only a minute to do the calculation; in real cases it may take days to do it, and often you have to make many attempts and adjust and calibrate the different parameters before developing an efficient neural network that is very accurate at class recognition, or at performing any other task.

If you see the results from a graphical point of view, it may be easier and faster to understand. You can use the matplotlib to do this.

```
In [ ]: plt.plot(epoch_set,avg_set,'o',label = 'SLP Training phase')
    ...: plt.ylabel('cost')
    ...: plt.xlabel('epochs')
    ...: plt.legend()
    ...: plt.show()
    ...:
```

You can analyze the learning phase of the neural network by following the trend of the cost value, as shown in Figure 9-10.

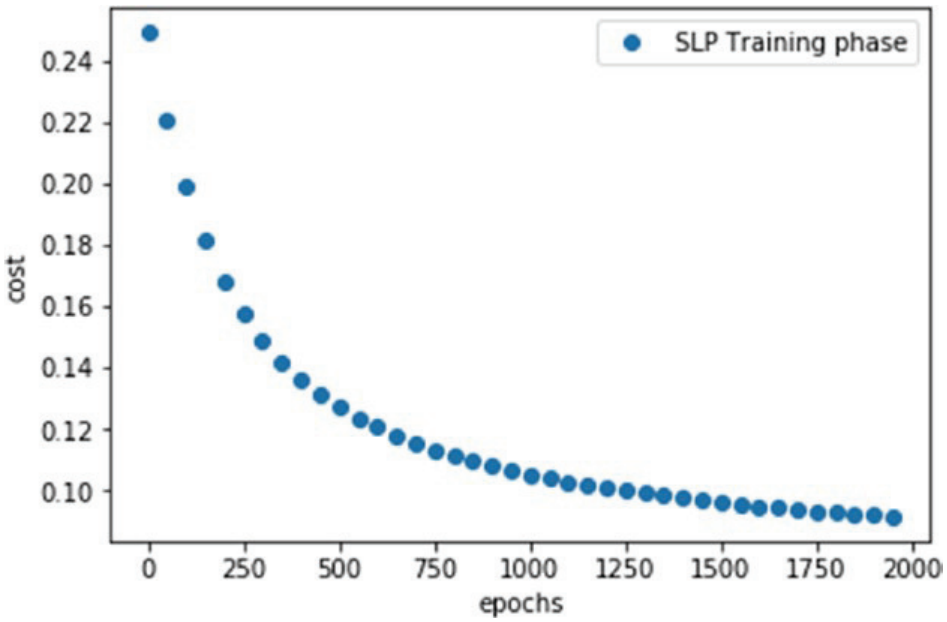

Figure 9-10. *The cost value decreases during the learning phase (cost optimization)*

Now you can move on to see the results of the classification during the last step of the learning phase.

```
In [ ]: yc = last_result[:,1]
    ...: plt.scatter(inputX[:,0],inputX[:,1],c=yc, s=50, alpha=1)
    ...: plt.show()
    ...:
```

You will get the representation of the points in the Cartesian plane, as shown in Figure 9-11.

