Single Neuron

# Introduction

In this chapter we will discuss what is a neuron and what are its components. We will clarify the mathematical notation we will need and discover the many activation functions that are used today in neural networks. Gradient descent optimization will be discussed in details and the concept of learning rate and its quirks will be introduced. To make things a bit more fun, we will then use a single neuron to do linear and logistic regression on real datasets. We will then discuss and understand how to implement the two algorithms with tensorflow.

To keep the chapter focused and the learning efficient we have left few things out on purpose. For example, we will not split the dataset in training and test parts. We simply use all the data. Using the two would force us to do some proper analysis, and that would distract from the main goal of this chapter and make it way too long. Later in the book we will do a proper analysis of the consequences of using several datasets and how to do it properly, especially in the context of deep learning. This is a subject that requires its own chapter.

You can do wonderful, amazing and fun things with deep learning. Let's start and have fun!

# The structure of a neuron

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 billion 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. Neural networks are similar. Using relatively simple computational units you can build very complex systems. We can vary the basic units changing how they compute the result, how they are connected to each other, how they use the input values and so on. Roughly formulated all those aspects define what is known as the network architecture. Changing it, will change how the network learn, how good the predictions are and so on.

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 will be 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. In the dataset we will use for our logistic regression example later in the chapter we will have .

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

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

Due to a biological parallel, the function is called the neuron activation function (and sometimes transfer function) that will be discussed at length in the next sections.

Let’s summarize again the neuron computational steps.

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

You remember that in chapter 1 we have discussed computational graphs. In Figure 1-1 you can find the one for the neuron described previously.

Figure 1-1: the computational graph for the neuron described in the text.



This is not what you usually find in blogs, books and tutorials. Is rather complicated and not very practical to use especially when you want to draw networks with many neurons. In the literature you can find numerous representations for neurons. In this book we will use the one in Figure 1-2 because is widely used and is easy to understand.

Figure 1-2: the neuron representation mostly used by practitioners.



Figure 1-2 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.

All neurons we will deal with in this book will have exactly this structure. Very often an even simpler representation is used, as in Figure 1-3. In such a case, unless otherwise stated, is understood that the output is

Figure 1-3: the following representation is a simplified version of Figure 1-2. Unless otherwise stated it is usually understood that the output is . The weights are often not explicitly reported in the neuron representation.



## 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 will use in this book:

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

### Python implementation tip: loops and numpy

The calculation that we have outlined in the equation (3) can be done in Python by standard lists and with 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 as we will see later) 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'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 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.

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.

## Activation functions

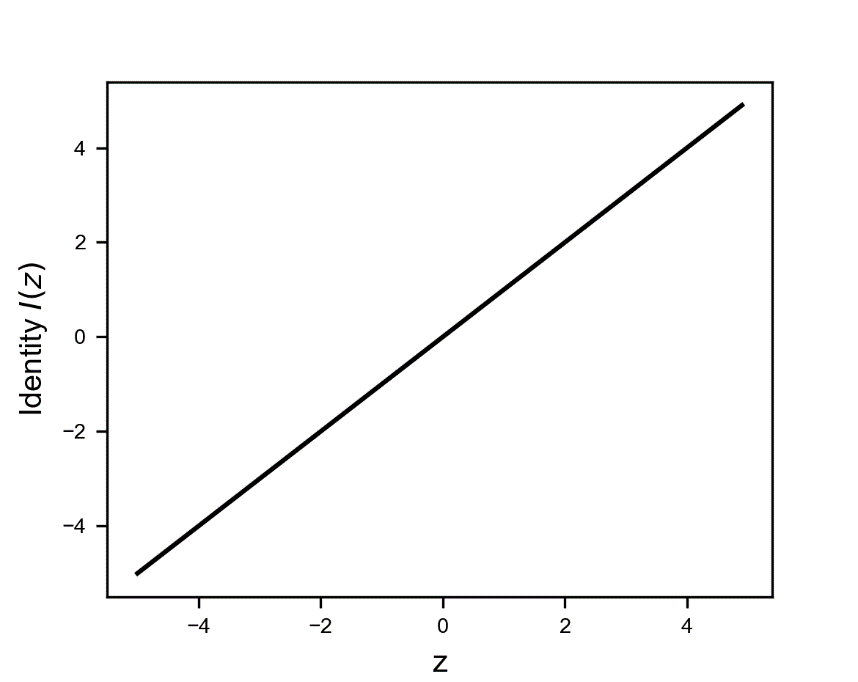
There are many activations functions at our disposal to change the output of our neuron. Remember an activation function is simply a mathematical function that transform in the output . Let's have a look at the most used.

### 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.

Figure 1-4: The identity function



Implementing it in Python with numpy is particularly trivial

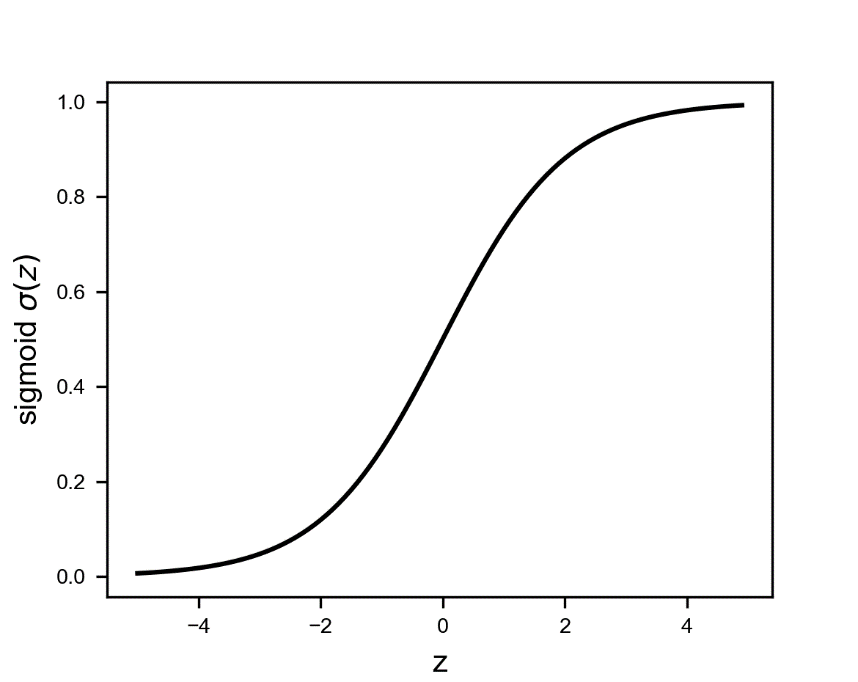
def identity(z):

return z

### Sigmoid function

This is a very commonly used function that gives only values between 0 and 1. It is usually indicate with

Figure 1-5: The sigmoid activation function is a s-shaped function that goes from 0 to 1.



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). Note that in Python, if is big enough, it can happen that the function returns exactly 0 or 1 (depending on the sign of ) for rounding errors. In classification problems we will calculate or very often, and therefore this can be a source of errors in Python since it will try to calculate that is not defined. For example, you can start seeing nan appearing while calculating the cost function (more on that later). We will see a practical example of this phenomena later in the chapter.

Although 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 (positive or negative) Python will round the results to exactly 0 or 1. This may give you errors while calculating the cost function (we will give you a detailed explanation and practical example later in the chapter) for classification since we will need to calculate and and therefore Python will try to calculate that is not defined. This may happen, for example, if we don't normalize correctly our input data or if we don't initialize correctly our weights. For the moment is important to remember that although mathematically everything seems under control, the reality while programming can be more difficult. Is something that is good to keep in mind while debugging models that for example gives nan as a result for the cost function.

The behavior with can be seen in Figure 1-5. The calculation can be written in this form using numpy functions

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

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))

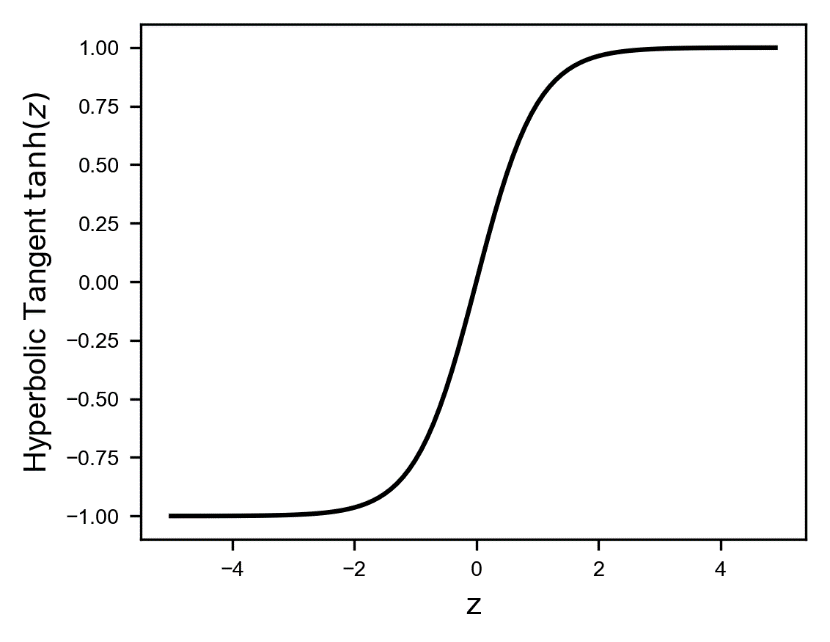
return s

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*[[1]](#footnote-1)*. 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 don't 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.

