CHAPTER 2

Hands-on with a Single Neuron

After reading about learning with neural networks in the previous chapter, you are now ready to explore their most fundamental component, the neuron. In this chapter, you learn the main components of the neuron. You also learn how to solve two classical statistical problems (i.e., linear regression and logistic regression) by using a neural network with just one neuron. To make things a bit more fun, you do that using real datasets. We discuss the two models and explain how to implement the two algorithms in Keras.

First, we briefly explain what a neuron is, including its typical structure and its main characteristics (e.g., the activation function, weights, etc.). Then we look at how you can formally express it in a matrix (this step is fundamental to obtaining optimized codes and exploiting all TensorFlow and NumPy functionalities). Finally, we look at some code examples in Keras. You can find the complete Jupyter Notebooks discussed in this chapter at `https://adl.toelt.ai`.

A Short Overview of a Neuron's Structure

Deep learning is based on large and complex networks made up of a large number of simple computational units. Companies at the forefront of research are dealing with networks with 160 billion parameters [1]. To put things in perspective, this number is half of the number of stars in our galaxy or 1.5 times the number of people that ever lived. On a basic level, neural networks are large sets of differently interconnected units,[1]

[1] More advanced architecture, such as convolutional or recurrent neural networks, have a structure that is more complex than what is described here. In this chapter, we describe the neurons used to build so-called Feed-Forward Neural Networks (FFNN).

U. Michelucci, *Applied Deep Learning with TensorFlow 2*, https://doi.org/10.1007/978-1-4842-8020-1_2

each performing a specific (and usually relatively easy) computation. They remind me of Lego building blocks, where you can build very complicated things using elementary and fundamental units.

Due to the biological parallel with the brain [2], these basic units are known as *neurons*. Each neuron (at least the ones most commonly used and the ones we use in this book) does a straightforward operation: it takes a certain number of inputs (real numbers) and calculates an output (also a real number). Remember that the inputs are indicated in this book with $x_i \in \mathbb{R}$ (real numbers) where $i = 1, 2, \ldots, n_x$, $i \in \mathbb{N}$ is an integer, and n_x is the number of input attributes (often called features). As an example of input features, you can imagine the age and weight of a person (so we would have $n_x = 2$). x_1 could be the age and x_2 could be the weight. In real life, the number of features can be easily very large (of the order of $10^2 - 10^3$ or higher).

There are several kinds of neurons that have been extensively studied. In this book we concentrate on the most commonly used one. The neuron we are interested in simply applies a function to a linear combination of all the inputs. In a more mathematical form, given n_x real parameters $w_i \in \mathbb{R}$ (with $i = 1, 2, \ldots, n_x$) and a constant $b \in \mathbb{R}$ (usually called bias), the neuron will first calculate what is traditionally indicated in literature with z:

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x} + b$$

It will then apply a function f (normally non-linear) to z, giving the output $\hat{y}$

$$\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x} + b\right)$$

Note Practitioners generally use the following terminology: w_i are called *weights*, b is the *bias*, x_i are the *input features*, and f is the *activation function*.

Let's summarize the neuron computational steps again.

1. Linearly combine all inputs x_i, calculating $z = w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x} + b$.
2. Apply f to z giving the output $\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x} + b\right)$.

In the literature, you can find numerous representations for neurons. Figure 2-1 shows a graphical representation of the mathematical operations we just discussed to obtain the output $\hat{y}$ from the inputs.

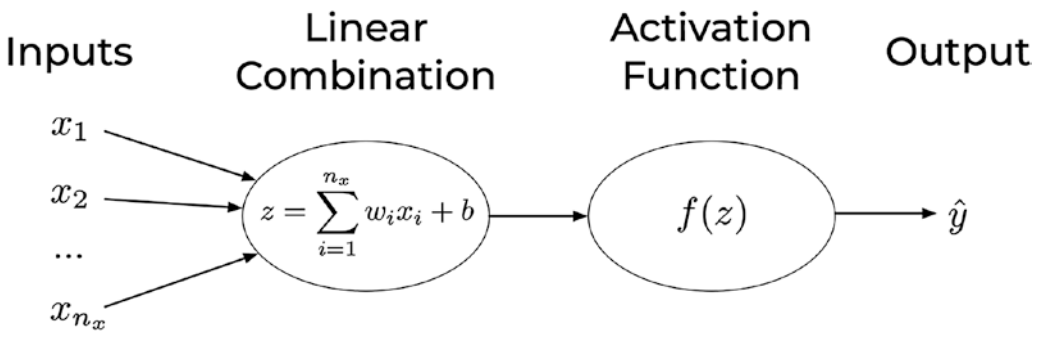

Figure 2-1. *A representation of a single neuron with the operation highlighted. This is also called the computational graph of a single neuron, or in other words, a graphical representation of the operations needed to calculate* $\hat{y}$ *from the inputs*

Figure 2-1 must be interpreted in this way:

- The inputs are not placed in a bubble, simply to distinguish them from nodes that perform an actual calculation.
- The weight's names are typically not written. The expected behavior is that before passing the inputs to the central bubble (or node), the inputs will be multiplied by the relative weight. The first input x_1 will be multiplied by w_1, x_2 by w_2, and so on.
- The first bubble (or node) will sum the inputs multiplied by the weights (the $x_i w_i$ for $i = 1, 2, \ldots, n_x$) and then sum the result to the bias b.
- The last bubble (or node) will finally apply to the resulting value the activation function f.

All the neurons we deal with in this book have exactly this structure. Very often an even simpler representation is used, as in Figure 2-2. In such a case, unless otherwise stated, it's understood that the output is as follows

$$\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x} + b\right)$$

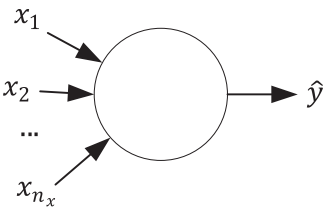

Figure 2-2. *A simplified version of Figure 2-1. Unless otherwise stated, it is usually understood that the output is* $\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x} + b\right)$. *The weights are often not explicitly reported in the neuron representation*

A Short Introduction to Matrix Notation

When dealing with big datasets, the number of features is typically large (n_x will be large) and so it is better to use a vector notation for the features and weights. The inputs can be indicated with

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_{n_x} \end{pmatrix}$$

where we have indicated the vector with a bold-faced $\boldsymbol{x}$. For the weights, we use the same notation

$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_{n_x} \end{pmatrix}$$

For consistency with formulas that we will use later, to multiply $\boldsymbol{x}$ and $\boldsymbol{w}$, we use matrix multiplication notation and therefore we write

$$\boldsymbol{w}^T \boldsymbol{x} = \left(w_1 \ldots w_{n_x}\right) \begin{pmatrix} x_1 \\ \vdots \\ x_{n_x} \end{pmatrix} = w_1 x_1 + w_2 x_2 + \ldots + w_{n_x} x_{n_x}$$

Where $\boldsymbol{w}^T$ indicates the transpose of $\boldsymbol{w}$. z can then be written with this vector notation as

$$z = \boldsymbol{w}^T \boldsymbol{x} + b$$

and the neuron output $\hat{y}$ as

$$\hat{y} = f(z) = f\left(\boldsymbol{w}^T \boldsymbol{x} + b\right)$$

Let's now summarize the different components that define this neuron and the notation we use in this book:

- $\hat{y}$: Neuron (and later network) output
- $f(z)$: Activation function (sometimes called a transfer function) applied to z
- $\boldsymbol{w}$: Weights (vector with n_x components)
- b: Bias

An Overview of the Most Common Activation Functions

There are many activation functions at your disposal to change the output of a neuron. Remember, an activation function is simply a mathematical function that transforms z in the output $\hat{y}$. Let's look at the most common ones.

