Training Neural Networks

# Introduction

Building complex networks with tensorflow is quite easy, as you have probably realized by now. A few lines of code are enough to construct networks with thousands (and even more) parameters. It should be clear by now that the problems come while training such networks. Is difficult, unstable, slow and is very difficult to test hyperparameters since a run over few hundred of epochs may take hours. This is not only a performance problem, otherwise it would be enough to use faster and faster hardware. The problem is that very often the convergence process (the learning) does not work at all. It stops, it diverges, or it never gets close to the minimum of the cost function. We need ways of making the training process efficient, fast and reliable. We will look at two of the main strategies that will help with the training of complex networks: dynamical learning rate decay and optimizers that are smarter than plain gradient descent (like RMSProp, Momentum and Adam).

# Dynamical learning rate decay

## The problem

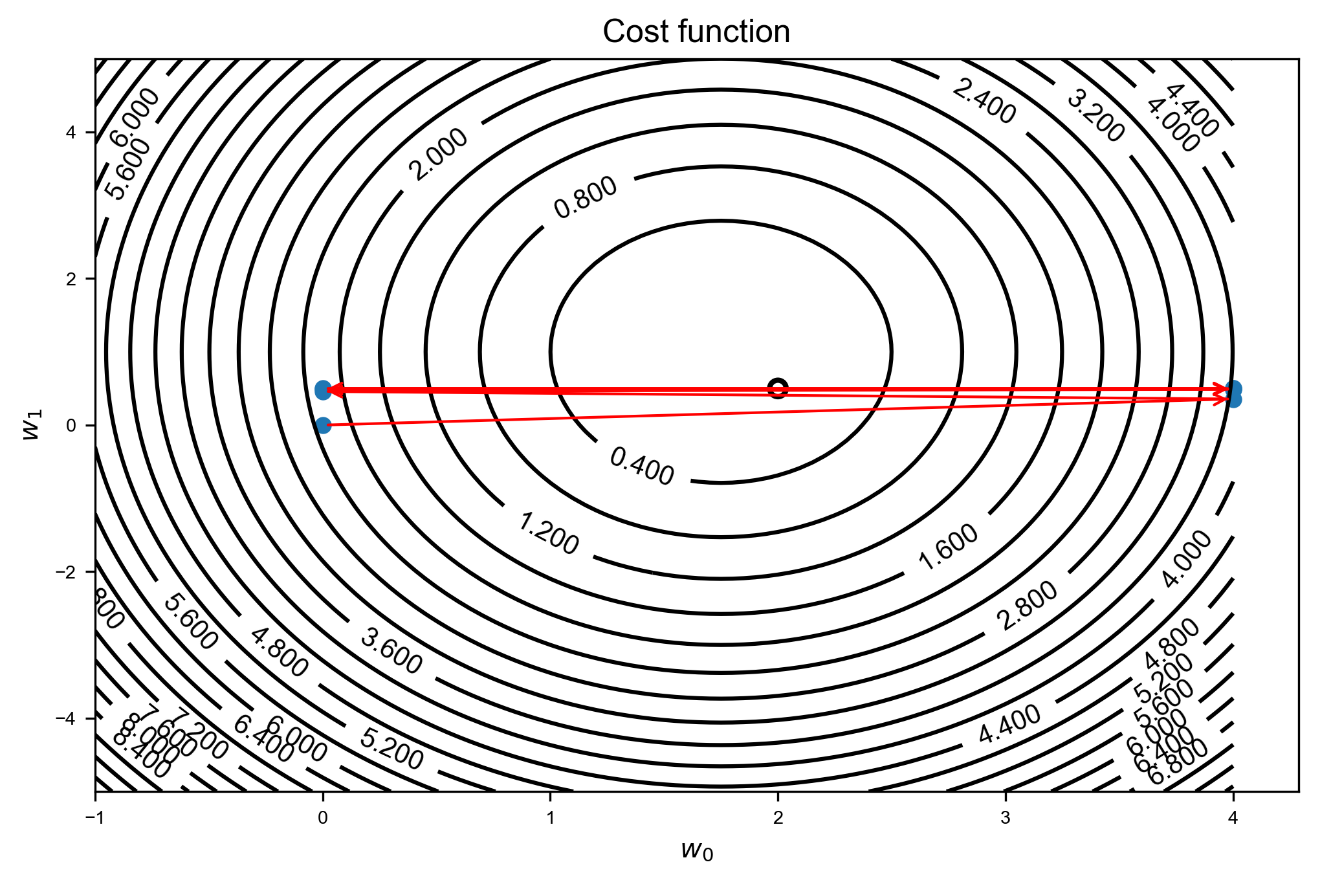
I mentioned several times that the learning rate is a very important paramer and that choosing it badly will make your model do not work. Check again Figure 2-12 where I showed you how choosing a learning rate that is too big will make your gradient descent algorithm bounce around the minimum and do not converge. Without discussing them let's rewrite the equations that describe the weight and bias update we described in Chapter 2 when discussing the gradient descent algorithm (remember we described the algorithm for a problem with two weights and ):

as a reminder here is an overview of the notation (please check again Chapter 2 if you don't remember how the gradient descent is working):

* - Weight 0 at iteration
* - Weight 1 at iteration
* - Cost function at iteration
* - learning rate

To show the effect of the what we will discuss, we will consider the same problem we have described in the Section " The learning rate in a practical example" of Chapter 2. Plotting the weights on the contour lines of the cost function for (see Figure 4-1) shows (as you will remember from Chapter 2) how the values of weights oscillates around the minimum of .

Figure 4-1: 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.



But you may have noticed that in our algorithm we took a pretty important decision (without explicitly saying that): **we keep the learning rate constant for each iteration**. But there is no reason for doing that, on the contrary is a quite bad idea. Intuitively a big learning rate will make the convergence moves fast at the beginning, but as soon as you are around the minimum you want to use a much smaller learning rate, to allow the algorithm to converge in the most efficient way toward the minimum. We want to have a learning rate that starts (relatively) big and then decreases with the iterations. But how should it decrease? There are several methods that are used today and in the next section we will look at the most used and how to implement them in Python and tensorflow. We will use again the same problem that we have used to generate Figure 4-1 and 2-12 and compare the behavior of the different algorithms. Please take some time and review the section in Chapter 2 on the gradient descent to have the material clear in your head before reading the next sections.

## Iterations or epochs?

Before looking at the various methods I would like to shed some light on the question: what are the iterations we are talking about? Are maybe the epochs? Technically this is not the case. An iteration is when you update your weights. Consider for example mini-batch gradient descent. In that case an iteration occurs after each mini-batch (when you update the weights). Consider the Zalando dataset of Chapter 3: 60000 training cases and a mini-batch size of 50. In this case you would have 1200 iterations in an epoch. What is important for the decrease of the learning rate is the number of updates you perform on the weights, not the number of epochs. If you use SGD on the Zalando dataset (update the weights after each observation) you would have 60000 iterations and you may need to decrease the learning more than with mini-batch gradient descent since is updated more often. In the case of batch gradient descent, where you update your weight after one complete sweep over the training data, you would update the learning rate exactly once each epoch.

When talking about iterations in dynamical learning rate decay, we intend the step in the algorithm when the weights are updated. For example, if you use SGD on the Zalando dataset in Chapter 3 with a mini-batch size of 50, in one epoch (sweep over the 60000 training observations) you have 1200 iterations.

This is very important to understand correctly. If you do, you can choose the parameters of the different algorithm for learning rate decay properly. If you choose them thinking that the learning rate is updated only after each epoch, you may make big mistakes.

Note that each algorithm that decrease dynamically the learning rate will introduce new hyperparameters that you need to optimize, adding some complexity to your model selection process.

## Staircase decay

The staircase decay method is the most basic one to use. It consists in reducing the learning rate manually in the code, hard coding the changes based on what seems to work. For example, how can we make the GD algorithm converge in Figure 4-1 starting with a ? Let's consider the following decay (where we have indicated with the iteration number)

Simply including this with the Python code

gamma0 = 2.0

if (j < 4):

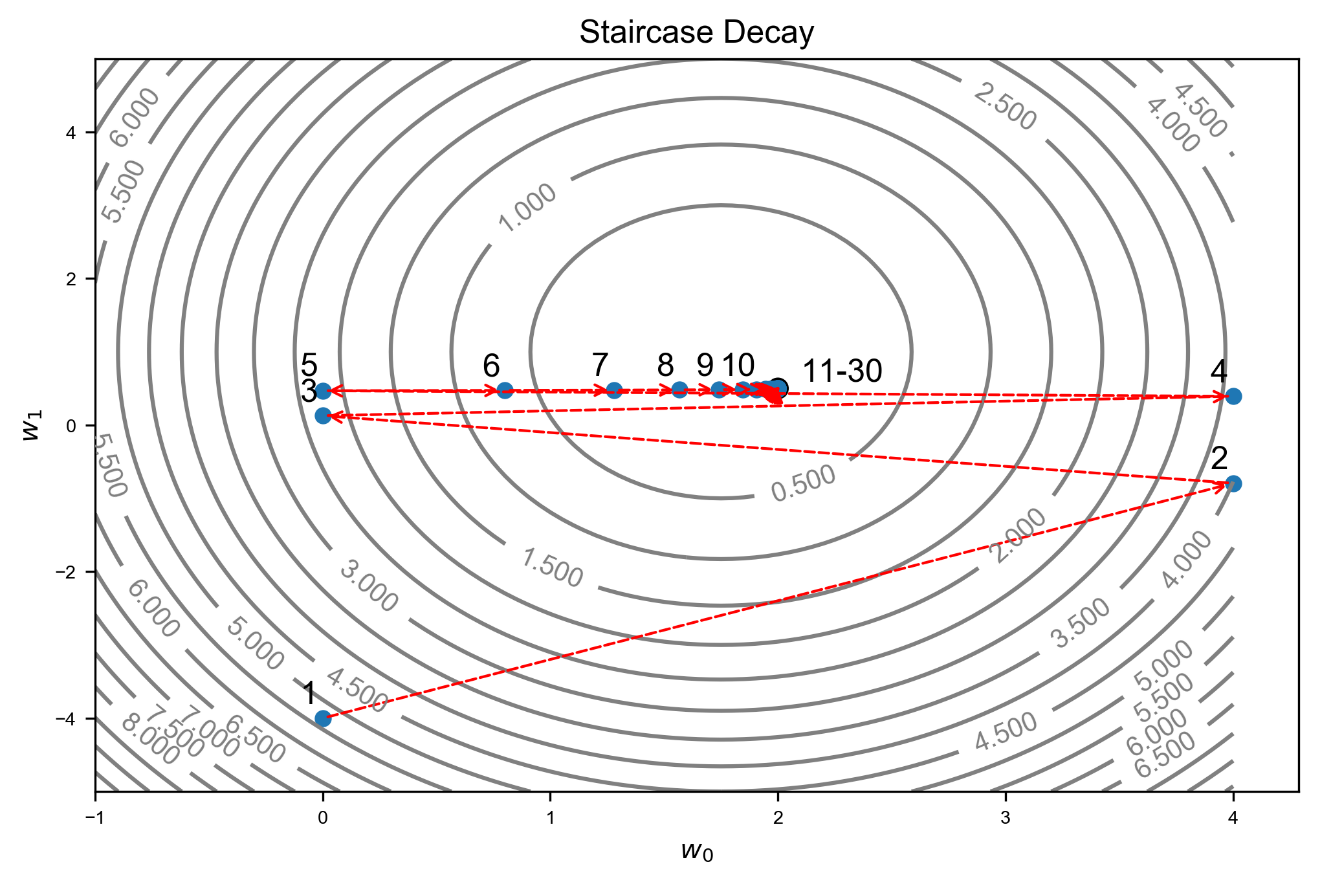
gamma = gamma0

elif j>=4:

gamma = gamma0 /5.0

will give a converging algorithm (see Figure 4-2).

Figure 4-2: Illustration of gradient descent algorithm. Here the initial learning rate of has been chosen and from iteration 4, has been used. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



The first steps are big and then, as we decrease the learning rate to 0.4 at iteration 4 they become smaller and the GD is able to converge toward the minimum. With this simple modification we have achieved already a nice result. The problem is that when dealing with complicated datasets and models (like we did in Chapter 3) this process requires (if it works) a lot of tests. You will need to reduce the learning rate several times and finding the right iteration and values for the learning rate decrease is a really challenging task, so much that it is actually not usable, unless you are dealing with very easy datasets and networks. The method is also not very stable and depending on the data you have may require continuous tuning. TL;DR[[1]](#footnote-1): don't use it.

Additional hyperparameters introduced

|  |  |
| --- | --- |
| Hyper parameter | Example |
| The iteration at which the algorithm will update the learning rate | In this example we choose iteration number 4 |
| The values of the learning rate after each change (multiple values) | In this example we had from iteration 1 to 3 and from iteration 4 |

## Step decay

Something slightly more automatic is the so-called step decay. This method reduces the learning rate by a constant factor every certain number of iterations. Mathematically it can be written as

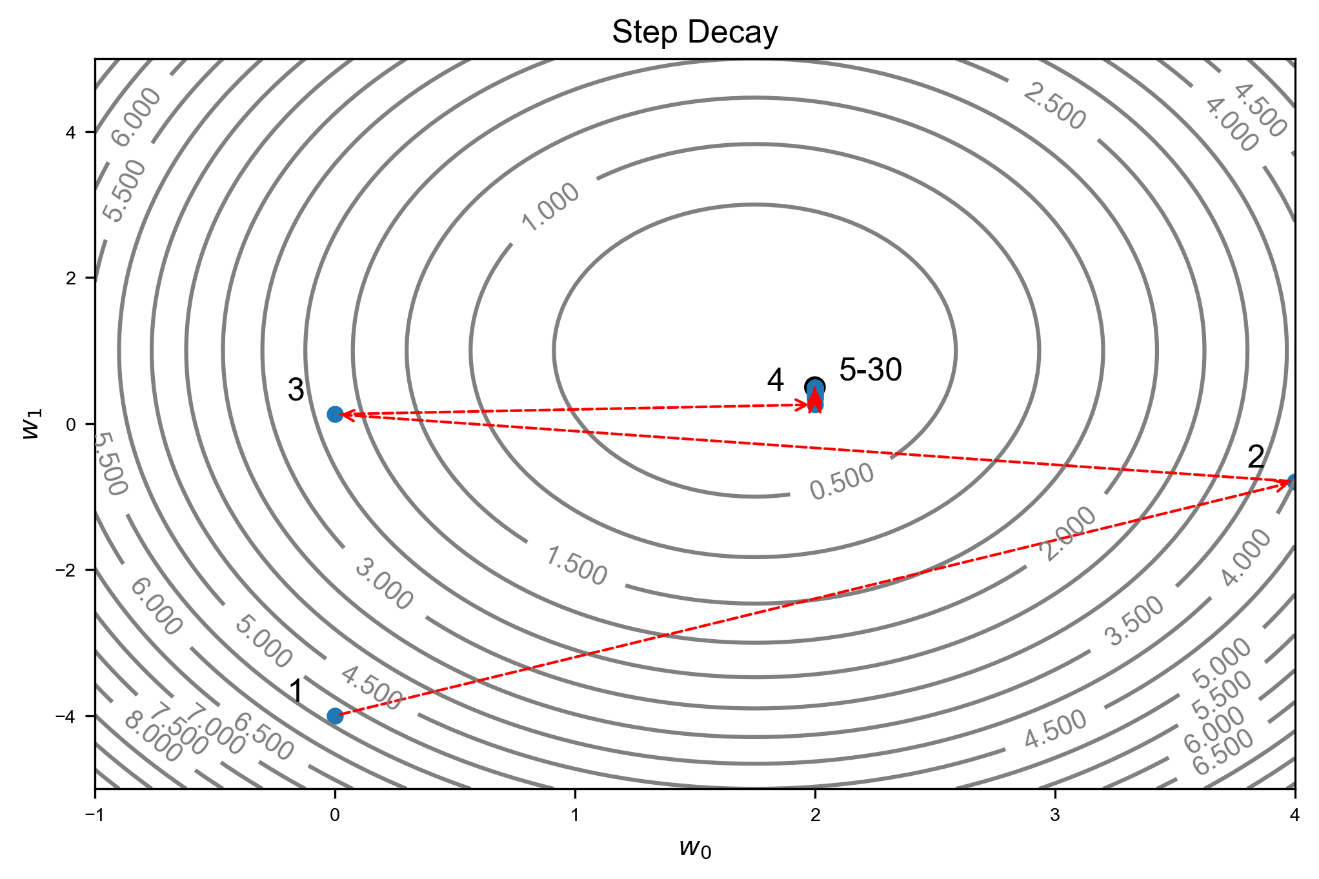
where indicates the integer part of , and (indicated in the code later with epoch\_drop) is an integer constant that we can tune. For example, using the following code

epochs\_drop = 2

gamma = gamma0 / (np.floor(j/epochs\_drop)+1)