### Tanh or hyperbolic tangent Activation Function

The hyperbolic tangent is also a s-shaped curve that goes from -1 to 1:

Figure 1-6: The tanh (or hyperbolic function) is a s-shaped curve that goes from -1 to 1.



In Python this can be easily implemented

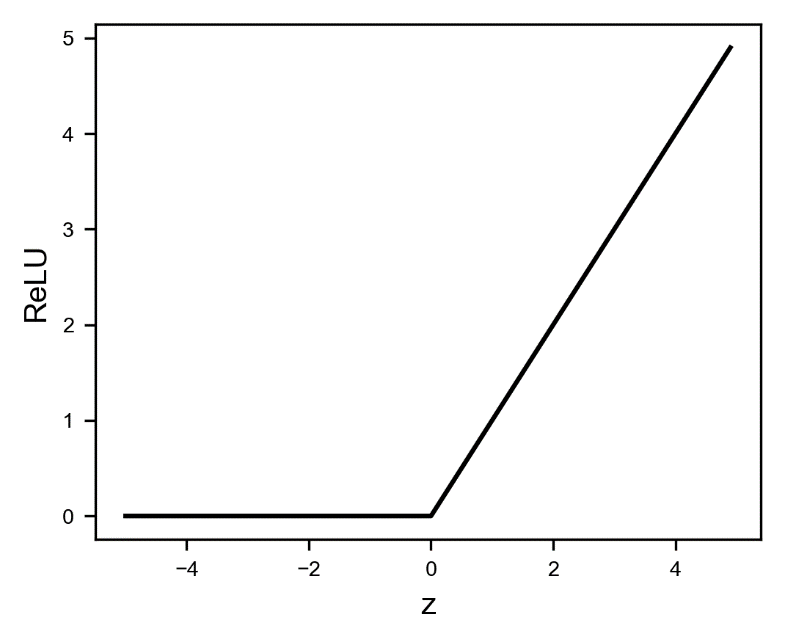
def tanh(z):

return np.tanh(z)

### ReLU (Rectified Linear Unit) Activation Function

The ReLu has the following formula

Figure 1-7: the ReLU function



It is interesting to spend a few moments to see how can we implement it in a smart way in Python. Note that when we will start using tensorflow we will have it already implemented for us, but is very instructive to see how different Python implementation can make a difference when implementing complex deep learning models.

In Python you can implement the ReLU function in several ways. Listed below are 4 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 speed. Let's generate a numpy array with elements

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

and let's 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. The method 1 is 4 times faster than the 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 np.maximum(x, 0, x) is faster than np.maximum(x, 0)? The first version update 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)

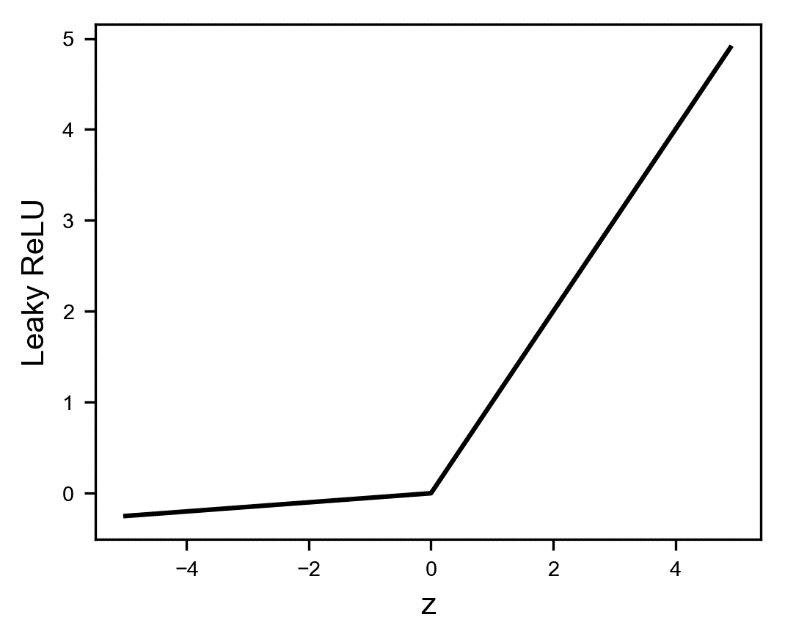
Remember when optimizing your code even small changes may make a huge difference. In deep learning programs the same chunk of code will be repeated millions and billions of times, so even a small improvement will have a huge impact on the long run. Spending time in optimizing your code is a necessary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

### Leaky ReLU

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

with a parameter typically of the order of 0.01.

Figure 1-8: The Leaky ReLU activation function with . This value has been chosen to make the difference between and more marked. Usually smaller values for are used. But testing with your model is required to find the best value.



In Python this can be for example implemented if the relu(z) function have already been defined as

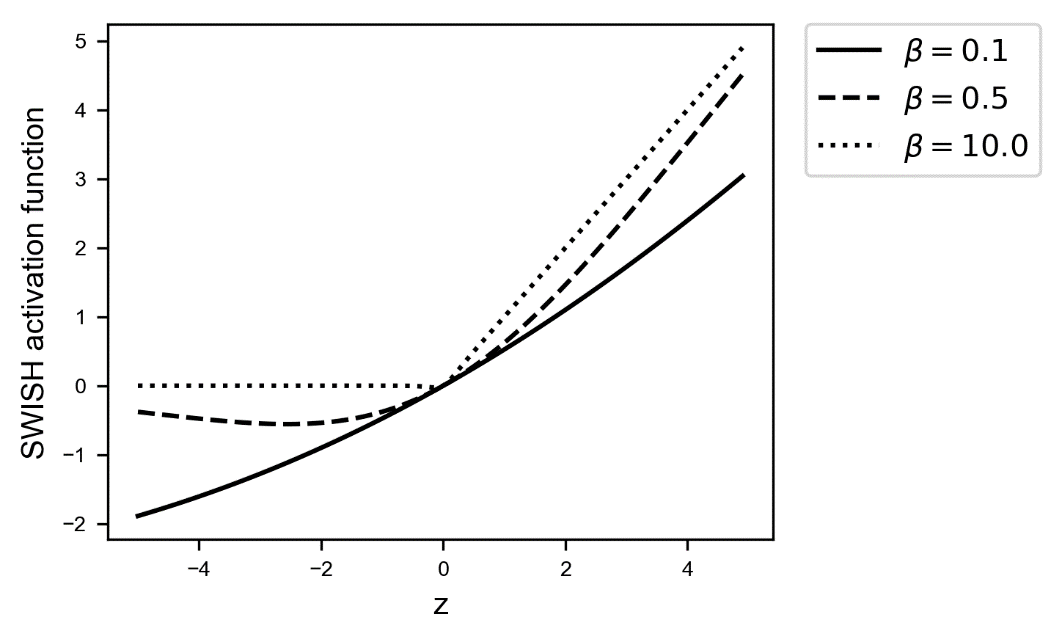
def lrelu(z, alpha):

return relu(z) - alpha \* relu(-z)

### Swish activation function

Recently Ramachandran, Zopf and Le from Google Brain [4] studied a new activation function that show great promise in the deep learning world that they called Swish. It is defined as

Figure 1-9: the Swish activation function for three different values of the parameter



where is a learnable parameter. Their studies have shown that simply replacing ReLU activation functions with Swish improves classification accuracy on ImageNet by 0.9%, that in today's deep learning world is a lot. You can find more information on ImageNet here

<http://www.image-net.org/>

Imagenet is a large database of images that is often used to benchmark new network architectures or algorithm, as in this case networks with a different activation function.

### 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
* Exponential Linear unit (ELU)
* Softplus

Practitioners uses almost always only two activation functions: the sigmoid and the ReLU (with probably the ReLU in the lead). With both you can achieve good results, and both can, given a complex enough network architecture, approximate any nonlinear function [5,6]. Remember that when using tensorflow you will not have to implement them by yourself. Tensorflow will offer an efficient implementation for you to use. But is important to know how each activation function behave to understand when to use which one.

## Cost function and gradient descent: the quirks of the learning rate

Once it is clear what a neuron is, let's discuss what it means for it (and in general for a neural network) to learn. This will allow us to introduce concepts like hyperparameters and learning rate. In almost all neural networks problems, learning means simply finding the weights (remember that a neural network is composed of many neurons, and each neuron will have its own set of weights) and biases of the network that minimize a chosen function, that is usually called the cost function and typically indicated with .

In calculus there are several methods for finding the minimum of a given function analytically. Unfortunately, in all neural networks applications, the number of weights is so big that is not possible to use those methods. Numerical methods must be relied on: the most famous being the gradient descent. It is the easiest to understand and it will give the reader the perfect basis to understand more complex algorithms that we will study later in the book. Let's give a brief overview on how it works because is one of the best algorithm in machine learning that can introduce the reader to the concept of learning rate and its quirks.

