CHAPTER 9

Deep Learning with TensorFlow

This chapter introduces an overview of the world of deep learning and the artificial neural networks on which its techniques are based. Furthermore, among the Python frameworks for deep learning, you will use *TensorFlow,* which is 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. In particular, in this third edition, the explanations and example codes are based on the new TensorFlow 2.x version, which has seen the incorporation of Keras and a complete upheaval in the modules and implementation paradigms.

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 read detailed definitions of these three terms. You 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 machine learning. Then you 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 hope 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. They had 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, can 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.*

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-9532-8_9

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, AI matured in its ability to assess risks, resources, and make 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 problems are still under study, thanks to 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 (the 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, this chapter refers 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.

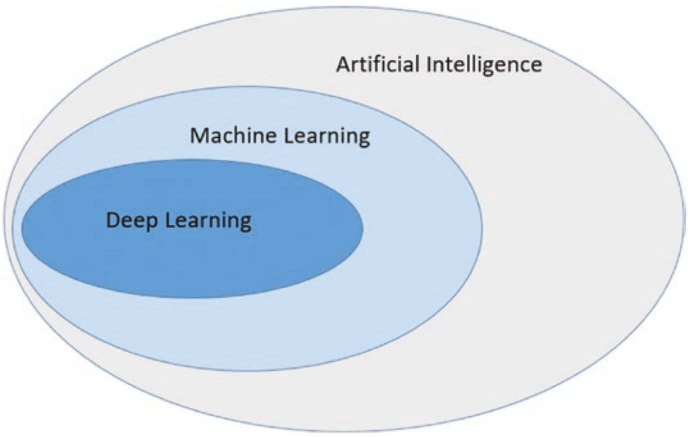


Figure 9-1. *Schematization of the relationship between artificial intelligence, machine learning, and deep learning*

Deep Learning

In this section, you learn about some significant factors that led to the development of deep learning and see how, in the last few years, there have been 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 they were powerful, were not able to perform these kinds of operations efficiently. 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 not required by neural networks.

A new kind of hardware has developed in recent decades, the *Graphics Processing Unit (GPU),* thanks to the enormous commercial drive of the video game 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, tensors (multidimensional matrices) are used, carrying out many mathematical operations. It is precisely this kind of work that GPUs can 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 the 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*).

Today a lot of data related to the problems you want to solve with the deep learning techniques are easily available, many of them in free form (as 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.

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 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 can learn, thanks to their particular structure that refers to the biological neural circuits.

In this section, you 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 in 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.

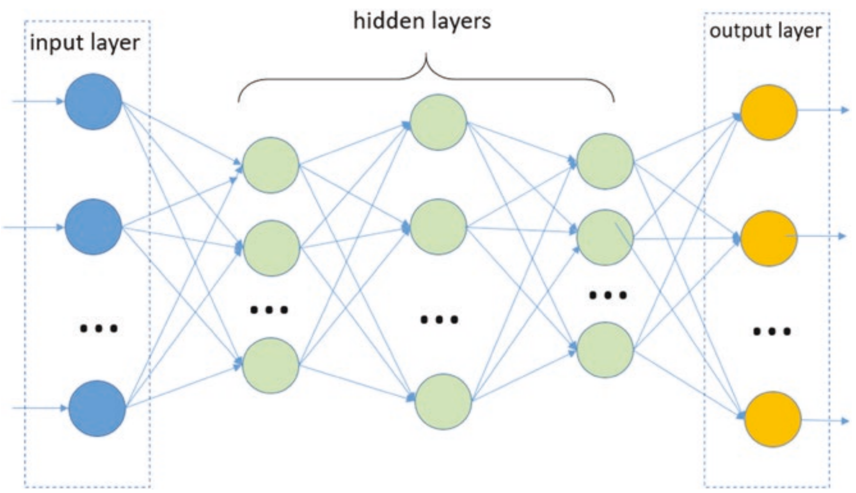


Figure 9-2. *A schematization of how a generic artificial neural network is structured*

The basic units are called *nodes* (the darker 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 their 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 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 preestablished 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 it 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.

In general, you have a flow of data that will enter the neural network (from left to right), 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 neural network model and was designed by Frank Rosenblatt in 1958. Its architecture is represented in Figure 9-3.

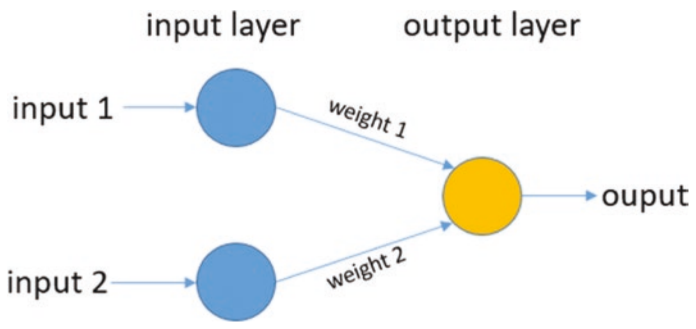


Figure 9-3. *The Single Layer Perceptron (SLP) architecture*

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.

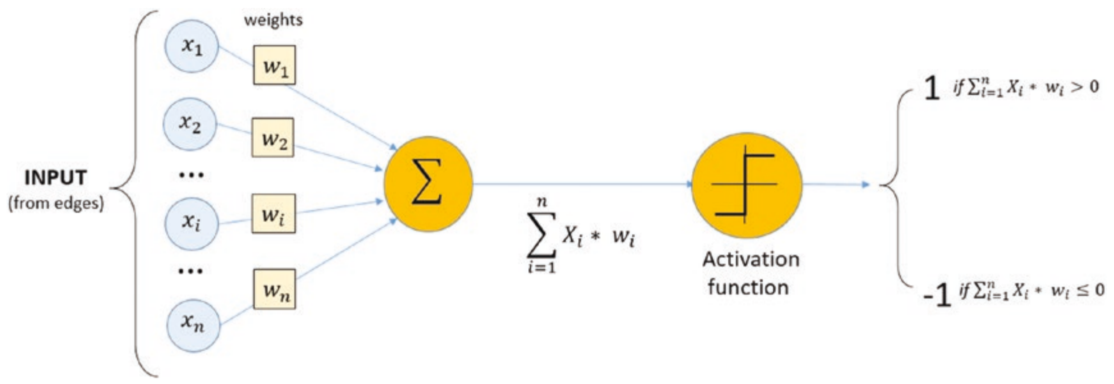

weights
INPUT
(from edges)
x_1
x_2
x_i
x_n
w_1
w_2
w_i
w_n
$\sum$
$\sum_{i=1}^{n} X_i * w_i$
Activation function
1 $if \sum_{i=1}^{n} X_i * w_i > 0$
-1 $if \sum_{i=1}^{n} X_i * w_i \leq 0$

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.

$$\sum_{i=0}^{n} w_i\, x_i = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = s$$

The total signal s is 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.