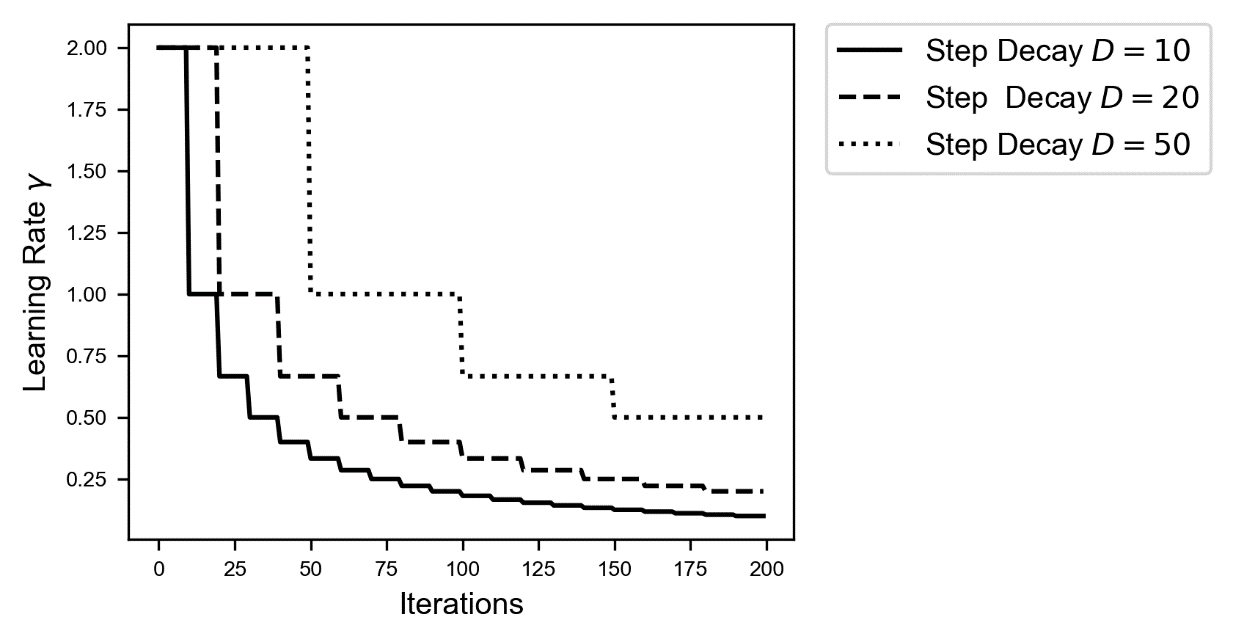
will again give a convergent algorithm (see Figure 4-3).

Figure 4-3: Illustration of gradient descent algorithm. Here the initial learning rate of has been chosen and and every 2 iterations the learning rate has decreased according to . The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



It is important to have an idea of how fast the learning rate is decreasing. You don't want to have a learning rate close to zero after only a few iterations, otherwise your convergence will never succeed. In Figure 4-4 you can see a comparison of how fast (or slow) the learning rate is decreasing for 3 values for .

Figure 4-4: the learning decreasing with the step decay algorithm for three values of 10,20 and 50.



Note for example how with the learning rate is roughly 10 times smaller after only 100 iterations! If you make your learning rate decrease too fast, you may see your convergence grind to a halt after only a few iterations. Always try to get an idea of how fast your is decreasing.

A good way of getting a feeling of how fast your learning rate is decreasing is trying to determine after how many iterations is ten times smaller than the initial value. Keep in mind that if you get a after iterations, you will get after only iterations, and after only iterations and so on. If it is what is needed can be answered only by testing it properly with several values of .

Let's make a concrete example. Suppose you are training your model with 1e5 observations for 5000 epochs, with mini-batches sizes of 50 with a starting learning rate of . If you choose without thinking a you will have

after only 100 epochs, so you will not gain much by using 5000 epochs if you reduce the learning rate so quickly.

Additional hyperparameters introduced

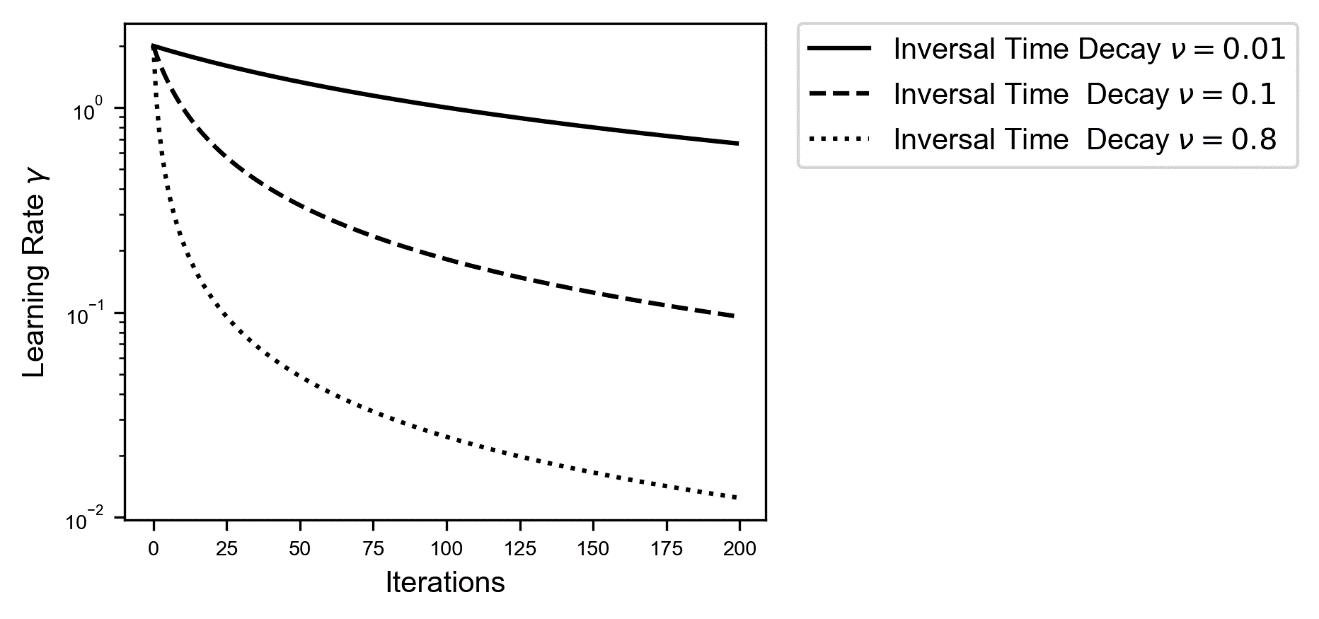
|  |  |
| --- | --- |
| Hyper parameter | Example |
| Parameter D |  |

## Inverse time decay

Another way of updating the learning rate is with the formula called inverse time decay

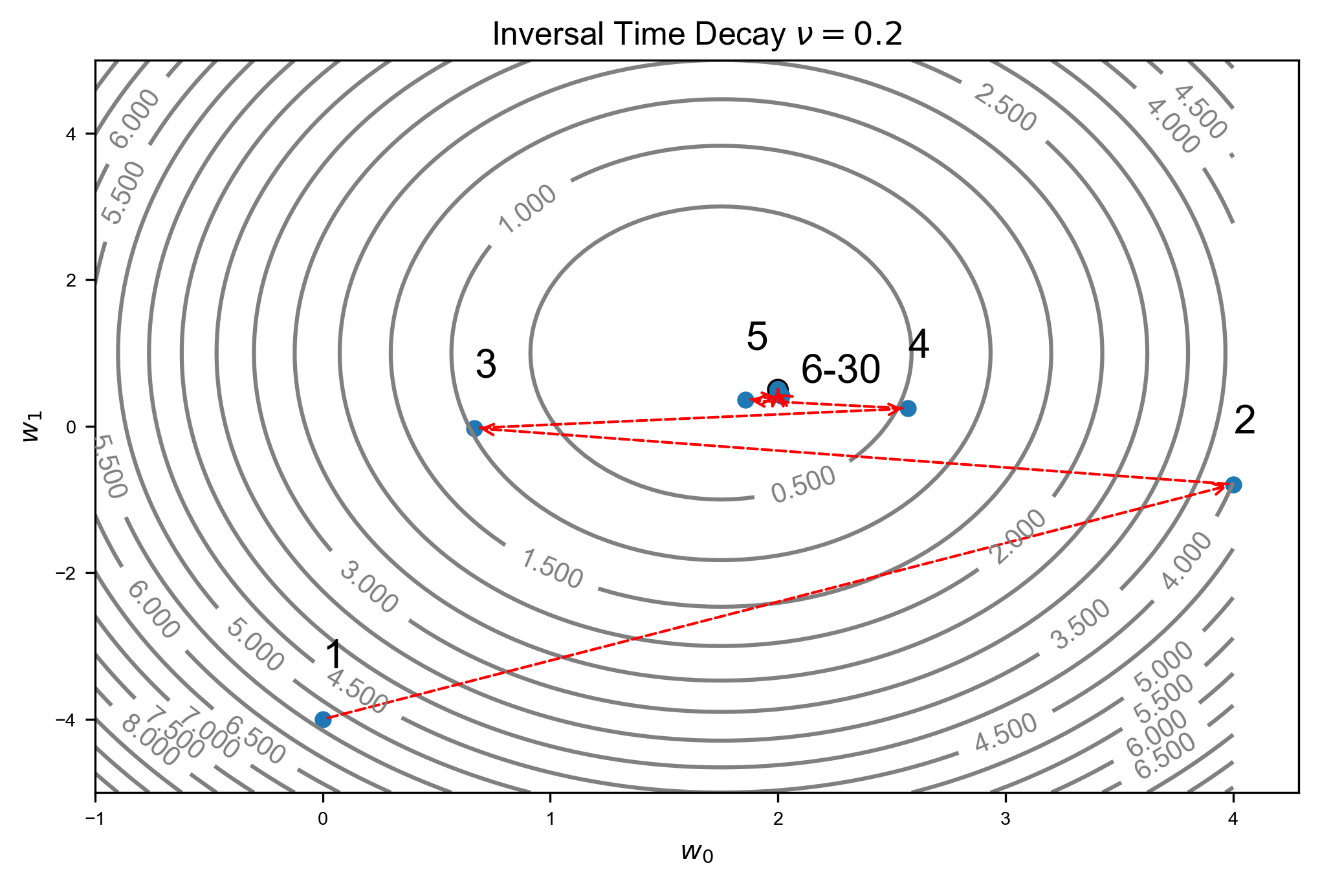
where is a parameter called the decay rate. In Figure 4-5 you can see a comparison of the learning rate decrease for three parameters of : 0.01, 0.1 and 0.8.

Figure 4-5: the learning decreasing with the inverse time decay algorithm for three values of 0.01,0.1 and 0.8. Note that the y-axis has been plotted in logaritmic scale to make the entity of the changes easier to compare. Note that the -axis is logarithmic.



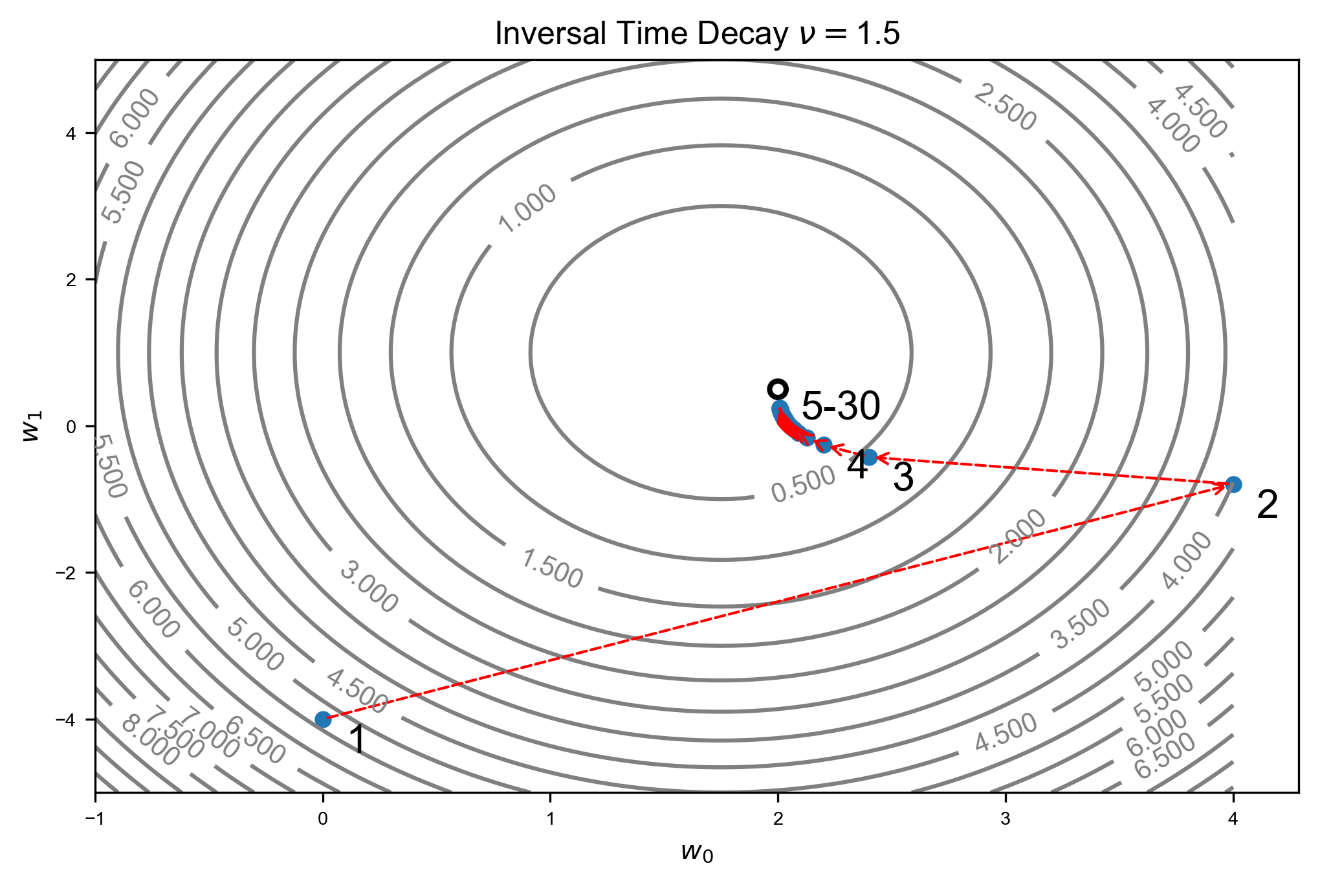
This method makes also our GD algorithm we discussed in Chapter 2 converge. In Figure 4-6 you can see how the weights converge toward the minimum location after only a few iterations when choosing .