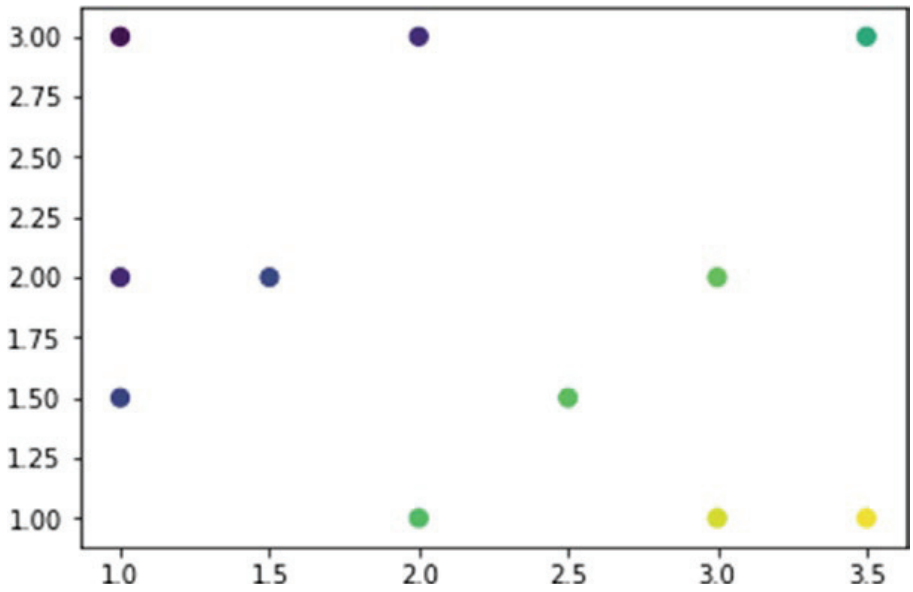

Figure 9-11. *The estimate of the class to which the points belong in the last epoch of the learning phase*

The graph represents the points in the Cartesian plane (see Figure 9-11), with a color ranging from blue (belonging to 100% to the first group) to yellow (belonging to 100% to the second group). As you can see, the division in the two classes of the points of the training set is quite optimal, with an uncertainty for the four points on the central diagonal (green).

This chart shows in some way the learning ability of the neural network used. As you can see, despite the learning epochs with the training set used, the neural network failed to learn that point 6 (x = 2.5, y = 1.5) belongs to the first class. This is a result you could expect, as it represents an exception and adds an effect of uncertainty to other points in the second class (the green dots).

Test Phase and Accuracy Calculation

Now that you have an educated neural network, you can create the evaluations and calculate the accuracy.

First you define a testing set with elements different than the training set. For convenience, these examples always use 11 elements.

```
In [ ]: #Testing set
    ...: testX = np.arr
ay([[1.,2.25],[1.25,3.],[2,2.5],[2.25,2.75],[2.5,3.],
[2.,0.9],[2.5,1.2],[3.,1.25],[3.,1.5],[3.5,2.],[3.5,2.5]])
    ...: testY = [[1.,0.]]*5 + [[0.,1.]]*6
    ...: print(testX)
    ...: print(testY)
    ...:
[[ 1. 2.25]
[ 1.25 3. ]
[ 2. 2.5 ]
[ 2.25 2.75]
[ 2.5 3. ]
[ 2. 0.9 ]
[ 2.5 1.2 ]
[ 3. 1.25]
[ 3. 1.5 ]
[ 3.5 2. ]
[ 3.5 2.5 ]]
[[1.0, 0.0], [1.0, 0.0], [1.0, 0.0], [1.0, 0.0], [1.0, 0.0],
[0.0, 1.0], [0.0, 1.0], [0.0, 1.0], [0.0, 1.0], [0.0, 1.0], [0.0, 1.0]]
```

To better understand the test set data and their membership classes, show the points on a chart using matplotlib.

```
In [ ]: yc = [0]*5 + [1]*6
    ...: print(yc)
    ...: plt.scatter(testX[:,0],testX[:,1],c=yc, s=50, alpha=0.9)
    ...: plt.show()
    ...:
[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
```

You will get the representation of the points in the Cartesian plane, as shown in Figure 9-12.

