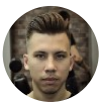
Stochastic Gradient Descent with momentum



Vitaly Bushaev
Dec 4, 2017 · 7 min read

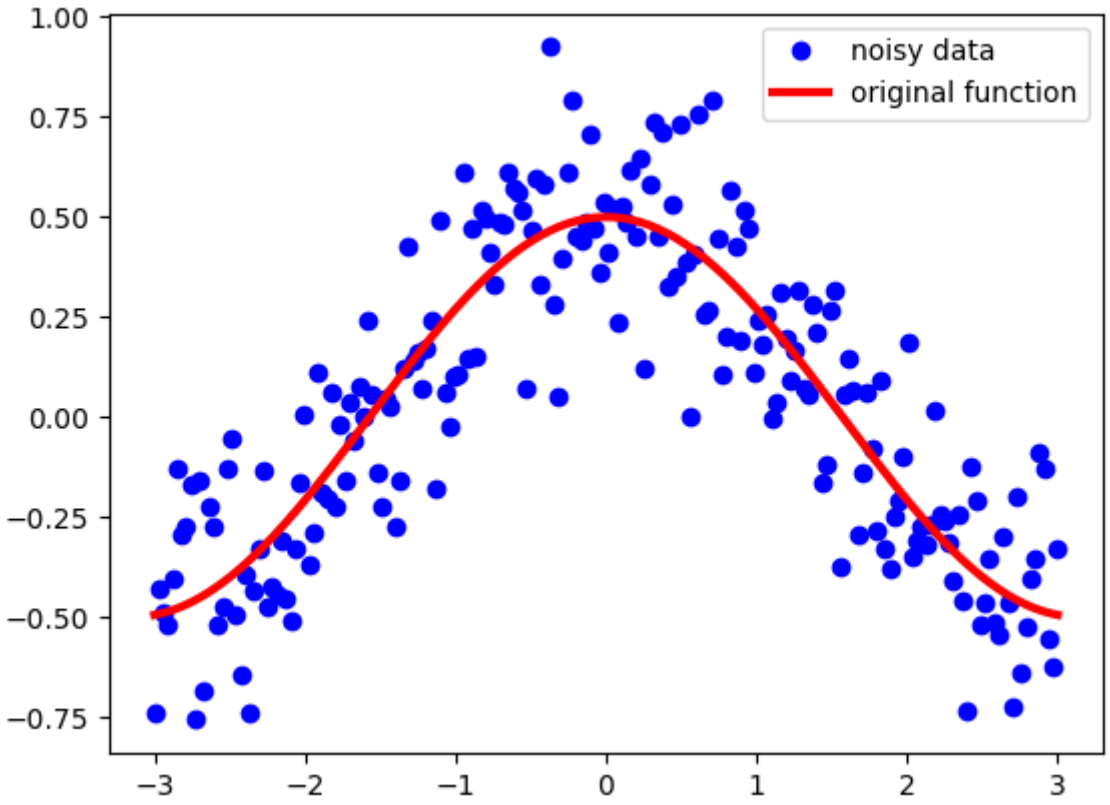
This is part 2 of my series on optimization algorithms used for training neural networks and machine learning models. Part 1 was about Stochastic gradient descent. In this post I presume basic knowledge about neural networks and gradient descent algorithm. If you don't know anything about neural networks or how to train them, feel free to read my first post before reading this one.

In this post I'll talk about simple addition to classic SGD algorithm, called momentum which almost always works better and faster than Stochastic Gradient Descent. Momentum [1] or SGD with momentum is method which helps accelerate gradients

vectors in the right directions, thus leading to faster converging. It is one of the most popular optimization algorithms and many state-of-the-art models are trained using it. Before jumping over to the update equations of the algorithm, let's look at some math that underlies the work of momentum.