Figure 4-6: Illustration of gradient descent algorithm. Here the initial learning rate of has been chosen and and inversal time decay algorithm with has been used. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



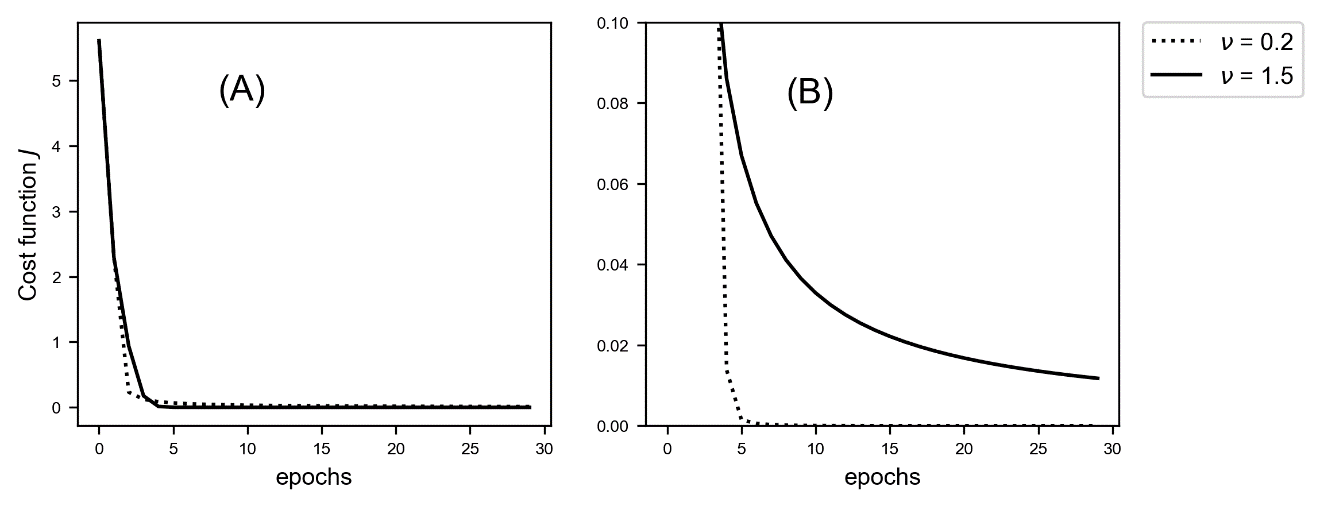
Actually, is very interesting to see what happens if we choose a bigger value for . Check Figure 4-7 where we have chosen .

Figure 4-7 Illustration of gradient descent algorithm. Here the initial learning rate of has been chosen and and inversal time decay algorithm with has been used. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



What we observe in Figure 4-7 makes perfect sense. Increasing makes the learning rate decreases faster and therefore more steps are needed to reach the minimum, since the learning rate is increasingly smaller in comparison with what happens in Figure 4-6. We can compare the behavior of the cost functions for the two values of In Figure 4-8 you can see in plot (A) the cost function vs the number of epochs. At first sight the two seems to converge exactly as fast. But let's zoom around in plot (B). You can clearly see how with the convergence is much faster, since the learning rate is bigger than for .

Figure 4-8: Cost function vs. the number of epochs. In plot (A) the entire range of values that the cost functions assume are plotted. In plot (B) the area around has been zoomed in to show how the cost functions decreases much faster for smaller values of .



Additional hyperparameters introduced

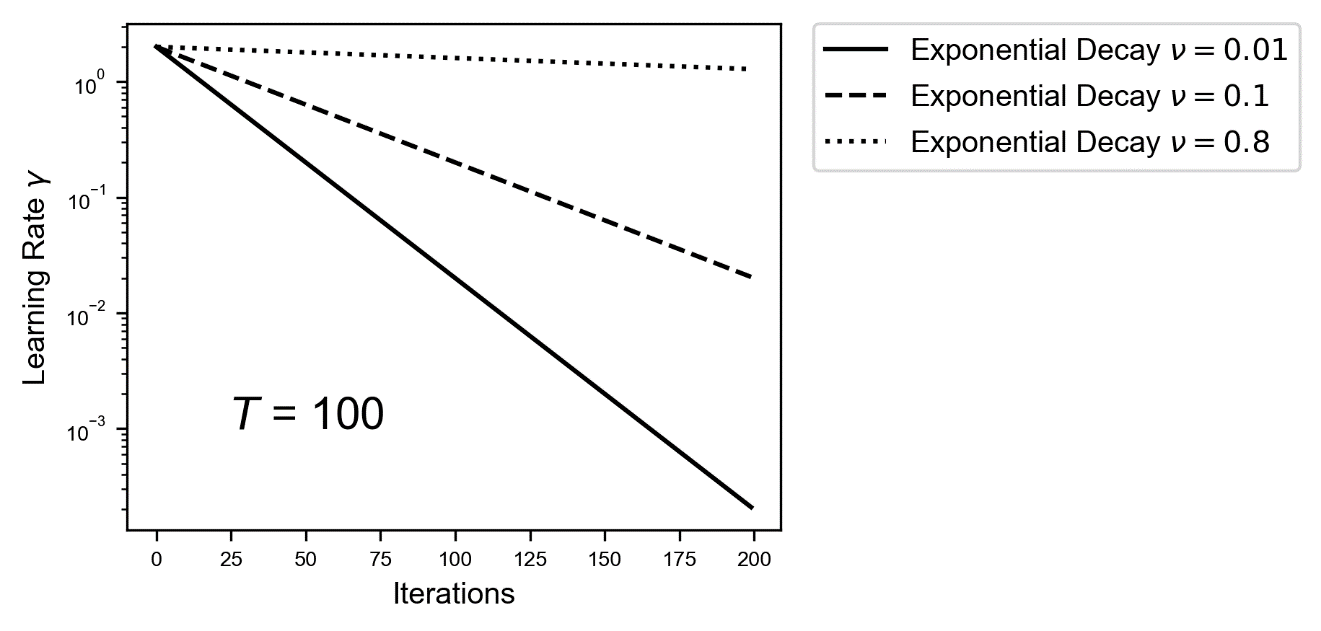
|  |  |
| --- | --- |
| Hyper parameter | Example |
| Decay rate |  |

## Exponential decay

Another way of reducing the learning rate is according to the formula called exponential decay

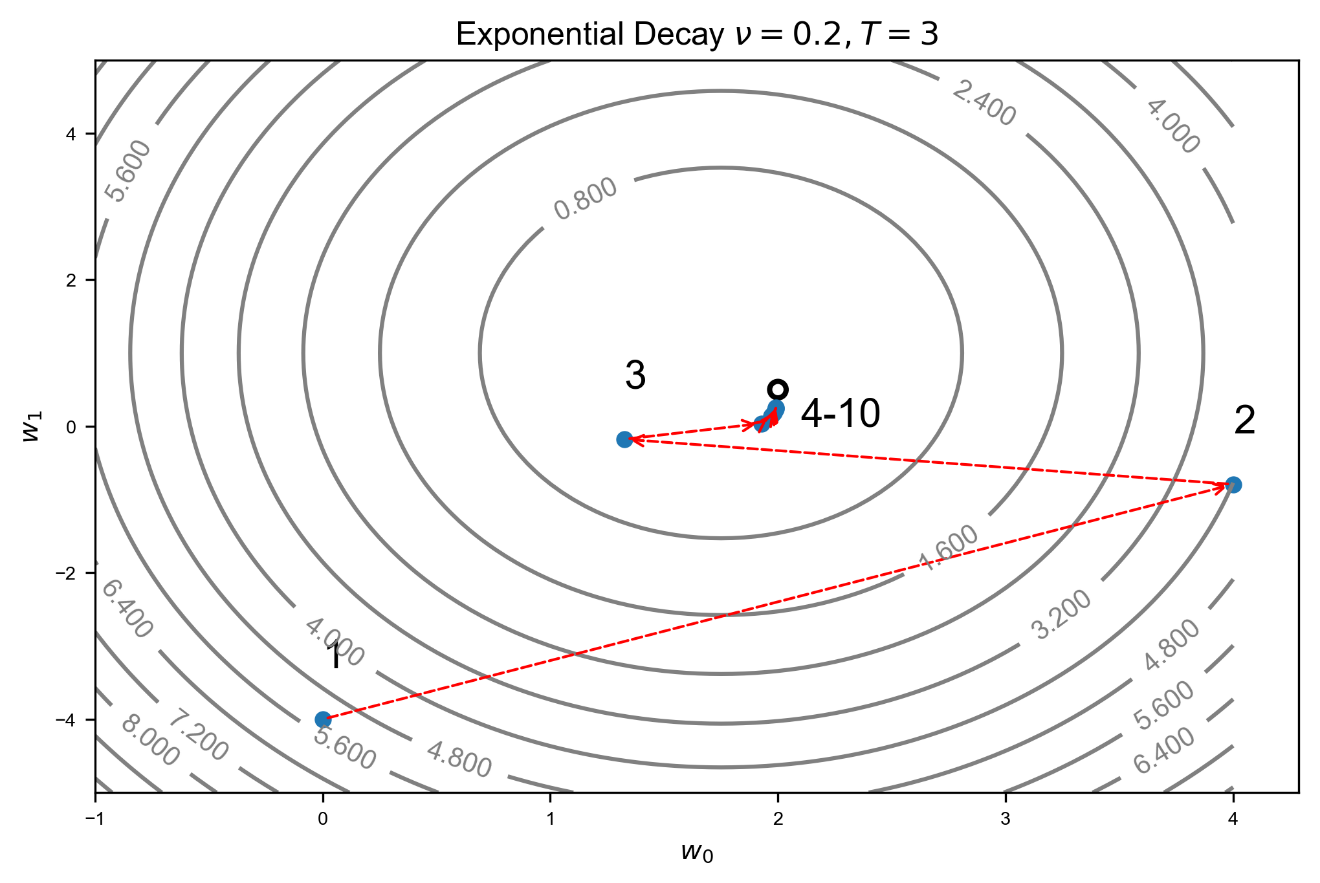
Check Figure 4-8 to get an idea of the speed with which the learning rate decreases (note that the y-axis is in logarithmic scale).

Figure 4-9: the learning decreasing with the exponential decay algorithm for three values of 0.01,0.1 and 0.8 and . Note that the y-axis has been plotted in logaritmic scale to make the entity of the changes easier to compare. Note how for after 200 iterations (not epochs) the learning rate is already a factor 1000 smaller than the starting one! The y-axis is logarithmic.



We can apply this method to our problem (see Figure 4-9) with and and again the algorithm converges.

Figure 4-9 Illustration of gradient descent algorithm. Here the initial learning rate of has been chosen and and exponential decay algorithm with and has been used. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



Additional hyperparameters introduced

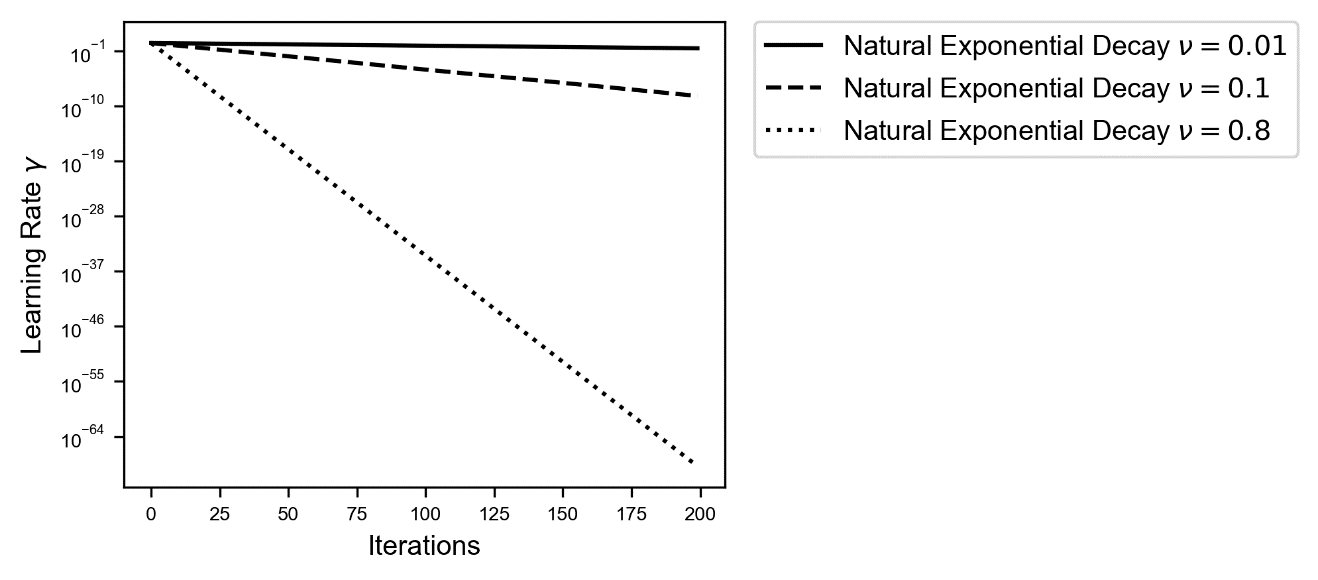
|  |  |
| --- | --- |
| Hyper parameter | Example |
| Decay rate |  |
| Decay steps |  |

## Natural exponential decay

Another way of reducing the learning rate is according to the formula called natural exponential decay

This case is particularly interesting because it allows us to learn a few important things. Consider first Figure 4-10 to compare how difference values of relates to different decrease speed for the learning rate.

