A picture containing clock, drawing

Description automatically generatedHands-on with One Single Neuron

Chapter by Michela Sperti and Umberto Michelucci

In this chapter, we will see how to solve two classic statistical problems (i.e., **linear regression** and **logistic regression**) by using the simplest and most basic unit of neural networks: the neuron.

To make things a bit more fun, we will apply linear and logistic regression methods (deployed with only one neuron) on real datasets. We will discuss the two models and understand how to implement the two algorithms with **Keras**.

First of all, we will shortly recap what a neuron is, its typical structure, and its main characteristics (e.g., the activation function). Then we will look at how we can formally express it in matrix form (this step is fundamental to obtain optimized codes, exploiting all TensorFlow functionalities). Then we will discuss in detail the code examples to see in detail each step of linear regression and logistic regression implementations.

# A short overview of a neuron's structure

Deep learning is based on large and complex networks made 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 a large set of differently interconnected units, each performing a specific (and usually relatively easy) computation. They remind me of the game LEGO where you can build very complicated things using elementary and fundamental units.

Due to a biological parallel with the brain [2], those basic units are known as neurons. Each neuron does a straightforward thing: take a certain number of inputs (real numbers) and calculate an output (also a real number). The inputs are indicated in this book with (real numbers) with , where is an integer and 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 ). could be the age and could be the weight. In real life, the number of features can be easily huge.

There are several kinds of neurons that have been extensively studied. We will concentrate in this book 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 real parameters (with ) and a constant (usually called bias), the neuron will calculate first what is traditionally indicated in literature and in books with :

it will then apply a function to , giving the output

**Note** Practitioners mostly use the terminology: are called **weights**, **bias**, **input features** and the **activation function**.

Let's summarize the neuron computational steps again.

1. Combine linearly all inputs calculating ;
2. Apply f to z giving the output .

In the literature, you can find numerous representations for neurons. This book uses the one in Figure 14-1 because it is widely used and is easy to understand.

Diagram

Description automatically generated

Figure 14-1. A representation of a single neuron with the operation highlighted.

Figure 14-1 must be interpreted in this way:

* the inputs are not put in a bubble, simply to distinguish them from nodes that perform an actual calculation;
* the weight's names are written on the arrow. The meaning is that before passing the inputs to the central bubble (or node), the inputs will be first multiplied by the relative weight, as labelled on the arrow. The first input will be multiplied by , by and so on;
* the central bubble (or node) will perform several calculations at the same time: first it will sum the inputs (the for ), then sum to the result the bias and finally apply to the resulting value the activation function.

## A short introduction to matrix notation

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

where we have indicated the vector with a bold-faced . For the weights, we use the same notation

For consistency with formulas that we will use later, to multiply and we will use matrix multiplication notation and therefore we will write

Where indicates the transpose of . can then be written with this vector notation as

and the neuron output as

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

* 🡪 neuron output
* 🡪 activation function (or transfer function) applied to z
* 🡪 weights (vector with components)
* 🡪 bias

## An overview of the most common activation functions

There are many activation functions at our disposal to change the output of our neuron. Remember, an activation function is simply a mathematical function that transforms in the output . Let's have a brief look at the most common ones.

### Identity function

This is the most basic function that you can use. Usually is indicated with . It merely returns the input value unchanged. Mathematically we can write it as

This simple function will come in handy when discussing linear regression with one neuron later in the chapter. Implementing it[[1]](#footnote-1) 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 indicate with

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)))

**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). In case you know object-oriented programming, we say that in NumPy basic operations like /, \*, + and - are **overloaded**. Note also that all those four basic operations in NumPy act element by element.

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 above 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))). I 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 what is called **broadcasting**. Python, subject to certain constraints, "broadcast" 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 it 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.

# How to implement a neuron in Keras

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

model = keras.Sequential([

layers.Dense(1, input\_shape = [...])

])

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 one single neuron, defined by the command layers.Dense which specifies 1 unit (neuron) inside a layer and the shape of our input dataset. The Dense class implements densely connected neural networks' layers.