### Gradient descent: an introduction

Given a generic function , where is a vector of weights, the minimum location in weight space (that means the value for for which has a minimum) can be found with an algorithm based on the following steps:

1. Iteration 0: Choose a random initial guess
2. Iteration (with starting from 0): The weights at iteration , will be updated from the previous values at iteration , using the formula

With we have indicated the gradient of the cost function, that is a vector whose components are the partial derivatives of the cost function with respect to all the components of the weight vector :

To decide when to stop, we could check when the cost function stop changing too much, or in other words you could define a threshold and stop at any iteration (with an integer that you have to find) that satisfy for all . The problem with this approach is that is complicated and this check is very expensive in terms of performance when implemented in Python (remember you will have do this step a very large number of times), so usually people simply let the algorithm run for a fix big number of iterations and check the final results. If the final result is not what is expected then they increase this fix big number. How big? Well that depends on your problem. What you do is you choose a certain number of iterations (for example 10000 or 1000000) and let the algorithm run. At the same time, you plot the cost function vs. the number of iterations and you check that the number of iterations you have chosen was sensible. Later in this chapter you will see a practical example where I will show you how to check if the number you chose was big enough. For the moment you should know that you simply stop the algorithm after a fixed number of iterations.

Why this algorithm converges toward the minimum (and how to show it) is beyond the scope of this book and would make this chapter too long and distract the reader from the main learning goal. The latter being making you understand what is the effect of choosing a specific learning rate and what are the consequences of choosing it too big or too small.

We will assume here that the cost function is differentiable. This is not usually the case, but a discussion of this issue goes well beyond the scope of this book. People tend to use a practical approach in this case: the implementations works very well, and so those kinds of theoretical problems are usually ignored by a large part of practitioners. Remember that in deep learning models the cost function becomes an incredibly complex function, and studying it is almost impossible.

The series will hopefully converge toward the minimum location after a reasonable amount of iterations. The parameter is called the learning rate and is one of the most important parameter that is needed in the neural networks learning process.

To distinguish it from the weights the learning rate is called a hyperparameter. We will encounter more of those. A hyperparameter is parameter whose value is not determined by training and usually set before the learning process begins. By contrast, the values of parameters and are derived via training.

The word "hopefully" has been chosen for a good reason. Is possible that the algorithm will not converge towards the minimum. It is even possible that the series will oscillates between values without converging at all. Or diverge outright. Chose too big or too small and your model will not converge (or converge too slowly). To understand why this is the case let's consider a practical case and let's see how the method works while choosing different learning rates.

## The learning rate in a practical example

Let's consider the data set formed by observations y generated by the code

m = 30

w0 = 2

w1 = 0.5

x = np.linspace(-1,1,m)

y = w0 + w1 \* x

As a cost function we choose the classical Mean Squared Error (MSE)

Where we have indicated with the superscript the ith observation. Remember that with the subscript ( we have indicated the feature. To recap our notation, we will indicated with the feature and the observation. In the example here, we have just one feature, so we don’t need the subscript . The cost functioe can be implemented in Python easily as

np.average((y-hypothesis(x, w0, w1))\*\*2, axis=2)/2

where we have defined

def hypothesis(x, w0, w1):

return w0 + w1\*x

Our goal is to find the values for and that minimize .

To apply the gradient descent method, we need to calculate the series for and as in equation (2.8). We have the following equations

Simplifying equations (2.10) by calculating the partial derivatives gives:

Since and .

Equations (2.11) are the ones that need to be implemented in Python if we want to code the gradient descent algorithm by ourselves.

The derivation of the equation in (2.11) has the goal of showing how the equations for gradient descent becomes very complicated very quickly even for a very easy case. In the next section we will build our first model with tensorflow. One of the best aspect of the library is that all those formulas are calculated automatically for you and you don't have to bother calculating anything. Implementing equations like the ones in (2.11) and debugging them can take quite some time and prove to be impossible the moment you are dealing with large neural networks of interconnected neurons.

We omit in the book the complete Python implementation of the example since it would take too much space.

It is instructive to check how the model works varying the learning rate. In Figures 1-10, 1-11 and 1-12 we have drawn the contour lines[[2]](#footnote-2) of the cost functions and we have plotted on top of it the series as points to visualise how the series converges (or don't). The minimum is indicated in the Figures with a circle approximately placed in the center. We will consider the values (in Figure 1-10), (in Figure 1-11) and (in Figure 1-12). In the first case (in Figure 1-10) the converging is well behaved and in just 8 steps the method converges toward the minimum. When (Figure 1-11) the method makes steps that are too big (remember the steps are given by and therefore the bigger the bigger the steps) and is unable to get close to the minimum. It keeps oscillating around it without reaching it. In this case the model will never converge. In the last case when (Figure 1-12) the learning is so slow that it will take many more steps to get close to the minimum. In some cases, the cost function may be so flat around the minimum it may even happen that the method would take such a big number of iterations to converge that practically you will not get close enough to the real minimum in a reasonable amount of time. In Figure 1-12 we plotted 300 iterations, but the method is not even very close to the minimum.

Choosing the right learning rate is of paramount importance when coding the learning part of a neural network. Chose it too big and it may happen that the method will just bounce around the minimum without ever reaching it. Chose it too small and the algorithm may become so slow, that you will not be able to find the minimum in a reasonable amount of time (or number of iterations). A typical sign of a learning rate that is too big is that the cost function may become nan (not a number in Python slang). Printing the cost function at regular intervals during the training process is a good way of checking such kind of problems. This will give you a chance of stopping the process and avoid wasting time (in case you see nan appearing). We will see a concrete example later in the chapter.

In deep learning problems each iteration will cost time and you will have to perform this process several times. Choosing the right learning rate is a key part of designing a good model because it will make training much faster (or make it impossible).

Figure 1-10: Illustration of gradient descent algorithm. Here the learning rate of has been chosen. The algorithm converges rather quickly toward the minum in just 8 steps. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image.

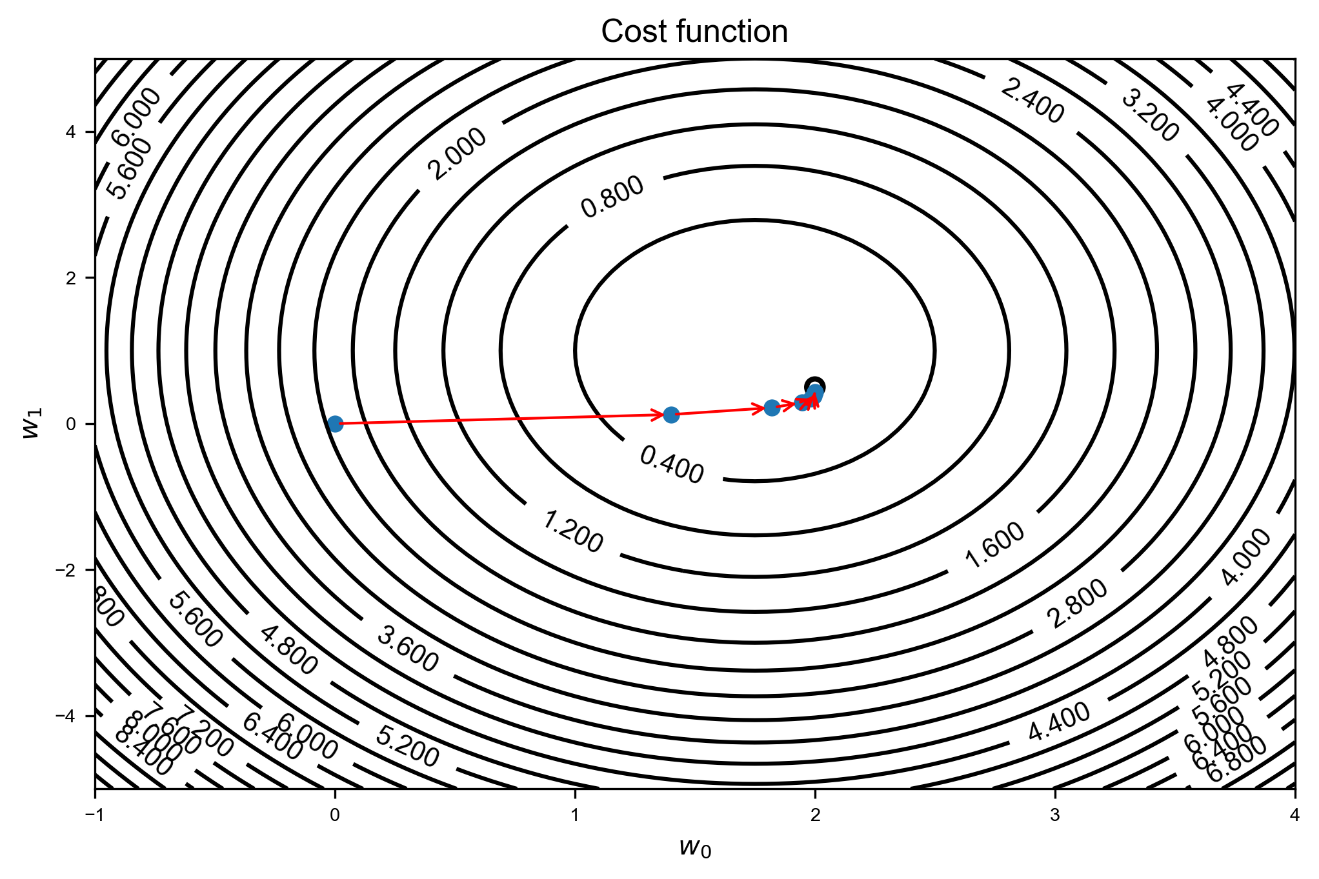


Figure 1-11: Illustration of gradient descent algorithm. Here the learning rate of has been chosen. Here the problem of a learning rate that is too big is clearly visible. The algorithm cannot converge because the steps that it takes are too big to be able to get close to the minimum. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image.