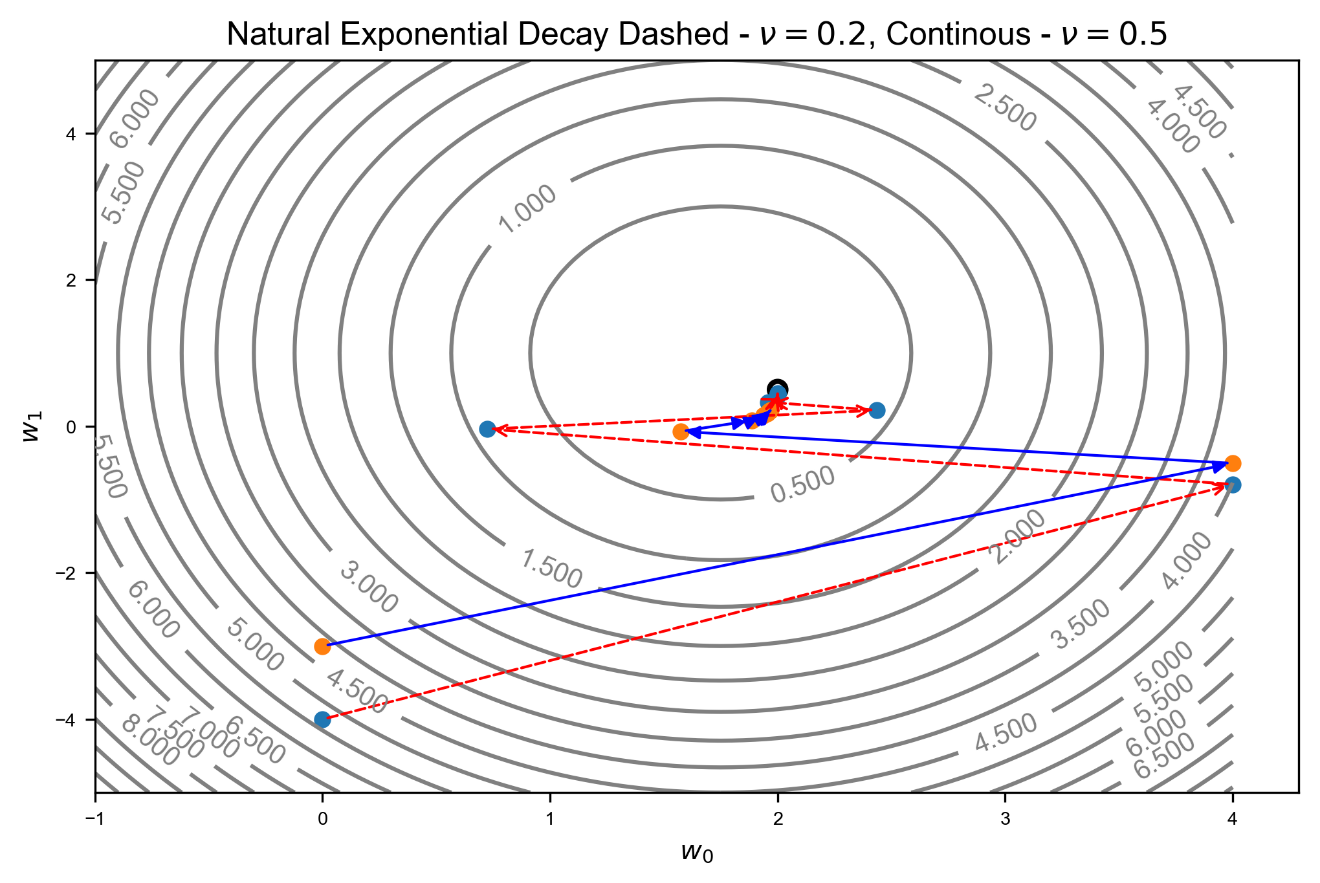
Figure 4-10: the learning decreasing with the natural exponential decay algorithm for three values of 0.01,0.1 and 0.8 and . Note that the y-axis has been plotted in logaritmic scale to make the entity of the changes easier to compare. Note how for after 200 iterations (not epochs) the learning rate is already a factor smaller than the starting one! The y-axis is logarithmic.



I would like to draw your attention to the values on the y-axis (note that is using a logarithmic scale). For after 200 iterations the learning rate is a factor of of the initial value! Practically zero. That means that already after a few iterations no more updates can happen, since the learning rate is too small! To give you an idea of the scale of : a hydrogen atom is "only" roughly ! So, unless you are very careful with the choice of you will not get very far.

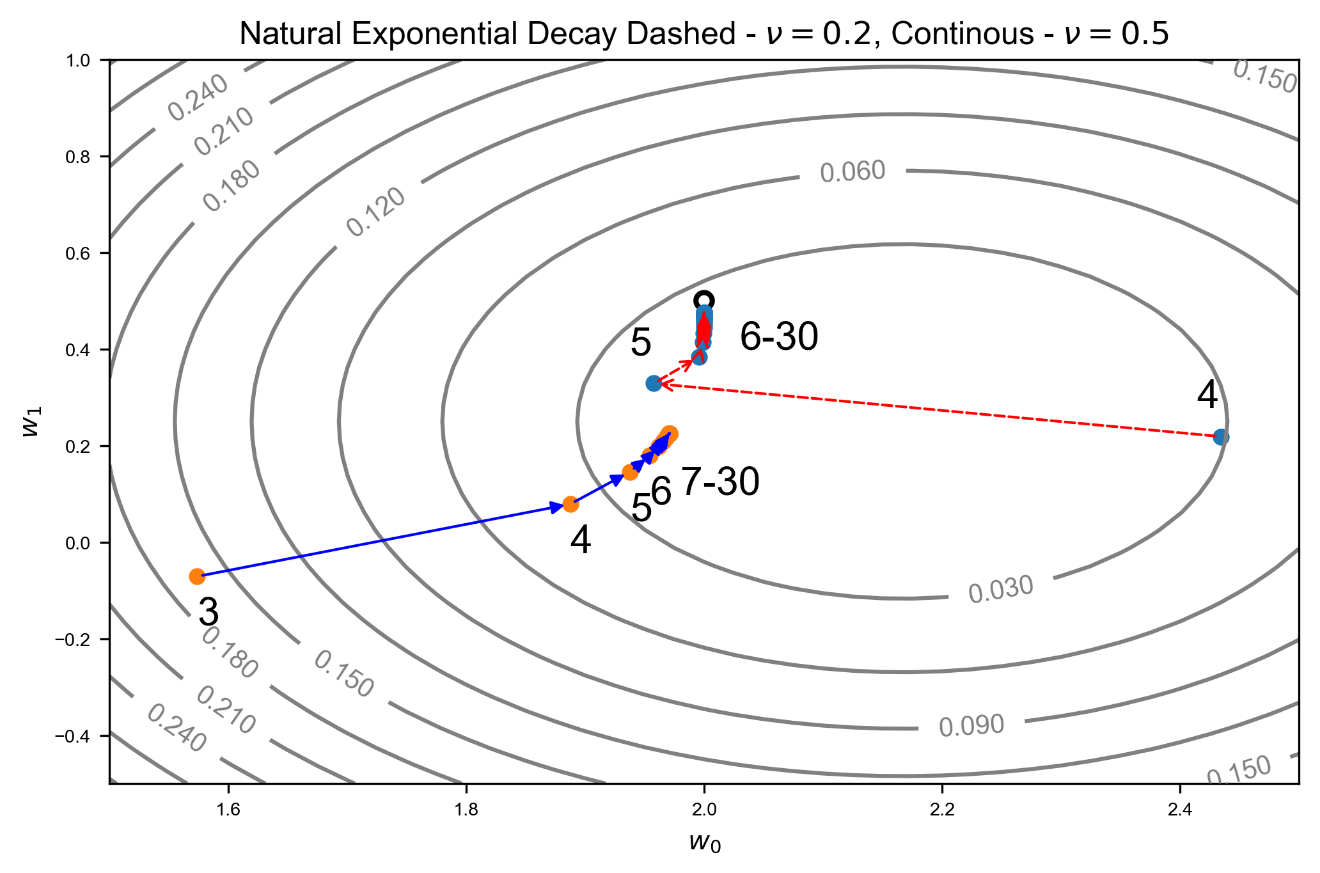
Consider Figure 4-11, where we have plotted our weights as they are updated with the GD algorithm for two values of the learning rate: 0.2 (dotted) and 0.5 (continuous).

Figure 4-11 Illustration of gradient descent algorithm. Here the initial learning rate of has been chosen and and exponential decay algorithm with (continous) and (dashed) has been used. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



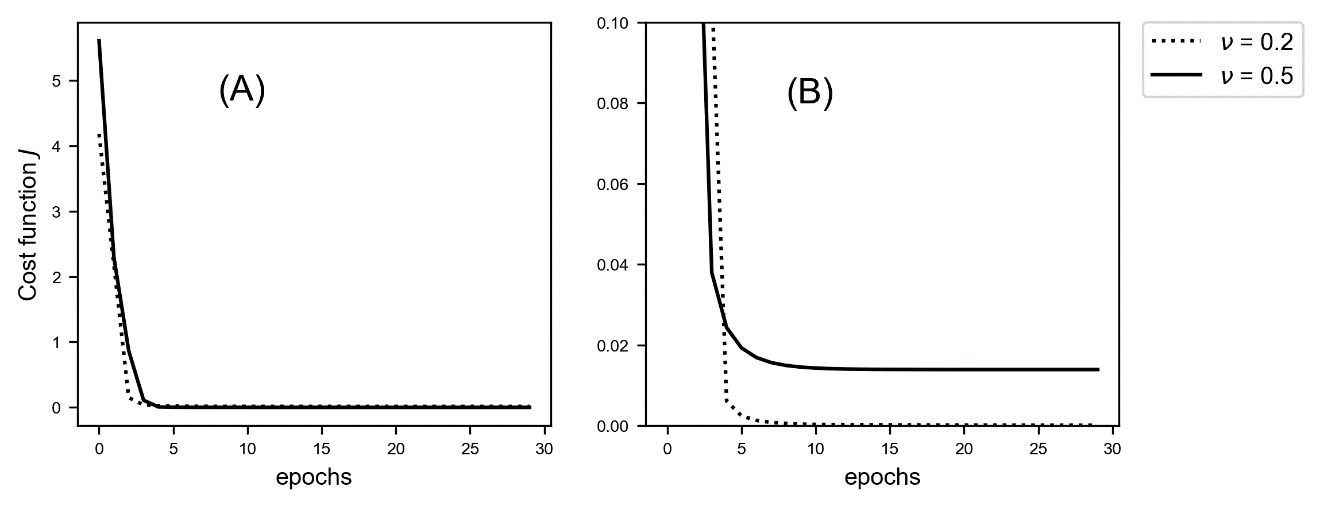
To check the convergence, we need to zoom in around the minimum. Let's do that in Figure 4-12. In case you are wondering why the minimum seems to be in a different position relative to the contours lines as in Figure 4-11, this is because the contour lines are not the same since in Figure 4-12 we are much closer to the minimum.

Figure 4-12 Illustration of gradient descent algorithm zoomed around the minimum. Here the initial learning rate of has been chosen and and exponential decay algorithm with (continous) and (dashed) has been used. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



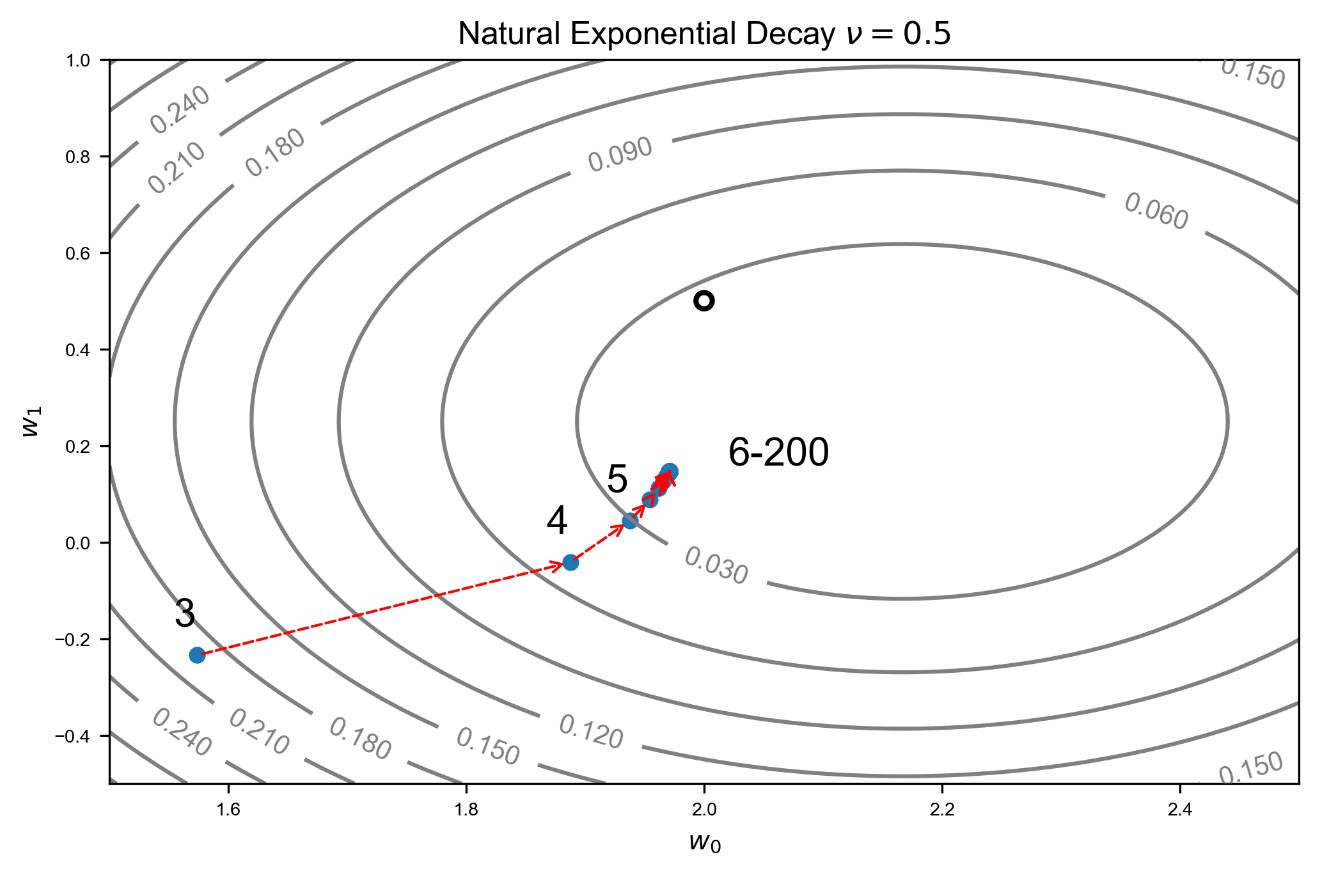
Now we see something that makes again sense. The continuous line is for , therefore the learning rate decreases much faster and does not manage to reach the minimum. In fact, after only 7 iterations we have , and after 20 iterations we have , a value so small that the convergence does not manage anymore to proceed at a reasonable speed! Is again very instructive to check the cost function decrease for the two parameters (see Figure 4-13).

Figure 4-13: Cost function vs. the number of epochs for natural exponential decay for two values of and 0.5. In plot (A) the entire range of values that the cost functions assume are plotted. In plot (B) the area around has been zoomed in to show how the cost functions decreases much faster for smaller values of .



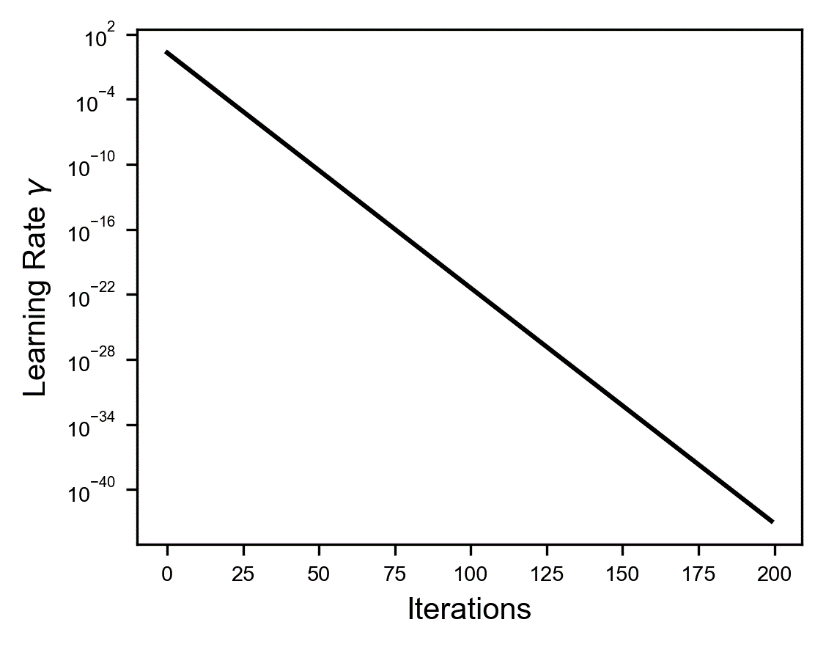
we notice in plot (B) how the cost function for does not reach zero and becomes practically constant, since the learning rate is too small. You may think that using more iterations the method will eventually converge, but that is not the case. Check Figure 4-14 to see that the convergence process actually stops due to the learning rate being almost zero after a while.