In the next paragraphs, you will see two practical examples of how you use this simple command, choosing the right activation function and the proper loss function (given as additional parameters), 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 job for you. Of course, you can also implement the neuron from scratch, using Python standard functionalities such as lists and loops, but those tends to be 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 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 where the code runs on. We are not interested in the absolute values, but only on how much faster NumPy is in comparison with standard Python loops. In case you are using a Jupyter notebook, it is useful to know how to time Python code in a cell. To do this, we can use a "magic command". Those commands start (in a Jupyter notebook) usually with %% or with %. A good idea is to check the official documentation to better understand how they work (<http://ipython.readthedocs.io/en/stable/interactive/magics.html>).

Coming back to our test, let us 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 2 seconds averaged over 7 runs. Now let's try to do the same multiplication, but this time using NumPy

%%timeit

out2 = np.multiply(list1\_np, list2\_np)

Where we have 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 21 ms, or in other words, 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 matrices 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 one single neuron

The first type of regression will allow us to understand how to build a model in Keras and how to use it to solve one of the most basic statistical problems. 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 in Chapter 14 (<https://adl.toelt.ai/Chapter14/Introduction.html>) of the online version of the book. However, it is very instructive to follow this example, since it gives a practical grasp 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 is essential to write our equations in matrix form and feed the matrices to NumPy. In this way, our code will be as efficient as possible. Remember: avoid loops at all costs whenever possible.

## The dataset for our real case example

To make things a bit more interesting, let's use an instructive dataset. We will use the so-called radon data set [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 mentioned above contains measured radon levels in U.S. homes by county and state. The **activity** label is the measured radon concentration in pCi/L (we can refer to it as the **target** variable, i.e., the one we want to predict using a linear regression model). Important predictors are:

* floor (the floor of the house in which the measurement was taken),
* county (the U.S. county in which the house is located), and
* uppm (a measurement of uranium level of the soil by county).

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

We do not need for our example to understand or study the features. Our goal here is to understand how to build a linear regression model with what we have learned. Generally, in a Machine Learning project, you would first study your input data, check their distribution, quality, missing values, and so on. But we will skip this part here 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 us have a look at our data. We will skip all the import and load details for simplicity, and we will concentrate on the fundamental steps of our code, such as dataset preparation, model creation, and performance evaluation. You can find the complete code on the online hands-on 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. Now, let us use the command radon\_features.head() which will give the following table as output (the first 5 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 4 features (floor, county, log\_uranium\_ppm, pcterr) that we will use as predictors of radon activity.

As already suggested before, we have prepared our data using **matrix form**. Let us briefly discuss the notation, which will come in handy when building our neuron. Normally we have many observations (919 in our case). We will use an upper index to indicate the different observations between round parentheses. Our observation will be indicated with , and the feature of the observations will be indicated as . We will indicate the number of observations with .

**Note** In this book is the **number of observations** and the **number of features**. Our feature of the observation will be indicated with . In deep learning projects the bigger the better. So be prepared to deal with a massive number of observations.

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

where each row is an observation, and each column represents a feature in the matrix that has dimensions .

### Dataset splitting

In any machine learning project, it is a good behavior to split the dataset into different subsets. Plenty of theoretical explanations about this need is present in literature [4, 5]. To simply explain the concept: 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 data never seen before). The most used 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.

The following code will split the dataset randomly in 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)

The above 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 one neuron model is an overkill for a linear regression task. We could solve linear regression exactly without the need of using gradient descent or a similar optimization algorithm. You can find an exact regression solution example, implemented with NumPy library, in the book's hands-on online version in Chapter 14.

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

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 measure how close are the neuron’s outputs to the target variables? The difference is measured by the Mean Squared Error (MSE) function.

where the sum is over all observations. As discussed in the Theory Part of the book, this is the typical loss function chosen in a regression problem. By minimizing with respect to the weights and bias, we can find their optimal values.

### Keras implementation

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

def build\_model():

# one unit as network's output

# identity function as activation function

# sequential groups a linear stack of layers into a

# tf.keras.Model

# activation parameter: if you don't specify anything, no

# activation is applied (i.e. "linear" activation:

# a(x) = x).

model = keras.Sequential([

layers.Dense(1, input\_shape = [len(train\_x.columns)])

])

# optimizer that implements the RMSprop algorithm

optimizer = tf.keras.optimizers.RMSprop(

learning\_rate = 0.001)

# the compile() method takes a metrics argument,

# which is a list of metrics

# loss = Mean Square Error (mse),

# metrics = Mean Absolute Error (mae),

# Mean Square Error (mse)

model.compile(loss = 'mse',

optimizer = optimizer,

metrics = ['mse'])

return model

Let analyze what this code does.

