[MUSIC] In this video, we'll discuss advanced

optimization techniques that allow to improve gradient descent methods. As we remember from the previous video,

**Problem:***if our function is difficult, for example, it has elliptic*

*level lines like on this graph? Then gradient descent will oscillate,*

*and it will take many iterations for it to converge to the minimum.*

***Solution:***

To improve this, we can somehow

change our gradient descent methods. For example,

take mini-batch gradient descent. In this method, we take some random

m examples from our training set, we approximate the gradient

based on these m examples, and then we'll make a gradient step. So in this video, we'll suppose that we have some way to approximate gradient. Maybe by mini-batch, maybe by stochastic

gradient descent with just one point, or maybe we calculate full gradient. That doesn't matter, we just have some approximation of gradient, and we allot it by **gt**. And in this video, we'll discuss how can we modify the way we make a gradient step, how we add a gradient or antigradient

to our previous approximation, w t-1.

**Method 1 :**

So let's start with **momentum method.** In this method, we maintain additional

vector h at every iteration. To calculate h in the step t, we take this vector h at the step t-1, multiply by some coefficient alpha. And add the gradient at the current iteration, gt, multiplied by the learning rate, eta t. So actually, ht is just a weighted sum of gradients from all previous iteration, and from this iteration, also. And then we just make a gradient step in the direction of -ht. **So what is the point of this method**? Suppose that we have some function, it'll make gradient descent, and maybe for some coordinates of our parameter vector. Gradients always have the same sign,

so they lead us to the minimum. And for some coordinates, the sign of the gradient changes

from iteration to iteration. So, vector ht would be large for component where gradients have the same sign on every iteration, and will make large steps by this coordinates. And for coordinates that change sign, they will just cancel each other and ht will be close to zero. So, ht cancels some coordinates that lead to oscillation of gradients, and help us to achieve better convergence. Here, we should somehow choose parameter alpha, for example, we could just set it to 0.9, and

to somehow choose learning rate, eta t. In practice, this momentum method indeed leads to faster convergence. So for our last function, with elliptic level lines, we'll see picture like this one. So here, we don't oscillate that much as on the previous picture, and we converge better. There is also an extension of momentum method called **Nesterov momentum**. So in simple momentum method,

on every iteration, we calculate the gradient at current point w t-1. We take a gradient step from it by gt, and we then get our momentum, ht. But since it's certain that we'll move

in the direction of the momentum, it will be more clever to, first,step in the direction of ht to get some new approximation of parameter vector. And then to calculate gradient

at the new point, w t-1 plus ht. So in this case, we'll get a better approximation on the next step. Mathematically, we take the current point, w t-1, we subtract alpha multiplied by h t-1, and

calculate the gradient at this new point. And then we add the gradient at this new point, multiplied by the learning rate, to the h from the previous iteration, alpha multiplied by h t-1. And in practice, this method indeed leads to better convergence than momentum method. So momentum method and Nesterov momentum method work better with difficult functions with complex level sets. But they still require to choose learning rate, and they're very sensitive to it. So now we'll discuss some other

optimization methods that try to choose learning rate adaptively, so we don't have to choose it ourselves. One of such methods is AdaGrad. So let's remember how we make

a gradient step, just for one coordinate g of parameter vector w. To make a gradient step,

we take j-th component of parameter vector from the previous iteration,

is w t-1 j. And we subtract j-th component of

the gradient at the current point, to find out the next parameter

approximation, w t j. In AdaGrad,

we've obtained additional value for each parameter from all

our parameter vector, G. So to calculate it, we take G t-1,

G from the previous iteration. We take j-th component of it, and we just add a square of gradient

at the current iteration. So essentially, G is a sum of squares of

gradients from all previous iterations, and then we modify our gradient step. We divide our learning rate, eta t, by the square root of G t j,

plus some small number, epsilon. We add epsilon, just to make sure that

we don't divide learning rate by zero. So this AdaGrad method,

it chooses learning rate adaptively. Here, we can just set eta t,

our learning rate, to some constant, for example, 0.01, and

don't remember about it at all. So this doesn't need to somehow wisely choose learning rate, but

it also has some disadvantages. Our auxiliary parameter, G,

accumulates squares of gradient, and at some step it can become too large. We'll divide our learning rate,

eta t, by a large number, and gradient descent will stop building. So to somehow overcome this,

we need some other methods like AdaGrad. By the way, AdaGrad has another advantage, it chooses its own learning rate for

each example. So suppose that we are analyzing texts,

and each feature from our sample

corresponds to one word. So for some frequent word that

we see in every document, we have gradient updates on every step,

and we'll make smaller steps. And for some rare words that we

can met only in one of thousand or ten thousand documents,

we'll make large updates, because they are rare,

we don't need them very often. And we need to move faster in

the direction of these words. Another matter that can

improve AdaGrad is RMSprop. This method is very similar to AdaGrad,

but here we take an exponentially weighted average of squares of

gradients on every step. So here, to calculate G j at the step t,

we take G j from the previous step, t-1, we multiply it by

some coefficient alpha. And then we add the square of

the gradient, G t j, at this iteration, multiplied by 1- alpha. And then we use this G to divide our

learning rate, just like in AdaGrad. So this method overcomes the problem

of large sums of square gradients. And here, our learning rate depends only on last

examples from our gradient descent method. Let's start with RMSprop method and

slightly modify it. So in RMSprop,

we maintained an additional variable, and it will be augmented by v t j. And is just an exponentially weighted

sum of gradients from all iterations. So here, to calculate v for

j-th component at the step t, we take v from the previous step, v j t-1,

multiplied by some coefficient beta 2. And we add square of gradient

in the current iteration, g t j squared, multiplied by 1- beta 2. And we can notice, that this

approximation, v has some bias towards zero, especially at first steps,

because we initialize it with zero. So to overcome this, we just divide

it by 1- beta 2, in the degree of t. So this normalization allows

us to get rid of this bias. At first steps,

this normalization is large, and for large t's,

this normalization almost equals to 1. And then we use this v to divide

our learning rate, eta t. But as we go from momentum method, an approximation of gradient from one step

can be noisy and lead to oscillations. So let's smooth our gradients. To do it,

we maintain another auxiliary variable, m, that is essentially a sum of gradients. Not squares of gradients,

like in v, but just gradients. And it's calculated in the same way, it's just an exponentially weighted

average with coefficient beta 1. And then we replace g, the gradient

approximation in our gradient step, by m, by weighted average of gradients

from all previous iterations. So this method, Adam,

combines both momentum methods and adaptive learning rate methods. And in practice, it achieves better

convergence and faster convergence. In this video,

we discussed some advanced optimization techniques that can

improve gradient decent. For example, momentum method and

Nesterov momentum method, that allows us to work with some complex

functions with complex level sets. And AdaGrad, RMSprop method,

that have adaptive learning rates, so we don't have to choose eta t manually. And there is also an Adam method,

that combines both of this methods. [MUSIC]