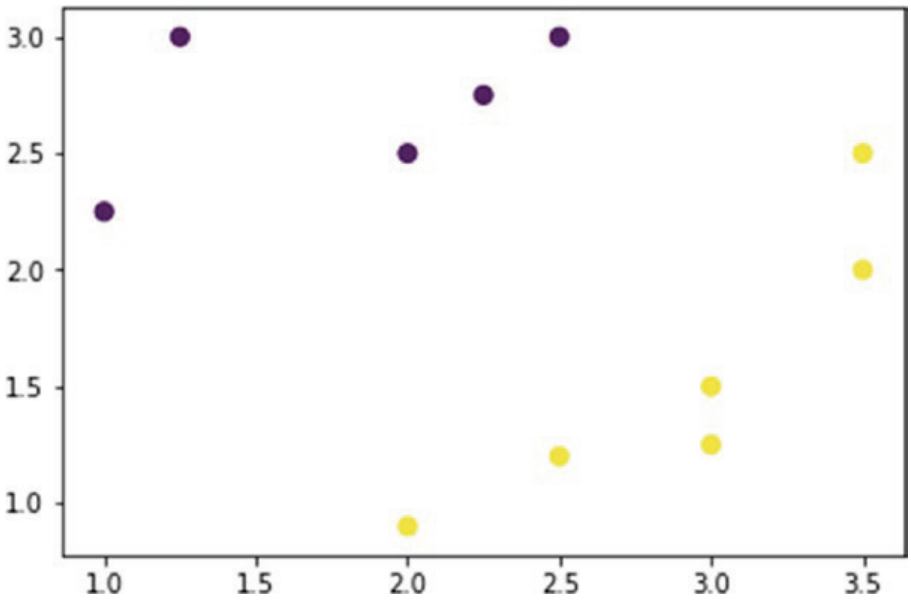

Figure 9-12. *The testing set*

Now you will use this testing set to evaluate the SLP neural network and calculate the accuracy.

```
In [ ]: init = tf.global_variables_initializer()
    ...: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
    ...:     for i in range(training_epochs):
    ...:             sess.run(optimizer, feed_dict = {x: inputX, y: inputY})
    ...:
    ...:     pred = tf.nn.softmax(evidence)
    ...:     result = sess.run(pred, feed_dict = {x: testX})
    ...:     correct_prediction = tf.equal(tf.argmax(pred, 1),
             tf.argmax(testY, 1))
    ...:
    ...:     # Calculate accuracy
    ...:     accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    ...:     print("Accuracy:", accuracy.eval({x: testX, y: testY}))
    ...:
Accuracy: 1.0
```

Apparently, the neural network was able to correctly classify all 11 past champions. It displays the points on the Cartesian plane with the same system of color gradations ranging from dark blue to yellow.

```
In [ ]: yc = result[:,1]
    ...: plt.scatter(testX[:,0],testX[:,1],c=yc, s=50, alpha=1)
    ...: plt.show()
```

You will get the representation of the points in the Cartesian plane, as shown in Figure 9-13.