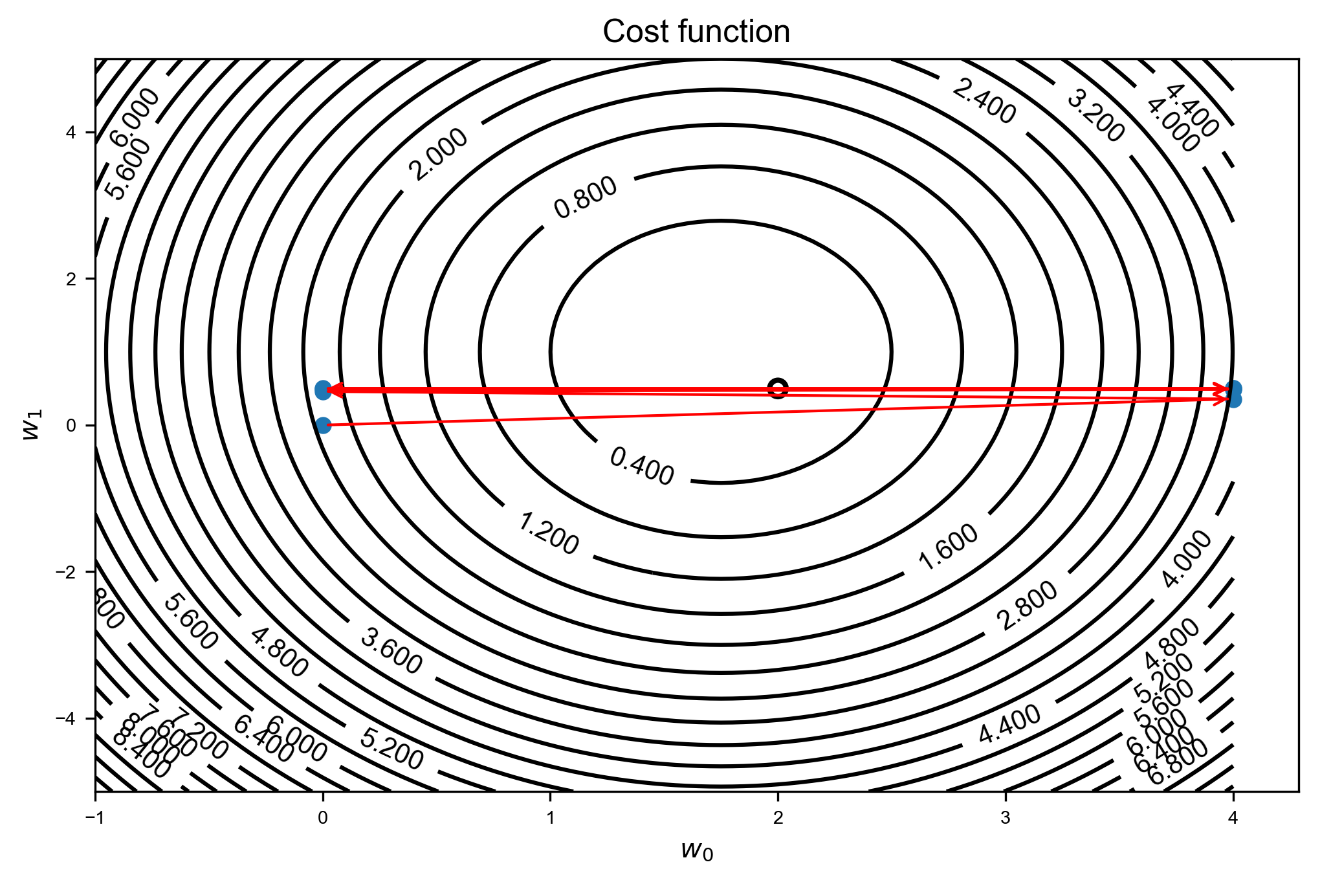
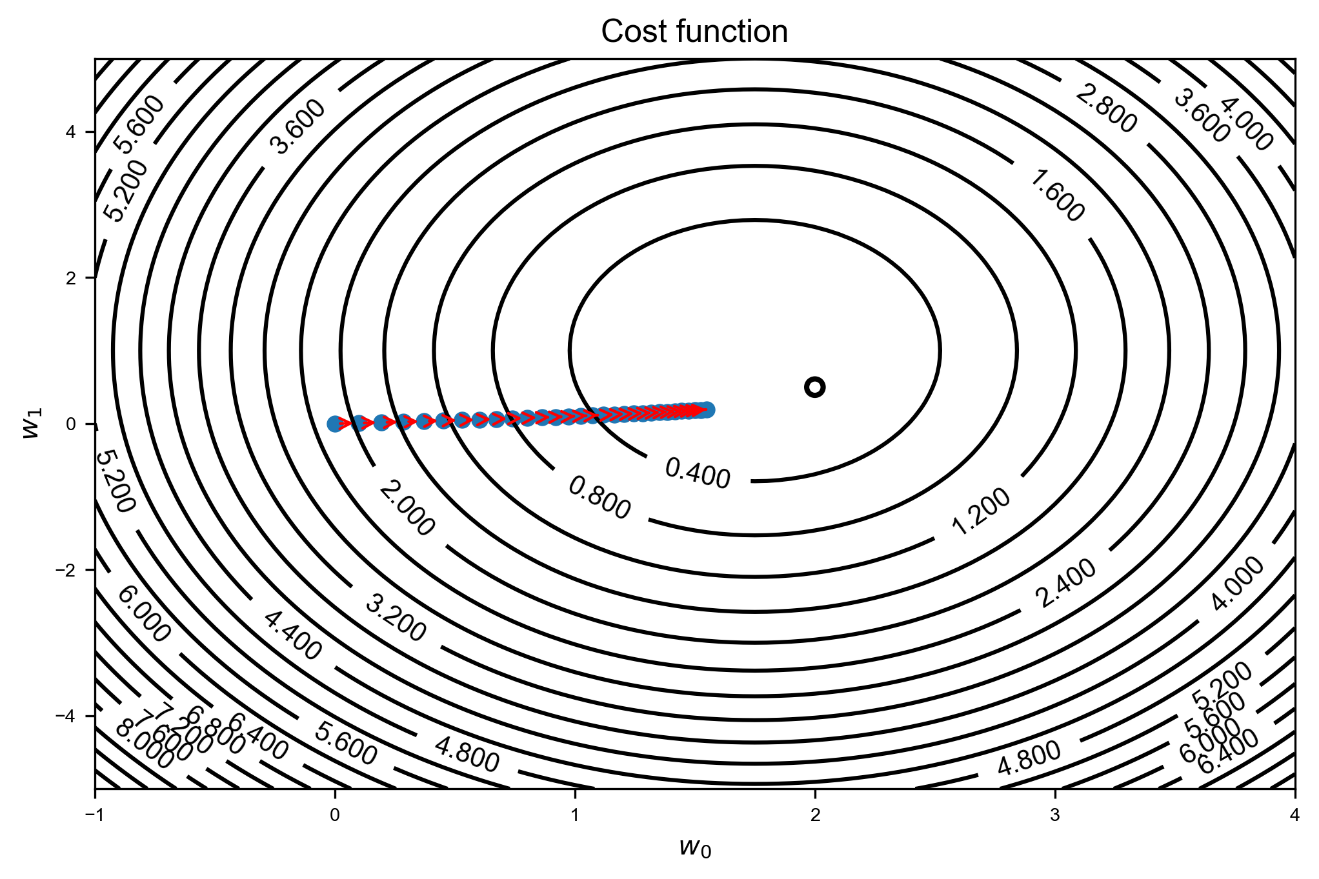


Figure 1-12: Illustration of gradient descent algorithm. Here the learning rate of has been chosen. Choosing the learning rate too small will make the algorithm quite slow in converging, requiring a lot more steps. In the Figure we have the first 300 iterations. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image.