Identity Function

This is the most basic function that you can use. It's typically indicated with $I(z)$. It merely returns the input value, unchanged (see Figure 2-3). Mathematically we can write it as

$$f(z) = I(z) = z$$

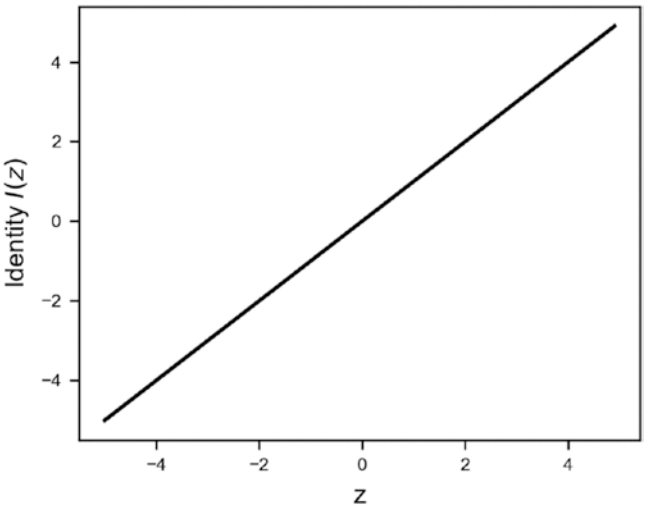

Figure 2-3. *The identity function*

This simple function will come in handy when discussing linear regression with one neuron later in the chapter. Implementing it[2] in Python with NumPy is incredibly trivial:

```
def identity(z):
    return z
```

Sigmoid Function

This is a very commonly used function that gives only values between 0 and 1. It is usually indicated with $\sigma(z)$

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

It is primarily used for classification models, where we want to predict the probability as an output (remember that a probability may only assume values between 0 and 1).

The calculation can be written in this form using NumPy functions

```
s = np.divide(1.0, np.add(1.0, np.exp(-z)))
```

[2] Note that you will not have to implement it yourself in Python. Keras offers this function, and many more, out of the box.

Note It is very useful to know that if we have two NumPy arrays, `A` and `B`, the following are equivalent: `A/B` is equivalent to `np.divide(A,B)`, `A+B` is equivalent to `np.add(A,B)`, `A-B` is equivalent to `np.subtract(A,B)`, and `A*B` is equivalent to `np.multiply(A,B)`. If you know object-oriented programming, we say that in NumPy basic operations like `/`, `*`, `+` and `-` are *overloaded*. Note also that these four basic operations in NumPy act element-by-element (also called element-wise).

We can write the sigmoid function in a more readable (at least for humans) form as

```
def sigmoid(z):
    s = 1.0 / (1.0 + np.exp(-z))
    return s
```

As stated, `1.0 + np.exp(-z)` is equivalent to `np.add(1.0, np.exp(-z))` and `1.0 / (np.add(1.0, np.exp(-z)))` to `np.divide(1.0, np.add(1.0, np.exp(-z)))`. We want to draw your attention to another point in the formula. `np.exp(-z)` will have the dimensions of `z` (usually a vector that will have a length equal to the number of observations), while `1.0` is a scalar (a one-dimensional entity). How can Python sum the two? What happens is called *broadcasting*. Python, subject to certain constraints, "broadcasts" the smaller array (in this case, the `1.0`) across the larger one so that, at the end, the two have the same dimensions. In this case, the `1.0` becomes an array of the same dimensions of `z`, all filled with `1.0`. This is an important concept to understand, as it is very useful. You do not have to transform numbers in arrays, for example. Python will take care of this process for you. The rules on how broadcasting works in other cases are rather complex and go beyond this book's scope. However, it is important to know that Python is doing something in the background.

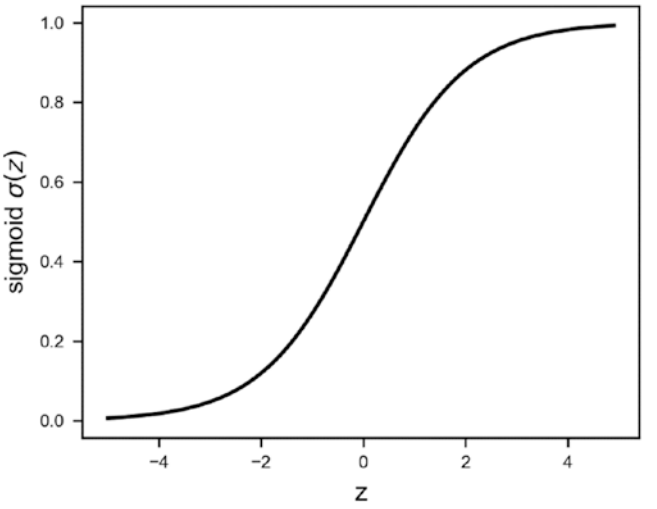

Figure 2-4. *The sigmoid activation function is a s-shaped function that goes from 0 to 1*

The sigmoid activation function (that you can see in Figure 2-4) is especially used for models where we must predict the probability as an output, as logistic regression (remember that a probability may only assume values between 0 and 1). Note that in Python, if z is big enough, it can happen that the function returns exactly 0 or 1 (depending on the sign of z) for rounding errors. In classification problems we will calculate $\log\sigma(z)$ or $\log(1-\sigma(z))$ very often, and therefore this can be a source of errors in Python since it will try to calculate log0, which is not defined. For example, you can start seeing `nan` appearing while calculating the cost function (more on that later).

Note Although $\sigma(z)$ should never be exactly 0 or 1, while programming in Python the reality can be quite different. It may happen that, due to a very big z (positive or negative), Python will round the results to exactly 0 or 1. This may give you errors while calculating the cost function for classification, since you need to calculate $\log\sigma(z)$ and $\log(1-\sigma(z))$. Therefore, Python will try to calculate log0, which is not defined. This may happen, for example, if you don't correctly normalize the input data or if you don't correctly initialize the weights. For the moment, it's important to remember that although everything seems under control mathematically, the reality while programming can be more difficult. It's good to keep this in mind while debugging models that give, for example, `nan` as a result for the cost function.