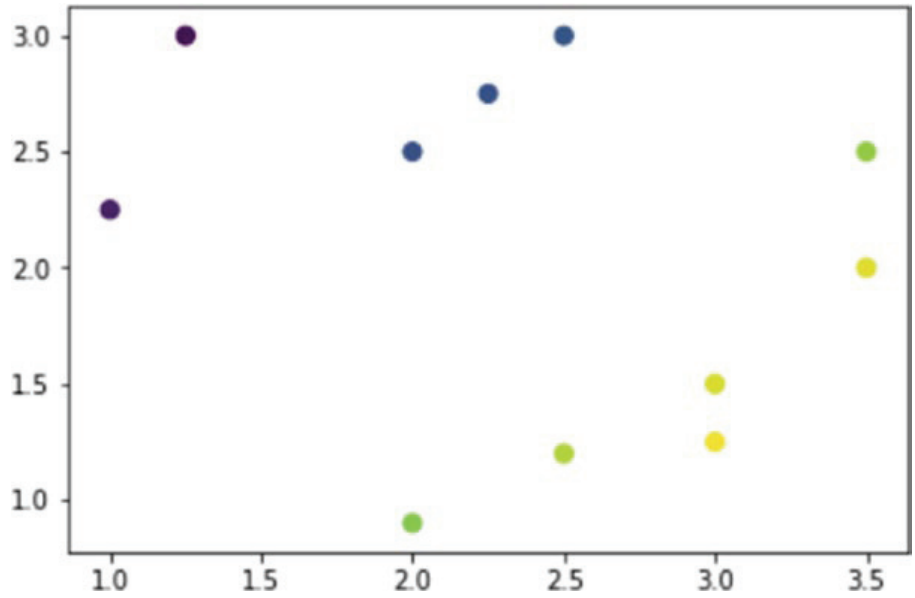

Figure 9-13. *The estimate of the class to which the testing set points belong*

The results can be considered optimal, given the simplicity of the model used and the small amount of data used in the training set. Now you will face the same problem with a more complex neural network, the Perceptron Multi Layer.

Multi Layer Perceptron (with One Hidden Layer) with TensorFlow

In this section, you will deal with the same problem as in the previous section, but using an MLP (Multi Layer Perceptron) neural network.

Start a new IPython session, resetting the kernel. As for the first part of the code, it remains the same as the previous example.

```
In [1]: import tensorflow as tf
   ...: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
   ...: #Training set
   ...: inputX = np.arr
ay([[1.,3.],[1.,2.],[1.,1.5],[1.5,2.],[2.,3.],[2.5,1.5],
[2.,1.],[3.,1.],[3.,2.],[3.5,1.],[3.5,3.]])
   ...: inputY = [[1.,0.]]*6+ [[0.,1.]]*5
   ...:
   ...: learning_rate = 0.001
   ...: training_epochs = 2000
   ...: display_step = 50
   ...: n_samples = 11
   ...: batch_size = 11
   ...: total_batch = int(n_samples/batch_size)
```

The MLP Model Definition

As you saw earlier in the chapter, a neural network MLP differs from a SLP neural network in that it can have one or more hidden layers.

Therefore, you will write parameterized code that allows you to work in the most general way possible, establishing at the time of definition the number of hidden layers present in the neural network and how many neurons they are composed of.

Define two new parameters that define the number of neurons present for each hidden layer. The `n_hidden_1` parameter will indicate how many neurons are present in the first hidden layer, while `n_hidden_2` will indicate how many neurons are present in the second hidden layer.

To start with a simple case, you will start with an MLP neural network with only one hidden layer consisting of only two neurons. Let's comment on the part related to the second hidden layer.

As for the `n_input` and `n_classes` parameters, they will have the same values as the previous example with the SLP neural network.

```
In [2]: # Network Parameters
   ...: n_hidden_1 = 2 # 1st layer number of neurons
   ...: #n_hidden_2 = 0 # 2nd layer number of neurons
   ...: n_input = 2 # size data input
   ...: n_classes = 2 # classes
   ...:
```

The definition of the placeholders is also the same as the previous example.

```
In [3]: # tf Graph input
   ...: X = tf.placeholder("float", [None, n_input])
   ...: Y = tf.placeholder("float", [None, n_classes])
   ...:
```

Now you have to deal with the definition of the various W and bias b weights for the different connections. The neural network is now much more complex, having several layers to take into account. An efficient way to parameterize them is to define them as follows, commenting out the weight and bias parameters for the second hidden layer (only for the MLP with one hidden layer as this example).

```
In [4]: # Store layers weight & bias
   ...: weights = {
   ...:     'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
   ...:     #'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
   ...:     'out': tf.Variable(tf.random_normal([n_hidden_1, n_classes]))
   ...: }
   ...: biases = {
   ...:     'b1': tf.Variable(tf.random_normal([n_hidden_1])),
   ...:     #'b2': tf.Variable(tf.random_normal([n_hidden_2])),
   ...:     'out': tf.Variable(tf.random_normal([n_classes]))
   ...: }
   ...:
```