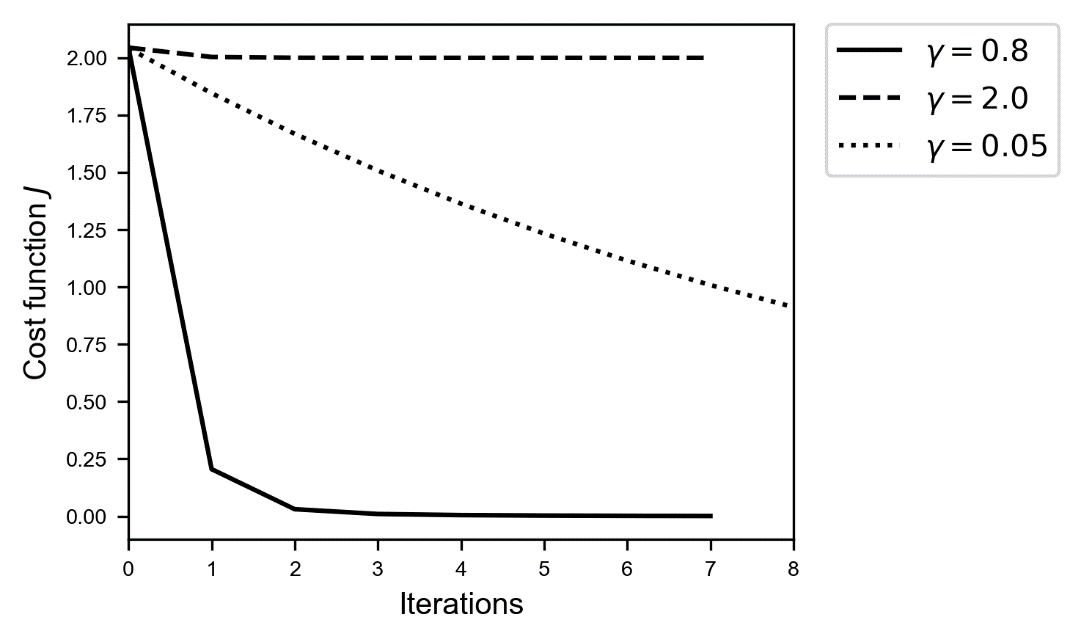


Sometimes it is efficient to change the learning rate during the process. You start with a bigger value to get close to the minimum faster, and then you reduce it progressively to make sure that you get as close as possible to the real minimum. We will discuss this approach later in the book.

There are no fix rules on how to choose the right learning rate. It depends on the model, on the cost function, on the starting point and so on. A good rule of thumb is to start with and then see how the cost function behave. It is rather common to plot vs. the number of iterations to check that it decreases and how fast is decreasing.

A good way of checking the convergence is to plot the cost function vs. the number of iterations. In this way you can check its behavior. How the cost function looks like in our three learning rates for the example above is shown in Figure 1-13.

Figure 1-13: The cost function vs. the number of iterations (only the first 8 are considered). You can clearly see how the case with goes to zero rather fast, showing that we have reached a minimum. The case with does not even start to go down. It continues to remain at almost the same initial value. And finally the case with starts to go down but is a lot slower that the first case.



So here are the conclusions we should draw from Figure 1-13 for the three cases:

* 🡪 is decreasing, so is good, but after 8 iterations we have not reached a plateau, so we need to use many more iterations until we see that is not changing much anymore
* 🡪 is not decreasing. We should check our learning rate to see if it helps. Trying smaller values would be a good starting point
* 🡪 The cost function decreasing rather quickly and then remain constant. That is a good sign and indicates that we have reached a minimum.

Remember also that the absolute value of the learning rate is not relevant. What is important is the behavior. We can multiply our cost function by a constant and that would not influence our learning at all. Don't look at the absolute values, check how fast and how the cost function is behaving. Additionally, the cost function will almost never reach zero, so don't expect it. The value of at its minimum is almost never zero (it depends on the functions itself). In the section about linear regression you will see an example where the cost function will not reach zero.

When training your models, remember to always check the cost function vs. the number of iterations (or number of swipes over the entire training set, called epochs). This will give you an efficient way of estimating if the training is efficient, if is working at all and can give you hints on how to optimize it.

Now that we have defined the basis, we will use a neuron to solve two simple problems with machine learning: linear and logistic regression.

## An example of linear regression in tensorflow

The first type of regression will offer us the opportunity to understand how to build a model in tensorflow. To explain how to perform linear regression efficiently with one neuron we must first explain some additional notation. In the previous sections we have talked about inputs . Those are the so-called features that describe an observation. Normally we have many observations. As briefly explained before we will use an upper index to indicate the different observations between round parenthesis. Our observation will be indicated with , and the feature of the observations will be indicated as . We will indicate the number of observations with .

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.

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. Let's spend some time now in writing all our equations in matrix form. In this way our Python implementation will be much easier later.

The entire set of inputs (features and observations) can be written in matrix form. We will use the following notation

where each column is an observation and each row represent a feature in the matrix that has dimensions . We can also write the output values in matrix form. If you remember our neuron discussion, we have defined a for one observation . Putting each observation in a column we can use the following notation

Where we have . We will define as

Where with we intend the function applied element by element to the matrix .

Although has dimensions we will use the term matrix for it and not vector, to use consistent names in the book. This will also help the reader remember that we should always use matrix operations. is for us simply a matrix with just one row.

You know from Chapter 1 that in tensorflow you need to declare explicitly the dimensions of our matrices (or tensors), so is a good idea to have them well under control. Here is an overview of the dimensions of all the vectors and matrices we will use:

* has dimensions
* has dimensions
* has dimensions
* has dimensions
* has dimensions

Now that the formalism is clear, we will prepare the dataset.

### The dataset for our linear regression model

To make things a bit more interesting let's use a real dataset. We will use the so-called Boston data set[[3]](#footnote-3). This contains information collected by the US Census service concerning housing around Boston. Each record in the database describes a Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970. The attributes are deﬁned as follows [3]:

1. CRIM: per capita crime rate by town
2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS: proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: nitric oxides concentration (parts per 10 million)
6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940
8. DIS: weighted distances to ﬁve Boston employment centers
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. B - 1000(Bk - 0.63)^2 - Bk proportion of blacks by town
13. LSTAT: % lower status of the population
14. MEDV: Median value of owner-occupied homes in $1000s

Our target variable MEDV, the one we want to predict, is the median price of the house in $1000s for each suburb. We don't need for our example to understand or study the features. My goal here is to show you 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 tensorflow.

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

Let's import the usual libraries including the sklearn.datasets. Importing the data and getting features and target is very easy with the help of the sklearn.datasets package. You don't need to download csv files and import them. Simply run the following code

import matplotlib.pyplot as plt

%matplotlib inline

import tensorflow as tf

import numpy as np

from sklearn.datasets import load\_boston

boston = load\_boston()

features = np.array(boston.data)

labels = np.array(boston.target)

Every dataset in the sklearn.datasets package comes with a description. You can check it with the command

print(boston["DESCR"])

Now let's check how many observations and features we have

n\_training\_samples = features.shape[0]

n\_dim = features.shape[1]

print('The dataset has',n\_training\_samples,'training samples.')

print('The dataset has',n\_dim,'features.')

To link the mathematical notation with the Python code n\_training\_samples is and n\_dim is . The code will give the following results

The dataset has 506 training samples.  
The dataset has 13 features.

Is a good idea to normalize each numerical feature defining normalized features according to the formula

Where is the average of the feature and is its standard deviation. This can be easily calculated in numpy with the following function

def normalize(dataset):

mu = np.mean(dataset, axis = 0)

sigma = np.std(dataset, axis = 0)

return (dataset-mu)/sigma

To normalize our features numpy array we simply have to call the function

features\_norm = normalize(features)

Now each feature contained in the numpy array features\_norm will have average zero and standard deviation of one.

It is almost always a good idea to normalize the features so that their average is zero and the standard deviation is one. Sometimes some features are much bigger than others and can have a stronger influence on the model, thus bringing wrong predictions. Particular care is needed when the dataset is split in training and test dataset to have consistent normalizations. We will discuss this at length in Chapter XX.

For this chapter we will simply use all the data for the training to concentrate on implementation details.

train\_x = features\_norm

train\_y = labels

print(train\_x.shape)

print(train\_y.shape)

The last two prints will give us the dimensions of our new matrices

(13, 506)  
(506,)

The train\_x array has dimensions of and that is exactly what we expect. Remember for our discussion that have dimensions .

The training target train\_y has dimensions of (506,), that is how numpy describe one-dimensional arrays. Tensorflow wants to have dimensions of (1, 506) (remember our discussion done previously?) so we need to reshape the array in this way

train\_y = train\_y.reshape(1,len(train\_y))

print(train\_y.shape)

and our print statements give us

(1, 506)

### Neuron and cost function for linear regression

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

where the sum is over all observations.

The tensorflow code to build this neuron and define the cost function is actually very easy

tf.reset\_default\_graph()

X = tf.placeholder(tf.float32, [n\_dim, None])

Y = tf.placeholder(tf.float32, [1, None])

learning\_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.ones([n\_dim,1]))

b = tf.Variable(tf.zeros(1))

init = tf.global\_variables\_initializer()

y\_ = tf.matmul(tf.transpose(W),X)+b

cost = tf.reduce\_mean(tf.square(y\_-Y))

training\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost)

Note that in tensorflow you don't have to explicitly declare the number of observations. You can use None in the code. In this way you will be able to run the model on any dataset independently on the number of observations without modifying your code.