Tanh (Hyperbolic Tangent) Activation Function

The hyperbolic tangent is also an s-shaped curve that goes from -1 to 1, as shown in Figure 2-5.

$$f(z) = \tanh(z)$$

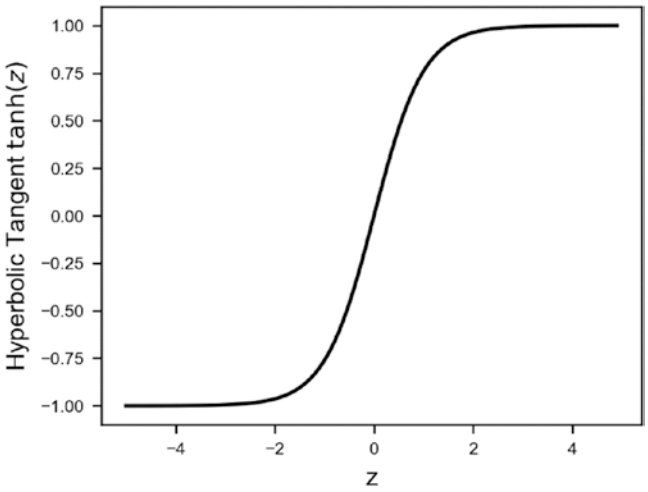

Figure 2-5. *The tanh (or hyperbolic function) is an s-shaped curve that goes from -1 to 1*

In Python, this can be easily implemented

```
def tanh(z):
    return np.tanh(z)
```

ReLU (Rectified Linear Unit) Activation Function

The ReLU has the following formula (see Figure 2-6):

$$f(z) = \max(0, z)$$

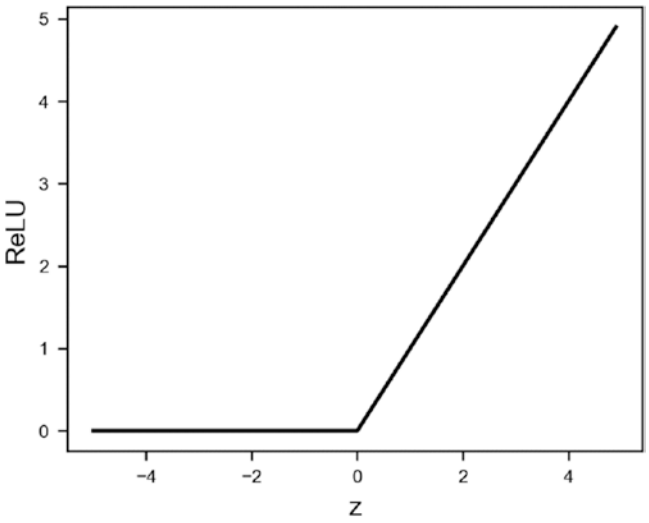

Figure 2-6. *The ReLU function*

It is worth it to spend a few moments to see how to implement ReLU in a smart way in Python. Note that when you start using TensorFlow, it is implemented for you, but it's still very instructive to see how different Python implementations can make a difference when implementing complex deep learning models.

In Python you can implement the ReLU function in several ways. Here are four different ways (try to understand why they work before going on):

1. `np.maximum(x, 0, x)`
2. `np.maximum(x, 0)`
3. `x * (x > 0)`
4. `(abs(x) + x) / 2`

The four methods have very different execution speeds. Let's generate a NumPy array with 10^8 elements:

```
x = np.random.random(10**8)
```

Now we measure the time needed by the four different versions of the ReLU function when applied to it. Let the following code run

```
x = np.random.random(10**8)
print("Method 1:")
%timeit -n10 np.maximum(x, 0, x)

print("Method 2:")
%timeit -n10 np.maximum(x, 0)

print("Method 3:")
%timeit -n10 x * (x > 0)

print("Method 4:")
%timeit -n10 (abs(x) + x) / 2
```

The results are

```
Method 1:
2.66 ms ± 500 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Method 2:
6.35 ms ± 836 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Method 3:
4.37 ms ± 780 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Method 4:
8.33 ms ± 784 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The difference is stunning. Method 1 is four times faster than method 4. The `numpy` library is highly optimized, with many routines written in C. But knowing how to code efficiently still makes a difference and can have a great impact. Why is `np.maximum(x, 0, x)` faster than `np.maximum(x, 0)`? The first version updates `x` in place, without creating a new array. This can save a lot of time, especially when the arrays are big. If you don't want to (or can't) update the input vector in place, you can still use the `np.maximum(x, 0)` version.

An implementation could look like this

```
def relu(z):
    return np.maximum(z, 0)
```

Note Remember that, when optimizing your code, even small changes can make a huge difference. In deep learning programs, the same chunk of code is repeated millions and billions of times, so even a small improvement can have a huge impact in the long run. Spending time optimizing your code is a necessary step that will pay off.

Leaky ReLU

The Leaky ReLU (also known as Parametric Rectified Linear Unit) is given by the formula

$$f(z)=\begin{cases}\alpha z & for \quad z<0\\ z & for \quad z\geq 0\end{cases}$$

with α a parameter typically of the order of 0.01. See Figure 2-7.

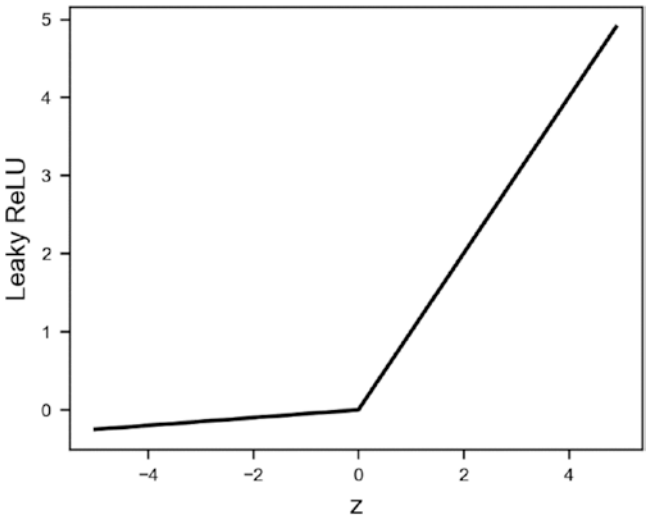

Figure 2-7. *The Leaky ReLU activation function with α = 0.05. This value has been chosen to make the difference between x > 0 and x < 0 more marked. Smaller values for α are typically used. Testing your model is required to find the best value*

In Python, this can be implemented if the `relu(z)` function has been defined as

