A picture containing clock, drawing

Description automatically generatedHands-on with One Single Neuron

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 deeply 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, what are its typical architecture and main characteristics (e.g. the activation function) and how we can formally express it in matrix form (this step is fundamental to obtain optimized codes, exploiting all TensorFlow assets). Then we will move deeply inside the code examples to see in detail each step of linear regression and logistic regression implementations.

# A short recall on neuron’s architecture

Deep learning is based on large and complex networks made of large number of simple computational units. Companies on the fore front of research are dealing with networks with 160 billions of parameters [1]. To put things in perspective this number is half of the number of the 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 of the game LEGO where you can build very complex things using very simple and basic units.

Those basic units are known, due to a biological parallel with the brain [2], as neurons. Each neuron does basically a very simple thing: take a certain number of inputs (real numbers) and calculates an output (also a real number). Our 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 person (so we would have ). could be the age and could be the weight. In real life the number of features can be easily very big.

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 usually indicated in literature and in books with :

it will then apply a function to , giving the output

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

Let’s summarize again the neuron computational steps.

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

In the literature you can find numerous representations for neurons. In this book we use the one in Figure 14-1 because is widely used and is easy to understand.

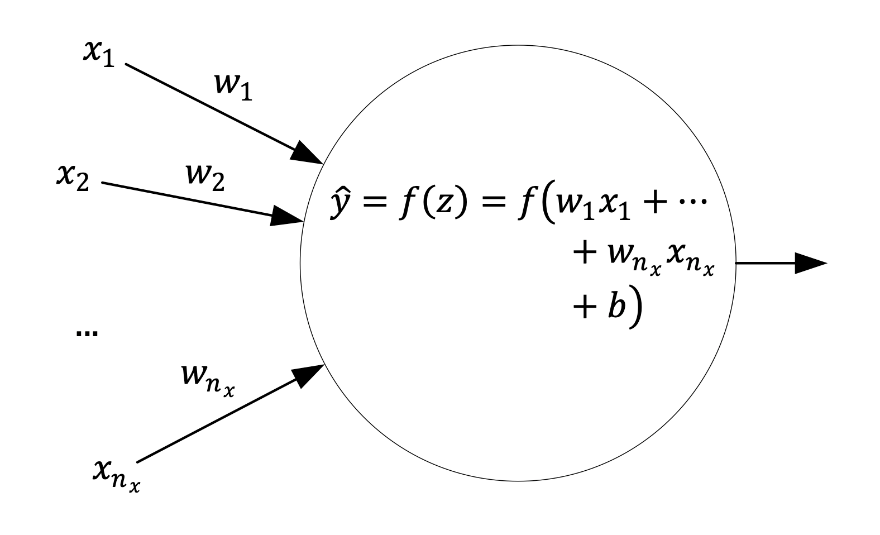


Figure 14-1. The neuron representation mostly used by practitioners.

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 input 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 recall on matrix notation

When dealing with big datasets, the number of features is large ( will be big) and so is better to use a vector notation for the features and the weights:

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

## A short recall on activation functions

There are many activations functions at our disposal to change the output of our neuron. Remember an activation function is simply a mathematical function that transform in the output . Let's have a brief look at the ones we will need to perform our examples.

### Identity function

This is the most basic function that you can use. Usually is indicated with . It returns simply the input value unchanged. Mathematically we have

This simple function will come handy when we will discuss linear regression with one neuron later in the chapter.

Implementing it in Python with NumPy is particularly 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 especially used for models where we must 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 goes beyond the scope of this book. However, 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 very simple case, we need just one layer made by one single neuron, and this is 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 on how you use this simple command, choosing the right activation function and the right 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 grow. A good rule of thumb is to avoid loops when possible, and to use NumPy (or TensorFlow) methods as often as possible.

Is easy to get an idea of 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 were 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. To time Python code in a Jupyter notebook 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 a different result):

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

The code needed roughly 2 seconds on average 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

list1\_np = np.array(lst1)

list2\_np = np.array(lst2)

and we have imported the library numpy with the command

import numpy

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 big amount of data.

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

While training deep learning models you will find yourself doing this kind of operations over and over, and therefore 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 offer us the opportunity 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 easily 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 the online hands-on version of the book (<https://adl.toelt.ai/Chapter14/Linear_regression_with_numpy.html>). 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 at the same time. To get the best performance possible, is important 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 a real 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 greatly from household to household. The above-mentioned dataset 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 by means of our 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 well a classical regression problem, since it contains a continuous variable (radon activity) which is interesting to be predicted. The model which will be built is made of one neuron and will fit a linear function to 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. Normally 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 how to implement what we learned 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. For simplicity, 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 hands-on version of the book ([https://adl.toelt.ai/Chapter14/Linear\_regression\_with\_one\_neuron.html#](https://adl.toelt.ai/Chapter14/Linear_regression_with_one_neuron.html)).