In the code we have indicated the neuron output as y\_ since we don't have a hat in Python. Let's clarify a bit which line of code does what

* X = tf.placeholder(tf.float32, [n\_dim, None]) 🡪 contains the matrix that must have dimensions . Remember that in our code n\_dim is and that m is not declared explicitly in tensorflow, and in its place we use None.
* Y = tf.placeholder(tf.float32, [1, None]) 🡪 contains the output values that must have dimensions . Here applies that instead of we use None since we want to use the same model for different data set (that will have a different number of observations);
* learning\_rate = tf.placeholder(tf.float32, shape=()) 🡪 contains the learning rate as a parameter instead of a constant so that we can run the same model varying it, without creating a new neuron each time;
* W = tf.Variable(tf.zeros([n\_dim, 1])) 🡪 defines and initialize the weights with zeros. Remember that the weights must have dimensions ;
* b = tf.Variable(tf.zeros(1)) 🡪 defines and initialize the bias with zero.

Remember that in tensorflow a placeholder is a tensor that will not change during the learning phase, while a variable is one that will change. Weights and bias will be updated during the learning. Now we must define what to do with all those quantities. Remember we need to calculate . The chosen activation function is the identity function, so will also be the output of our neuron.

* init = tf.global\_variables\_initializer() 🡪 creates a piece of the graph that initialize the variable and add it to the graph.
* y\_ = tf.matmul(tf.transpose(W),X)+b 🡪 calculates the output of the neuron. The output of a neuron is . Since the activation function for linear regression is the identity, the output is . Remember that being a scalar is not a problem. Python broadcasting will take care of it, expanding it to the right dimensions to make the sum between a vector and a scalar possible.
* cost = tf.reduce\_mean(tf.square(y\_-Y)) 🡪 defines the cost function. Tensorflow provides an easy and efficient way of calculating the average: tf.reduce\_mean() that simply performs the sum of all the element of the tensor and divide it by the number of the elements.
* training\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost) 🡪 tells tensorflow which algorithm to use to minimize the cost function. In tensorflow language, the algorithms used to minimize the cost function are called optimizers. We now use gradient descent with the given learning rate. Later in the book other optimizers will be extensively studied.

You will remember from the introduction in Chapter 1 that the previous code will not run any model. It simply defines the computational graph. Let's define a function that will perform the actual learning and will run our model. Is easier to define it in a function so that we can rerun it changing, for example, the learning rate or the number of iterations we want to use.

def run\_linear\_model(learning\_r, training\_epochs, train\_obs, train\_labels, debug = False):

sess = tf.Session()

sess.run(init)

cost\_history = np.empty(shape=[0], dtype = float)

for epoch in range(training\_epochs+1):

sess.run(training\_step, feed\_dict = {X: train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_ = sess.run(cost, feed\_dict={ X:train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_history = np.append(cost\_history, cost\_)

if (epoch % 1000 == 0) & debug:

print("Reached epoch",epoch,"cost J =", str.format('{0:.6f}', cost\_))

return sess, cost\_history

Let's go through the code again line by line

* sess = tf.Session() 🡪 creates a tensorflow session;
* sess.run(init) 🡪 runs the initialization of the different element of the graphs;
* cost\_history = np.empty(shape=[0], dtype = float) 🡪 creates an empty vector (for the moment with zero elements) where the value of our cost function at each iteration is stored;
* for loop… 🡪 In this loop tensorflow will perform the gradient descent steps that we have discussed earlier and update the weights and the bias. In addition it will save in the array cost\_history the value of the cost function each time: cost\_history = np.append(cost\_history, cost\_);
* if (epoch % 1000 == 0)… 🡪 Every 1000 epochs we will print the value of the cost function. 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 testes 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;
* return sess, cost\_history 🡪 returns the session (in case you want to calculate something else), and the array containing the cost function values (we will use this array to plot it).

Running the model is as easy as using the call

sess, cost\_history = run\_linear\_model(learning\_r = 0.01,

training\_epochs = 10000,

train\_obs = train\_x,

train\_labels = train\_y,

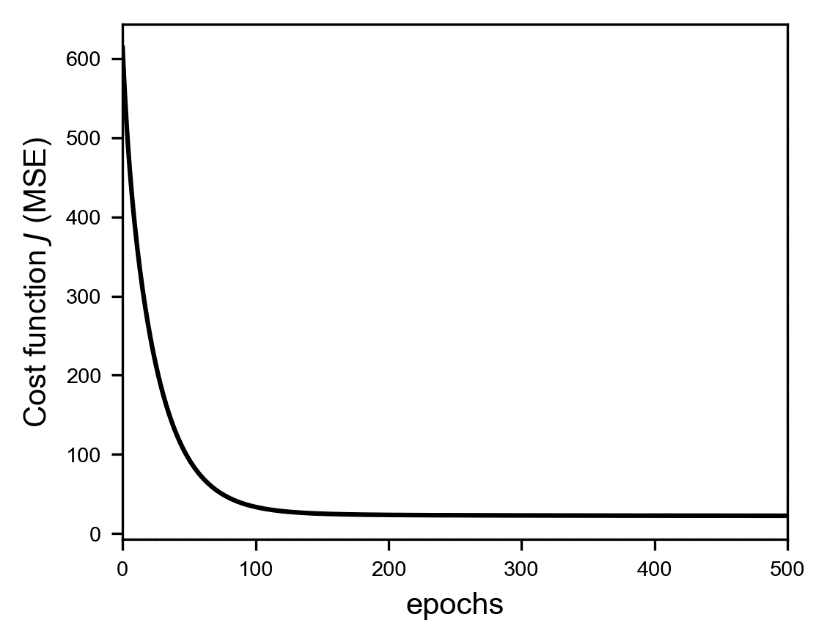
debug = True)

The output of the command will be the cost function every 1000 epochs (check in the function definition the if starting with if (epoch % 1000 == 0))

Reached epoch 0 cost J = 613.947144  
Reached epoch 1000 cost J = 22.131165  
Reached epoch 2000 cost J = 22.081099  
Reached epoch 3000 cost J = 22.076544  
Reached epoch 4000 cost J = 22.076109  
Reached epoch 5000 cost J = 22.07606  
Reached epoch 6000 cost J = 22.076057  
Reached epoch 7000 cost J = 22.076059  
Reached epoch 8000 cost J = 22.076059  
Reached epoch 9000 cost J = 22.076054  
Reached epoch 10000 cost J = 22.076054

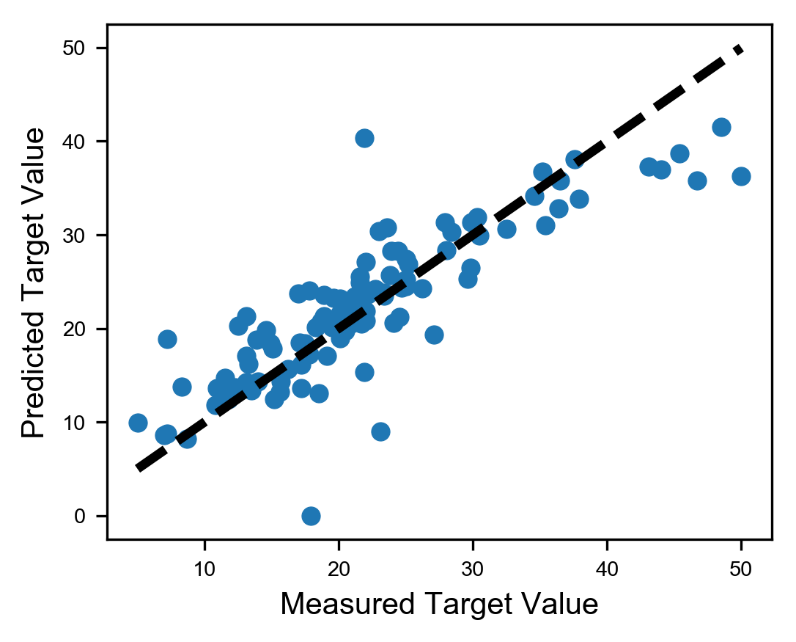
The cost function clearly decreases and then reach a value and stays almost constant. That is a good sign indicating that the cost function has reached a minimum. That does not mean that our model is good or that will give good prediction. This tells us only that the learning has worked efficiently.

Figure 1-14: the cost function resulting in our model applied to the Boston dataset with a learning rate of 0.01. We plot only the first 500 epochs, since the cost function has already reached almost its final value.



It would be nice to be able to visualize graphically how good our fit is. Since we have 13 features, it is not possible to plot the price versus the other features. However, it is helpful to get a feeling on how good the model predicts the observed values. This can be done by plotting our predicted target variable vs. the observed one. If we could predict perfectly our target variable all the points should be on the diagonal line in the plot. The more spread the points are around the line worst our model is in predicting. Let's check how our model is doing.

Figure 1-15: The predicted target value vs. the measured target value for our model applied to our trianing data.



The points lay reasonably well around the line, so it seems we can predict our price up to a certain degree. A more qualitative method for estimating the goodness of our regression is the MSE itself (that in our case is simply our cost function). If the value we are obtaining (22.08 in 1000 USD) is good enough or not depends on the problem you are trying to solve or the constraint and requirements you have been given.

### Satisficing and optimizing metric

We have seen that is not easy to decide if a model is good or not good. Figure 1-15 will not allow us to describe quantitively how good (or not good) our model is. For this we must define a metric.