```
def lrelu(z, alpha):
  return relu(z) - alpha * relu(-z)
```

The Swish Activation Function

Ramachandran, Zopf, and Le from Google Brain [4] recently studied a new activation function that shows great promise in the deep learning world; they named it Swish. It is defined as

$$f(z) = z\sigma(\beta z)$$

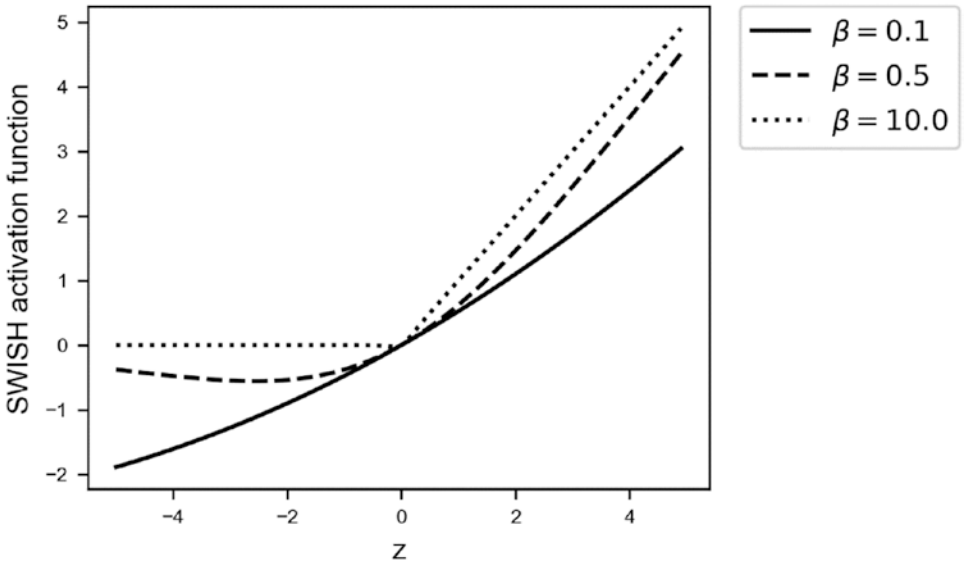

Figure 2-8. *The Swish activation function for three different values of the parameter* β

where β is a learnable parameter (see Figure 2-8). Their studies have shown that simply replacing ReLU activation functions with Swish improves classification accuracy on ImageNet by 0.9%, which in today's deep learning world is a lot. ImageNet is a large database of images that is often used to benchmark new network architectures or algorithms, as in this case, networks with a different activation functions. You can find more information about ImageNet at http://www.image-net.org/.

Other Activation Functions

There are many other activation functions, but those are rarely used. As a reference, here are some additional ones. The list is by no means comprehensive and should serve the purposes of giving you an idea of the variety of activation functions that can be used when developing neural networks.

- ArcTan

$$f(z)=\tan^{-1} z$$

- Exponential Linear unit (ELU)

$$f(z)=\begin{cases}\alpha\left(e^{z}-1\right) & for \quad z<0\\ z & for \quad z\geq 0\end{cases}$$

- Softplus

$$f(z)=\ln\left(1+e^{z}\right)$$

Note Practitioners almost always use only two activation functions: the sigmoid and the ReLU (with probably the ReLU in the lead). With both you can achieve good results. Both can, given a complex enough network architecture, approximate any nonlinear function.[3] Remember that when using TensorFlow, you do not have to implement them by yourself. TensorFlow offers an efficient implementation for you to use. But is important to know how each activation function behaves to understand when to use which one.

Now that we briefly discussed all the necessary components, you can use the neuron on some real problems. Let's first see how to implement a neuron in Keras and then how to perform linear and logistic regression with it.

How to Implement a Neuron in Keras

Building a network with a single neuron in Keras is straightforward and can be done with

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

[3] Montufar G., Pascanu R., Cho K. and Bengio Y., "On the Number of Linear Regions of Deep Neural Networks," https://papers.nips.cc/paper/5422-on-the-number-of-linear-regions-of-deep-neural-networks.pdf, last accessed 10th Jan. 2018; Fortuner B., "Can neural networks solve any problem?," https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6, last accessed 10th Jan. 2018.

The **Sequential** class groups a linear stack of layers into a `tf.keras.Model`. In this straightforward case, we need just one layer made by a single neuron, defined by the **layers.Dense** command, which specifies **1** unit (neuron) inside a layer and the shape of the input dataset. The `Dense` class implements densely connected neural networks' layers (more on that in the next chapters).

In the next paragraphs, you see two practical examples of how to use this simple approach—choosing the right activation function and the proper loss function—to solve two different problems, namely linear regression and logistic regression.

Python Implementation Tips: Loops and NumPy

As you have just seen, Keras does all the dirty work for you. Of course, you can also implement the neuron from scratch, using Python standard functionalities such as lists and loops, but those tend to become very slow as the number of variables and observations grows. A good rule of thumb is to avoid loops when possible and use NumPy (or TensorFlow) methods as often as possible.

It is easy to understand how fast NumPy can be (and how slow loops are). Let's start by creating two standard lists of random numbers in Python with 10^7 elements in each:

```
import random
lst1 = random.sample(range(1, 10**8), 10**7)
lst2 = random.sample(range(1, 10**8), 10**7)
```

The actual values are not relevant for our purposes. We are simply interested in how fast Python can multiply two lists, element by element. The time reported was measured on a 2017 Microsoft Surface laptop and will vary greatly depending on the hardware that the code runs on. We are not interested in the absolute values, but only in how much faster NumPy is in comparison to standard Python loops. If you are using a Jupyter Notebook, it is useful to know how to time Python code in a cell. To do this, you can use a "magic command." Those commands start (in a Jupyter Notebook) with %% or with %. It's a good idea to check the official documentation to better understand how they work (http://ipython.readthedocs.io/en/stable/interactive/magics.html).

Coming back to the test, let's measure how much time a standard laptop takes to multiply, element by element, the two lists with standard loops. Using the code

```
%%timeit
ab = [lst1[i]*lst2[i] for i in range(len(lst1))]
```

gives us the following result (note that on your computer, you will probably get different numbers):

```
2.06 s ± 326 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The code needed roughly two seconds averaged over seven runs. Now let's try to do the same multiplication, but this time using NumPy

```
%%timeit
out2 = np.multiply(list1_np, list2_np)
```

We first converted the two lists to numpy arrays with the following code

```
import numpy
list1_np = np.array(lst1)
list2_np = np.array(lst2)
```