* First of all, we defined the net structure with the keras.Sequential class, adding one layer 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 in this example the RMSprop algorithm.
* Finally, we compiled the model (i.e. we configured 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 performances evaluation (model.compile). The function returns the built model as one 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 see how the model's convergence changes [6].

Now, let us apply the build\_model function and have a look at the model summary

model = build\_model()

model.summary()

The above 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 5 parameters to be trained, i.e. the weights associated to the 4 features plus the bias.

## Model's learning phase

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

EPOCHS = 1000

history = model.fit(

train\_x, train\_y,

epochs = EPOCHS, verbose = 0,

callbacks = [tfdocs.modeling.EpochDots()])

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, the number of epochs, and tensorflow\_docs.modeling function as a callback function[[2]](#footnote-2) (this will generate a report on the network's performances step by step). The output is

Epoch: 0, loss:2543.7136, mse:2543.7136,

..............................................................

Epoch: 100, loss:19.7100, mse:19.7100,

..............................................................

Epoch: 200, loss:16.4799, mse:16.4799,

..............................................................

Epoch: 300, loss:16.0523, mse:16.0523,

..............................................................

Epoch: 400, loss:16.0160, mse:16.0160,

..............................................................

Epoch: 500, loss:15.9984, mse:15.9984,

..............................................................

Epoch: 600, loss:15.9890, mse:15.9890,

..............................................................

Epoch: 700, loss:15.9610, mse:15.9610,

..............................................................

Epoch: 800, loss:15.9715, mse:15.9715,

..............................................................

Epoch: 900, loss:15.9663, mse:15.9663,

..............................................................

Every 100 epochs, the cost function and the MSE calculated over the training dataset (in this case, they are the same thing) are plotted on the screen. This is an easy way of checking if the cost function is really decreasing or not or if nans are appearing. If you perform some initial tests in an interactive environment (like a Jupyter notebook), you can stop the process if you see that the cost function is not behaving as you expect.

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 worked efficiently. Another way to immediately visualize the loss function decrease is by plotting the **cost function vs. number of epochs**. This can be seen in Figure 14-2.

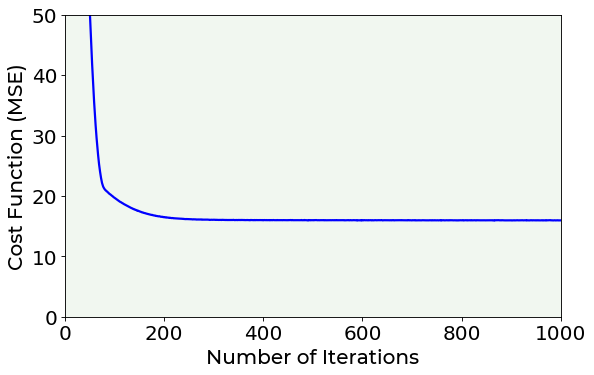


Figure 14-2. The cost function behaviour during the model training applied to the radon dataset with a learning rate of 0.001.

Looking at Figure 14-2, you can notice 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 may be interested in the coeffients of the linear equation. Those are the weights of our neuron model. 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 NumPy library (<https://adl.toelt.ai/Chapter14/Linear_regression_with_numpy.html>).

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)]

## Model's performance evaluation on unseen data

Now, to know if the model you have just trained is suited to be applied to unseen data, you must check its performances over a dataset containing only unseen data (the test set). In the following code the linear model is applied to the test set to make predictions (test\_predictions). Then, predicted radon activity values are compared with real values (test\_y) by simply plotting **predictions vs. true values as can be seen in Figure 14-3**. A perfect model will show points distributed over the black solid line present in the plot. The less precise our predictions are, the most spread the points will be around the diagonal.

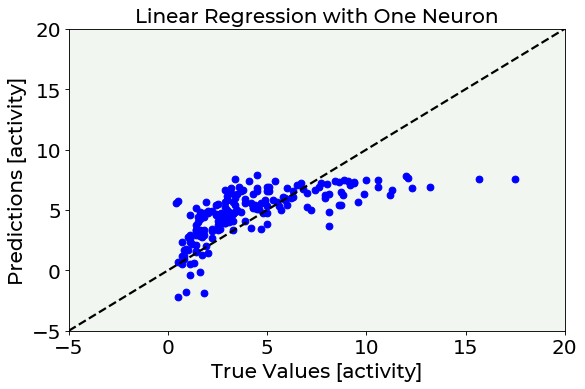


Figure 14-3. The predicted target value vs. measured target value for our model, applied to our testing data.

