Icon

Description automatically generatedOptimizers

In this chapter, we will look at optimizers. We will look at different optimizers as Momentum, RMSProp and Adam. We will look at the mathematics behind those and then at how to implement and use them in Keras.

# 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 faster and more 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 most used 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. So, let’s start.

## Exponentially weighted averages

Let's suppose you are measuring a quantity (it could be the temperature where you live for example) 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[[1]](#footnote-1). 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

Diagram

Description automatically generated

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 .Shape

Description automatically generated

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 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. 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.

A picture containing chart

Description automatically generated

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.

Chart

Description automatically generated

## 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 particular easy to use it simply with the code

optimizer = tf.keras.optimizers.RMSprop(learning\_rate=0.1)

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.

Diagram

Description automatically generated with low confidence

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.

Chart

Description automatically generated

## 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.keras.optimizers.Adam(learning\_rate=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-07)

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.

Diagram

Description automatically generated

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!

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 .

Graphical user interface

Description automatically generated with medium confidence

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

To understand how to develop your own optimizer, we first need to understand how Keras and TensorFlow implements automatic differentiation. Automatic differentiation is simply the ability of TensorFlow to compute the gradient of a computation (for example your network function) with respect to some inputs as the name says automatically. To do this TensorFlow offers the tf.GradientTape API. Let’s look at some very easy examples.

Suppose we want to evaluate the derivative of at the point . From calculus we know that

We can do it without calculating anything in TensorFlow with

x = tf.Variable(3.0)

with tf.GradientTape() as tape:

y = x\*\*2

# dy = 2x \* dx

dy\_dx = tape.gradient(y, x)

dy\_dx

print(dy\_dx.numpy())

that will get you the result 6.0 as expected. To completely understand how this works, we would need to understand many complex concepts as for example backpropagation and how TensorFlow builds the computational graph of the operations in the tf.GradentTape() context, but that would go beyond the scope of this book, therefore we will concentrate here on the use of the API.

What we really need in practical applications, is the gradient of a loss function with respect to the trainable parameters. Here is a very compact example that show how this works.

layer = tf.keras.layers.Dense(2, activation='relu')  
x = tf.constant([[1., 2., 3.]])  
  
with tf.GradientTape() as tape:  
  y = layer(x)  
  loss = tf.reduce\_mean(y\*\*2)  
  
# Calculate gradients with respect to every trainable variable  
grad = tape.gradient(loss, layer.trainable\_variables)

as you can see the line tape.gradient(loss, layer.trainable\_variables) needs as input the function you want to calculate the gradient of, and the list of variables that you want to derive the function with respect to.

# Exercises

Exercise 1

List the most useful tasks you can use an autoencoder for. Can you think of an application in your field of work?

Exercise 2

Can you explain briefly what is a sparse autoencoder? How is similar to an autoencoder with a bottleneck?

Exercise 3

How do you measure (with which metric) the performance of an autoencoder? List the most commonly used metrics that you can use. Can you think of any additional metric, in addition to those discussed in this chapter, that could be used?

Exercise 4

Describe how anomaly detection work with autoencoders.

# Further Readings

Deep Learning Tutorial from Stanford University

<http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/>

Building autoencoders in Keras

<https://blog.keras.io/building-autoencoders-in-keras.html>

Introduction to autoencoders in TensorFlow

<https://www.tensorflow.org/tutorials/generative/autoencoder>

Bank, D., Koenigstein, N., and Giryes, R., “Autoencoders”, arXiv e-prints, 2020,

<https://arxiv.org/abs/2003.05991>

R. Grosse, University of Toronto, Lecture on autoencoders

<http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec20.pdf>

1. Generally speaking a sequence is an enumerated collection of objects. [↑](#footnote-ref-1)