Let us 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 called 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 stated before, we have prepared our data using **matrix form**. Let us briefly recall the notation, which will come in handy when building our neuron. Each line of our dataset can be represented by the vector , where each number describe a different feature for the single observation. 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 huge 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 you have at your disposal in 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 (i.e. build) the model and then you have to test it (i.e. verify the model’s performances on never seen before data). The roughest 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.

Now we build a train and a test set splitting 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)

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

## Linear regression model

From now on, the interesting part begins. Keep in mind that a one neuron model is an overkill for a regression task. We could solve linear regression exactly without the need of using gradient descent or similar optimization algorithm (employed in the neuron’s architecture). You can find an exact regression solution example, implemented with NumPy library, in the hands-on online version of the book (<https://adl.toelt.ai/Chapter14/Linear_regression_with_numpy.html>).

Given that our dataset (as previously recalled) 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 are simply computing the following equation:

that is the linear combination of the input data and the network’s weights plus the constant term (bias) .

A neuron that can perform linear regression uses the **identity activation function**. The **cost function** that needs to be minimized is the **Mean Square Error** (MSE) that can be written as:

where the sum is over all observations.

### Keras implementation

Despite it is very important to keep in mind the previous considerations, developing this model 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 us recap what this code does.

* First of all, we defined the net structure by keras.Sequential class, adding one layer made of one neuron (layers.Dense) and with input dimensions 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 now use 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 inside one single Python object.

**Learning rate** is a very important parameter of 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 and 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 most famous numerical method to find the minimum of a given function is the gradient descent (it is suited for cases in which the solution cannot be found analytically, such as all neural network applications). In our case we used the **RMSprop algorithm** as optimizer, instead of the gradient descent. Optimizers will be deeply discussed in this book.

The **minimization process** is iterative; therefore, it is necessary to decide when to stop it. The simplest way is to set a fixed number of repetitions (called **epochs**) and to run the algorithm that specific number of times. Then, results are checked to see if an optimal point has been reached. If not, the number of epochs is increased.

We start training our model for 1000 epochs and we look at the summary in terms of performances (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 applying the fit method to our built model. fit takes as inputs the training data, the number of epochs and tensorflow\_docs.modeling function as a callback function (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 (as you can see 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 and then reach a value and 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 will give good prediction. This tells us only that the learning has worked efficiently. Another way to immediately visualize the loss function decreasing is by plotting of the **cost function vs. number of iterations**. Let us plot it. If you are interested in plotting details you can find the complete code inside the /module folder of the GitHub repository of the book.

# The following line contains the path to fonts that are used

# to plot result in a uniform way.

f = set\_style().set\_general\_style\_parameters()

# Cost Function vs. Number of Iterations PLOT

fig = plt.figure()

ax = fig.add\_subplot(111)

plt.plot(hist['epoch'], hist['mse'], color = 'blue')

plt.ylabel('Cost Function (MSE)',

fontproperties = fm.FontProperties(fname = f))

plt.xlabel('Number of Iterations',

fontproperties = fm.FontProperties(fname = f))

plt.ylim(0, 50)

plt.xlim(0, 1000)

plt.axis(True)

plt.show()

The previous code gives as input Figure 14-2.

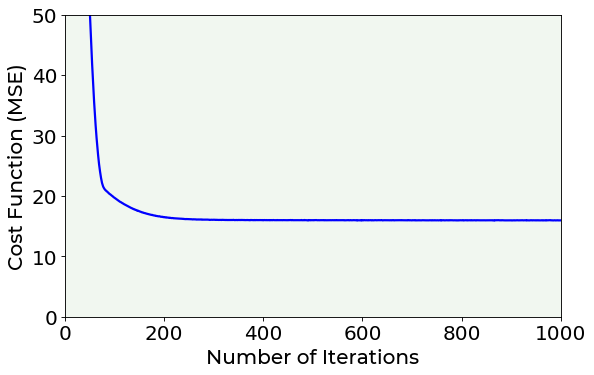


Figure 14-2. The cost function resulting in our model applied to the radon dataset with a learning rate of 0.001.

Looking at the previous plot, you can notice that, after 400 epochs, the cost function remains almost constant in its value, indicating that a minimum has been reached.

Let us have a look at the estimated 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)]

Finally, the cost function vs. number of iterations plot is also useful to evaluate the model’s convergence for different learning rates.