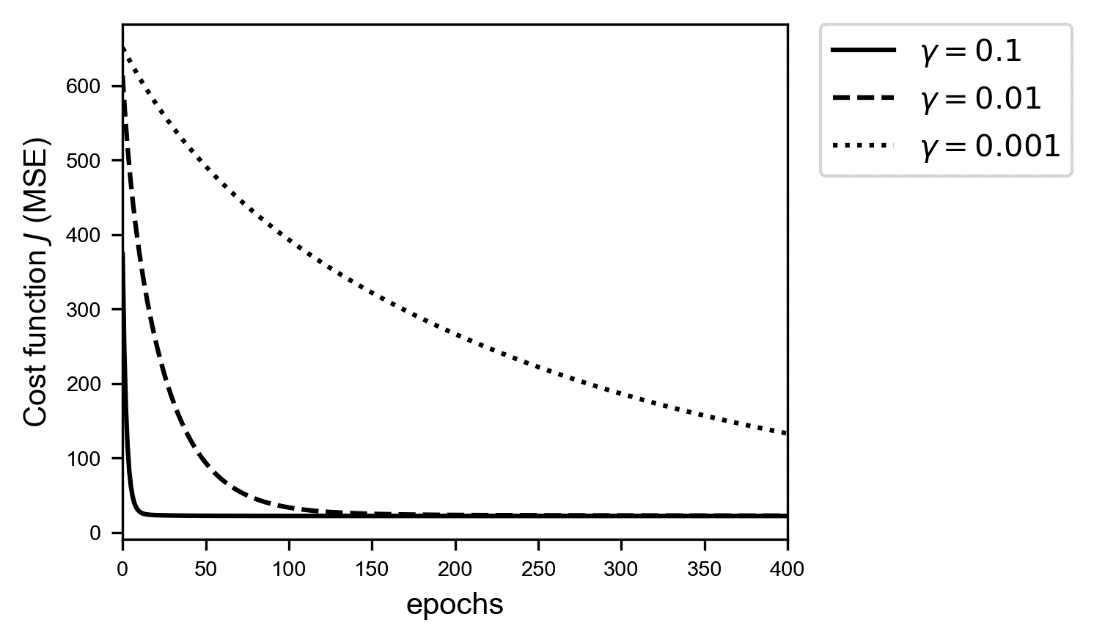
The easiest way is setting up what is called a single number evaluation metric. That means that you calculate one single number and base your model evaluation on that number. Is easy and very practical. For example, you could use the accuracy or the F1 score in case of classification or the MSE in case of regression. Normally in real life you will receive goals and constraints for your model. For example, your company may want to predict house prices with a MSE < 20 (in 1000 USD) and your model should be able to run on an ipad, or in less than 1 second. Is useful to distinguish between two types of metrics:

* **Satisficing metric** 🡪 searching through available alternatives until an acceptability threshold is met. For example: code running (RT) time; minimizing the cost function subject to RT < 1 sec or choosing between mode the one that have a RT < 1 sec.
* **Optimizing metric** 🡪 searching through available alternatives to maximize a specific metric. For example, choosing the model (or the hyperparameters) that maximize accuracy

If you have several metrics, you should always choose one optimizing and the rest satisficing.

We have written our code to be able to run our model with different parameters. It is very instructive now to do that. Here is how the cost function behaves for three different learning rates: 0.1, 0.01 and 0.001.

Figure 1.16: the cost function for linear regression applied to the Boston dataset for three learning rates: 0.1 (solid line), 0.01 (dashed line) and 0.001 (dotted line). The smaller the learning rate, the slowest the learning process.



As expected for very small learning rates (0.001) the gradient descent algorithm is very slow in finding the minimum. While with a bigger value (0.1) the method works quickly. This kind of plot is very useful to give you an idea on how fast and how good the learning process is going. We will see cases later in the book where the cost function is much less well behaved. For example, when applying dropout regularization, the cost function will not be smooth anymore.

## An example of logistic regression

### Introduction

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 .

### Cost function

As a cost function we will use the cross entropy[[4]](#footnote-4). The function for one observation is

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

In chapter XXXXX we will give a complete derivation of logistic regression from scratch, but for the moment, tensorflow will take care of all the details: derivatives, gradient descent implementation and so on. We just need to build the right neuron and we will be on our way.

### Activation function

Remember 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. For our logistic regression we will use the sigmoid function as the activation function

### The dataset

To build an interesting model we will use a modified version of the MNIST dataset. You will find all information on it at this link

<http://yann.lecun.com/exdb/mnist/>

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that we can use to train our model. The MNIST database contains 70000 images. "The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field" (from <http://yann.lecun.com/exdb/mnist/>).

Our features will be the gray value for each pixel, so we will have features whose values will go from 0 to 255 (gray values). The dataset contains all ten digits: from 0 to 9. With the following code you can prepare the data to use in the sections below. As usual, first let's import the necessary library

from sklearn.datasets import fetch\_mldata

Then let's load the data

mnist = fetch\_mldata('MNIST original')

X,y = mnist["data"], mnist["target"]

Now X contains the input images and y the labels (remember the value we want to predict is called target in machine learning jargon). Just typing X.shape will give you the shape of X: (70000, 784). Note that X have 70000 rows (each row is an image) and 784 columns (each column is a feature, or a pixel gray value in our case). Let's check how many digits we have in our dataset

for i in range(10):

print ("digit", i, "appears", np.count\_nonzero(y == i), "times")

That gives us

digit 0 appears 6903 times  
digit 1 appears 7877 times  
digit 2 appears 6990 times  
digit 3 appears 7141 times  
digit 4 appears 6824 times  
digit 5 appears 6313 times  
digit 6 appears 6876 times  
digit 7 appears 7293 times  
digit 8 appears 6825 times  
digit 9 appears 6958 times

It is useful to define a function to visualize the digits to get an idea of how they look like

def plot\_digit(some\_digit):

some\_digit\_image = some\_digit.reshape(28,28)

plt.imshow(some\_digit\_image, cmap = matplotlib.cm.binary, interpolation = "nearest")

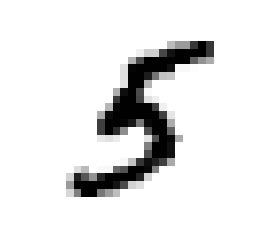
plt.axis("off")

plt.show()

For example we can plot one randomly (check Figure 1-17)

plot\_digit(X[36003])

Figure 1-17: the 36003 digit in the dataset. Is easly recognizable as a 5.



The model we want to implement here is a simple logistic regression for binary classification, so the dataset needs to be reduced to two classes, or in this case to two digits: we choose the ones and the twos. Let's extract from our dataset only the images that represent a 1 or a 2. Our neuron will try to recognize if a given image is of class 0 (a digit 1) or of class 1 (a digit 2).

X\_train = X[np.any([y == 1,y == 2], axis = 0)]

y\_train = y[np.any([y == 1,y == 2], axis = 0)]

Then the input observations need to be normalized (remember you don't want your input data to be too big when using the sigmoid activation function since you have 784 of them)

X\_train\_normalised = X\_train/255.0

We chose 255 since each feature is the gray value of a pixel in the image and gray levels in the source images go from 0 to 255. In Chapter XX we will discuss at length why we need to normalize the input features, for the moment believe me that is a necessary step. We want to have in each column an input observation and each row should represent a feature (a pixel gray value), so we need to reshape the tensors

X\_train\_tr = X\_train\_normalised.transpose()

y\_train\_tr = y\_train.reshape(1,y\_train.shape[0])

and we can define a variable n\_dim to contain the number of features

n\_dim = X\_train\_tr.shape[0]

Now comes a very important point. The labels in our dataset as imported will be 1 or 2 (they simply tell you which digit the image represents). But we will build our cost function with the assumptions that our class' labels are 0 and 1, so we need to rescale our y\_train\_tr array.

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.

y\_train\_shifted = y\_train\_tr - 1

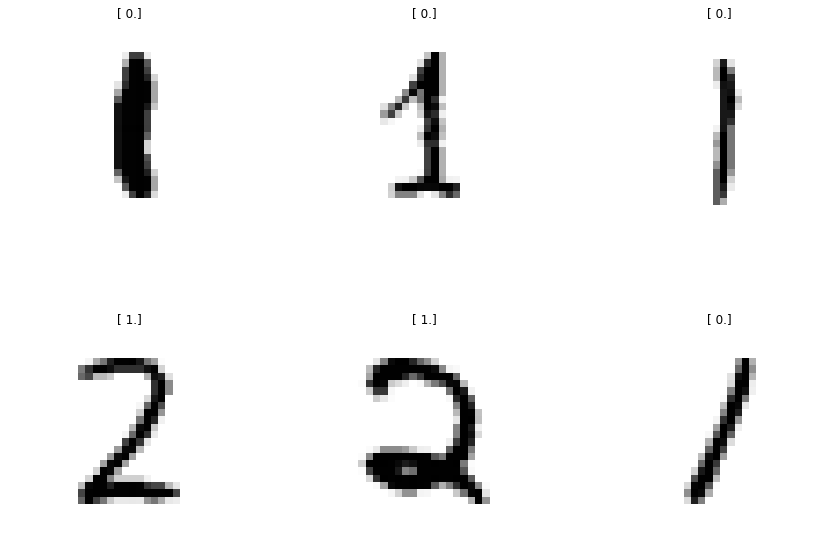
Now all images representing a 1 will have a label of 0, and all images representing a 2 will have a label of 1. Finally let's use some proper names for our Python variables

Xtrain = X\_train\_tr

ytrain = y\_train\_shifted

In Figure 1-18 there are some of the digits we are dealing with.