To create a neural network model that takes into account all the parameters you've specified dynamically, you need to define a convenient function, which you'll call `multilayer_perceptron()`.

```
In [5]: # Create model
   ...: def multilayer_perceptron(x):
   ...:     layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
   ...:     #layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
   ...:     # Output fully connected layer with a neuron for each class
   ...:     out_layer = tf.matmul(layer_1, weights['out']) + biases['out']
   ...:     return out_layer
   ...:
```

Now you can build the model by calling up the function you just defined.

```
In [6]: # Construct model
   ...: evidence = multilayer_perceptron(X)
   ...: y_ = tf.nn.softmax(evidence)
   ...:
```

The next step is to define the `cost` function and choose an optimization method. For MLP neural networks, a good choice is `tf.train.AdamOptimizer()` as an optimization method.

```
In [7]: # Define cost and optimizer
   ...: cost = tf.reduce_sum(tf.pow(Y-y_,2))/ (2 * n_samples)
   ...: optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).
        minimize(cost)
   ...:
```

With these last two lines you have completed the definition of the model of the MLP neural network. Now you can move on to creating a session to implement the learning phase.

Learning Phase

As in the previous example, you will now define two lists that will contain the number of epochs and the measured cost values for each of them. You will also initialize all the variables before starting the session.

```
In [8]: avg_set = []
   ...: epoch_set = []
   ...: init = tf.global_variables_initializer()
   ...:
```

Now you are ready to start implementing the learning session. Open the session with the following instructions (remember not to press Enter, but Enter+Ctrl to insert other commands later):

```
In [9]: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
```

Now it implements the code to execute for each epoch, and inside it a scan for each batch belonging to the training set. In this case you have a training set consisting of a single batch, so you will have only one iteration in which you will directly assign `inputX` and `inputY` to `batch_x` and `batch_y`. In other cases you will need to implement a function, such as `next_batch(batch_size)`, which subdivides the entire training set (for example, `inputdata`) into different batches, progressively returning them as a return value.

At each batch cycle the `cost` function will be minimized with `sess.run([optimizer, cost])`, which will correspond to a partial cost. All batches will contribute to the calculation of the average cost of all the `avg_cost` batches. However, in this case, since you have only one batch, the `avg_cost` is equivalent to the cost of the entire training set.

```
In [9]: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
    ...:     for epoch in range(training_epochs):
    ...:         avg_cost = 0.
    ...:         # Loop over all batches
    ...:         for i in range(total_batch):
    ...:             #batch_x, batch_y = inputdata.next_batch(batch_size)
                     TO BE IMPLEMENTED
    ...:             batch_x = inputX
    ...:             batch_y = inputY
    ...:             _, c = sess.run([optimizer, cost], feed_dict={X:
                     batch_x, Y: batch_y})
```

```
...:             # Compute average loss
...:             avg_cost += c / total_batch
```

Every certain number of epochs, you will certainly want to display the value of the current cost on the terminal and add these values to the avg_set and epoch_set lists, as in the previous example with SLP.

```
In [9]: with tf.Session() as sess:
    ...:      sess.run(init)
    ...:
    ...:     for epoch in range(training_epochs):
    ...:         avg_cost = 0.
    ...:         # Loop over all batches
    ...:         for i in range(total_batch):
    ...:             #batch_x, batch_y = inputdata.next_batch(batch_size)
                     TO BE IMPLEMENTED
    ...:             batch_x = inputX
    ...:             batch_y = inputY
    ...:             _, c = sess.run([optimizer, cost], feed_dict={X:
                     batch_x, Y: batch_y})
    ...:             # Compute average loss
    ...:             avg_cost += c / total_batch
    ...:         if epoch % display_step == 0:
    ...:             print("Epoch:", '%04d' % (epoch+1), "cost={:.9f}".
                     format(avg_cost))
    ...:             avg_set.append(avg_cost)
    ...:             epoch_set.append(epoch + 1)
    ...:
    ...:     print("Training phase finished")
```