**Figure 14-3** shows that a linear regression model is too simple to accurately model this dataset's behavior and a more complex model is needed.

# Logistic regression with one single neuron

The logistic regression is a classical classification algorithm. To maintain it simple we will consider here a binary classification problem: that means we will deal with the problem of recognizing two classes only (that we will label as 0 or 1). We will need an activation function different from the one we used for linear regression, and a different cost function to minimize and a slight modification of the output of our neuron. Our goal is to be able to build a model that can predict if a certain new observation is of one of two classes. The neuron should give as output the probability of the input to be of class 1. We will then classify our observation as of class 1 if or of class 0 if .

It is very instructive to compare this example with the one about linear regression, since they both are applications of the one neuron model, used to solve different tasks. You should try to pay attention to similarities and differences with the linear regression model discussed previously. You are going to see how simple is using Keras also in this case and how, changing a few parameters, 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 will use a dataset taken from the real world, to make things more interesting. We will employ the BCCD Dataset, a small-scale dataset for blood cells classification. The dataset can be downloaded from its GitHub repository. From this dataset, nicolaschen [8] developed two Python scripts to make preparation data for recognition of abnormalities in blood cells on medical images. All the code can be found in the online version of the book in Chapter 14. In the example, a slightly modified version of the two scripts will be used.

1. a script to create the pandas dataframe with all data needed: filename, cell\_type, xmin, xmax, ymin, ymax;
2. a script to plot the boxes for each image and save it in a new directory.

The dataset contains three kind of labels:

1. Red Blood Cell (**RBC**)
2. White Blood Cell (**WBC**)
3. Platelets

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

For simplicity, as in the linear regression example, we will skip all the import and load details and we will concentrate on the fundamental steps of our code, such as dataset preparation, model creation and performances evaluation. You can find the complete code on the online version of the book in Chapter 14. Note that the greatest differences between this case and the linear regression lie in the activation function and cost function chosen.

Let us have a look at our data:

num\_observations = len(bccd\_features)

print('Number of total samples: ', num\_observations)

The above lines of code return

Number of total samples: 4527

And let us display the first lines of our 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 of 4527 observations, 1 target column (cell\_type) and 4 features (xmin, xmax, ymin, ymax). When working with images, it is always useful to get an idea of how they look. You can see an example in Figure 14-4.

Immagine che contiene testo

Descrizione generata automaticamente

Figure 14-4. A sample image from the BCCD dataset.

Note that the features we will employ in our 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 only have 4 values (xmin, xmax, ymin and ymax).

## Dataset splitting

We have already stated, in the linear regression model's section, that in any machine learning project, we have to split the dataset in different subsets. Let’s create a train and a test dataset by splitting the dataset randomly in two parts with the following proportions: 80%/20%, as we have already performed for 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)

The above code will give as output the following lines

The training dataset dimensions are: (3631, 4)

The testing dataset dimensions are: (896, 4)

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

Now comes a particularly important point. The labels in our 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, so we need to change our train\_y and test\_y arrays.

**Note** When doing binary classification 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.

# The following lines are needed to convert the labels from

# RBC/WBC notation to 0/1 notation.

# This is fundamental to correctly train the net.

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.

## Logistic regression model

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

Differently from linear regression, the **activation function** will be a **sigmoid function** (leading to a different neuron's output) and the **cost function** will be the **cross-entropy** [7]. If you don’t remember what a sigmoid function is, check again the review at the beginning of this chapter. We use it since we want our neuron to output the probability of our observation to be of 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

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

### 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 will analyze more in detail in the testing phase).

def build\_model():

# one unit as network's output

# sigmoid function as activation function

# sequential groups a linear stack of layers into a

# tf.keras.Model

# activation parameter: if you don't specify anything, no

# activation is applied (i.e. "linear" activation:

# a(x) = x).

model = keras.Sequential([

layers.Dense(1, input\_shape = [len(train\_x.columns)], activation = 'sigmoid')

])

# optimizer that implements the RMSprop algorithm

optimizer = tf.keras.optimizers.RMSprop(

learning\_rate = 0.001)

# the compile() method takes a metrics argument, which can

# be a list of metrics

# loss = cross-entropy, metrics = accuracy,

model.compile(loss = 'binary\_crossentropy',

optimizer = optimizer,

metrics =

['binary\_crossentropy','binary\_accuracy'])

return model

Now, let us apply the build\_model function and have a look at the model summary

model = build\_model()

model.summary()

The above 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 5 parameters to be trained also in this case, i.e. the weights associated to the 4 features plus the bias.