Figure 4-14 Illustration of gradient descent algorithm zoomed around the minimum for 200 iterations. Here the initial learning rate of has been chosen and and exponential decay algorithm with has been used. The GD does not manage to reach the minimum. The different estimates are indicated with points. The minimum is indicated by the circle approximately in the middle of the image. The algorithm is now able to converge. Each point has been labelled with the iteration number to make following the weights update easier.



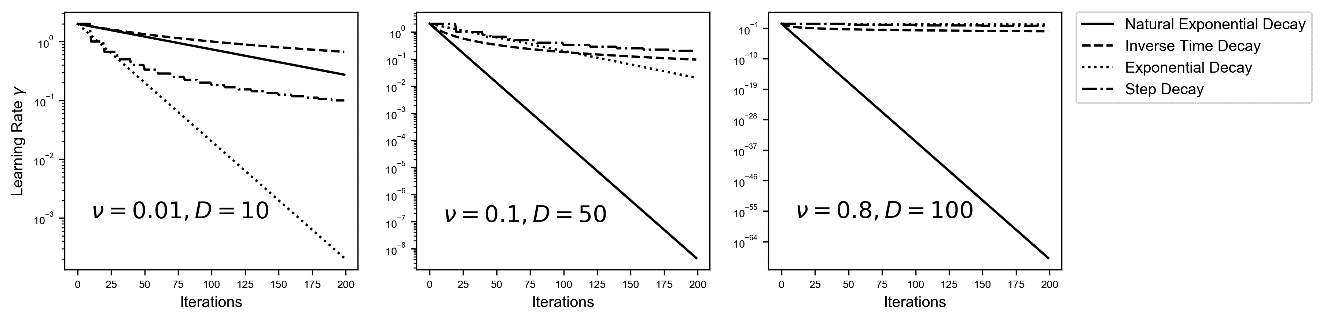
Let's check the learning rate during this process for (see Figure 4-15). Check the values on the y-axis. The learning rate reaches after roughly 175 iterations. For all practical purposes is zero. The GD algorithm will not update the weights anymore, regardless of how many iterations you let it run.

Figure 4-15: learning rate vs. the number of iterations with natural exponential decay for . Note that the y-axis is in logaritmic scale to better highlight the change of .



To finish let's compare the methods putting them on the same plot, to get an idea of the relative behavior. In Figure 4-16 you can see three plots where we plot the learning rate decay for each method with different parameters.

Figure 4-16: Comparison of the learning rate decay for the algorithm described for different parameters.



You should be aware of how fast your learning rate is decreasing to avoid it becoming practically zero and stopping your convergence altogether.

## Tensorflow implementation

We should briefly talk on how tensorflow implement the methods I just explained you, since there are a few details that you should know. In tensorflow you can find the following functions to perform dynamical learning rate decay[[2]](#footnote-2):

* Exponential decay 🡪 tf.train.exponential\_decay (<https://goo.gl/fiE2ML>)
* Inverse time decay 🡪 tf.train.inverse\_time\_decay (<https://goo.gl/GXK6MX>)
* Natural exponential decay 🡪 tf.train.natural\_exp\_decay (<https://goo.gl/cGJe52>)
* Step decay 🡪 tf.train.piecewise\_constant (<https://goo.gl/bL47ZD>)
* Polynomial decay 🡪 tf.train.polynomial\_decay (https://goo.gl/zuJWNo)

Polynomial decay is a slightly more complex way of decreasing the learning rate that we have not discussed, since is rarely used, but you can read the documentation on the tensorflow website to get an idea on how is working.

Tensorflow uses an additional parameter to give you a bit more flexibility. Take for example the inverse time decay method. Our equation for the learning rate decay was

Where we have two parameters: and . Tensorflow uses three parameters:

where is called in tensorflow code decay\_step. The formula you will find on the tensorflow official documentation in Python code is

decayed\_learning\_rate = learning\_rate / (1 + decay\_rate \* global\_step / decay\_step)

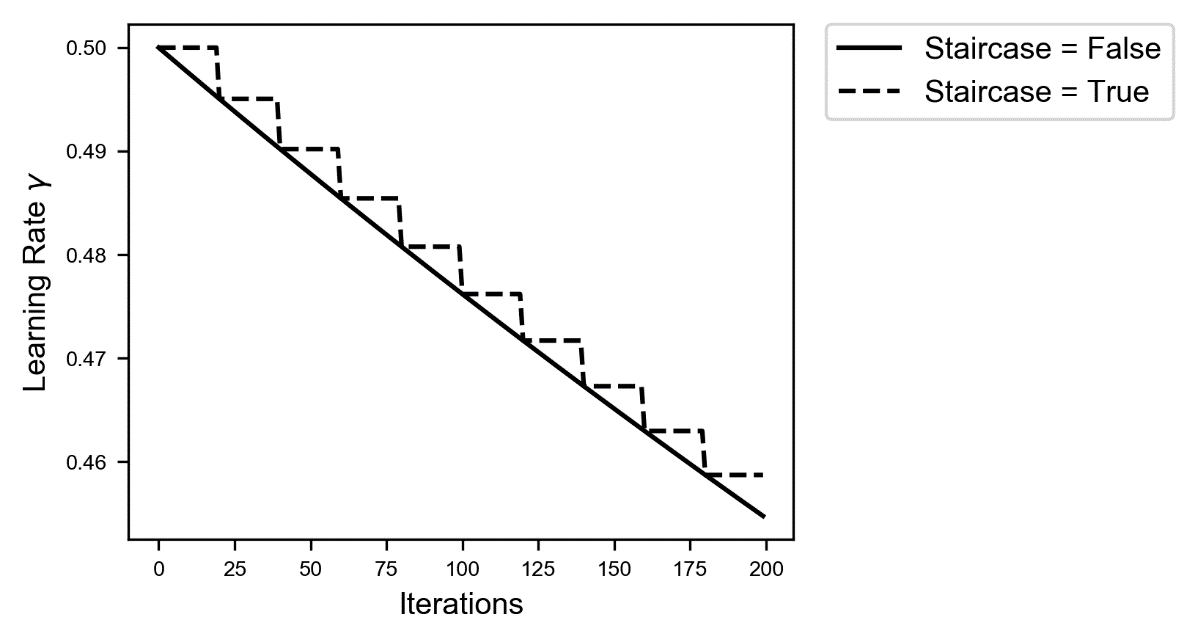
to link tensorflow language with our notation:

* global\_step 🡪 (number of iterations)
* decay\_rate 🡪
* decay\_step 🡪
* learning\_rate 🡪 (initial learning rate)

You may ask yourself why you want to have this additional parameter. The parameter, mathematically speaking, is redundant: we can simply set our to the same value of and we would get the same result. The problem, practically, is that (the number of iterations) gets very big very quickly and therefore our may need to assume very small values to be able to get a reasonable learning rate decrease. The goal of the parameter is to scale the number of iterations. For example, you can set this parameter to and therefore making the decrease of the learning rate happen on a scale of iterations instead of every single iteration. If you have a huge dataset with observation and you use a mini-batch size of 50 you will get each epoch iterations. Suppose you now want your learning rate to be of the initial value after 100 epochs. Then you would need a , a rather small value that more importantly is depending on the size of your dataset and the mini-batch size. If you "normalize", so to speak, the number of iterations you can choose a value for that can remain constant if you choose to change, for example, the mini-batch size. There is an additional practical reason (more important that what we just discussed) that is the following: the tensorflow function has an additional parameter: staircase that can assume the values of True or False. If set to True the following function is used

And therefore, you get an update only each iterations instead of continuously. In Figure 4-17 you can see the difference for and for 200 iterations. You may want to keep you learning rate constant for 10 epochs before updating it.

Figure 4-17: learning rate decay with the two variations obtained in tensorflow with staircase = True and False.



The same parameters are needed by the functions: tf.train.inverse\_time\_decay, tf.train.natural\_exp\_decay and tf.train.polynomial\_decay. They work in the same way and the purpose of the additional parameter is what we just described. Don't be confused when implementing the methods in tensorflow if you need this additional parameter. I will show you how to implement it for inverse time decay, but it works in the same exact way for all the others. You need the following additional lines of code

initial\_learning\_rate = 0.1

decay\_steps = 1000

decay\_rate = 0.1

global\_step = tf.Variable(0, trainable = False)

learning\_rate\_decay = tf.train.inverse\_time\_decay(initial\_learning\_rate, global\_step, decay\_steps, decay\_rate)

and then you need to modify the line of code where you specify which optimizer you are using with this

optimizer = tf.train.GradientDescentOptimizer(learning\_rate\_decay).minimize(cost, global\_step = global\_step)

the only difference being the additional parameter in the minimze function: global\_step = global\_step. The minimize function will update the global\_step variable with the iteration number each update for us. That's all. It works in the same way for the other functions.

The only difference is for the function piecewise\_constant that require different paraemters: x, boundaries and values. For example (from the tensorflow documentation)

use a learning rate that's 1.0 for the first 100000 steps, 0.5 for steps 100001 to 110000, and 0.1 for any additional steps

that would need

boundaries = [100000, 110000]

values = [1.0, 0.5, 0.1]

The following code

boundaries = [b1,b2,b3, …, bn]

values = [l1,l12,l23,l34,…, ln]

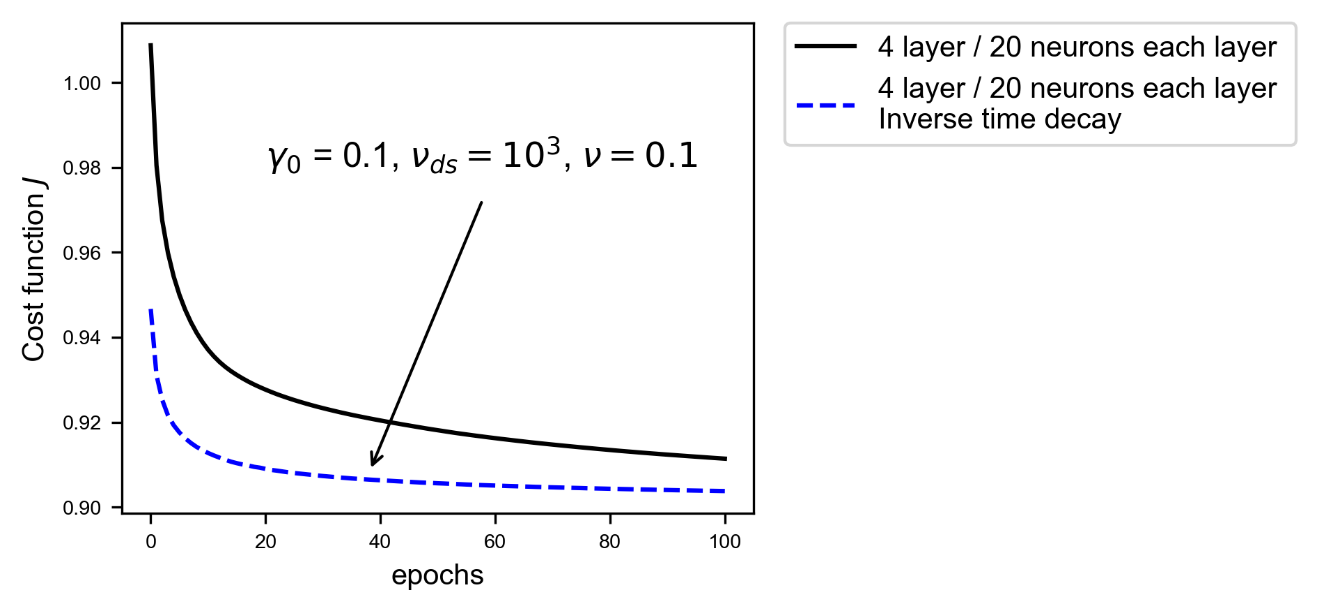
will give a learning rate of l1 before b1 iterations, l12 between b1 and b2 iterations, l23 between b2 and b3 iterations and so on. Keep in mind that with this method you must set manually all the values and boundaries in the code. This will require quite some patience if you want to test each combination to see if is working well or not. An implementation of step decay algorithm in tensorflow would look like this

global\_step = tf.Variable(0, trainable=False)  
boundaries = [100000, 110000]  
values = [1.0, 0.5, 0.1]  
learning\_rate = tf.train.piecewise\_constant(global\_step, boundaries, values)

## Applying it to the Zalando dataset

Let's try to apply the methods we just learned to a realistic scenario. For this we will use the Zalando dataset we used in Chapter 3. Please check again Chapter 3 to see how to load the dataset and how to prepare the data. At the end of the chapter we wrote the functions to construct a model with many layers and a function to train it. Let's consider a model with 4 hidden layers, each containing 20 neurons. Let's compare how the model learn with a starting initial learning rate of 0.01 and keep that constant, and then apply inverse time decay algorithm starting with a , and (see Figure 4-18).

Figure 4-18: cost function behaviour for a neural network with 4 layers each having 20 neurons applied to the Zalando dataset. Continous line is for a model with a constant learning rate of . Dashed for network where we have used inverse time decay algorithm with a , and .



So even with a starting learning rate ten times bigger the algorithm is much more efficient. Is has been proven in several research papers that applying dynamical learning rate makes the learning faster and more efficient as we have seen in this case.

Unless you are using optimization algorithms that include a learning rate change during training (we will see them in the next sections), is usually a good idea to use dynamical learning rate decay. This makes learning stable and usually faster. The downside is that you have more hyperparameters to tune.

Normally is a good idea, when using dynamical learning rate decay, to start with an initial learning rate bigger than you would normally use. Since is decreasing this won't normally create problems and will make the convergence at the beginning (hopefully) faster. As you should now expect there are not fix rules on which method works better. Each case and dataset are different, and some testing is always required to see which parameter value brings the best results.

# Common optimizers

Until now we have used gradient descent to minimize our cost function. That is not the most efficient way of doing that, and there are some modifications to the algorithm that can make it much more fast and efficient. This is a very active area of research and you will find an incredible number of algorithms based on different ideas to make the learning faster. I will look here at the most instructive and famous ones: Momentum, RMSProp and Adam. Additional material that you can read to investigate the most exotic ones has been written by S. Ruder in a paper called An overview of gradient descent optimization algorithms (<https://goo.gl/KgKVgG>). The paper is not for beginners and requires quite some mathematical background, but it gives an overview of more exotic algorithms as Adagrad, Adadelta and Nadam. Additionally it reviews weights update schemes applicable in distributed environments like Hogwild!, Downpour SGD and many more. Surely a read worth your time.

To understand the basic idea of Momentum (and partially also of RMSProp and Adam) you first need to understand what exponentially weighted averages are.