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

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

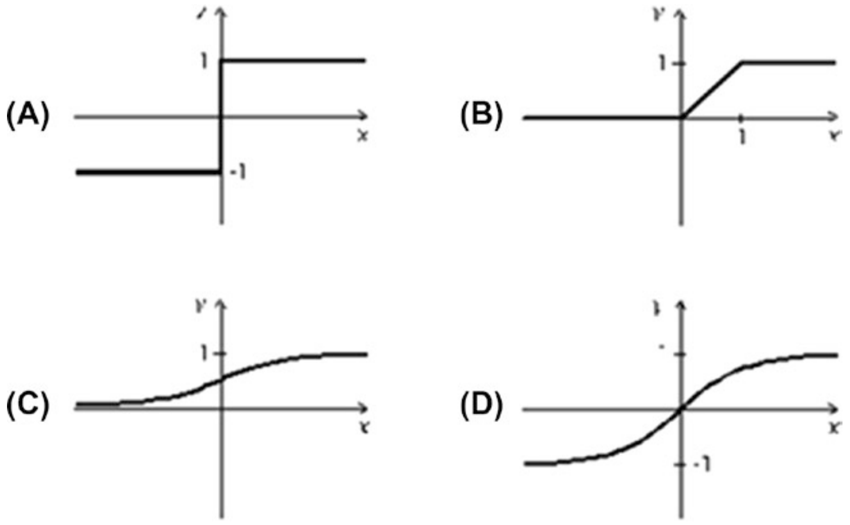


Figure 9-5. *Some examples of activation functions*

Now that you've seen the structure of the SLP neural network, you can now 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. The weights of the `wi` synapses are slightly modified in each cycle. Each learning cycle is called an *epoch*. In order to carry out the learning, you have to use appropriate input data, called the *training sets* (you have already used them in depth in 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 are 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 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 obtained and expected values, the degree of ability of the neural network to solve the problem of deep learning will be known. Often the percentage of cases guessed compared to the wrong ones is used to indicate this value, and it is called *accuracy*.

Multilayer Perceptron (MLP)

A more complex and efficient architecture is *Multilayer 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.

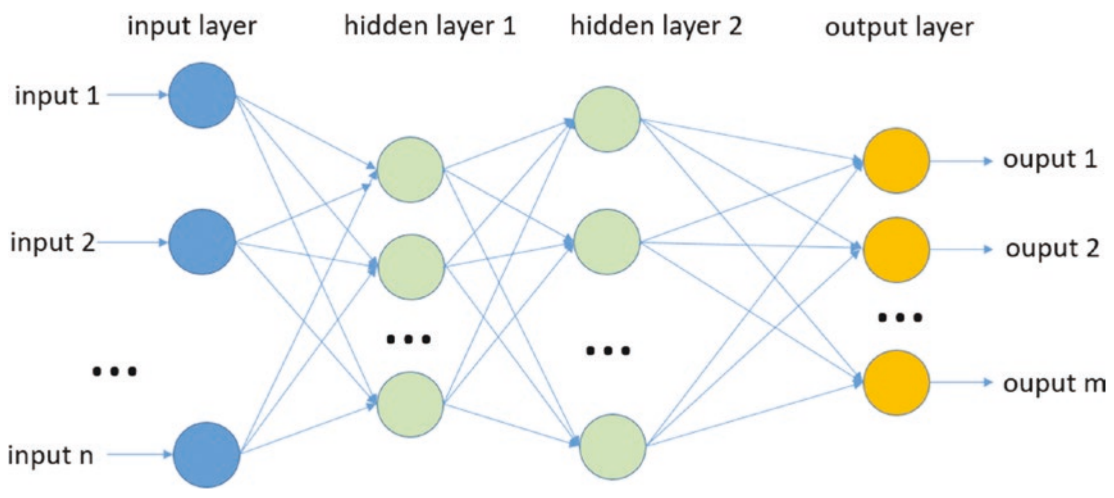


Figure 9-6. *The Multilayer Perceptron (MLP) architecture*

At the end of the learning phase, you 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 obtained and expected values, the degree of ability of the neural network to solve the problem of deep learning will be known. Often, the percentage of cases guessed compared to 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 makes the neural network capable of learning 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*, which is used to effectively modify the weights of the various connections to minimize the cost function, in order to converge the output values quickly and progressively 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 (`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 updates 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 of TensorFlow (`https://github.com/tensorflow/tensorboard`).
- *TensorFlow Fold*—Produces beautiful dynamic calculation charts (`https://github.com/tensorflow/fold`)
- *TensorFlow Transform*—Creates and manages input data pipelines (`https://github.com/tensorflow/transform`)

TensorFlow: Data Flow Graph

TensorFlow (`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 them, exploiting them in such a way as to make mathematical calculations.

The graph created internally in 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.

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.

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, for the study of complex physical systems through the calculation of partial differentials, and so on.

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 see how to install this framework, understand the differences between the old 1.x version and the new one, and the key features of the latter.

TensorFlow 2.x vs TensorFlow 1.x

As anticipated at the beginning of the chapter, in this third edition, the text and example code related to the use of TensorFlow for deep learning have been completely rewritten. This is because the new version of TensorFlow 2.x has been introduced since 2019. There are many changes that have been made. Many of the modules present in the TensorFlow 1.x release have been removed or moved. Keras has been completely incorporated as a neural network management module, and therefore most of the previous programming mechanisms and paradigms (for example, those present in the second edition with TensorFlow 1.x) are no longer compatible.

Given the large amount of programs developed in recent years with TensorFlow 1.x, it's important that these programs continued to compatible, and therefore usable. Rushing to address this issue, Google enabled a way to continue using the old code without having to rewrite much.

At the beginning of the code, you simply replace the classic import line:

```
import tensorflow as tf
```

with the following line:

```
import tensorflow.compat.v1 as tf
```

This replacement should guarantee in most cases the perfect execution of code developed with TensorFlow 1.x, even if your current version is TensorFlow 2.x.