## Model’s performances evaluation

Now, to know if the model you have just built is suited to be applied to unseen data, you must check its performances over the test set. In the following cell 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**. An optimal model shows points distributed over the black solid line present in the plot.

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

# predict radon activities with the built linear regression

# model

# Predictions vs. True Values PLOT

fig = plt.figure()

ax = fig.add\_subplot(111)

plt.scatter(test\_y, test\_predictions, marker = 'o',

c = 'blue')

plt.plot([-5,20], [-5,20], color = 'black', ls = '--')

plt.ylabel('Predictions [activity]',

fontproperties = fm.FontProperties(fname = f))

plt.xlabel('True Values [activity]',

fontproperties = fm.FontProperties(fname = f))

plt.title('Linear Regression with One Neuron',

fontproperties = fm.FontProperties(fname = f))

plt.ylim(-5, 20)

plt.xlim(-5, 20)

plt.axis(True)

plt.show()

The previous code gives as input Figure 14-3.

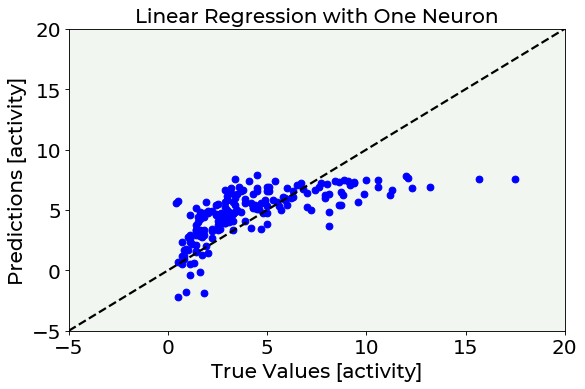


Figure 14-3. The predicted target value vs. measured target value for our model, applied to our testing data.

This plots shows that linear regression is too simple to accurately model this dataset's behaviour 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: 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, 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 will be provided with information about this model to notice similarities and differences with the linear regression one. You are going to see how simple is using Keras and how, changing a few parameters, you can easily obtain a different model that can solve a different problem.

## The dataset for our real case example

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 detection. The dataset will 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. 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 keep it simple, we will consider only **RBC** and **WBC** to be predicted. In detail, we will face a typical classification problem. The model which will be built is made of one neuron and will predict if an image contains RBC or WBC from 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 hands-on version of the book (<https://adl.toelt.ai/Chapter14/Logistic_regression_with_one_neuron.html#>).

As in the linear regression example, it is important to keep in mind matrix notation to express our data, since the neuron model works in a similar way, with greatest differences 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 useful to get an idea of how they look. Let us plot an example image from our dataset.

# The following line contains the path to fonts that are used

# to plot result in a uniform way.

f = set\_style().set\_general\_style\_parameters()

# Image Example

fig = plt.figure()

ax = fig.add\_subplot(111)

# add axes to the image

plt.axis('off')

# read and plot the image

image = plt.imread(

'BCCD\_Dataset/BCCD/JPEGImages/BloodImage\_00000.jpg')

plt.imshow(image)

the above lines of code produce Figure 14-4.

Immagine che contiene testo

Descrizione generata automaticamente

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

Notice that the features we will employ in our example are a simplified version of all the image, 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, it is a good behavior to split the dataset you have at your disposal in different subsets. [4, 5]

Now we build a train and a test set 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 build our logistic regression model and we 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 working.

# 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 an example of 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] In the case of linear regression an identity activation function and a MSE cost function were used. We havejust recalled what a sigmoid function is 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 0 or 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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Also in this case, we have 5 parameters to be trained, i.e. the weights associated to the 4 features and 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.

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 plotted 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.

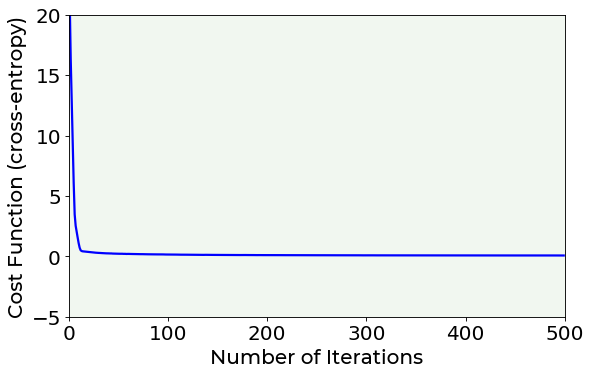


Figure 14-5. The cost function resulting in our model applied to the BCCD dataset with a learning rate of 0.001.

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 performances evaluation

Now, to know if the model you have just built 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 **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