## Exponentially weighted averages

Let's suppose you are measuring a quantity (it could be the temperature where you live) over time, once a day for example. You will have a series of measurements that we can indicate with where goes from 1 to a certain number . Now bear with me if at the beginning does not make much sense, but let's define recursively a quantity as

and so on with a real number with . Generally, we could write the term as

Now let's write all the terms , and so on just as a function of and (so not recursively). For we have

for

Generalizing we obtain

Or in a more elegant way (without the three dots)

Now let's try to understand what this formula means. First of all, let's note that the term disappear if we choose Let's do that (we set ) and consider now what remains

Are you still with me? Now comes the interesting part. Let's define the convolution between two sequences[[3]](#footnote-3). Let's consider two sequences and . The convolution between the two (that we indicate with the symbol ) is defined by

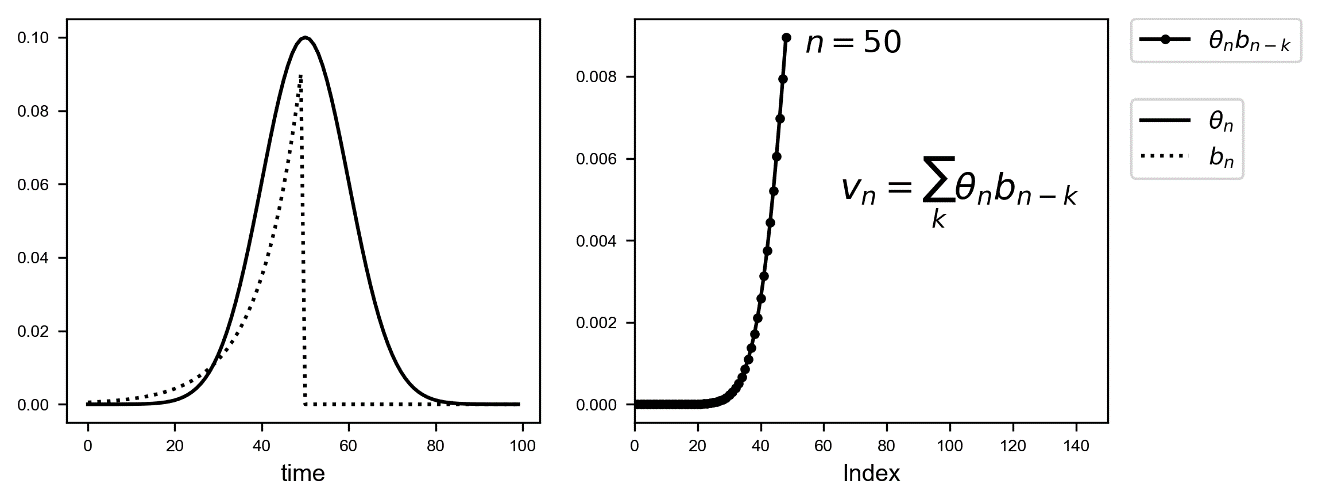
Now since we have only a finite number of measurements for our quantity , we will have

Therefore, we can write as a convolution as

Where we have defined

To get an idea of what that means let's plot together , and . To do that let's assume has a Gaussian shape (the exact form is not relevant, is just for illustrative purposes), and let's take (see Figure 4-19).

Figure 4-19: left side - a plot showing (solid line) and (dotted line) together. Right side - a plot showing the points that need to be summed to obtain for



Let's discuss briefly Figure 4-19. The Gaussian curve () will be convoluted with to obtain . The result can be seen in the right plot. All those terms for (plotted in the right plot) will be summed to obtain . Intuitively is the average of all for each term multiplied by a term () that is 1 for and then decreases rapidly for decreasing toward 1. Basically, this is a weighted average, with an exponentially decreasing weight (the name comes from here). The terms farther from are less and less relevant while the terms close to gets more weight. This is also a moving average. For each , all the preceding terms are added each multiplied by a weight ().

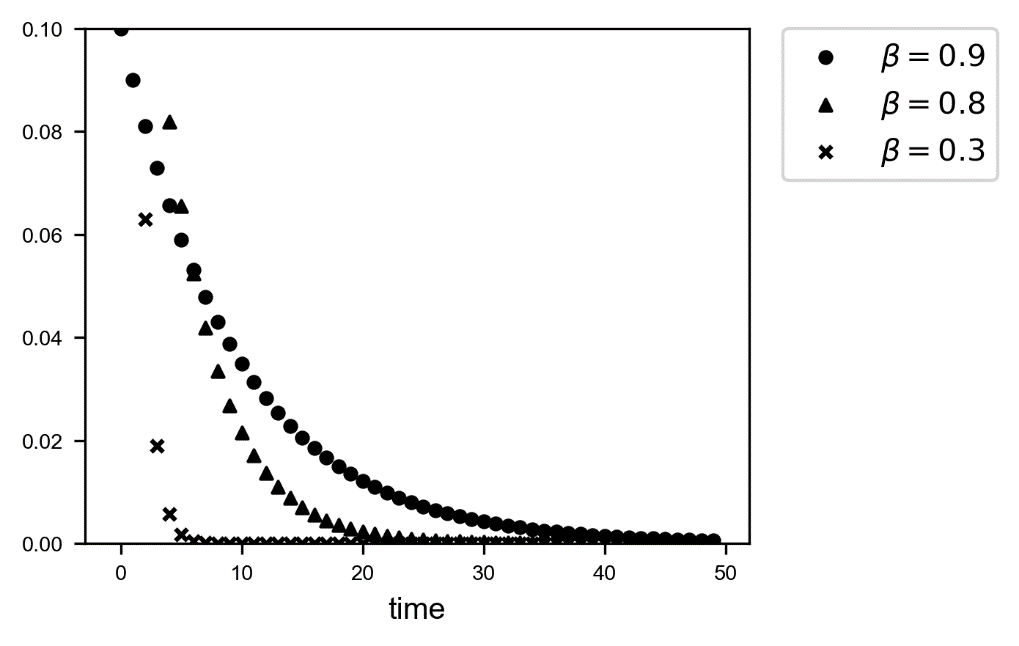
I would like now to show you why there is this factor in . Why not choose only ? The reason is very simple. The sum of over all positive is equal to 1. Let's see why.

where we have used the fact that for we have and that for a geometric series we have

The algorithm we described to calculate is nothing else than the convolution of our quantity with a series that have sum equal to one and has the form .

The exponentially weighted average of a series of a quantity is the convolution of our quantity with where has the property that its sum over the positive values of is equal to 1. It has the intuitive meaning of a moving average where each term is multiplied by weights given by the sequence .

As you choose smaller and smaller, the number of points that has a weight significantly different than zero decreases as you can see in Figure 4-20, where we have plotted the series for different values of .

Figure 4-20: the series for three values of : 0.9, 0.8 and 0.3. Note that as gest smaller the series is significantly different than zero for an increasingly smaller number of values around .

This method is at the very core of the Momentum optimizer and more advanced learning algorithms, and we will see in the next sections how it works in practice.

## Momentum

You will remember that in plain Gradient Descent the weights updates are calculated with the equations

The idea behind the momentum optimizer is to use exponentially weighted averages of the corrections of the gradient and then use them for the weight updates. More mathematically we calculate

And we will then perform the updates with the equations

Where usually and are chosen. Now that means, as you can now understand from our discussion about exponentially weighted averages from last section, that instead of using the derivatives of the cost functions with respect to the weights, we update the weights with a moving average of the derivatives. Usually, experience shows that a bias correction could theoretically be neglected.

The momentum algorithm uses an exponential weighted average of the derivates of the cost function with respect to the weights for the weight updates. In this way not only the derivatives at a given iteration are used, but also the past behavior is considered. It may happen that the algorithm oscillates around the minimum instead of converging directly. This algorithm can escape from plateaus much more efficiently than standard gradient descent.

Sometimes you find a slightly different formulation in books or blogs, that is (I report here only the equation for the weights for brevity):

The idea and meaning remain the same, is simply a slightly different mathematical formulation. I find that the method I described you is easier to understand intuitively with the notion of sequence convolution and of weighted average than this second formulation. Another formulation that you will find (and that is the one that tensorflow uses) is

Where is called by tensorflow momentum (the superscript indicates that this variable is used by tensorflow). In this formulation the weight update assumes the form

Where again the superscript indicates that the variable is the one used by tensorflow. Although it seems different, this formulation is completely equivalent to the formulation I gave you at the beginning:

The tensorflow formulation and the one we discussed at the beginning are equivalent if we choose

That can be seen by simply comparing the two different equations for the weight updates. Typically, values around in tensorflow implementations are used and they typically work well.

Implementing momentum in tensorflow is extraordinarily easy. Just replace the GradientDescentOptimizer with

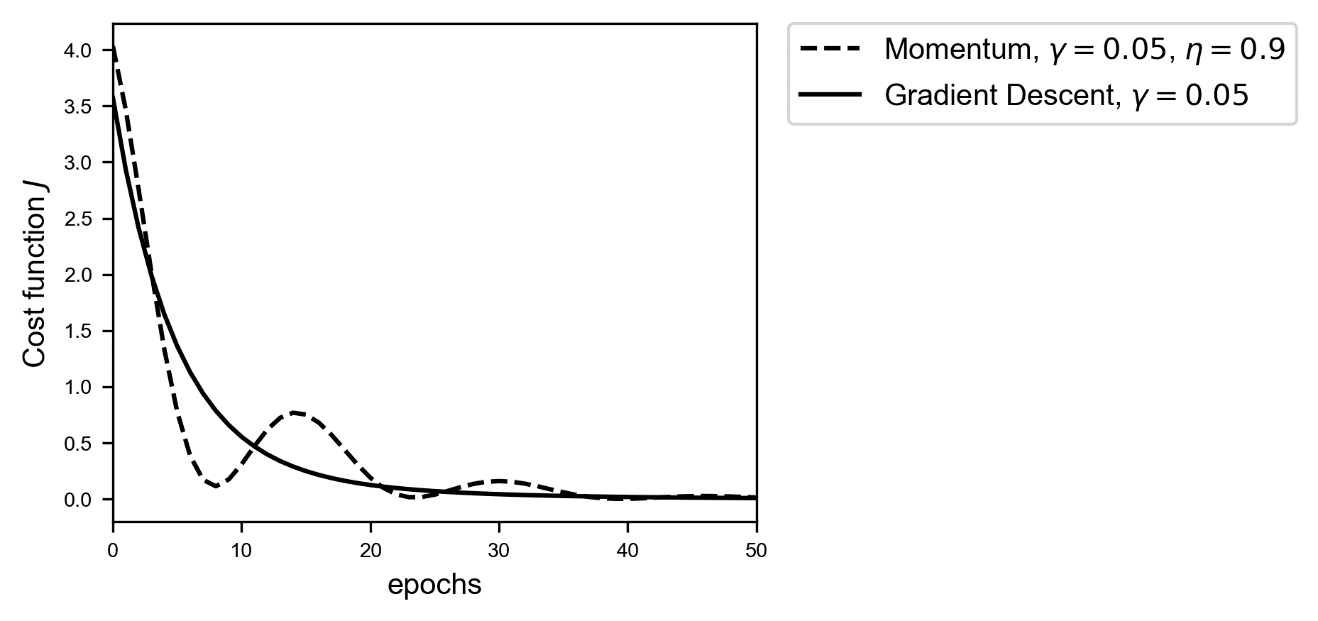
tf.train.MomentumOptimizer(learning\_rate = learning\_rate, momentum = 0.9)

The momentum almost always converges faster than plain gradient descent.

Note that comparing the different parameters in the different optimizers is wrong. When talking about the learning rate for example, it has a different meaning in the different algorithms. What you should compare is the best convergence speed you can achieve with several optimizers, regardless of the choice of parameters. Compare the GD for a learning rate of 0.01 with Adam (you will see it later) for the same learning rate does not make much sense. You should compare the optimizers with the parameters that gives you the best and fastest convergence to decide which one to use.

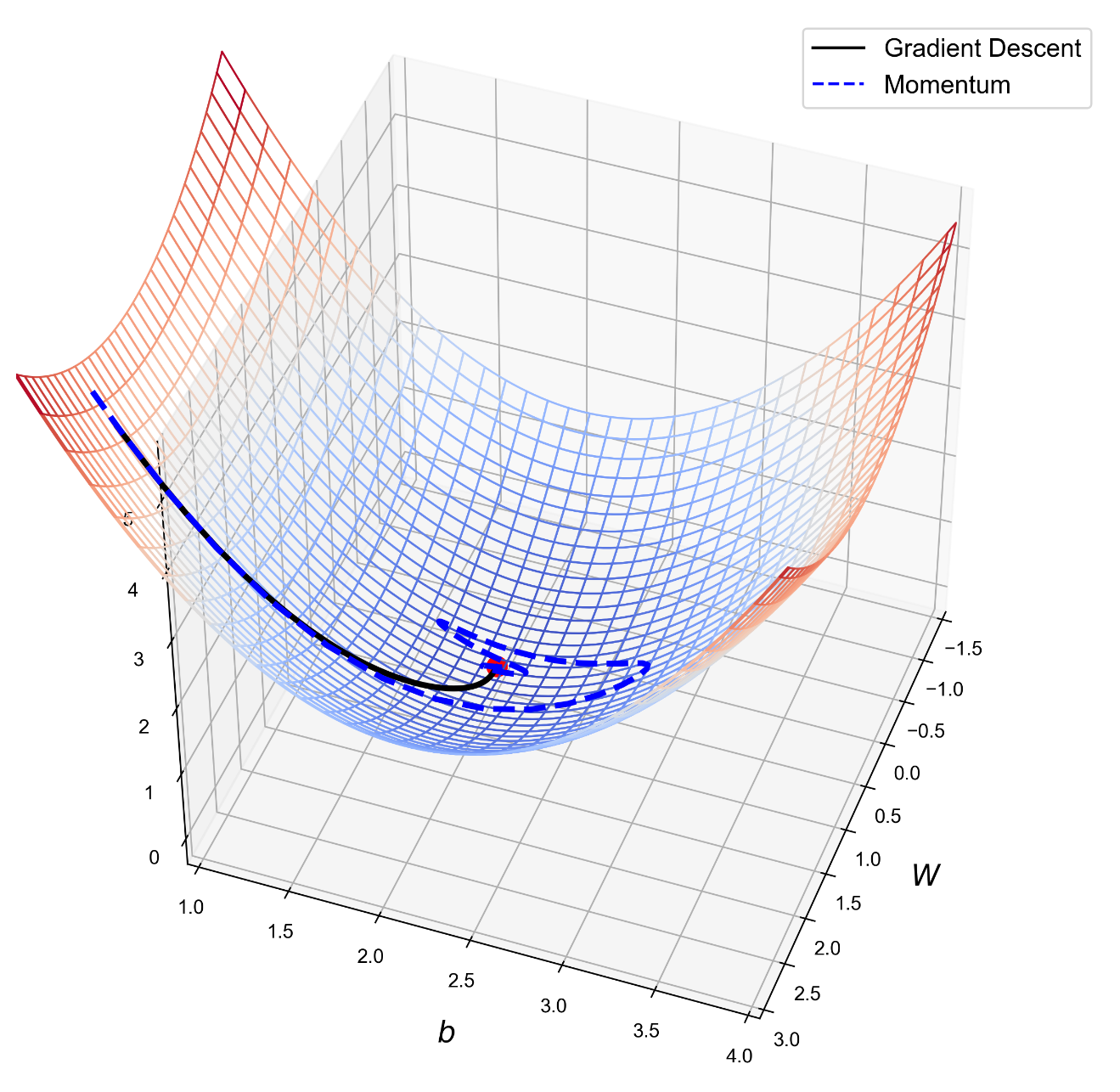
In Figure 4-21 you can see the cost function for the problem we have discussed in the previous section for plain gradient descent (with ) and for Momentum (with and ). You can see how the Momentum optimizer oscillates around the minimum. What is difficult to see on the y-scale, is that with Momentum reaches a much lower value.