As for present projects, if you are about to develop a new deep learning project with TensorFlow, the only reasonable path is to follow the TensorFlow 2.x paradigms. That's why this chapter only mentions this latest version, without transcribing the old code.

Installing TensorFlow

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

If the TensorFlow library is already installed, it's useful to determine which version is present, especially regarding the differences between TensorFlow 1.x and TensorFlow 2.x. You can easily find that out by opening a Python session and inserting the following lines of code:

```
import tensorflow as tf
tf.__version__
```

If the 1.x version is present, the best thing to do is create a new virtual environment on which to install the 2.x version without compromising the configuration of modules installed on your system. If the library is not present on your system, you can easily install TensorFlow 2.x. As in the previous chapters, the optimal solution is to have the Anaconda platform and graphically install the TensorFlow package from Navigator. If you prefer to use the command line, still on Anaconda, you can enter the following:

```
conda install tensorflow
```

If, on the other hand, you don't have (or don't want) the Anaconda platform, you can safely install TensorFlow via PyPI.

```
pip install tensorflow
```

■ **Note** At the time of this writing, I found some incompatibility issues in Anaconda between TensorFlow and other libraries for virtual environments based on Python 3.10 and 3.11. I then created a virtual environment with Python 3.9 and didn't encounter any problems.

Programming with the Jupyter Notebook

Once TensorFlow is installed, you can start programming with this library. The examples in this chapter use Jupyter Notebook, but you can do the same things by opening a normal Python session.

With the latest versions of TensorFlow, the demand for resources becomes more and more preponderant. Deep learning with applications like IPython and Jupyter Notebook may not be possible without proper precautions. So in this case it is necessary to add the following lines of code in the first cell of the Notebook before starting to work (also in IPython):

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

In particular, by varying this environment variable, the OpenMP API active in the system on which you are operating is informed not to create problems if another instance of it is created (a case that leads to the crash of Jupyter Notebook).

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).

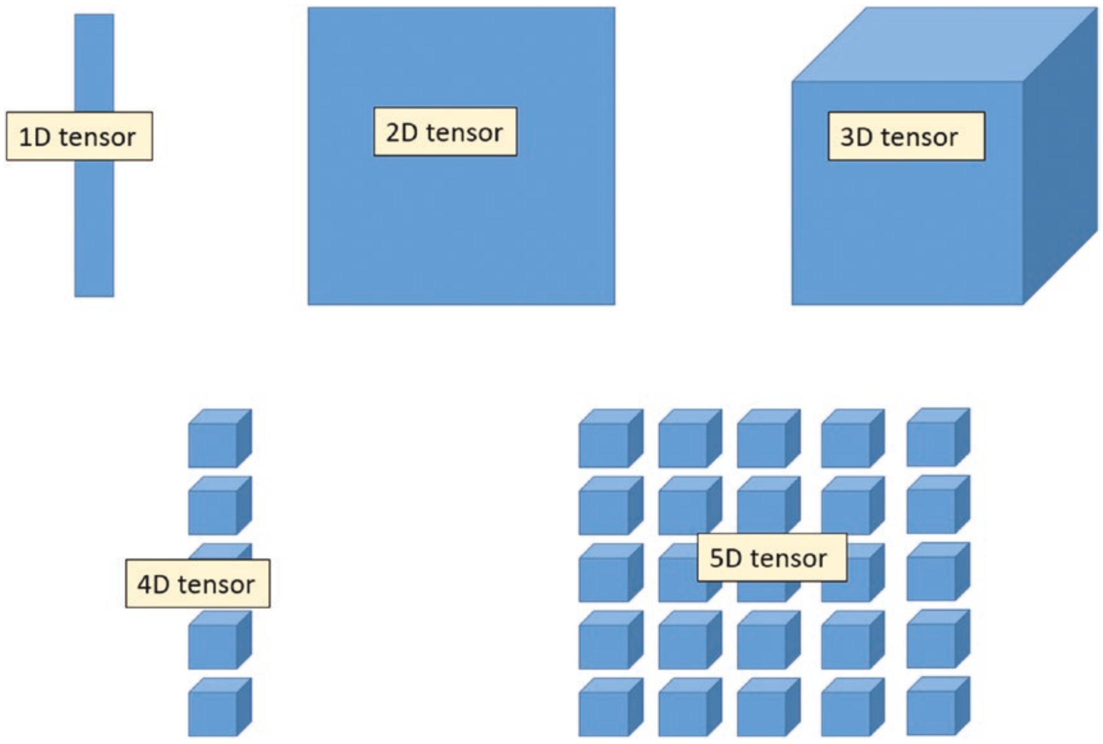


Figure 9-7. *Some representations of the tensors according to the different dimensions*

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**—The 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. You can start by defining one with this library.

```
import numpy as np
t = np.arange(9).reshape((3,3))
t
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

You can convert this multidimensional array into a TensorFlow tensor very easily, thanks to the `tf.convert_to_tensor()` function, which takes as a parameter the array to convert.

```
tensor = tf.convert_to_tensor(t)
tensor
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=int64, numpy=
    array([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])>
```

During the conversion from array to tensor, it is also possible to change the data type. In this case, you add the second optional parameter `dtyle`, specifying the new data type. For example, if you wanted to convert the integer input into floating numbers, you would write:

```
tensor2 = tf.convert_to_tensor(t, dtype='float64')
tensor2
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float64, numpy=
array([[0., 1., 2.],
           [3., 4., 5.],
           [6., 7., 8.]])>
```

As you can see, you have a tensor containing the same values and the 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 `0` values, you can use the `tf.zeros()` method.

```
t0 = tf.zeros((3,3),'float64')
t0
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float64, numpy=
array([[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.

```
t1 = tf.ones((3,3),'float64')
t1
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float64, numpy=
array([[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 exist), 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:

```
trand = tf.random.uniform((3, 3), minval=0, maxval=1, dtype=tf.float32)
trand
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0.99075377, 0.7289959 , 0.6183866 ],
           [0.51800334, 0.49188066, 0.01087034],
           [0.21716583, 0.29331267, 0.91550064]], dtype=float32)>
```

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 of 0 and standard deviation of 3, you will write:

```
tnorm = tf.random.normal((3, 3), mean=0, stddev=3)
tnorm
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[-1.2079163 , -0.88857937, 5.041537 ],
           [-3.7309105 , 3.157123 , -3.4958515 ],
           [ 4.219907 , 1.7997034 , -5.020906 ]], dtype=float32)>
```

Loading Data Into a Tensor from a pandas Dataframe

So far you've seen how to manually define the data within a tensor, or how to convert a NumPy array to a tensor. But a much more common operation in data analysis is having to insert the data present in a dataframe into a tensor. In fact, dataframes are one of the most commonly used formats during the data analysis process in Python. As you will see now, this operation is very simple in TensorFlow.

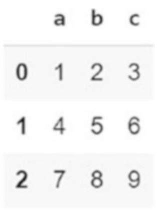
First import the pandas library into your Notebook.