This time we get the following results

```
20.8 ms ± 2.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The NumPy code needed only 21ms or, in other words, it was roughly 100 times faster than the code with standard loops. NumPy is faster for two reasons: the underlying routines are written in C, and it uses vectorized code as much as possible to speed up calculations on a large amount of data.

Note Vectorized code refers to operations performed on multiple components of a vector (or a matrix) at the same time (in one statement). Passing matrixes to NumPy functions is an excellent example of *vectorized code*. NumPy will perform operations on big chunks of data simultaneously, obtaining much better performance than standard Python loops since the latter must operate on one element at a time. Note that part of the excellent performance NumPy shows is also due to the underlying routines being written in C.

While training deep learning models, you will find yourself doing this kind of operation over and over. Such a speed gain will make the difference between having a model that can be trained and one that will never give you a result.

Linear Regression with a Single Neuron

This section explains how to build your first model in Keras and how to use it to solve a basic statistical problem. You can, of course, perform linear regression quickly by applying traditional math formulas or using dedicated functions such as those found in Scikit-learn. For example, you can find the complete implementation of linear regression from scratch with NumPy, using the analytical formulas (see `http://adl.toelt.ai/single_neuron/Linear_regression_with_numpy.html`) in the online version of the book. However, it is instructive to follow this example, since it gives a practical understanding of how the building blocks of deep learning architectures (i.e., neurons) work.

If you remember, we have said many times that NumPy is highly optimized to perform several parallel operations simultaneously. To get the best performance possible, it's essential to write your equations in matrix form and feed the matrixes to NumPy. In this way, the code will be as efficient as possible. Remember: avoid loops at all costs whenever possible.

The Dataset for the Real-World Example

To make things a bit more interesting, let's use an instructive dataset. We will use the so-called radon dataset [3]. Radon is a radioactive gas that enters homes through contact points with the ground. It is a carcinogen that is the primary cause of lung cancer in non-smokers. Radon levels vary significantly from household to household. The dataset contains measured radon levels in U.S. homes by county and state. The `activity` label is the measured radon concentration in pCi/L (referred to as the *target* variable, i.e., the one we want to predict using a linear regression model). Important features are:

- Floor (the floor of the house in which the measurement was taken)
- County (the U.S. county in which the house is located)
- uppm (a measurement of the uranium level of the soil by county)

This dataset fits a classical regression problem well since it contains a continuous variable (radon activity) to predict. The model that we will build is made of one neuron and will fit a linear function to the different features.

You do not need to understand or study the features. The goal here is to understand how to build a linear regression model with what you have learned. Generally, in a machine learning project, you would first study your input data, check its distribution, quality, missing values, and so on. But we skip this part to concentrate on the implementation with Keras.

Note In machine learning, the variable we want to predict is usually called the *target variable*.

Now, let's look at the data. We will skip all the import and load details for simplicity and concentrate on the fundamental steps of the code, such as dataset preparation, model creation, and performance evaluation. Remember that you can find the complete code on the online version of the book. We start by checking how many observations we have

```
num_counties = len(county_name)
num_observations = len(radon_features)
print('Number of counties included in the dataset: ', num_counties)
print('Number of total samples: ', num_observations)
```

The code will give the following results

```
Number of counties included in the dataset:  85
Number of total samples:  919
```

So, we have 919 different measurements of radon activity in 85 distinct counties. The `radon_features.head()` command will give the following table as output (the first five lines of the `pandas` dataframe)

	floor	county	log_uranium_ppm	pcterr
0	1	0	0.502054	9.7
1	0	0	0.502054	14.5
2	0	0	0.502054	9.6
3	0	0	0.502054	24.3
4	0	1	0.428565	13.8

We have four features (`floor`, `county`, `log_uranium_ppm`, and `pcterr`) that we will use as predictors of radon activity.

As suggested, we have prepared the data in matrix form. Let's briefly review the notation, which will come in handy when building the neuron. Normally we have many observations (919 in this case). We use an upper index to indicate the different observations between parentheses. The i^{th} observation is indicated with $\boldsymbol{x}^{(i)}$, and the j^{th} feature of the i^{th} observation is indicated with $\boldsymbol{x}_j^{(i)}$. We indicate the number of observations with m.

Note In this book, m is the *number of observations* and n_x is the *number of features.* Our j^{th} feature of the i^{th} observation is indicated with $\boldsymbol{x}_j^{(i)}$. In deep learning projects, the bigger the m, the better. So be prepared to deal with a massive number of observations.

---

n_x in this example is equal to 4, while m is equal to 919. The entire set of inputs (features and observations) can be therefore written using the following notation

$$X = \begin{pmatrix} x_1^{(1)} & \dots & x_4^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_4^{(m)} \end{pmatrix}$$

where each row is an observation, and each column represents a feature in the matrix X that has the dimensions $m \times 4$.

Dataset Splitting

In any machine learning project, to check how the model generalizes to unseen data, you need to split the dataset into different subsets.[4] When you build a machine learning model, you first need to train the model, and then you have to test it (i.e., verify the model's performances on novel data). The most common way to do this is to split the dataset into two subsets: 80% of the original dataset to train the model (the more data you have, the better your model will perform) and the remaining 20% to test it.[5]

[4] It is assumed here that you know why it is important.

[5] The ratio 80/20 is simply a convention; you can choose 75/25, 70/30, or anything in between. Normally 80/20 is chosen since you want to have as much training data as possible and enough test data to make your testing reasonable.

The following code splits the dataset randomly into two parts with the following proportions: 80%/20%.

```
np.random.seed(42)
rnd = np.random.rand(len(radon_features)) < 0.8

train_x = radon_features[rnd] # training dataset (features)
train_y = radon_labels[rnd] # training dataset (labels)
test_x = radon_features[~rnd] # testing dataset (features)
test_y = radon_labels[~rnd] # testing dataset (labels)

print('The training dataset dimensions are: ', train_x.shape)
print('The testing dataset dimensions are: ', test_x.shape)
```

This code will give as output the following lines

```
The training dataset dimensions are:  (733, 4)
The testing dataset dimensions are:  (186, 4)
```

We will use 733 observations to train our linear regression model, and we will then evaluate it on the remaining 186 observations.

Linear Regression Model

Keep in mind that using a single-neuron model is overkill for a linear regression task. We could solve linear regression exactly without using gradient descent or a similar optimization algorithm. As mentioned, you can find an exact regression solution example, implemented with NumPy library, in the book's online version.

Given that the dataset can be expressed as a matrix (X) and the label we want to predict as a column vector (y), when we employ one neuron to perform linear regression, we simply try to find the best weights (W) that appear in the following equation:

$$\hat{y} = WX + b$$

The weights and bias need to be chosen so that that the network output is as similar as possible to the expected target variable.

If you remember how a neuron is structured, you can easily see that, for the neuron to give as output a linear combination of the inputs, we need to use the *identity activation function.* How can we measure how close the neuron's outputs are to the target variables? The difference is measured by using the Mean Squared Error (MSE) function.

$$L(\boldsymbol{w},b)=\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}-\boldsymbol{w}^{T}\boldsymbol{x}^{(i)}-b\right)^{2}$$

where the sum is over all m observations. This is the typical loss function chosen in a regression problem. By minimizing $L(\boldsymbol{w}, b)$ with respect to the weights and bias, we can find their optimal values.

Minimizing $L(\boldsymbol{w}, b)$ is done with an optimizer. If you remember from the last chapter, gradient descent is the most basic example of an optimizer and could be used to solve this problem. Since is not available out of the box in TensorFlow, we use the RMSProp optimizer for practical reasons. Don't worry if you don't know how it works. Just know that it is simply a more intelligent version of the GD algorithm. You learn how it works in detail in the following chapters.

Keras Implementation

If you have no experience with Keras, you can consult the appendixes of this book. There you will find an introduction to Keras that will give you enough information to be able to understand the following discussion.

Implementing what we discussed with Keras is straightforward. The following function builds the one-neuron model for linear regression.

