Optimizers

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

# Available optimizers in Keras in TensorFlow 2.5

TensorFlow has evolved a lot in the last years and while at the beginning you could find gradient descent as a class, now it is not available anymore (not directly). Keras offers as of TensorFlow version 2.5 only advanced algorithms. In particular you will find: Adadelta, Adagrad, Adam. Adamax, Ftrl, Nadam, RMSProp and SGD (gradient descent with momentum).

In general, you should try which optimizers work best for your problem, but if you don’t know where to start, Adam is always a good choice.

# Advanced optimizers

Until now we have only discussed how gradient descent works. That is not the most efficient optimizer (although the easiest to understand), and there are some modifications to the algorithm that can make it 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 famous ones: Momentum, RMSProp and Adam.

Additional material that 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 with some mathematics.

## 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 with a recursive formula 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 can define

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 6-1).

Figure 6-1: 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 6-1. 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 6-2, where we have plotted the series for different values of .

Figure 6-2: 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

Where the subscript in square brackets indicates the iteration. 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. Mathematically we formulate the previous statement as

And we will then perform the weights and biases 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.

## 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)

## 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 have 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.

# Comparison of Optimizers’ Performance

It is interesting to see how the optimizers behave to better understand why, for example, Adam is so efficient. To do that let’s create a toy problem. Let’s consider a dataset of 30 tuples with

And

And let’s compare Gradient Descent, Adam and RMSProp. The goal of the problem is, starting from the 30 datapoints, to determine the linear relations between and , in other words determine the two parameter 2 and 1/2 in the last equation. For this example, we can write the linear relation as

The optimizers will need try to minimize the MSE with respect to and . You will find the entire code to do the comparison in the online version of the book at <https://adl.toelt.ai>. In the online code you will see an implementation of the Gradient Descent from scratch, in case you want to check it out. In Figure 6-3 you can see the patch that the GD algorithm takes in parameter space to reach the minimum of the MSE function in 200 iterations.

Chart, line chart

Description automatically generated

Figure 6-3: the path that the GD algorithm follows in parameter spaces while minimizing the MSE when starting from (0,0). The number of iterations used for this plot is 200.

In Figure 6-4 you can see the different paths taken by the different optimizers when solving the same problem.

Chart, line chart

Description automatically generated

Figure 6-4: the path that the GD algorithm, Adam and RMSProp optimizers follow in parameter spaces while minimizing the MSE when starting from (0,0). The number of iterations used for this plot is 200.

One striking difference in Figure 6-4 is that whil the GD path is rather direct the others tends to be less direct, with Adam making loops around the minimum and RMSProp oscillating when close to the minimum. From the plot is difficult to see which one is faster, and the best way of checking that is to plot and vs. , where indicates the iteration number. In Figure 6-5 you can see how clearly how quickly converges to the expected value of 2.0 with the different algorithms. In this case Adam and GD are on par, while RMSProp seems to be slower.

Chart

Description automatically generated

Figure 6-5: Plot of vs. the number of iterations with different optimizers.

A different picture comes when considering . In Figure 6-6 you can see how fast the different algorithms converge.

Chart, histogram

Description automatically generated

Figure 6-6: Plot of vs. the number of iterations with different optimizers.

In this case you can see how much faster Adam and RMSProp are in converging. It is interesting to note how Adam oscillates around the expected value of 0.5. This property makes Adam very efficient in escaping areas in parameters space where training can get stuck. But you should notice something else that is very interesting. Let’s zoom the picture after iteration 150, as you can see in Figure 6-6.

Chart

Description automatically generated

Figure 6-6: Zoom of Figure 6-5. Plot of vs. the number of iterations with different optimizers.

In Figure 6-6 you can see how while GD is still converging, Adam is already at the expected value, while RMSProp oscillates around it, since probably it cannot converge completely. This should convince you how efficient Adam is in comparison with other optimizers.

## Small coding digression

I wanted to explain you briefly how you can setup the optimizers to use (0,0) as starting point for the updates. To do that, after you have created and compiled the model, you can set the weights of the model with

model.set\_weights([np.array([w0\_start]).reshape(1,1),np.array([w1\_start]).reshape(1,)])

where w0\_start = 2.0 and w1\_start = 0.5.

After having done that, you need to save the value of the weights after each iteration. The easiest way is to implement your update as a custom training loop by using GradientTape(). You will find the entire code online, but your training loops will look like

for epoch in range(200):

with tf.GradientTape() as tape:

# Run the forward pass of the layer.

ypred = model(x\_, training=True)

loss\_value = loss\_fn(y\_, ypred)

grads = tape.gradient(loss\_value, model.trainable\_weights)

optimizer.apply\_gradients(zip(grads, model.trainable\_weights))

w1\_rmsprop\_list.append(float(model.get\_weights()[0][0]))

w0\_rmsprop\_list.append(float(model.get\_weights()[1][0]))

where w1\_rmsprop\_list and w0\_rmsprop\_list are lists that will contain the values of the weights for each iteration. In this way you can test any other optimizer you are interested in.

## 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!

EXERCISE 1 (LEVEL MEDIUM)

Generate Figure 6-5 and 6-6 for other optimizers that Keras offers, like Adagrad. What is their behaviour? Are they faster than Adam for the toy problem we are discussing?

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