Before running the session, add a few lines of instructions to view the results of the learning phase.

```
In [9]: with tf.Session() as sess:
    ...:      sess.run(init)
    ...:
    ...:     for epoch in range(training_epochs):
    ...:         avg_cost = 0.
```

```
...:         # Loop over all batches
...:         for i in range(total_batch):
...:              #batch_x, batch_y = inputdata.next
                  _batch(batch_size) TO BE IMPLEMENTED
...:              batch_x = inputX
...:              batch_y = inputY
...:              _, c = sess.run([optimizer, cost], feed
                  _dict={X: batch_x, Y: batch_y})
...:              # Compute average loss
...:              avg_cost += c / total_batch
...:         if epoch % display_step == 0:
...:             print("Epoch:", '%04d' % (epoch+1), "cost={:.9f}".
                 format(avg_cost))
...:             avg_set.append(avg_cost)
...:             epoch_set.append(epoch + 1)
...:
...:     print("Training phase finished")
...:     last_result = sess.run(y_, feed_dict = {X: inputX})
...:     training_cost = sess.run(cost, feed_dict = {X:
         inputX, Y: inputY})
...:     print("Training cost =", training_cost)
...:     print("Last result =", last_result)
...:
```

Finally, you can execute the session and obtain the following results of the learning phase.

```
Epoch: 0001 cost=0.454545379
Epoch: 0051 cost=0.454544961
Epoch: 0101 cost=0.454536706
Epoch: 0151 cost=0.454053283
Epoch: 0201 cost=0.391623020
Epoch: 0251 cost=0.197094142
Epoch: 0301 cost=0.145846367
Epoch: 0351 cost=0.121205062
Epoch: 0401 cost=0.106998600
Epoch: 0451 cost=0.097896501
```

```
Epoch: 0501 cost=0.091660112
Epoch: 0551 cost=0.087186322
Epoch: 0601 cost=0.083868250
Epoch: 0651 cost=0.081344165
Epoch: 0701 cost=0.079385243
Epoch: 0751 cost=0.077839941
Epoch: 0801 cost=0.076604150
Epoch: 0851 cost=0.075604357
Epoch: 0901 cost=0.074787453
Epoch: 0951 cost=0.074113965
Epoch: 1001 cost=0.073554687
Epoch: 1051 cost=0.073086999
Epoch: 1101 cost=0.072693743
Epoch: 1151 cost=0.072361387
Epoch: 1201 cost=0.072079219
Epoch: 1251 cost=0.071838818
Epoch: 1301 cost=0.071633331
Epoch: 1351 cost=0.071457185
Epoch: 1401 cost=0.071305975
Epoch: 1451 cost=0.071175829
Epoch: 1501 cost=0.071063705
Epoch: 1551 cost=0.070967078
Epoch: 1601 cost=0.070883729
Epoch: 1651 cost=0.070811756
Epoch: 1701 cost=0.070749618
Epoch: 1751 cost=0.070696011
Epoch: 1801 cost=0.070649780
Epoch: 1851 cost=0.070609920
Epoch: 1901 cost=0.070575655
Epoch: 1951 cost=0.070546091
Training phase finished
Training cost = 0.0705207
Last result = [[ 0.994959 0.00504093]
[ 0.97760069 0.02239927]
[ 0.95353836 0.04646158]
[ 0.91986829 0.0801317 ]
```

```
[ 0.93176246 0.06823757]
[ 0.27190316 0.7280969 ]
[ 0.40035316 0.59964687]
[ 0.04414944 0.9558506 ]
[ 0.17278962 0.82721037]
[ 0.01200284 0.98799717]
[ 0.19901533 0.80098462]]
```

Now you can view the data collected in the `avg_set` and `epoch_set` lists to analyze the progress of the learning phase.

```
In [10]: plt.plot(epoch_set,avg_set,'o',label = 'MLP Training phase')
    ...: plt.ylabel('cost')
    ...: plt.xlabel('epochs')
    ...: plt.legend()
    ...: plt.show()
    ...:
```

You can analyze the learning phase of the neural network by following the trend of the cost value, as shown in Figure 9-14.