Figure 4-21: the cost function vs. the number of epochs for plain gradient descent (with ) and for Momentum (with and ). You can see how the Momentum optimizer oscillates a bit around the minimum.



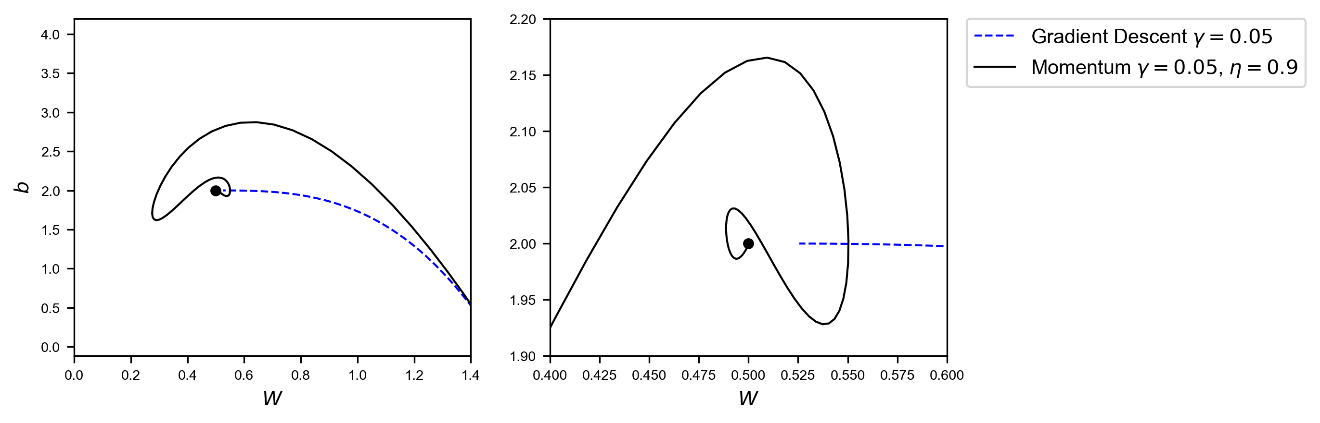
More interesting is to check how the Momentum optimizer chooses its path along the cost function surface. In Figure 4-22 you can see a 3D surface plot of the cost function. The continuous line is the path that the Gradient Descent optimizer chooses, along the maximum steepness as expected. The dashed line is the one that the Momentum optimizers chooses as it oscillates around the minimum.

Figure 4-22: a 3D surface plot of the cost function . The continuous line is the path that the Gradient Descent optimizer chooses, along the maximum steepness as expected. The dashed line is the one that the Momentum optimizers chooses as it oscillates around the minimum.



I want to convince you that Momentum is faster and better at converging. To do that let's check in the weights plane how the two optimizers behave. In Figure 4-23 you can see the path that the two optimizers have chosen. On the right plot you can see a zoom around the minimum. You can see how Gradient descent after 100 epochs does not manage to reach the minimum, although it seems to choose a more direct path toward the minimum. It gets very close, but not close enough. The Momentum optimizer oscillates around the minimum, and very efficiently it reaches it.

Figure 4-23: the path that the two optimizers have chosen. On the right plot you can see a zoom around the minimum. You can see how Momentum reaches the minimum after oscillating around it, while GD does not manage to reach it in 100 epochs.



## RMSProp

Let's move to something a bit more complex, but usually more efficient. Let me give you the mathematical equations and then we will compare it to the others we have seen so far. At each iteration we need to calculate

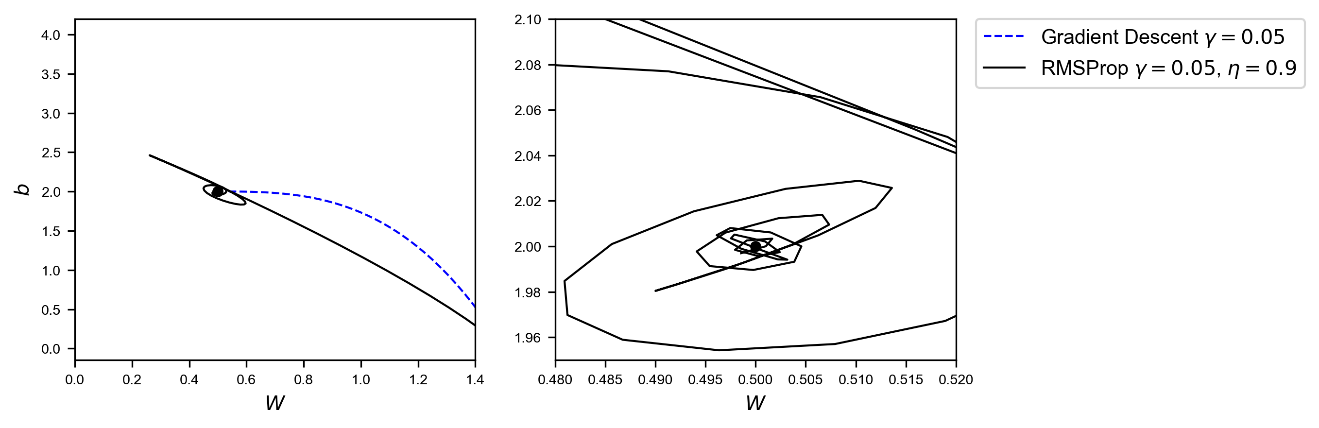
where the symbol indicates an element wise product. Then we will do the update of our weights with the equations

So first you do an exponential weighted average of the quantities and and then use them to modify the derivates that you use to do your weight updates. The , usually chosen is there to avoid the denominator going to zero in case the quantities and goes to zero. The intuitive idea is that if the derivative is big then the quantities are big therefore the factors or will be smaller and therefore the learning will slow down. The other way around is also true, so if the derivatives are small than the learning will be faster. This algorithm will make the learning faster for the parameters that are slowing it down. In tensorflow is again particular easy to use it simply with the code

optimizer = tf.train.RMSPropOptimizer(learning\_rate, momentum = 0.9).minimize(cost)

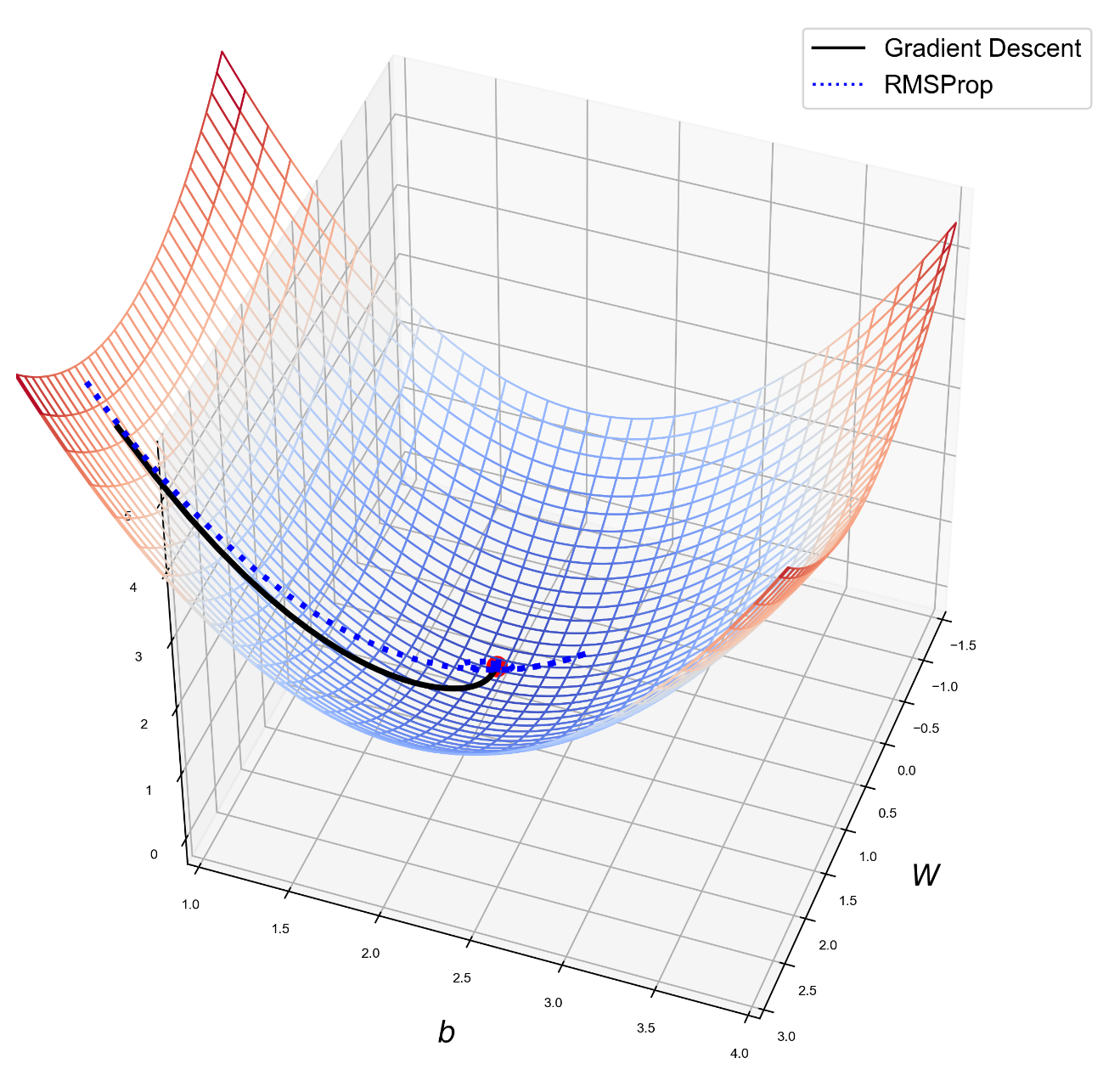
Let's check what path this optimizer chooses. In Figure 4-24 you can see that RMSProp oscillating around the minimum. While the GD does not reach it, the RMSProp algorithm has the time of doing several loops around it before reaching it.

Figure 4-24: The path chosen toward the cost function minimum by plain Gradient Descent and RMSProp. The latter make loops around the minimum and then reaches it. In the same number of epochs the GD does not even gets that close. Note the scale on the plot on the right. The zoom level is very high, we are looking at an extreme closeup (the GD path is not even visible on this scale) around the minimum.



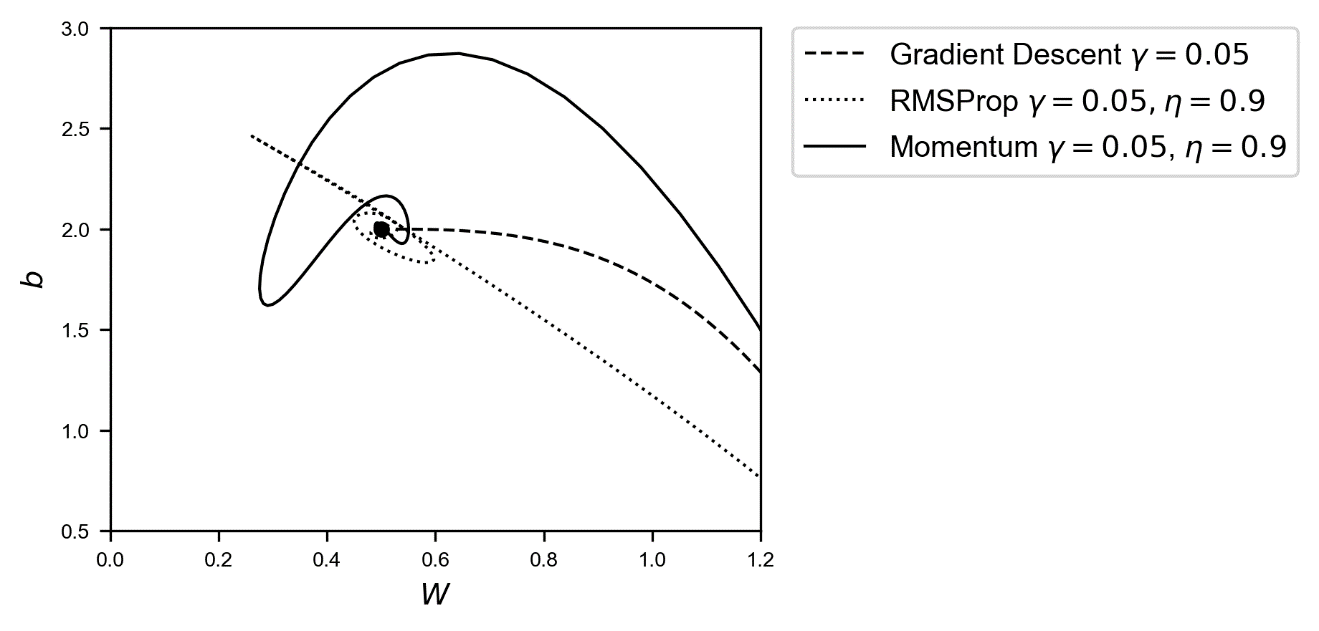
In Figure 4-25 you can see the same path in 3D along the cost function surface.

Figure 4-24: the path choosen by the GD () and RMSProp (, , ) along the surface of the cost function. The red point indicates the minimum. RMSProp, especially at the beginnig, chooses a more direct path toward the minimum than GD.



In Figure 4-26 you can see GD, RMSProp and Momentum paths. You can see how RMSProp path is much more direct toward the minimum. It gets close to it very quickly and then oscillating gets closer and closer. It overshoots a bit at the beginning but then corrects itself quickly and comes back.

Figure 4-26: the path toward the minimum choosen by GD, RMSProp and Momentum. You can see how RMSProp path is much more direct toward the minimum. It gets around it very quickly and then oscillating gets closer and closer.



## Adam

The last algorithm we will look at is called Adam (Adaptive Moment estimation). It combines the ideas of RMSProp and Momentum in one optimizer. Like Momentum it uses an exponential weighted average of past derivatives and like RMSProp it uses the expentially weigthed averages of past squared derivatives.

You will need to calculate the same quantities that you need for Momentum and for RMSProp and then you need to calculate the following quantities

And in the same way

Where we have used for the hyperparameter we will use in Momentum and for the one we used in RMSProp. Then, similar to what we did in RMSProp, we will update our weights with the equations

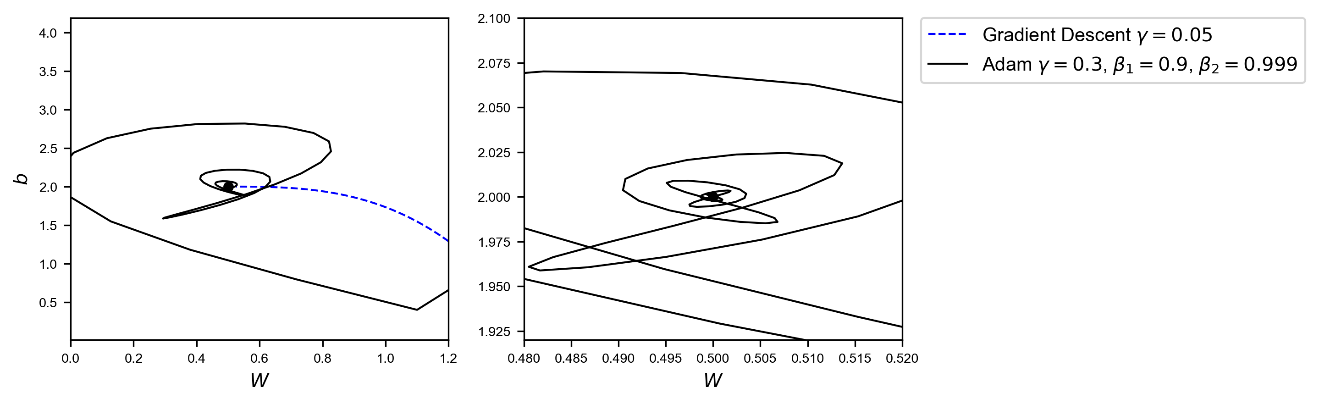
Tensorflow does everything for us if we simply use the following line

optimizer = tf.train.AdamOptimizer(learning\_rate = learning\_rate, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8).minimize(cost)

where in this case the typical values for the parameters has been chosen: , , and . Note how, since this algorithm adapt the learning rate to the situation we can start with a bigger learning rate to speed up the convergence.