Exponentially weighed averages

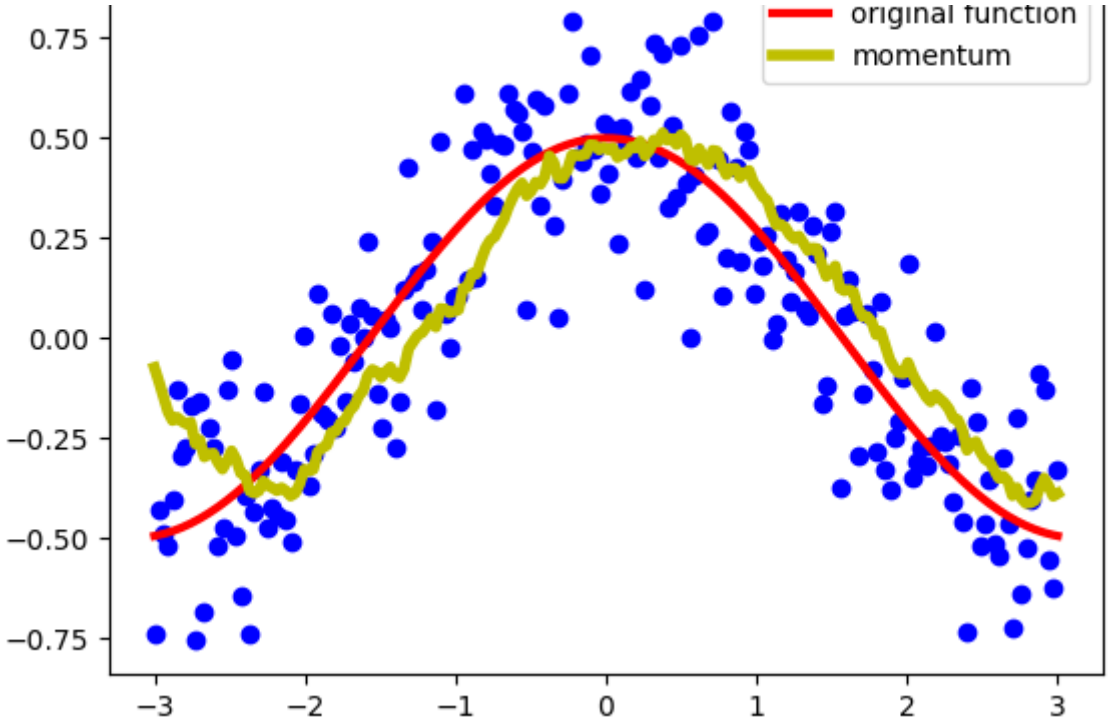
Exponentially weighed averages deal with sequences of numbers. Suppose, we have some sequence S which is noisy. For this example I plotted cosine function and added some Gaussian noise. It looks like this:



Out sequence.

Note, that even though these dots seem very close to each over, none of them share x coordinate. It is a unique number for each point. That's the number the defines the index of each point in our sequence S.

What we want to do with this data is, instead of using it, we want some kind of 'moving' average which would 'denoise' the data and bring it closer to the original function. Exponentially weighed averages can give us a pictures which looks like this:

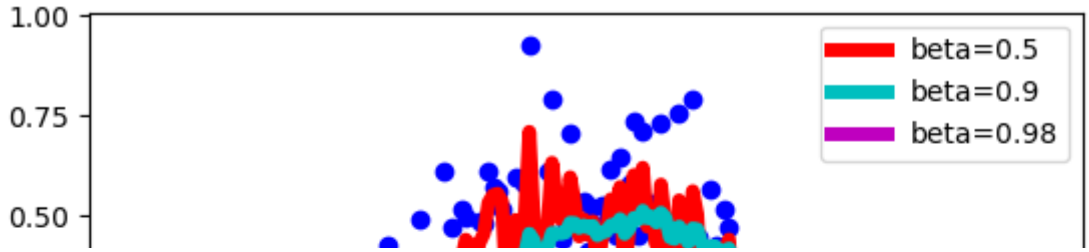


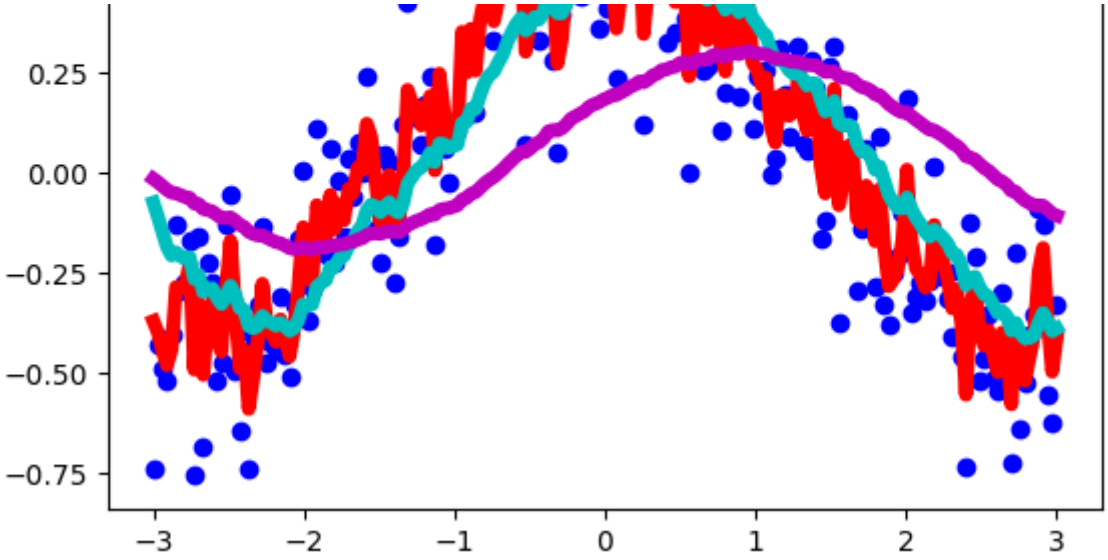
momentum — data from exponentially weighed averages.

As you can see, that's a pretty good result. Instead of having data with a lot of noise, we got much smoother line, which is closer to the original function than data we had. Exponentially weighed averages define a new sequence V with the following equation:

$$V_t = \beta V_{t-1} + (1-\beta) S_t$$
$$\beta \in [0,1]$$

That sequence V is the one plotted yellow above. *Beta* is another hyper-parameter which takes values from 0 to one. I used *beta* = 0.9 above. It is a good value and most often used in SGD with momentum. Intuitively, you can think of *beta* as follows. We're approximately averaging over last *1 / (1- beta)* points of sequence. Let's see how the choice of *beta* affects our new sequence V.





Exponentially weighed averages for different values of beta.

As you can see, with smaller numbers of beta, the new sequence turns out to be fluctuating a lot, because we're averaging over smaller number of examples and therefore are 'closer' to the noisy data. With bigger values of beta, like *beta=0.98*, we get much smother curve, but it's a little bit shifted to the right, because we average over larger number of example(around 50 for *beta=0.98*). *Beta = 0.9* provides a good balance between these two extremes.

Math section

This section isn't necessary to use momentum in your projects, so feel free to skip it. But it provides a little bit more intuition of how momentum works.

Let's expand our definition of exponentially weighed averages for three consecutive elements of the new sequence V.

$$V_t = \beta V_{t-1} + (1-\beta) S_t$$
$$V_{t-1} = \beta V_{t-2} + (1-\beta) S_{t-1}$$
$$V_{t-2} = \beta V_{t-3} + (1-\beta) S_{t-2}$$

V — New sequence. S — original sequence.

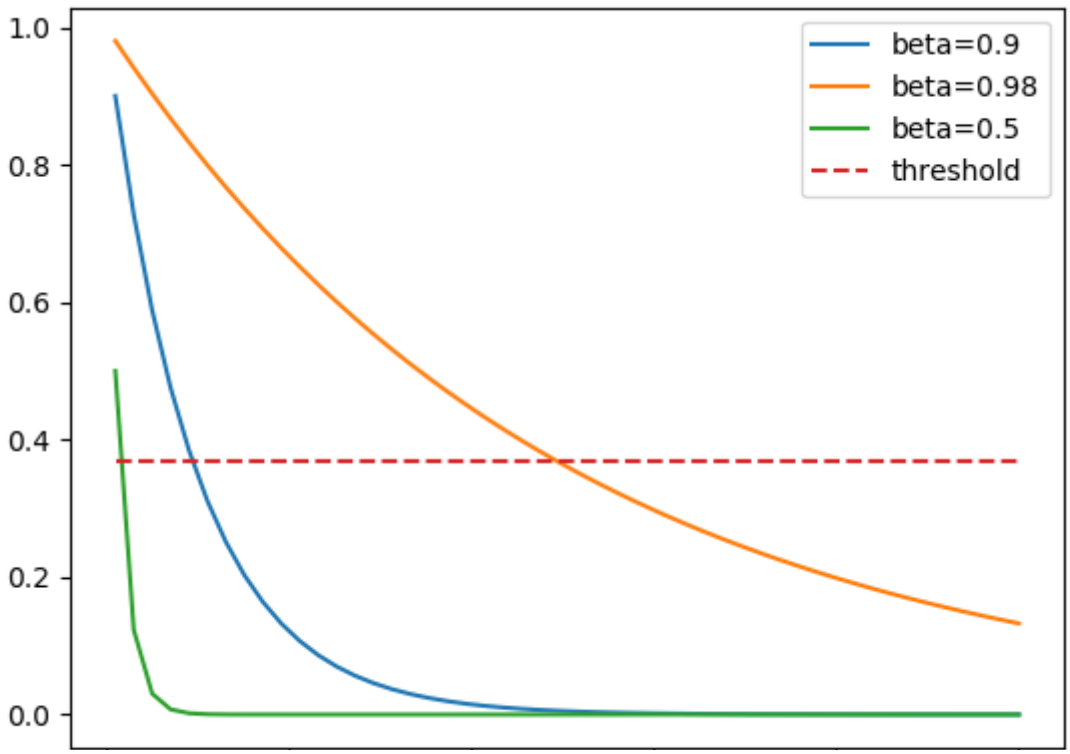
Combining all of them together we get:

$$V_t = \beta(\beta(\beta V_{t-3} + (1-\beta) S_{t-2}) + (1-\beta) S_{t-1}) + (1-\beta) S_t$$

And then simplifying it a bit:

$$V_t = \beta\beta(1-\beta) S_{t-2} + \ldots + \beta(1-\beta) S_{t-1} + \ldots + (1-\beta) S_t$$

From this equation we see, that the value of *Tth* number of the new sequence is dependent on all the previous values *1..t* of the original sequence S. All of the values from S are assigned some weight. This weight is *beta* to power of *i* multiplied by *(1-beta)* for *(t - i)*th value of S. Because beta is less than 1, it becomes even smaller when we take beta to the power of some positive number. So the older values of S get much smaller weight and therefore contribute less for overall value of the current point of V. At some point the weight is going to be so small that we can almost say that we 'forget' that value because its contribution becomes too small to notice. A good value to use for this approximation is when weight becomes smaller than *1 / e*. Bigger values of beta require larger values of the power to be smaller than *1 / e*. This is why with bigger values of beta we're averaging over bigger numbers of points. The following graph shows how fast weights get smaller with older values of S compared to threshold = *1 / e*, where we mostly 'forget' values that are older.



The last thing to notice is that the first couple of iterations will provide a pretty bad averages because we don't have enough values yet to average over. The solution is instead of using V, we can use what's called bias-corrected version of V.

$$V_t = \frac{V_t}{1 - b^t}$$

where $b = beta$. With large values of t, b to the power of t will be indistinguishable from zero, thus not changing our values of V at all. But for small values of t, this equation will produce a little bit better results. With momentum, people usually don't bother to implement this part, because learning stabilizes pretty fast.

SGD with momentum

We've defined a way to get 'moving' average of some sequence, which changes alongside with data. How can we apply this for training neural networks ? The can average over our gradients. I'll define how it's done in momentum and then move on to explain why it might work better.

I'll provide two definitions of SGD with momentum, which are pretty much just two different ways of writing the same equations. First, is how Andrew Ng, defines it in his Deep Learning Specialization on coursera. The way he explains it, we define a momentum, which is a moving average of our gradients. We then use it to update the weight of the network. This could written as follows:

$$V_t = \beta V_{t-1} + (1 - \beta) \nabla_w L(W, X, y)$$
$$W = W - \alpha V_t$$

Where L — is loss function, triangular thing — gradient w.r.t weight and *alpha* — learning rate. The other way, which is most popular way to write momentum update