Figure 1-18: six random digits chosen from the dataset. In squarebrackets the relative rescaled labels (remember labels in our dataset are now 0 or 1).



### Tensorflow implementation

The tensorflow implementation is not difficult and is almost the same as for the linear regression. First let's define placeholders and variables

tf.reset\_default\_graph()

X = tf.placeholder(tf.float32, [n\_dim, None])

Y = tf.placeholder(tf.float32, [1, None])

learning\_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n\_dim]))

b = tf.Variable(tf.zeros(1))

init = tf.global\_variables\_initializer()

Notice that the code is the same we used for the linear regression model. However, we must define a different cost function (as discussed earlier) and a different neuron output (the sigmoid function)

y\_ = tf.sigmoid(tf.matmul(W,X)+b)

cost = - tf.reduce\_mean(Y \* tf.log(y\_)+(1-Y) \* tf.log(1-y\_))

training\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost)

We have used the sigmoid function for the output of our neuron, using tf.sigmoid(). The code that will run the model is the same of what we have used for the linear regression. We have just changed the name of the function

def run\_logistic\_model(learning\_r, training\_epochs, train\_obs, train\_labels, debug = False):

sess = tf.Session()

sess.run(init)

cost\_history = np.empty(shape=[0], dtype = float)

for epoch in range(training\_epochs+1):

sess.run(training\_step, feed\_dict = {X: train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_ = sess.run(cost, feed\_dict={ X:train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_history = np.append(cost\_history, cost\_)

if (epoch % 500 == 0) & debug:

print("Reached epoch",epoch,"cost J =", str.format('{0:.6f}', cost\_))

return sess, cost\_history

Let's run the model and see the results. We will choose to start with a learning rate of 0.01.

sess, cost\_history = run\_logistic\_model(learning\_r = 0.01,

training\_epochs = 5000,

train\_obs = Xtrain,

train\_labels = ytrain,

debug = True)

The output of our code (stopped after 3000 epochs) is

Reached epoch 0 cost J = 0.678598  
Reached epoch 500 cost J = 0.108655  
Reached epoch 1000 cost J = 0.078912  
Reached epoch 1500 cost J = 0.066786  
Reached epoch 2000 cost J = 0.059914  
Reached epoch 2500 cost J = 0.055372  
Reached epoch 3000 cost J = nan

What happened? Suddenly at some point our cost function assumes the value nan (not a number). It seems that the model does not seems to do well after a certain point. If the learning rate is too big or you initialize your weights wrongly your values for may get very close to zero or one (remember that the sigmoid function assumes values very close to 0 or 1 for very big negative or positive values of ). Remember that in the cost function you have the two terms tf.log(y\_) and tf.log(1-y\_) and since the log function is not defined for a value of zero, if y\_ is 0 or 1 you will get a nan since you the code will try to evaluate tf.log(0). As an example, we can run the model with a learning rate of 2.0. After only one epoch you will get already a nan value for the cost function. And is easy to understand why if you print out the value for before and after the first training step. Simply modify your model code and use the following version

def run\_logistic\_model(learning\_r, training\_epochs, train\_obs, train\_labels, debug = False):

sess = tf.Session()

sess.run(init)

cost\_history = np.empty(shape=[0], dtype = float)

for epoch in range(training\_epochs+1):

print ('epoch: ', epoch)

print(sess.run(b, feed\_dict={X:train\_obs, Y: train\_labels, learning\_rate: learning\_r}))

sess.run(training\_step, feed\_dict = {X: train\_obs, Y: train\_labels, learning\_rate: learning\_r})

print(sess.run(b, feed\_dict={X:train\_obs, Y: train\_labels, learning\_rate: learning\_r}))

cost\_ = sess.run(cost, feed\_dict={ X:train\_obs, Y: train\_labels, learning\_rate: learning\_r})

cost\_history = np.append(cost\_history, cost\_)

if (epoch % 500 == 0) & debug:

print("Reached epoch",epoch,"cost J =", str.format('{0:.6f}', cost\_))

return sess, cost\_history

You will get the following result (after stopping the training after just 1 epoch)

epoch: 0  
[ 0.]  
[-0.05966223]  
Reached epoch 0 cost J = nan  
epoch: 1  
[-0.05966223]  
[ nan]

You see how goes from 0 to -0.05966223, and then goes to nan. Therefore turns into nan, then turns also into nan and finally the cost function, being a function of will also result into a nan. Simply because the learning rate is way too big.

What is the solution? You should try a different (read: much smaller) learning rate.

Let's try and see if we can get a result that is more stable after 2500 epochs). We run the model with the call

sess, cost\_history = run\_logistic\_model(learning\_r = 0.005,

training\_epochs = 5000,

train\_obs = Xtrain,

train\_labels = ytrain,

debug = True)

The output of the command is

Reached epoch 0 cost J = 0.685799  
Reached epoch 500 cost J = 0.154386  
Reached epoch 1000 cost J = 0.108590  
Reached epoch 1500 cost J = 0.089566  
Reached epoch 2000 cost J = 0.078767  
Reached epoch 2500 cost J = 0.071669  
Reached epoch 3000 cost J = 0.066580  
Reached epoch 3500 cost J = 0.062715  
Reached epoch 4000 cost J = 0.059656  
Reached epoch 4500 cost J = 0.057158  
Reached epoch 5000 cost J = 0.055069

No more nan in our output. To evaluate our model, we need to choose an optimizing metric (as discussed before). For a binary classification problem, a classical metric is the accuracy (that we can indicate with ) that can be understood as a measure of the difference between a result and its "true" value. Mathematically it can be calculated as

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

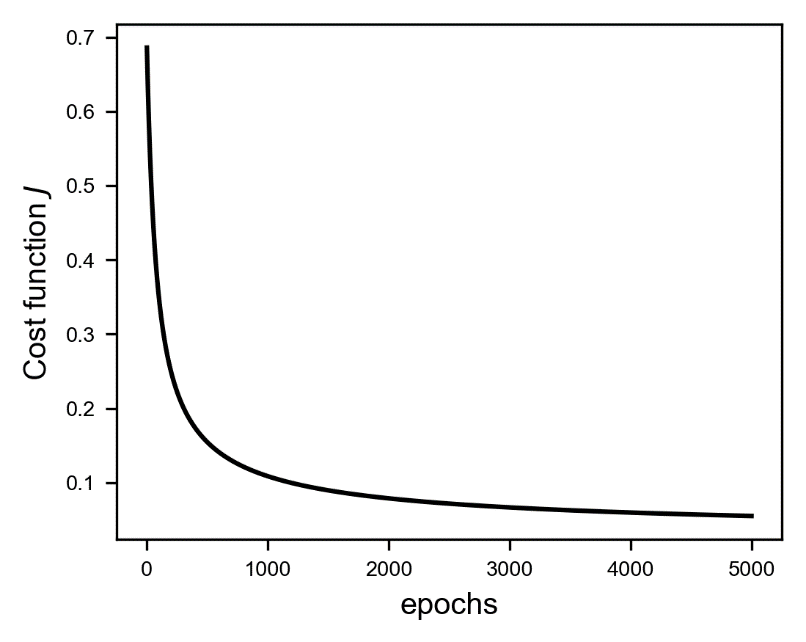
correct\_prediction1 = tf.equal(tf.greater(y\_, 0.5), tf.equal(Y,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction1, tf.float32))

print(sess.run(accuracy, feed\_dict={X:Xtrain, Y: ytrain, learning\_rate: 0.05}))

With this model we reach an accuracy of 98.6%. Not bad for a network with just one neuron.

Figure1-19: The cost function vs. epochs for a learning rate of 0.005.



You could also try to run the previous model (with a learning rate of 0.005) for more epochs. You will discover that around 7000 epochs the nan will reappear. The solution here would be to reduce the learning rate with increasing number of epochs. A simple approach like halving the learning rate every 500 epochs will get rid of the nans. We will discuss a similar approach in more detail later in the book.

## 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.cs.toronto.edu/~delve/data/boston/bostonDetail.html>, last accessed 27.12.2017

[4] P. Ramachandran, B. Zoph and Q.V. Le (2017), "Searching for activation functions" arXiv:1710.05941 [cs.NE]

[5] Montufar G., Pascanu R., Cho K. and Bengio Y., On the Number of Linear Regions of Deep Neural Networks, thttps://papers.nips.cc/paper/5422-on-the-number-of-linear-regions-of-deep-neural-networks.pdf , last accessed 10th Jan. 2018

[6] Fortuner B., Can neural networks solve any problem?, <https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6>, last accessed 10th Jan. 2018

1. You can check a more extensive explanation on how numpy uses broadcasting on the official documentation https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html [↑](#footnote-ref-1)
2. A contour line of a function is a curve along which the function has a constant value. [↑](#footnote-ref-2)
3. https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html [↑](#footnote-ref-3)
4. A discussion about the meaning of cross-entropy is beyond the scope of this book. A nice introduction can be found here <https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/> and in many introductory machine learning books. [↑](#footnote-ref-4)