In Figure 4-27 you can see the path chosen by GD and Adam optimizer around the minimum. Adam oscillates too around the minimum, but it reaches it without problems. On the right plot (a zoom around the minimum) you can see how the algorithm gets very close to the minimum. To give you an idea of how good the optimizer is, after just 200 epochs the weights and bias gets to 0.499983, 2.000047, that is really close to the minimum (remember the minimum is at and .

Figure 4-27: The path that GD and Adam optimizers choose after 200 epochs. Note the amount of loops that Adam does around the minimum. Regardless the opimizer is really efficient with comparison with the plain GD.



I don’t show you all optimizers together since you would see a lot of loops and it would not teach you anything really.

## Which optimizer should I use?

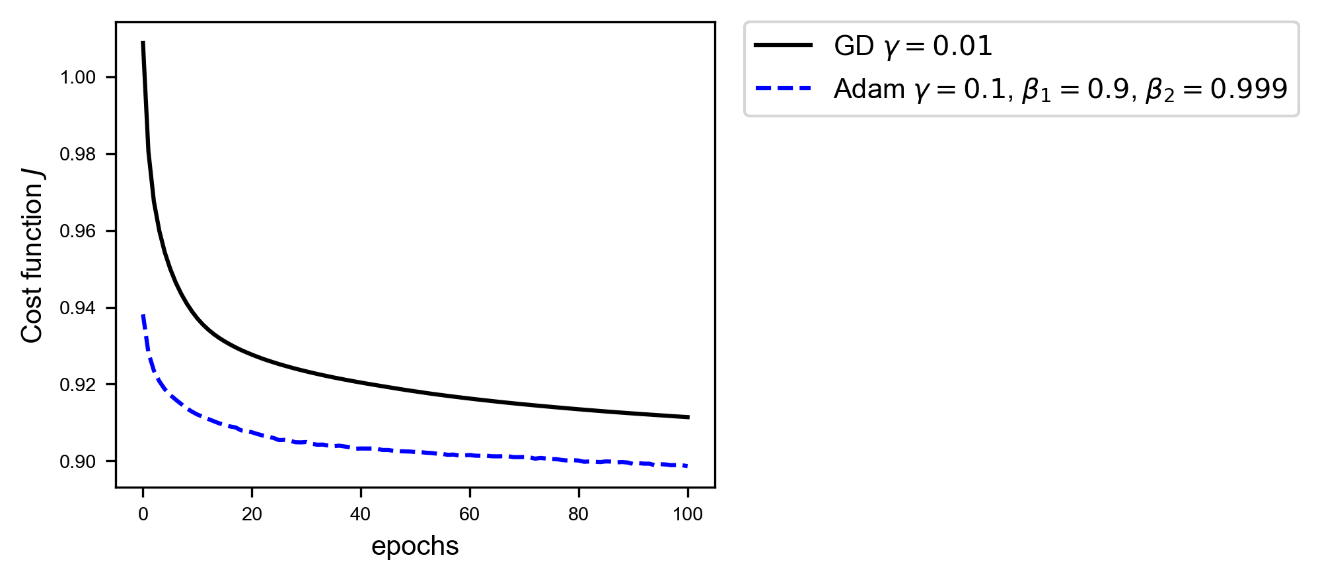
To make the story short you should use **Adam**, it is generally considered faster and better than other methods. That does not mean that this is always the case. There are recent research papers that indicates how those optimizers could generalize poorly on new datasets (check for example https://goo.gl/Nzc8bQ). And there are other papers that simply use GD with a dynamical learning rate decay. It mostly depends on your problem. But generally, Adam is a very good starting point.

If you are unsure with which optimizer to start: use Adam. Is generally considered faster and better than other methods.

To give you an idea of how good it can be, let's apply it to the Zalando dataset. We will use a network with 4 hidden layers each with 20 neurons. The model we will use is the one we discussed at the very end of Chapter 3. Check in Figure 4-28 how the cost function converges faster when using Adam optimization in comparison with GD. Additionally, in 100 epochs, GD reaches an accuracy of 86%, while Adam reaches 90%. Note that we have not changed anything in the model, except the optimizer! For the Adam we have used the following code

optimizer = tf.train.AdamOptimizer(learning\_rate = learning\_rate, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8).minimize(cost)

Figure 4-28: the cost function for the Zalando dataset for a network with 4 hidden layers, each with 20 neurons. The continous line is plain GD with a learning rate of , and the dashed line is Adam optimisation with , , and .



As I suggested, when testing complex networks on big dataset, the Adam optimizer is a good place to start. But you should not limit your tests only to this optimizer. A test of other methods is always worthwhile! Maybe for your problem other approaches work better!

# Example of self-developed optimizer

Before finishing with this chapter, I want to show you how to develop your own optimizer using tensorflow. This is very useful when you want to use an optimizer that is not directly available. Take for example the paper from Neelakantan et al.[[4]](#footnote-4). In their research they show how adding random noise to the gradients when training complex networks, allows plain gradient descent to become very effective. They showed how a 20-layer deep networks can be trained efficiently with standard GD even starting with poor weight initialization. If you want to test this method, for example, you cannot use the tf.GradientDescentOptimizer function, since this implements a plan GD, without the noise they describe in the paper. To test it you need to have access to the gradients in the code, add the noise to them and then update the weights with the modified gradients. We will not test their method here, that would require too much time and would go beyond the scope of the book, but is instructive to see how to develop plain gradient descent without using the tf.GradientDescentOptimizer and without calculating any derivative manually!

Before constructing our network, we must know the dataset we want to use and what problem (regression, classification, etc.) we want to solve. Let's make something new with a known dataset. Let's use the MNIST dataset we used in Chapter 2, but this time let's perform multiclass classifcation using the softmax function as we have done on the Zalando dataset in Chapter 3. We have discussed at length how to load the MNIST dataset with sklearn in Chapter 2, so let's do it in a different (and more efficient) way here. Tensorflow have a method to download the MNIST dataset, including labels already one-hot encoded. This can be simply done with the lines

from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets("/tmp/data/", one\_hot=True)

This gives you the output

Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.   
Extracting /tmp/data/train-images-idx3-ubyte.gz   
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.   
Extracting /tmp/data/train-labels-idx1-ubyte.gz   
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.   
Extracting /tmp/data/t10k-images-idx3-ubyte.gz   
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.   
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz

You will find the files in the folder (if you are under windows) c:\tmp\data. If you want to change the location where the files are stored, you need to change the "/tmp/data" parameter of the function read\_data\_sets. Now, as you probably remember from Chapter 2, the MNIST images are 28x28 pixels (784 pixels in total) images, in gray scale (so each pixel can assume the value 0 to 254). Having this information, we can now construct our network

X = tf.placeholder(tf.float32, [784, None]) # mnist data image of shape 28\*28=784

Y = tf.placeholder(tf.float32, [10, None]) # 0-9 digits recognition => 10 classes

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

W = tf.Variable(tf.zeros([10, 784]), dtype=tf.float32)

b = tf.Variable(tf.zeros([10,1]), dtype=tf.float32)

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

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

grad\_W, grad\_b = tf.gradients(xs=[W, b], ys=cost)

new\_W = W.assign(W - learning\_rate\_ \* grad\_W)

new\_b = b.assign(b - learning\_rate\_ \* grad\_b)

The line

grad\_W, grad\_b = tf.gradients(xs=[W, b], ys=cost)

gives you the tensors that contains the gradients of the cost node with respect to W and b respectively. Tensorflow calculates them for you automatically! If you are interested to know how, check the official documentation of the tf.gradients function at <https://goo.gl/XAjRkX>. Now we need to add nodes to the computational graph that update the weights, and that is what we do with the lines

new\_W = W.assign(W - learning\_rate\_ \* grad\_W)

new\_b = b.assign(b - learning\_rate\_ \* grad\_b)

when we ask tensorflow to evaluate the nodes new\_W and new\_b during our session, the weights and bias gets updated. Finally, we need to modify the function that evaluates the graph using the line (for the mini-batch GD)

\_, \_, cost\_ = sess.run([new\_W, new\_b , cost], feed\_dict = {X: X\_train\_mini, Y: y\_train\_mini, learning\_rate\_: learning\_r})

In this way the new nodes new\_W and new\_b gets evaluated, and in doing so tensorflow updates the weights and bias. The following lines is not needed anymore

sess.run(optimizer, feed\_dict = {X: X\_train\_mini, Y: y\_train\_mini, learning\_rate\_: learning\_r})

since we don't have the optimizer node anymore. The entire function you need is the following

def run\_model\_mb(minibatch\_size, training\_epochs, features, classes, logging\_step = 100, learning\_r = 0.001):

sess = tf.Session()

sess.run(tf.global\_variables\_initializer())

total\_batch = int(mnist.train.num\_examples/minibatch\_size)

cost\_history = []

accuracy\_history = []

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

for i in range(total\_batch):

batch\_xs, batch\_ys = mnist.train.next\_batch(minibatch\_size)

batch\_xs\_t = batch\_xs.T

batch\_ys\_t = batch\_ys.T

\_, \_, cost\_ = sess.run([new\_W, new\_b ,

cost], feed\_dict = {X: batch\_xs\_t, Y: batch\_ys\_t, learning\_rate\_: learning\_r})

cost\_ = sess.run(cost, feed\_dict={ X:features, Y: classes})

accuracy\_ = sess.run(accuracy, feed\_dict={ X:features, Y: classes})

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

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

if (epoch % logging\_step == 0):

print("Reached epoch",epoch,"cost J =", cost\_)

print ("Accuracy:", accuracy\_)

return sess, cost\_history, accuracy\_history

This function is slightly different that what we have used before, since I used here some features of tensorflow to make our life a bit easier. In particular the line

total\_batch = int(mnist.train.num\_examples/minibatch\_size)

calculate the total number of mini batches that we have, since the variable mnist.train.num\_examples contains the number of observations we have at our disposal. Then to get the batches we use

batch\_xs, batch\_ys = mnist.train.next\_batch(minibatch\_size)

this returns two tensors, containing the training input data (batch\_xs) and the labels one-hot encoded (batch\_ys). We then simply need to transpose them, since tensorflow returns the array with observations as rows. We do that with the lines

batch\_xs\_t = batch\_xs.T

batch\_ys\_t = batch\_ys.T

I also added to the function the accuracy calculation, to make it easier to see how good we are doing. Letting the model run with the python call

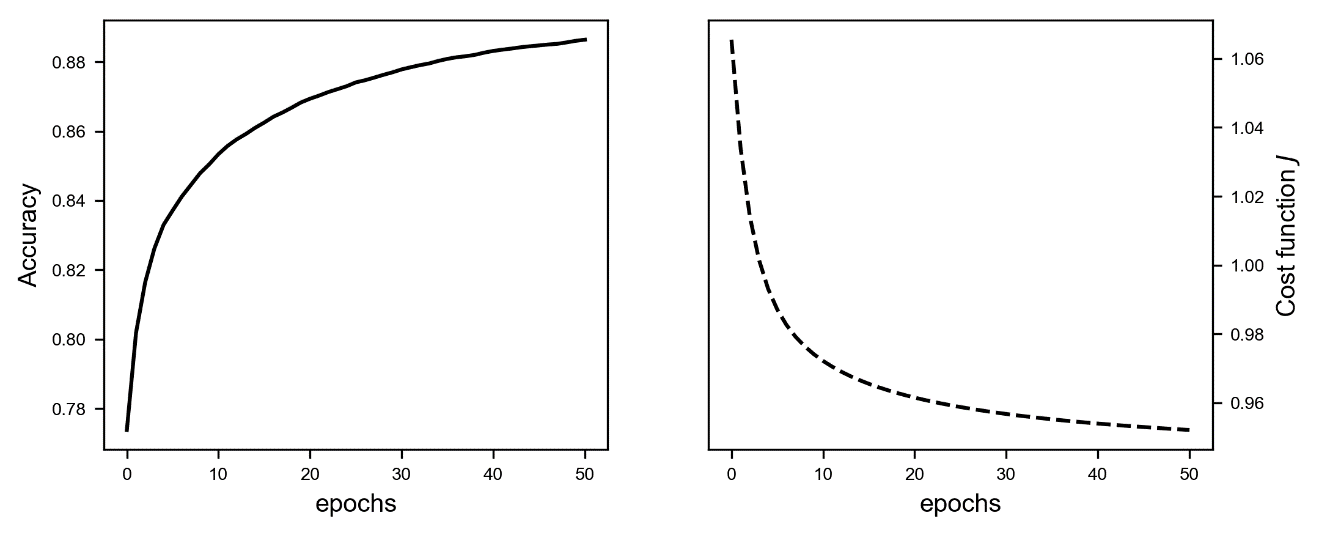
sess, cost\_history, accuracy\_history = run\_model (100, 50, X\_train\_tr, labels\_, logging\_step = 10, learning\_r = 0.01)

will give you the output

Reached epoch 0 cost J = 1.06549   
Accuracy: 0.773786   
Reached epoch 10 cost J = 0.972171   
Accuracy: 0.853371   
Reached epoch 20 cost J = 0.961519   
Accuracy: 0.869357   
Reached epoch 30 cost J = 0.956766   
Accuracy: 0.877814   
Reached epoch 40 cost J = 0.953982   
Accuracy: 0.883143   
Reached epoch 50 cost J = 0.952118   
Accuracy: 0.886386

This model will work exactly in the same way as the one with the gradient descent optimizer provided by tensorflow. But now you have access to the gradients and you can modify them, add noise to them (if you want to try) and so on. In Figure 4-29 you can see the cost function behavior (on the right side) and the accuracy vs. the epochs (on the left side) that we get with this model.

Figure 4-29 you can see the cost function behaviour (on the right side) and the accuracy vs. the epochs (on the left side) for the neural network with one neuron and with the gradient descent developed with the tf.gradients function.



1. In case you don't know what TL;DR means: Too Long; Don't Read. Is internet slang to say that some text has been ignored because too long [Source: https://en.wikipedia.org/wiki/TL;DR] [↑](#footnote-ref-1)
2. Check the overview on the tensorflow official documentation: https://goo.gl/vpFNp7 [↑](#footnote-ref-2)
3. Generally speaking a sequence is an enumerated collection of objects. [↑](#footnote-ref-3)
4. Neelakantan A. et al., Adding gradient noise improves learning for very deep learning, conference paper at ICLR 2016, available at https://arxiv.org/abs/1511.06807 [↑](#footnote-ref-4)