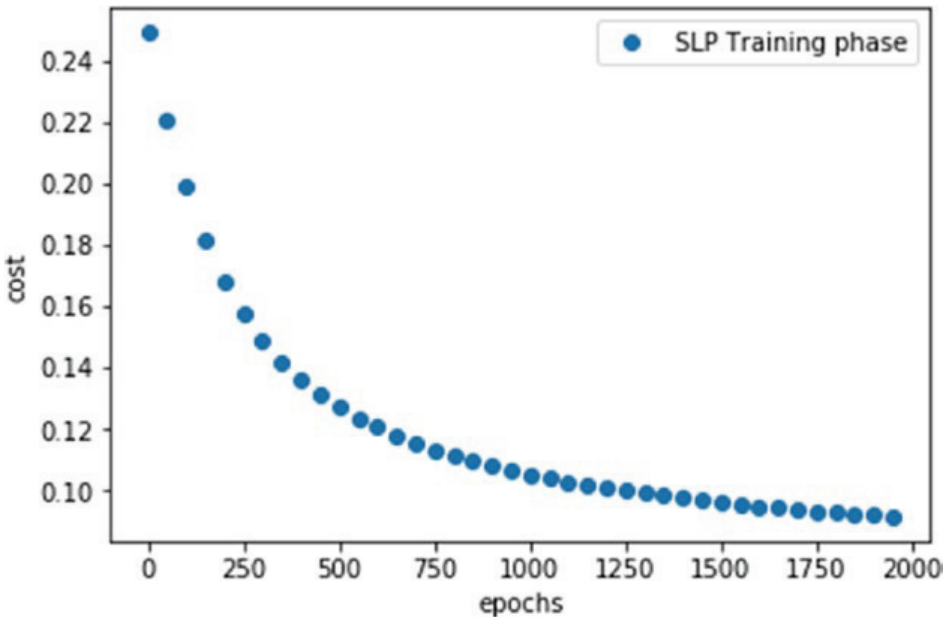

Figure 9-14. *The cost value decreases during the learning phase (cost optimization)*

In Figure 9-14, you can see that during the learning epochs there is a huge initial improvement as far as the cost optimization is concerned, then in the final part, the epoch improvements become smaller and then converge to zero.

From the analysis of the graph, however, it can be ascertained that the learning cycle of the neural network has been completed in the assigned epoch cycles. So you can consider the neural network as learned. Now you can move on to the evaluation phase.

Test Phase and Accuracy Calculation

Now that you have an educated neural network, you can make the evaluation text and calculate the accuracy.

Now that you have a neural network instructed to perform the assignment, you can move on to the evaluation phase. Then you will calculate the accuracy of the model you generated.

To test this MLP neural network model, you will use the same testing set used in the SLP neural network example.

```
In [11]: #Testing set
    ...: testX = np.arr
ay([[1.,2.25],[1.25,3.],[2,2.5],[2.25,2.75],[2.5,3.],
[2.,0.9],[2.5,1.2],[3.,1.25],[3.,1.5],[3.5,2.],[3.5,2.5]])
    ...: testY = [[1.,0.]]*5 + [[0.,1.]]*6
    ...:
```

It is not necessary now to view this testing set since you already did so in the previous section (you can check it if necessary).

Launch the session with the training test and evaluate the correctness of the results obtained by calculating the accuracy.

```
In [12]: with tf.Session() as sess:
    ...:     sess.run(init)
    ...:
    ...:     for epoch in range(training_epochs):
    ...:         for i in range(total_batch):
    ...:             batch_x = inputX
    ...:             batch_y = inputY
    ...:             _, c = sess.run([optimizer, cost],
```