```
import pandas as pd
```

Now define a simple dataframe as a basic example.

```
df = pd.DataFrame(np.array([[1, 2, 3],
                           [4, 5, 6],
                           [7, 8, 9]]),
                   columns=['a', 'b', 'c'])
df
```

For the output, you will get a dataframe like the one shown in Figure 9-8.



	a	b	c
0	1	2	3
1	4	5	6
2	7	8	9

Figure 9-8. *A simple example of a pandas dataframe*

To convert the data inside the dataframe into a tensor, you can use the `tf.convert_to_tensor()` function that you used previously. This in fact also accepts a pandas dataframe as an argument.

```
tensor_df = tf.convert_to_tensor(df)
tensor_df
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=int64, numpy=
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])>
```

As you can see from the result, the conversion was very easy. Rarely, however, will you have to load all the data present within a dataframe, but rather the values of one or more columns. So you will use the following form more often, making a selection of the columns (or data) that interest you within the dataframe.

```
tensor_df = tf.convert_to_tensor(df[['a','b']])
tensor_df

Out[ ]:
<tf.Tensor: shape=(3, 2), dtype=int64, numpy=
array([[1, 2],
       [4, 5],
       [7, 8]])>
```

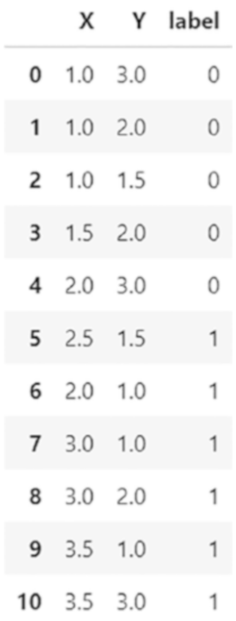
Loading Data in a Tensor from a CSV File

Another format in which the data to be analyzed is often available is within files, especially CSV files. Also in this case the data of a CSV file can be loaded into a tensor using pandas. In fact, the pandas library provides many functions for loading data contained in CSV files (and other formats) within the dataframe. These can then be converted into tensors using the procedure seen earlier with the `tf.convert_to_tensor()` function.

For example, you can load the data present in the `training_data.csv` file containing the training data that you will use in the following examples of the chapter during the development and study of some models. To do this, you use the pandas function called `read_csv()`.

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

By loading the data contained in the CSV, you will obtain as a result a dataframe like the one shown in Figure 9-9.



	X	Y	label
0	1.0	3.0	0
1	1.0	2.0	0
2	1.0	1.5	0
3	1.5	2.0	0
4	2.0	3.0	0
5	2.5	1.5	1
6	2.0	1.0	1
7	3.0	1.0	1
8	3.0	2.0	1
9	3.5	1.0	1
10	3.5	3.0	1

Figure 9-9. *The pandas dataframe containing the training dataset*

The training dataset contained in the dataframe is composed of three columns. The first two represent the X,Y coordinates of the points on a Cartesian plane, while the third column contains the labels, that is, the classes to which the correlated points belong. As you learned for machine learning (Chapter 8), training datasets are composed of two parts—the features and the labels. The same rule applies to deep learning, and therefore you have to extract two distinct tensors from the dataframe, one for the features and one for the labels.

```
df_features = df.copy()
df_labels = df_features.pop('label')
data_features = tf.convert_to_tensor(df_features)
data_features
Out [ ]:
<tf.Tensor: shape=(11, 2), dtype=float64, numpy=
array([[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. ]])>
```

Run the same operation for the tensor of the labels.

```
data_labels = tf.convert_to_tensor(df_labels)
data_labels
Out [ ]:
<tf.Tensor: shape=(11,), dtype=int32, numpy=array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])>
```

Using the `tf.conver_to_tensor()` function, you now have two tensors: one for the features containing the X,Y coordinates of the points and one for the labels containing the classes to which the points belong.

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.

```
t1 = tf.random.uniform((3, 3), minval=0, maxval=1, dtype=tf.float32)
t1
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0.29003692, 0.92972696, 0.41073143],
           [0.46694946, 0.46367037, 0.11636639],
           [0.31574678, 0.70260215, 0.0642364 ]], dtype=float32)>

t2 = tf.random.uniform((3, 3), minval=0, maxval=1, dtype=tf.float32)
t2
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0.23392928, 0.7185135 , 0.64518535],
           [0.6719583 , 0.7983806 , 0.10201716],
           [0.92533255, 0.32889807, 0.4179113 ]], dtype=float32)>
```

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

```
sum = tf.add(t1,t2)
sum
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0.5239662 , 1.6482404 , 1.0559168 ],
           [1.1389078 , 1.262051 , 0.21838355],
           [1.2410793 , 1.0315002 , 0.4821477 ]], dtype=float32)>

mul = tf.matmul(t1,t2)
mul
Out [ ]:
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1.0726491 , 1.0857601 , 0.453625 ],
         [0.5284779 , 0.7439676 , 0.39720213],
         [0.6054218 , 0.8089395 , 0.3022378 ]], dtype=float32)>
```

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

```
det = tf.linalg.det(t1)
det
Out [ ]:
<tf.Tensor: shape=(), dtype=float32, numpy=0.06581897>
```

The new `tf.linalg` module (introduced in TensorFlow 2.x) contains, in addition to the determinant calculation, many other algebraic operations on matrices, which are very useful during tensor calculations. These functions, along with the basic operations, allow you to implement many mathematical expressions that use tensors.

Developing a Deep Learning Model with TensorFlow

Once you have seen the tensors that are the basis of the data to be used in TensorFlow, you can proceed further, analyzing in brief the fundamental steps to create a deep learning model with neural networks and its training and testing phases with TensorFlow 2.x. Those steps are as follows:

- Definition of tensors (training and testing sets)
- Model building
- Model compiling
- Model training
- Model testing
- Predictions making

You learned about the first part, the one related to the preparation of tensors from the training and testing datasets, in the previous section. The next section covers model building.

Model Building

Regarding the construction of the model based on a neural network, you must define how the layers will be configured. As you saw in the first part of the chapter, each neural network is composed of one or more layers of neurons. With TensorFlow 2.x, it is not necessary to define each neuron and the individual connections that compose the network. But you can use one of the layers that's predefined in Keras.

For example:

```
tf.keras.layers.Dense
```

This corresponds to a layer of neurons where all the connections are made with the adjacent layer of neurons, called a "regular densely-connected NN layer."

Another widely used layer is as follows:

```
tf.keras.layers.Flatten
```

This layer is typically used at the beginning of the neural network, when the feature dataset is not one-dimensional. This allows you to flatten the data, making them single-dimensional and thus usable by the subsequent layers of the neural network.

To better understand, if you wanted to submit an image to a neural network, this would be composed of (nxn) pixels and therefore would be two-dimensional. Inserting a `Flatten` layer would make this training dataset one-dimensional.

Another widely used layer is as follows:

```
tf.keras.layers.Normalization
```

This layer, like the previous one, is also placed as the first layer of the neural network and is used to normalize the input data. This practice is very common in machine learning, making the data more easily actionable.

Therefore, thanks to the integration of Keras in TensorFlow 2.x, the building of the model can have many predefined layers, with a whole series of learning parameters inside that will vary during the training phase. These are already defined and therefore the model building operation is much easier.

In fact, to build a simple neural network like the ones you saw at the beginning of the chapter, it is sufficient to define `tf.keras.Sequential()` with the various predefined layers inside, in the order of construction.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape(128,128)),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dense(12)
])
```

The model just described therefore represents a two-layer neural network (without hidden layers) in which two-dimensional tensors, such as 128x128 pixel images, are made one-dimensional in the first layer, to then be passed to the first layer composed of 128 neurons. These in turn are connected to another layer of 12 neurons, which will probably correspond to 12 classes of belonging (as you will see later).

Model Compiling

Now that a model has been defined, it is necessary to compile it. This second step requires the choice of functions and training parameters:

- Loss function: Measures how accurate the model has to be during the training phase.
- Optimizer: Takes care of how and which parameters should be updated during the training phase.
- Metrics: The parameters used to monitor the progress of the training (and testing) of the model.

All these elements are already available within the Keras module, thus already providing a set of tools ready to use in a model.

Thus, compiling the model is reduced to a simple `compile()` function, in which these three elements are defined as arguments.

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

As you can see, in the new version of TensorFlow 2.x, the construction of a model and its compilation are very fast and high-level operations, contrary to the previous version, which required the detailed definition of most of these elements.

Model Training and Testing

Now that the model is compiled, you can move on to training and testing it. For these two phases, a dataset is used in which there are *features* (a series of variables that describe the subject under study) and *labels* (the solutions such as, for example, the classes to which they belong). This is very similar to what you saw with machine learning in the previous chapter (Chapter 8). The dataset is divided into a training dataset and a testing dataset, with the former being much larger.

Each feature is then subdivided into two tensors, until four tensors are obtained:

- `train_features`
- `train_labels`
- `test_features`
- `test_labels`

The first two tensors are used for model learning, using the `fit()` function.

```
model.fit(train_features, train_labels, epochs=100)
```

The number of epochs is the third parameter, which is the number of times the learning phase is performed. At each of these stages, the accuracy metrics should improve, thus signaling successful model learning. Once done, you need to have an educated model ready for testing. Here, you use the other two tensors (`test_features` and `test_labels`) to evaluate the model's ability to make predictions. The difference between these and the values contained in `test_labels` will provide the accuracy of these predictions.

For the testing phase, the `evaluate()` function is used.

```
test_loss, test_acc = model.evaluate( test_features, test_labels, verbose=2)
```

This function returns the loss and accuracy of the model as values.

Prediction Making

The last phase submits the model to the purpose for which it was created, making predictions on input data whose solution you do not know (for example, the class it belongs to).

During this phase, an additional layer is often added to the model, called Softmax. This is placed at the end of the neural network and is used to convert the output of the last layer (logits) into probability values. In this case, it is not necessary to recompile the newly trained model again, but this can be extended by adding this layer at the end. You have thus defined a new extended model.

```
probability_model = tf.keras.Sequential([
       model,
       tf.keras.layers.Softmax()
])
```

To complete the deep learning steps, the model makes predictions from data whose solutions you do not know. You create a new tensor and submit it to the `predict()` function.

```
predictions = probability_model.predict(samples_features)
```

An array with all predictions divided by the percentage is obtained as the returned value. It can be used to obtain the class to which it belongs.

```
np.argmax(prediction[i])
```

`i` is the ith element of the dataset.

Practical Examples with TensorFlow 2.x

At this point, in theory, you should have acquired enough basic knowledge to be able to start working with real examples. Let's put into practice what you have seen so far with different types of neural networks.

- Single Layer Perceptron (SLP)
- Multilayer Neural Network with One Hidden Layer
- Multilayer Neural Network with Two Hidden Layers

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. By using the concepts you learned during the chapter and gradually introducing new ones, you can implement a Single Layer Perceptron (SLP) neural network.

Before Starting

Before starting, open a new Jupyter Notebook or shut down and start the kernel again. Once the session is open it imports all the necessary modules:

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

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

Data To Be Analyzed

For the examples that you consider in this chapter, you will use a series of data that you used in Chapter 8, in particular in the section entitled "Support Vector Machines (SVMs)."

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 floating point values is due to the optimization calculation of deep learning. You will see later that the test results of the neural networks are 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 91 percent to the first class and 9 percent to the second class. You will see this in practice at the end of the section, but it is important to explain the concept to better understand the purpose of some values.

Based on the values taken from the example of SVMs in Chapter 8, you can define the following values.

```
#Training set
inputX = np.array([[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)
Out [ ]:
[[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]]
```

In reality, you have already seen these values previously, when we talked about how to import data from a CSV file by converting it to a pandas dataframe. Here, you use the NumPy array version (the same one used for machine learning, in particular in the section about the Support Vector Machines technique) to give an additional version of the starting data to be converted into tensors. Feel free to choose the format you prefer for the initial datasets.

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.