```
def build_model():

  model = keras.Sequential([
    layers.Dense(1, input_shape = [len(train_x.columns)])
  ])

  optimizer = tf.keras.optimizers.RMSprop(
    learning_rate = 0.001)
```

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

    return model
```

Let's analyze what this code does:

- First of all, we defined the network structure (also called network architecture) with the `keras.Sequential` class, adding one layer[6] made of one neuron (`layers.Dense`) and with `input dimensions` equal to the number of features used to build the model. The activation function is the one set by default, i.e. the identity function.
- Then, we defined the optimizer (`tf.keras.optimizers.RMSprop`), setting the learning rate to 0.001. The optimizer is the algorithm that Keras will use to minimize the cost function. We use the RMSprop algorithm in this example.
- Finally, we compile the model (i.e., we configure the model for training), setting its loss function (i.e., the cost function to be minimized), its optimizer, and the metric to be calculated during performance evaluation (`model.compile`). The function returns the built `model` as a single Python object.

The *learning rate* is a very important parameter for the optimizer. In fact, it strongly influences the convergence of the minimization process. It is a common and good behavior to try different learning rate values and observe how the model's convergence changes.

Now, let's apply the `build_model` function and look at the model summary

```
model = build_model()
model.summary()
```

[6] We discuss at length what layers are in the next chapter. For the moment, just ignore this if you don't understand what it means.

This code gives the following output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 5
=================================================================
Total params: 5
Trainable params: 5
Non-trainable params:
_________________________________________________________________
```

Thus, we have five parameters to be trained—the weights associated with the four features, plus the bias.

The Model's Learning Phase

Training our neuron means finding the weights and biases that minimize the chosen cost function (the MSE in this case). The *minimization process* is iterative; therefore, it is necessary to decide when to stop it. For this example, we simply set a fixed number of epochs. We train the model for 1000 epochs and then look at the results in terms of the MSE.

```
EPOCHS = 1000

history = model.fit(
  train_x, train_y,
  epochs = EPOCHS, verbose = 0
)
```

As you can see, training the model in Keras is straightforward. It is enough to apply the `fit` method to our model object. `fit` takes as inputs the training data and the number of epochs.

The cost function clearly decreases for a while and then stays almost constant. That is a good sign, indicating that the cost function has reached a minimum. That does not mean that our model is good or that it will give good predictions. This tells us only that learning has somehow worked. A very good way to immediately visualize the loss function decrease is by plotting the cost function vs. the number of epochs. This can be seen in Figure 2-9.

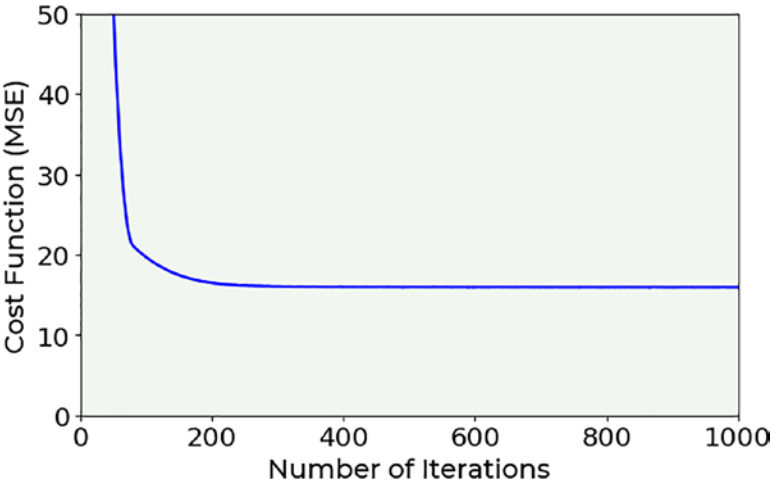

Figure 2-9. *The cost function behavior during the model training, applied to the radon dataset with a learning rate of 0.001*

Looking at Figure 2-9, you can see that, after 400 epochs, the cost function remains almost constant in its value, indicating that a minimum has been reached.

Since we are doing a linear regression, we are interested in the coefficients of the linear equation. Those are the weights of our neuron. The first four are the linear regression coefficients, while the last one is the bias term. You can compare these numbers with the ones obtained by performing linear regression with traditional math formulas and using the NumPy library in the online version of the book. To get the weights in Keras, you simply use the `get_weights()` call.

```
weights = model.get_weights() # return a numpy list of weights
print(weights)
```

which returns

```
[array([[-6.6795307e-01],
       [ 2.7279984e-03],
       [ 2.8733387e+00],
       [-2.0828046e-01]], dtype=float32),
 array([4.2394686], dtype=float32)]
```

Those are the weights we were expecting.

Model's Performance Evaluation on Unseen Data

To determine if the model you just trained is suited for unseen data, you must check its performance over a dataset containing only unseen data (the test set). Then, predicted radon activity values are compared with real values (`test_y`) by simply plotting predictions vs. true values, as shown in Figure 2-10. A perfect model will show points distributed over the black line. The less precise the predictions are, the more spread out the points will be around the diagonal.

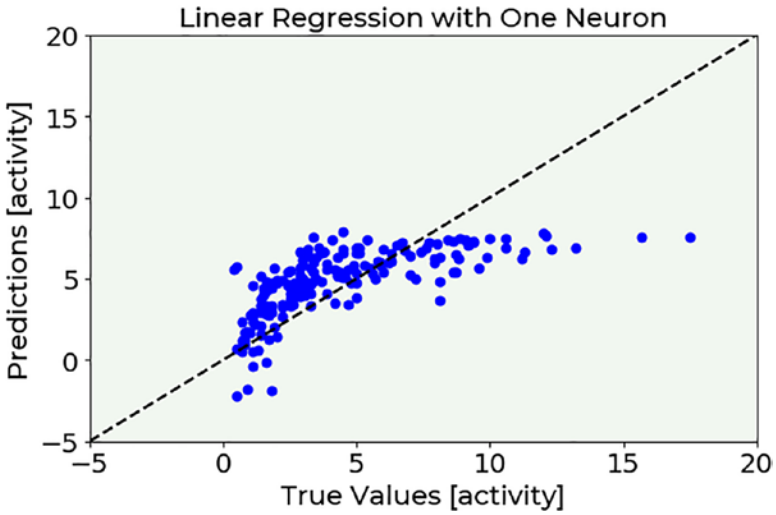

Figure 2-10. *The predicted target value vs. measured target value for our model, applied to our testing data*

If you have followed so far, congratulations! You just built your very first neural network, with just one neuron, but still a neural network!

Logistic Regression with a Single Neuron

Now let's try to solve a classification problem with a single neuron. Logistic regression is a classical algorithm and probably the simplest classification. We will consider a binary classification problem: that means we will deal with the problem of recognizing two classes (that we label as 0 or 1). We need an activation function that's different from the one we used for linear regression, a different cost function to minimize, and a slight modification of the output of our neuron. The goal is to be able to build a model that can

predict if a certain new observation is in one of two classes. The neuron should give as output the probability $P(y = 1|x)$ of the input x to be of class 1. We will then classify our observation as being in class 1 if $P(y = 1|x) > 0.5$ or in class 0 if $P(y = 1|x) < 0.5$.

It is instructive to compare this example with the one about linear regression, since they both are applications of the one-neuron model yet are used to solve different tasks. You should pay attention to the similarities and differences with the linear regression model discussed previously. You are going to see how simple using Keras is in this case and how, by changing a few things (such as the activation and loss functions), you can easily obtain a different model that can solve a different problem.

The Dataset for the Classification Problem

As in the linear regression example, we use a dataset taken from the real world, to make things more interesting. We employ the BCCD dataset, a small-scale dataset for blood cell classification. The dataset can be downloaded from its GitHub repository [8]. For this dataset, two Python scripts have been developed to make preparing the data easier. All the code can be found in the online version of the book. In the example, a slightly modified version of the two scripts is used. Remember, you are not interested at this point in how the data looks or how it is cleaned. You should focus on how to build the model with Keras.

The dataset contains three kinds of labels:

1. Red blood cells (RBC)
2. White blood cells (WBC)
3. Platelets

To make it a binary classification problem, we will consider only the RBC and WBC types. The model that will be trained has one neuron and will predict if an image contains an RBC or a WBC type by using the `xmin`, `xmax`, `ymin`, and `ymax` variables as features.

For simplicity, as in the linear regression example, we will skip all the import and load details and concentrate on the fundamental steps of the code, such as dataset preparation, model creation, and performance evaluation. You can find the complete code on the online version of the book. Note that the greatest differences between this case and the linear regression lie in the chosen activation and cost function.

Here is the data:

```
num_observations = len(bccd_features)
print('Number of total samples: ', num_observations)
```

This code returns

```
Number of total samples:  4527
```

Let's display the first lines of the data

```
bccd_features.head()
```

which prints to the screen

```
   xmin xmax ymin ymax
0   192  292  376  473
1   301  419  320  424
2   433  510  273  358
3   434  528  368  454
4   507  574  381  454
```

The dataset is made up of 4527 observations, one target column (`cell_type`), and four features (`xmin`, `xmax`, `ymin`, and `ymax`). When working with images, it is always useful to get an idea of how they look. You can see an example in Figure 2-11.

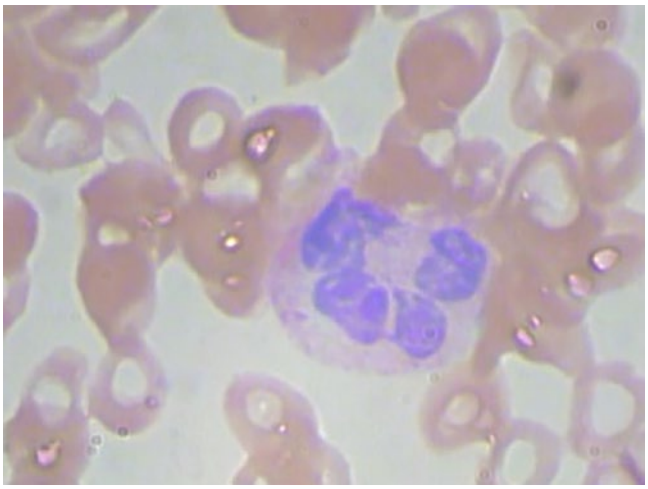

Figure 2-11. *A sample image from the BCCD dataset*

Note that the features we employ in this example are not the pixel values of the image, but the location of the edges of the bounding box of the cell. In fact, for each image, we have only four values (`xmin`, `xmax`, `ymin`, and `ymax`).

Dataset Splitting

As stated, in any machine learning project, you have to split the dataset in different subsets. Let's create a `train` and a `test` dataset by splitting the dataset randomly into two parts with a ratio of 80/20, as we did with the radon dataset.

```
np.random.seed(42)
rnd = np.random.rand(len(bccd_features)) < 0.8

train_x = bccd_features[rnd] # training dataset (features)
train_y = bccd_labels[rnd] # training dataset (labels)
test_x = bccd_features[~rnd] # testing dataset (features)
test_y = bccd_labels[~rnd] # testing dataset (labels)

print('The training dataset dimensions are: ', train_x.shape)
print('The testing dataset dimensions are: ', test_x.shape)
```

This code will give as output the following lines

```
The training dataset dimensions are:  (3631, 4)
The testing dataset dimensions are:  (896, 4)
```

So, we use 3631 observations to train our logistic regression model and then evaluate it on the remaining 896 observations.

Now comes a particularly important point. The labels in this dataset as imported will be `'WBC'` or `'RBC'` strings (they simply tell you if an image contains white or red blood cells). But we will build our cost function with the assumptions that our class labels are 0 and 1. That means we need to change our `train_y` and `test_y` arrays.

Note When doing binary classifications, remember to check the values of the labels you are using for training. Sometimes using the wrong labels (not 0 and 1) may cost you quite some time in understanding why the model is not learning.

```
train_y_bin = np.zeros(len(train_y))
train_y_bin[train_y == 'WBC'] = 1

test_y_bin = np.zeros(len(test_y))
test_y_bin[test_y == 'WBC'] = 1
```

Now all images containing RBC will have a label of `0`, and all images containing WBC will have a label of `1`.

The Logistic Regression Model

This model will be made up of one neuron and its goal will be to recognize two classes (labeled `0` or `1`, referring to RBC or WBC inside a cell image). This is a *binary classification problem.*

As opposed to linear regression, the activation function will be a *sigmoid function* (leading to a different neuron's output) and the cost function will be *cross-entropy* [7]. If you don't remember what a sigmoid function is, reread the beginning of this chapter. We use it because we want our neuron to output the probability of our observation being in class 1. Therefore, we need an activation function that can assume only values between 0 and 1; otherwise we cannot regard it as a probability. The cross-entropy for one observation is

$$L\left(\hat{y}^{(i)},y^{(i)}\right)=-\left(y^{(i)}\log\hat{y}^{(i)}+\left(1-y^{(i)}\right)\log\left(1-\hat{y}^{(i)}\right)\right)$$

In presence of more than one observation, the cost function is the sum over all observations

$$L(\boldsymbol{w},b)=\frac{1}{m}\sum_{i=1}^{m}L\left(\hat{y}^{(i)},y^{(i)}\right)$$

Explaining the cross-entropy loss function goes beyond the scope of this book, but if you are interested, you can find it described in many books and websites. For example, you can check out [7].

Keras Implementation

The following function builds the one-neuron model for logistic regression. The implementation is remarkably similar to that of linear regression. The differences,

as already mentioned, are the activation function, the cost function, and the metrics (accuracy in this case, which we analyze more in detail in the testing phase).