## Model's learning phase

As in 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, as we discussed in the previous section.

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

EPOCHS = 500

history = model.fit(

train\_x, train\_y\_bin,

epochs = EPOCHS, verbose = 0,

callbacks = [tfdocs.modeling.EpochDots()])

The above lines of code print to the screen

Epoch: 0, binary\_accuracy:0.6651, binary\_crossentropy:33.9200, loss:33.9200,

..............................................................

Epoch: 100, binary\_accuracy:0.9628, binary\_crossentropy:0.1531, loss:0.1531,

..............................................................

Epoch: 200, binary\_accuracy:0.9730, binary\_crossentropy:0.1051, loss:0.1051,

..............................................................

Epoch: 300, binary\_accuracy:0.9763, binary\_crossentropy:0.0869, loss:0.0869,

..............................................................

Epoch: 400, binary\_accuracy:0.9799, binary\_crossentropy:0.0781, loss:0.0781,

..............................................................

Every 100 epochs the cost function (i.e. the cross-entropy) and the accuracy calculated over the training dataset are printed on the screen.

The cost function clearly decreases and reaches a value near to zero, while the accuracy increases. That is a good sign indicating that the cost function has reached a minimum.

In Figure 14-5 you can see the **cost function vs. number of iterations** plot associated to the learning phase.

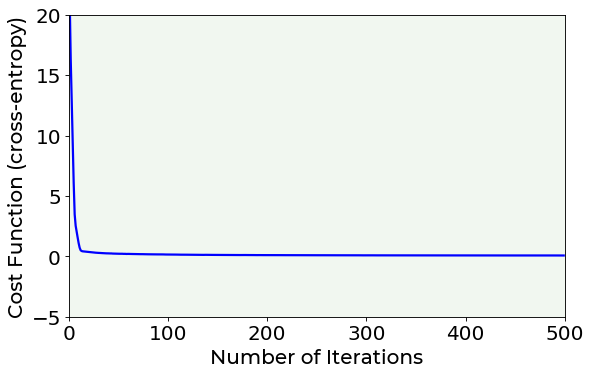


Figure 14-5. The cost function resulting in our model applied to the BCCD dataset with a learning rate of 0.001.

Looking at the previous plot, you can notice that, after 100 epochs, the cost function remains almost constant in its value, indicating that a minimum has been reached.

## Model's performance evaluation

Now, to know if the model you have just trained is suited to be applied to unseen data, you must check its performances over 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

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, usually called true positives and true negatives.

To get the accuracy we can run this code (remember we will classify an observation of class 0 if or in class 1 if

test\_predictions = model.predict(test\_x).flatten()

# predict cell type with the built logistic regression model

# The following lines compute the accuracy on the test set.

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), '%.')

The above code prints 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.

# Exercises

EXERCISE 1 (linear regression)

Try using only one feature to predict radon activity and see how results change. Difficulty: easy.

exercise 2 (linear regression)

Try to change the learning\_rate parameter and see how the model's convergence changes. Then try to reduce the EPOCHS parameter and see when the model cannot reach convergence. Difficulty: medium.

exercise 3 (linear regression)

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

exercise 4 (logistic regression)

Try to change the learning\_rate parameter and see how the model's convergence changes. Then try to reduce the EPOCHS parameter and see when the model cannot reach convergence. Difficulty: medium.

exercise 5 (logistic regression)

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

exercise 6 (logistic regression)

Try to add to labels Platelets samples and generalize the binary classification model to a multiclass one (3 possible classes). Difficulty: hard.

# 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

1. Note that you will not have to implement it yourself in Python. Keras offers already this function, and many more, out of the box. [↑](#footnote-ref-1)
2. If you don’t remember what a callback function is, check Chapter 13 again. [↑](#footnote-ref-2)