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

```
Out [ ]:
[0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
```

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

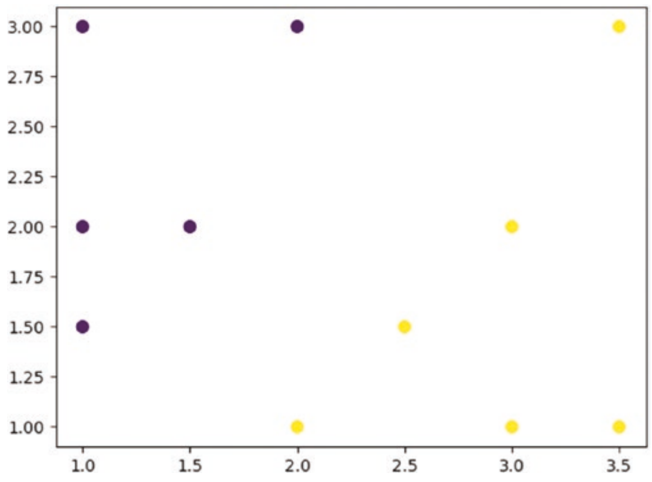


***Figure 9-10.** The training set is a set of Cartesian points divided into two classes of membership (light and dark)*

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

As you can see, the two classes are easily identifiable in two opposite regions. The first region covers the upper-left part, and 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 6 that is internal to the other points.

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

You then convert the values of the training arrays into tensors, as you have seen done previously.

```
train_features = tf.convert_to_tensor(inputX)
train_labels = tf.convert_to_tensor(inputY)
train_features
Out [ ]:
<tf.Tensor: shape=(11, 2), dtype=float64, numpy=
array([[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. ]])>
```

```
train_labels
Out [ ]:
<tf.Tensor: shape=(11, 2), dtype=float32, numpy=
array([[1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.]], dtype=float32)>
```

Now you can build the Single Layer Perceptron model based on the neural network in Figure 9-11, adding the various Keras layers to the model definition.

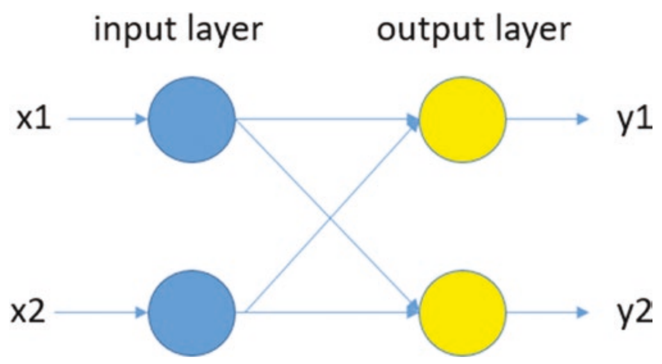


Figure 9-11. *The Single Layer Perceptron model used in this example*

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2),
    tf.keras.layers.Dense(2)
])
```

As you can see, it is a very simple model, but more than sufficient for making predictions in simple cases such as the one in question.

Because this is a binary classification problem, you can choose `BinaryCrossentropy` as the function loss for the compilation.

```
model.compile(optimizer='SGD',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Once the model is compiled, you can move on to training it. Choose 200 epochs, assuming more than enough time for the model to learn.

```
h = model.fit(train_features, train_labels, epochs=2000)
Out [ ]:
Epoch 1/2000
1/1 [==============================] - 1s 637ms/step - loss: 2.0271 - accuracy: 0.5455
Epoch 2/2000
1/1 [==============================] - 0s 4ms/step - loss: 1.9807 - accuracy: 0.5455
Epoch 3/2000
1/1 [==============================] - 0s 5ms/step - loss: 1.9356 - accuracy: 0.5455
Epoch 4/2000
1/1 [==============================] - 0s 9ms/step - loss: 1.8918 - accuracy: 0.5455
Epoch 5/2000
1/1 [==============================] - 0s 8ms/step - loss: 1.8493 - accuracy: 0.5455
Epoch 6/2000
1/1 [==============================] - 0s 6ms/step - loss: 1.8079 - accuracy: 0.5455
Epoch 7/2000
...
```

In the output, you will have all the learning situations epoch by epoch, through a scroll bar that shows the completion for each of them. The loss and accuracy value will then be shown next to each line of output.

However, it is clear that this output is not the easiest way to understand how this neural network model behaved during the learning phase. For this purpose, I have saved the output in the return value `h` (for history), which you can use for graphical visualizations that can help you.

Extract from the history variable the loss values corresponding to the various periods.

```
acc_set = h.history['loss']
epoch_set = h.epoch
```

Arrange these values in a plotting chart to see the learning progress graphically, thanks to the `matplotlib` library.

```
# return list of every 100th item in a larger list
plt.plot(epoch_set[0::100],acc_set[0::100], 'o', label='Training phase')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
```

As a result, you will obtain a graph similar to the one shown in Figure 9-12.

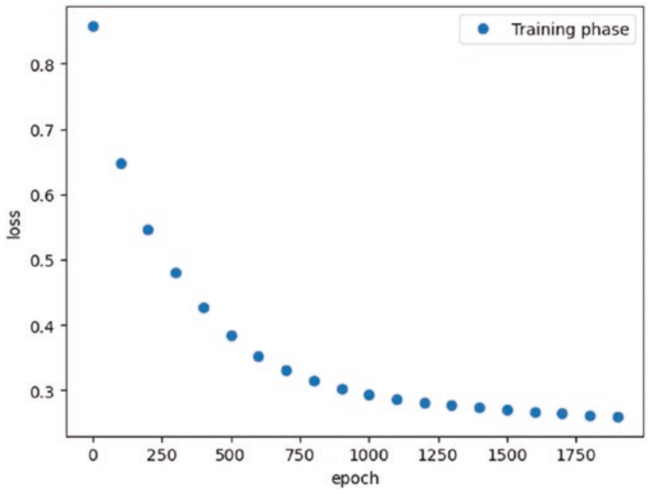


***Figure 9-12.** The less value decreases during the learning phase (less optimization)*

Now move on to the testing phase, defining the other two tensors.

```
testX = np.array([[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

test_features = tf.convert_to_tensor(testX)
test_labels = tf.convert_to_tensor(testY)
```

Evaluate the accuracy of the newly educated SLP model.

```
test_loss, test_acc = model.evaluate(test_features, test_labels, verbose=2)
Out [ ]:
1/1 - 0s - loss: 0.1812 - accuracy: 1.0000 - 233ms/epoch - 233ms/step
```

As you can see, the accuracy is at best, given the simplicity of the classification. So you can expect a very good level of prediction of the newly educated model.