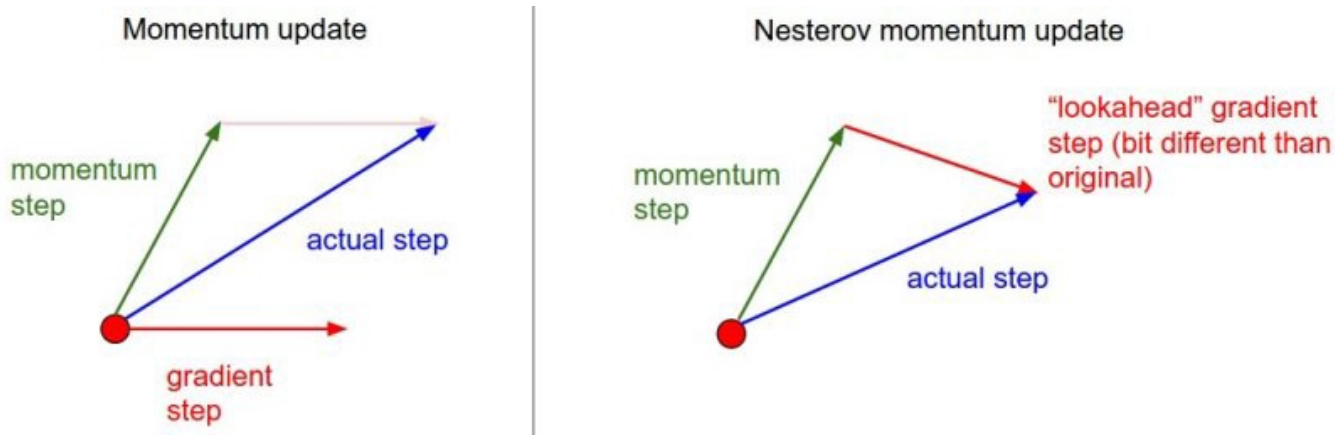
rules, is less intuitive and just omits *(1 - beta)* term.

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W, X, y)$$
$$W = W - V_t$$

This is pretty much identical to the first pair of equation, the only difference is that you need to scale learning rate by *(1 - beta)* factor.

Nesterov accelerated gradient

Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity. In this version we're first looking at a point where current momentum is pointing to and computing gradients from that point. It becomes much clearer when you look at the picture.



Source (Stanford CS231n class)

Nesterov momentum can be define with the following formulas:

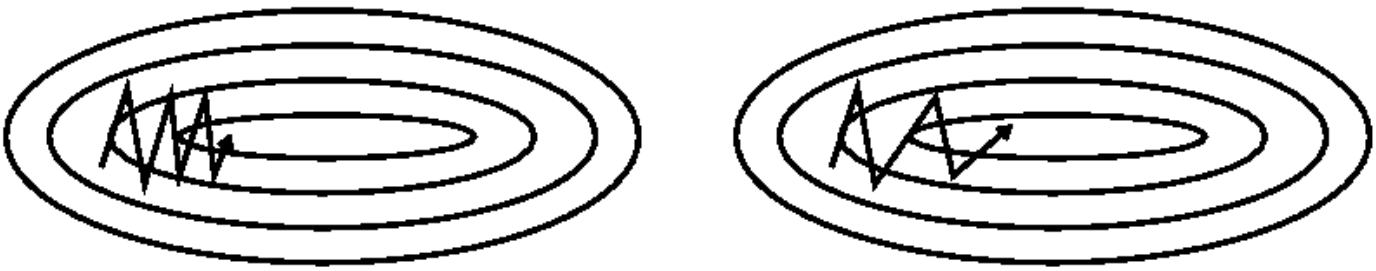
$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W - \beta V_{t-1}, X, y)$$
$$W = W - V_t$$

Why momentum works

In this section I'd like to talk a little bit about why momentum most of the times will be better than classic SGD.

With Stochastic Gradient Descent we don't compute the exact derivate of our loss function. Instead, we're estimating it on a small batch. Which means we're not always going in the optimal direction, because our derivatives are 'noisy'. Just like in my graphs above. So, exponentially weighed averages can provide us a better estimate which is closer to the actual derivate than our noisy calculations. This is one reason why momentum might work better than classic SGD.

The other reason lies in ravines. Ravine is an area, where the surface curves much more steeply in one dimension than in another. Ravines are common near local minimas in deep learning and SGD has troubles navigating them. SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. Momentum helps accelerate gradients in the right direction. This is expressed in the following pictures:



Left — SGD without momentum, right— SGD with momentum. (Source: Genevieve B. Orr)

Conclusion

I hope this post gave you some intuition on how and why SGD with momentum works. It's actually one of the most popular optimization algorithms in deep learning and used even more frequently than more advanced ones.

Below I provide some references, where you can learn more about optimization in deep learning.

References

[1] Ning Qian. On the momentum term in gradient descent learning algorithms. Neural networks : the official journal of the International Neural Network Society, 12(1):145–151, 1999

[2] Distill, Why Momentum really works

[3] deeplearning.ai

[4] **Ruder** (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747

[5] Ruder (2017) Optimization for Deep Learning Highlights in 2017.

[6] Stanford CS231n lecture notes.

[7] fast.ai