Gradient descent with momentum

(DESCRIPTION)
Text, Optimization Algorithms. Gradient descent with momentum. Website, deep learning, dot, A.I.

(SPEECH)
There's an algorithm called momentum, or gradient descent with momentum that almost always works faster than the standard gradient descent algorithm.

In one sentence, the basic idea is to compute an exponentially weighted average of your gradients, and then use that gradient to update your weights instead.

In this video, let's unpack that one sentence description and see how you can actually implement this.

(DESCRIPTION)
New slide, Gradient descent example.

(SPEECH)
As a example let's say that you're trying to optimize a cost function which has contours like this.

So the red dot denotes the position of the minimum.

Maybe you start gradient descent here and if you take one iteration of gradient descent either or descent maybe end up -heading there.

But now you're on the other side of this ellipse, and if you take another step of gradient descent maybe you end up doing that.

And then another step, another step, and so on.

And you see that gradient descents will sort of take a lot of steps, right?

Just slowly oscillate toward the minimum.

And this up and down oscillations slows down gradient descent and prevents you from using a much larger learning rate.

In particular, if you were to use a much larger learning rate you might end up over shooting and end up diverging like so.

And so the need to prevent the oscillations from getting too big forces you to use a learning rate that's not itself too large.

Another way of viewing this problem is that on the vertical axis you want your learning to be a bit slower, because you don't want those oscillations.

But on the horizontal axis, you want faster learning.

Right, because you want it to aggressively move from left to right, toward that minimum, toward that red dot.

So here's what you can do if you implement gradient descent with momentum.

On each iteration, or more specifically, during iteration t you would compute the usual derivatives dw, db.

I'll omit the superscript square bracket l's but you compute dw, db on the current mini-batch.

And if you're using batch gradient descent, then the current mini-batch would be just your whole batch.

And this works as well off a batch gradient descent.

So if your current mini-batch is your entire training set, this works fine as well.

And then what you do is you compute vdW to be Beta vdw plus 1 minus Beta dW.

So this is similar to when we're previously computing the theta equals beta v theta plus 1 minus beta theta t.

Right, so it's computing a moving average of the derivatives for w you're getting.

And then you similarly compute vdb equals that plus 1 minus Beta times db.

And then you would update your weights using W gets updated as W minus the learning rate times, instead of updating it with dW, with the derivative, you update it with vdW.

And similarly, b gets updated as b minus alpha times vdb.

So what this does is smooth out the steps of gradient descent.

For example, let's say that in the last few derivatives you computed were this, this, this, this, this.

If you average out these gradients, you find that the oscillations in the vertical direction will tend to average out to something closer to zero.

So, in the vertical direction, where you want to slow things down, this will average out positive and negative numbers, so the average will be close to zero.

Whereas, on the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big.

So that's why with this algorithm, with a few iterations you find that the gradient descent with momentum ends up eventually just taking steps that are much smaller oscillations in the vertical direction, but are more directed to just moving quickly in the horizontal direction.

And so this allows your algorithm to take a more straightforward path, or to damp out the oscillations in this path to the minimum.

One intuition for this momentum which works for some people, but not everyone is that if you're trying to minimize your bowl shape function, right?

This is really the contours of a bowl.

I guess I'm not very good at drawing.

They kind of minimize this type of bowl shaped function then these derivative terms you can think of as providing acceleration to a ball that you're rolling down hill.

And these momentum terms you can think of as representing the velocity.

And so imagine that you have a bowl, and you take a ball and the derivative imparts acceleration to this little ball as the little ball is rolling down this hill, right?

And so it rolls faster and faster, because of acceleration.

And data, because this number a little bit less than one, displays a row of friction and it prevents your ball from speeding up without limit.

But so rather than gradient descent, just taking every single step independently of all previous steps.

Now, your little ball can roll downhill and gain momentum, but it can accelerate down this bowl and therefore gain momentum.

I find that this ball rolling down a bowl analogy, it seems to work for some people who enjoy physics intuitions.

But it doesn't work for everyone, so if this analogy of a ball rolling down the bowl doesn't work for you, don't worry about it.

Finally, let's look at some details on how you implement this.

(DESCRIPTION)
New slide, Implementation details. Text, On iteration T: Compute D, W, D, B, on the current mini-batch. V, subscript D, W, Second line, equals, Beta, V, subscript D W, plus, 1 minus beta, D W. Third line, V, subscript D B, equals, beta, V, subscript D B, plus 1, minus, beta, D B. Fourth line, W, equals, W, minus, A V, subscript D W, B, equals, B, minus, A V, subscript, D B. Hyperparameters, A, Beta. Beta equals 0.9.

(SPEECH)
Here's the algorithm and so you now have two hyperparameters of the learning rate alpha, as well as this parameter Beta, which controls your exponentially weighted average.

The most common value for Beta is 0.9.

We're averaging over the last ten days temperature.

So it is averaging of the last ten iteration's gradients.

And in practice, Beta equals 0.9 works very well.

Feel free to try different values and do some hyperparameter search, but 0.9 appears to be a pretty robust value.

Well, and how about bias correction, right?

So do you want to take vdW and vdb and divide it by 1 minus beta to the t.

In practice, people don't usually do this because after just ten iterations, your moving average will have warmed up and is no longer a bias estimate.

So in practice, I don't really see people bothering with bias correction when implementing gradient descent or momentum.

And of course this process initialize the vdW equals 0.

Note that this is a matrix of zeroes with the same dimension as dW, which has the same dimension as W.

And Vdb is also initialized to a vector of zeroes.

So, the same dimension as db, which in turn has same dimension as b.

Finally, I just want to mention that if you read the literature on gradient descent with momentum often you see it with this term omitted, with this 1 minus Beta term omitted.

So you end up with vdW equals Beta vdw plus dW.

And the net effect of using this version in purple is that vdW ends up being scaled by a factor of 1 minus Beta, or really 1 over 1 minus Beta.

And so when you're performing these gradient descent updates, alpha just needs to change by a corresponding value of 1 over 1 minus Beta.

In practice, both of these will work just fine, it just affects what's the best value of the learning rate alpha.

But I find that this particular formulation is a little less intuitive.

Because one impact of this is that if you end up tuning the hyperparameter Beta, then this affects the scaling of vdW and vdb as well.

And so you end up needing to retune the learning rate, alpha, as well, maybe.

So I personally prefer the formulation that I have written here on the left, rather than leaving out the 1 minus Beta term.

But, so I tend to use the formula on the left, the printed formula with the 1 minus Beta term.

But both versions having Beta equal 0.9 is a common choice of hyper parameter.

It's just at alpha the learning rate would need to be tuned differently for these two different versions.

So that's it for gradient descent with momentum.

This will almost always work better than the straightforward gradient descent algorithm without momentum.

But there's still other things we could do to speed up your learning algorithm.

Let's continue talking about these in the next couple videos.