Now move on to the proper classification, passing to the neural network a very large amount of data (points on the Cartesian plane) without knowing to which class they belong. This is, in fact, the moment that the neural network informs you about the possible classes.

To this end, the program simulates experimental data, creating points on the Cartesian plane that are completely random. For example, you can generate an array containing 1,000 random points.

```
exp_features = 3*np.random.random((1000,2))
```

Now you extend the model with Softmax to obtain an output of the probability of the different points belonging to the two classes.

```
probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])
```

Now make the predictions of the experimental data with the model you just extended.

```
predictions = probability_model.predict(exp_features)
```

Let's determine the probability of a single point belonging to the two classes. For example, the first:

```
predictions[0]
Out [ ]:
array([0.073105 , 0.9268949], dtype=float32)
```

If, on the other hand, we want to know directly which of the two classes it belongs to, we write the following code obtaining the class it belongs to (0 for the first and 1 for the second):

```
np.argmax(predictions[0])
Out [ ]:
1
```

Instead of analyzing point by point in a textual way, there is a graphical way to visualize the result of these predictions. In the previous scatterplots, you classified the points on the Cartesian plane with two colors (yellow and purple). Now that you considered the probability of a point belonging to these two classes, for all intermediate probabilities, an intermediate color of the gradient will be displayed. It will fade from yellow to purple depending on how close it is to one of the two classes.

```
yc = predictions[:,1]
plt.scatter(exp_features[:,0],exp_features[:,1],c=yc, s=50, alpha=1)
plt.show()
```

Running the code, you will get a scatterplot like the one shown in Figure 9-13.

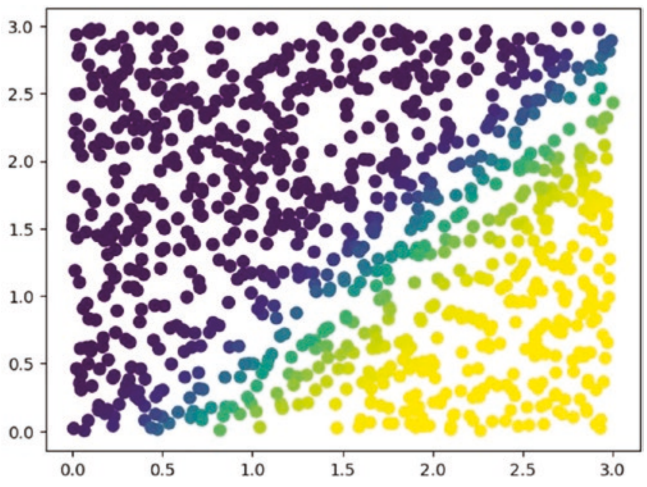


Figure 9-13. *A scatterplot with all the experimental points and the estimate of the classes to which they belong*

As you can see according to the shades, two areas of classification are delimited on the plane, with a color gradient in the central part (green color) indicating the zones of uncertainty.

The classification results can be made more comprehensible and clearer by deciding to establish based them on the probability of the point belonging to one or the other class. If the probability of a point belonging to a class is greater than 0.5, it will belong to it.

You can modify the previous scatterplot by requiring that each point belong to one or another class, according to the most probable option.

```
yc = np.round(predictions[:,1])
plt.scatter(exp_features[:,0],exp_features[:,1],c=yc, s=50, alpha=1)
plt.show()
```

Running the code, you will get a scatterplot similar to the one shown in Figure 9-14.

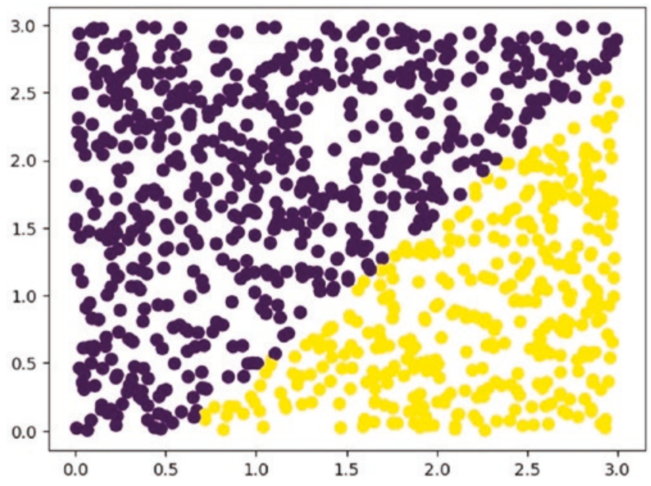


Figure 9-14. *The points delimit the two regions corresponding to the two classes*

In the scatterplot shown in Figure 9-14, you can clearly see the two regions of the Cartesian plane that characterize the two classes of belonging.

Multilayer Perceptron (with One Hidden Layer) with TensorFlow

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

Start a new Jupyter Notebook, or continue with the same one, but reset the kernel.

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

To build this model, which compared to the previous one has a hidden layer of two neurons, you define a model similar to the previous one, but with an intermediate Dense layer.

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2),
    tf.keras.layers.Dense(2),
    tf.keras.layers.Dense(2)
])
```

The next step is to compile the model, choosing an optimization method. For MLP neural networks, a good choice is the Adam optimization method. Instead, keep the loss function and the metrics unchanged.

```
model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

As with the previous SLP model, train it using the same training dataset.

```
h = model.fit(train_features, train_labels, epochs=2000)
Out [ ]:
Epoch 1/2000
1/1 [==============================] - 0s 426ms/step - loss: 1.8568 - accuracy: 0.4545
Epoch 2/2000
1/1 [==============================] - 0s 3ms/step - loss: 1.8473 - accuracy: 0.4545
Epoch 3/2000
1/1 [==============================] - 0s 3ms/step - loss: 1.8378 - accuracy: 0.4545
Epoch 4/2000
1/1 [==============================] - 0s 646us/step - loss: 1.8284 - accuracy: 0.4545
Epoch 5/2000
1/1 [==============================] - 0s 0s/step - loss: 1.8190 - accuracy: 0.4545
Epoch 6/2000
1/1 [==============================] - 0s 0s/step - loss: 1.8097 - accuracy: 0.4545
Epoch 7/2000
1/1 [==============================] - 0s 14ms/step - loss: 1.8004 - accuracy: 0.4545
Epoch 8/2000
...
```

Extract the training history and create the same graph showing the behavior of the model during the training process.

```
acc_set = h.history['loss']
epoch_set = h.epoch# return list of every 50th item in a larger list
plt.plot(epoch_set[0::50],acc_set[0::50], 'o', label='Training phase')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
```

Running the previous code, you will get a chart like the one shown in Figure 9-15.