```
def build_model():

  model = keras.Sequential([
    layers.Dense(1, input_shape = [len(train_x.columns)], activation =
    'sigmoid')
  ])

  optimizer = tf.keras.optimizers.RMSprop(
              learning_rate = 0.001)

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

  return model
```

Now, let's apply the `build_model` function and look at the model summary

```
model = build_model()
model.summary()
```

This code gives the following output

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 5
=================================================================
Total params: 5
Trainable params: 5
Non-trainable params: 0
_________________________________________________________________
```

We have five parameters to be trained in this case as well—the weights associated with the four features plus the bias.

The Model's Learning Phase

As with the linear regression example, training our neuron means finding the weights and biases that minimize the cost function. The cost function we chose to minimize in our logistic regression task is the cross-entropy function, as discussed in the previous section.

We start training our model for 500 epochs and then look at the summary in terms of performances (accuracy).

```
EPOCHS = 500

history = model.fit(
  train_x, train_y_bin,
  epochs = EPOCHS, verbose = 1
)
```

In Figure 2-12, you can see the cost function vs. the number of iterations associated with the learning phase.

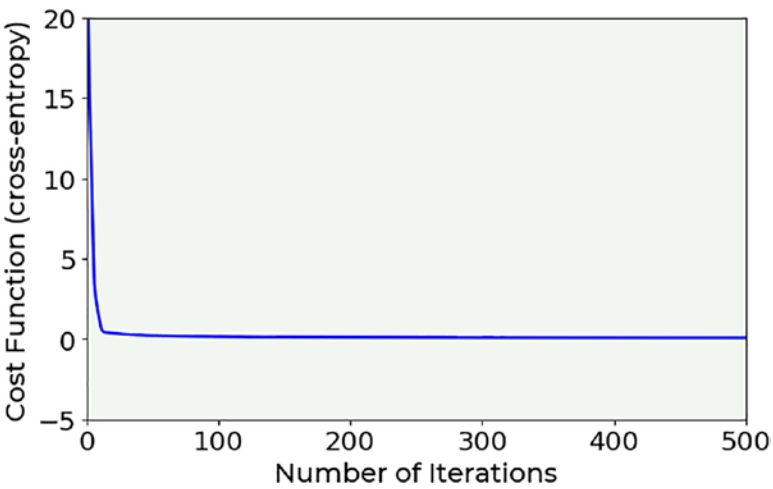

Figure 2-12. *The cost function resulting in our model applied to the BCCD dataset with a learning rate of 0.001*

Note that, after 100 epochs, the cost function remains almost constant, indicating that a minimum has been reached.

The Model's Performance Evaluation

Now, to determine if the model you have just trained is suited for unseen data, you must check its performance against the test set. Moreover, an *optimizing metric* must be chosen. For a binary classification problem, a classic metric is the *accuracy,* which can be understood as a measure of how well the classifier correctly identified the two classes of the dataset. Mathematically it can be calculated as

$$accuracy = \frac{number\ of\ cases\ correctly\ identified}{total\ number\ of\ cases}$$

where the number of cases correctly identified is the sum of all positive samples and negative samples (i.e., all 0s and 1s) that were correctly classified. These are usually called true positives and true negatives.

To get the accuracy, we can run this code (remember, we will classify an observation i to be in class 0 if $P(y^{(i)} = 1|\boldsymbol{x}^{(i)}) < 0.5$ or to be in class 1 if $P(y^{(i)} = 1|\boldsymbol{x}^{(i)}) > 0.5$)

```
test_predictions = model.predict(test_x).flatten()
test_predictions1 = test_predictions > 0.5
tp = np.sum((test_predictions1 == 1) & (test_y_bin == 1))
tn = np.sum((test_predictions1 == 0) & (test_y_bin == 0))
accuracy_test = (tp + tn)/len(test_y)
print('The accuracy on the test set is equal to: ',
       int(accuracy_test*100), '%.')
```

This code prints this to the screen

```
The accuracy on the test set is equal to:  98 %.
```

With this model, we reach an accuracy of 98%. Not bad for a network with just one neuron.

Conclusion

This chapter looked at many things. You learned how a neuron works and what its main components are. You also learned about the most common activation functions and saw how to implement a model with a single neuron in Keras to solve two problems: linear and logistic regression. In the next chapter, we look at how to build neural networks with a large number of neurons and how to train them.

Note Linear and logistic regression are two classical statistical models that can be implemented in many ways. This chapter used the neural network language to implement them and to show you how flexible neural networks are. When you understand their inner components, you can use them in many ways.

Exercises

EXERCISE 1 (LINEAR REGRESSION) (LEVEL: EASY)

Try using only one feature to predict radon activity and see how the results change.

EXERCISE 2 (LINEAR REGRESSION) (LEVEL: MEDIUM)

Try to change the `learning_rate` parameter and then observe how the model's convergence changes. Then try to reduce the `EPOCHS` parameter and observe when the model cannot reach convergence.

EXERCISE 3 (LINEAR REGRESSION) (LEVEL: MEDIUM)

Try to see how a model's results change based on the training dataset's size (reduce it and use different sizes, comparing the results).

EXERCISE 4 (LOGISTIC REGRESSION) (LEVEL: MEDIUM)

Try to change the `learning_rate` parameter and observe how the model's convergence changes. Then try to reduce the `EPOCHS` parameter and observe when the model cannot reach convergence.

EXERCISE 5 (LOGISTIC REGRESSION) (LEVEL: MEDIUM)

Try to see how a model's results change based on the training dataset's size (reduce it and use different sizes comparing the results).

EXERCISE 6 (LOGISTIC REGRESSION) (LEVEL: DIFFICULT)

Try to add to labels to `Platelets` samples and generalize the binary classification model to a multiclass one (three possible classes).

References

[1] https://spectrum.ieee.org/tech-talk/computing/software/biggest-neural-network-ever-pushes-ai-deep-learning, last accessed 27.12.2017.

[2] R. Rojas (1996), *Neural Networks: A Systematic Introduction,* Springer-Verlag Berlin Heidelberg.

[3] https://www.tensorflow.org/datasets/catalog/radon, last accessed 09.01.2021.

[4] Lever, Jake, Martin Krzywinski, and Naomi Altman. "Points of significance: model selection and overfitting." (2016): 703.

[5] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15.1 (2014): 1929-1958.

[6] Bengio, Yoshua. "Practical recommendations for gradient-based training of deep architectures." Neural networks: Tricks of the trade. Springer, Berlin, Heidelberg, 2012. 437-478.

[7] https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/, last accessed 10.01.2021.

[8] https://www.tensorflow.org/datasets/catalog/bccd, last accessed 10.01.2021.