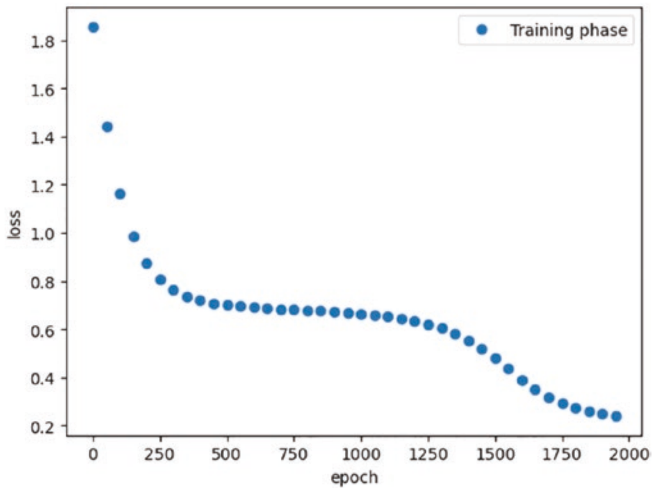


Figure 9-15. *The learning curve of the MLP model shows two distinct optimization phases*

As you can see, even a more complex neural network shows different stages of learning. In this case, the choice of 2,000 epochs was fundamental to obtain an increase in the forecasting capacity of the model. If you had stopped at 1,000 epochs, the loss would have been 0.7 instead of 0.2.

As the result of testing this last model, you get an accuracy of 100 percent and a loss value of 01.

```
test_loss, test_acc = model.evaluate(test_features, test_labels, verbose=2)
Out [ ]:
1/1 - 0s - loss: 0.1602 - accuracy: 1.0000 - 105ms/epoch - 105ms/step
```

Multilayer Perceptron (with Two Hidden Layers) with TensorFlow

In this section, you extend the previous structure by adding two neurons to the first hidden layer (four in all) and adding a second hidden layer with two neurons.

As you did previously, start a new Jupyter Notebook and write the necessary code of the previous examples, or restart the kernel and execute the cells with the necessary code.

```
).
model = tf.keras.Sequential([
    tf.keras.layers.Dense(2),
    tf.keras.layers.Dense(4),
    tf.keras.layers.Dense(2),
    tf.keras.layers.Dense(2)
])
model.compile(optimizer='Adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Then train the new two hidden layer MLP model with the same training dataset you used previously.

```
h = model.fit(train_features, train_labels, epochs=2000)
Out [ ]:
Epoch 1/2000
1/1 [==============================] - 1s 826ms/step - loss: 0.6980 - accuracy: 0.4545
Epoch 2/2000
1/1 [==============================] - 0s 3ms/step - loss: 0.6970 - accuracy: 0.4545
Epoch 3/2000
1/1 [==============================] - 0s 3ms/step - loss: 0.6961 - accuracy: 0.4545
Epoch 4/2000
1/1 [==============================] - 0s 0s/step - loss: 0.6953 - accuracy: 0.4545
Epoch 5/2000
1/1 [==============================] - 0s 0s/step - loss: 0.6945 - accuracy: 0.4545
Epoch 6/2000
1/1 [==============================] - 0s 0s/step - loss: 0.6938 - accuracy: 0.4545
Epoch 7/2000
1/1 [==============================] - 0s 3ms/step - loss: 0.6931 - accuracy: 0.4545
Epoch 8/2000
1/1 [==============================] - 0s 0s/step - loss: 0.6925 - accuracy: 0.4545
...
```

Also for this model, you can analyze the learning phase of the neural network by displaying the loss values as the epochs increase in a chart.

```
acc_set = h.history['loss']
epoch_set = h.epoch
# return list of every 50th item in a larger list
plt.plot(epoch_set[0::50],acc_set[0::50], 'o', label='Training phase')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
```

Running the code, you will get a plot like the one in Figure 9-16.

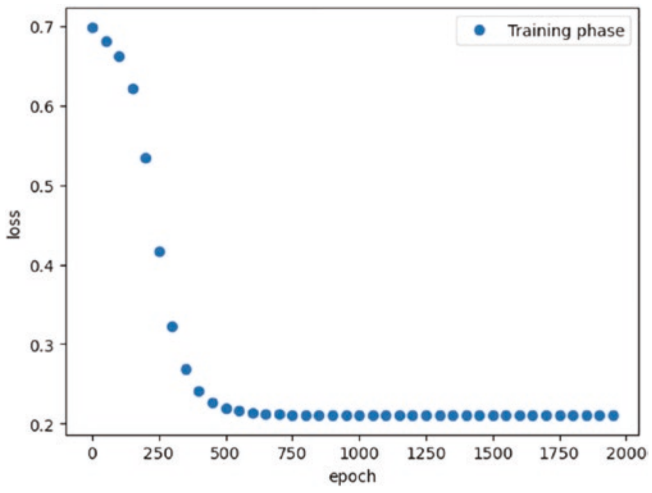


Figure 9-16. *The trend of the loss during the learning phase for an MLP with two hidden layers*

From what you can see in Figure 9-16, learning in this case is much faster than the previous case (at 1,000 epochs, you would be fine).

```
test_loss, test_acc = model.evaluate(test_features, test_labels, verbose=2)
Out [ ]:
1/1 - 0s - loss: 0.0951 - accuracy: 1.0000 - 124ms/epoch - 124ms/step
```

The optimized loss is the best of those obtained so far, and only at 700 epochs (0.0951 versus 0.16 in the previous case, at 2,000 epochs). It is clear that adding the hidden layer of four neurons has made the model faster and more efficient.

Conclusions

In this chapter, you learned about the branch of machine learning that uses neural networks as a computing structure, called deep learning. You read an overview of the basic concepts of deep learning, which involves neural networks and their structure. Finally, thanks to the TensorFlow library, you implemented different types of neural networks, such as Perceptron Single Layer and Perceptron Multilayer.

Deep learning, with all its techniques and algorithms, is a very complex subject, and it is practically impossible to treat it properly in one chapter. However, you have now become familiar with deep learning and can begin implementing more complex neural networks.