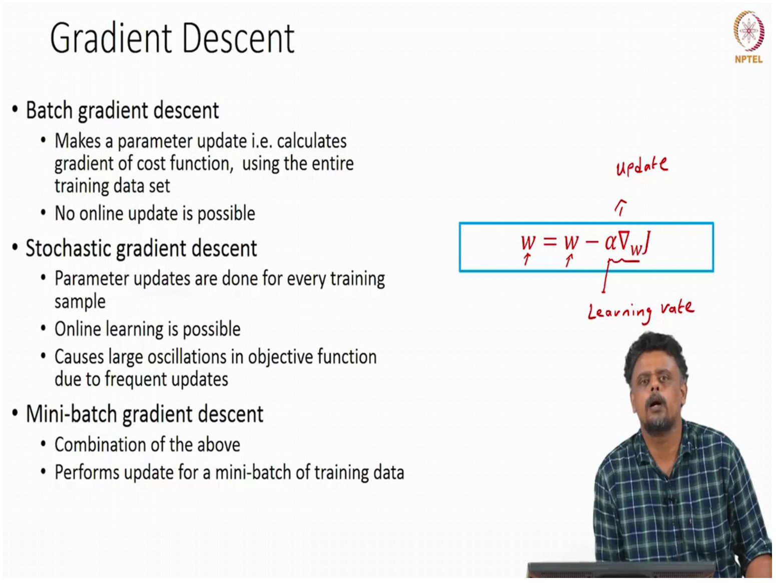
Machine Learning for Engineering and Science Applications

Professor Dr Ganapathy Krishnamurthi
Department of Engineering Design
Indian Institute of Technology, Madras

Gradient Descent Algorithms

(Refer Slide Time: 0:18)



Gradient Descent

- Batch gradient descent
  - Makes a parameter update i.e. calculates gradient of cost function, using the entire training data set
  - No online update is possible
- Stochastic gradient descent
  - Parameter updates are done for every training sample
  - Online learning is possible
  - Causes large oscillations in objective function due to frequent updates
- Mini-batch gradient descent
  - Combination of the above
  - Performs update for a mini-batch of training data

$$\boxed{w = w - \overset{\text{update}}{\underbrace{\alpha}_{\text{Learning rate}}\nabla_w J}}$$

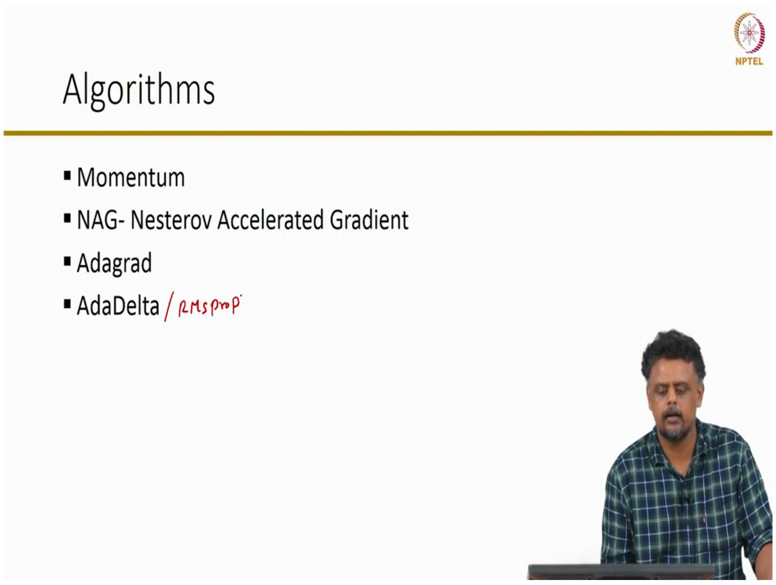
Hello and welcome back, in this video will look at some alternatives to gradient descent algorithms. Just to refresh your memory, we have looked at gradient descent techniques for optimisation. So there are basically 3 different versions, one is called the batch gradient descent, where the parameter update is made based on the entire training dataset. So you would calculate an average gradient for based on the individual grades that you calculate for your training data points.

The other extreme is the stochastic gradient descent where you would update the parameter for every individual training data points. Your online learning is possible because of can update your parameter as soon as a new data point arrives. But this method causes some large oscillations in your objective functions, as well as your parameter updates. To get the best of both worlds, what is typically done is minimise the gradient descent, which is a combination of the above, that you take a subset of, take subsets of your training data and then calculate the average gradient and use that to obtain the parameters.

So, for reference we have given the gradient descent update equation here, so the current parameter estimate is the previous parameter estimate plus the update, so this is the update. Update is your gradient, with respect to the parameters, multiplied by the alpha which is called the learning rate. So, what will do now is to go through some of the variants of this gradient descent algorithm. So, primarily these have been, these have evolved primarily by looking at how we can make networks converge faster, which means deep learning networks converge faster to the optimal solution.

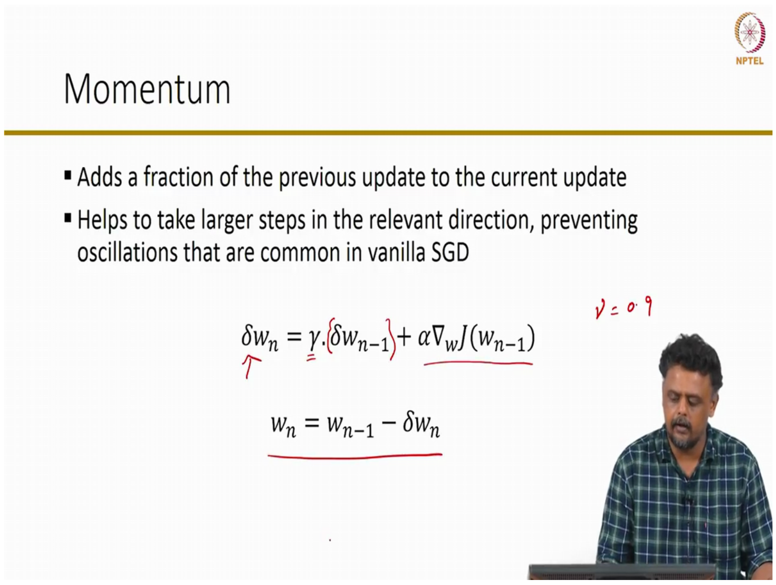
Almost all of the algorithms are, you can use them like black boxes in most of the packages that they introduced last time. So in that sense you do not have to really have to code them but you just have to understand how they work and try out different things for your particular implementation of a deep learning technique or machine learning techniques. So for instance tensor flow or pie torch will have many of their algorithms, we will see, we will have them already coded, and will just have to use that option.

(Refer Slide Time: 2:53)



Algorithms

- Momentum
- NAG- Nesterov Accelerated Gradient
- Adagrad
- AdaDelta / RMSprop



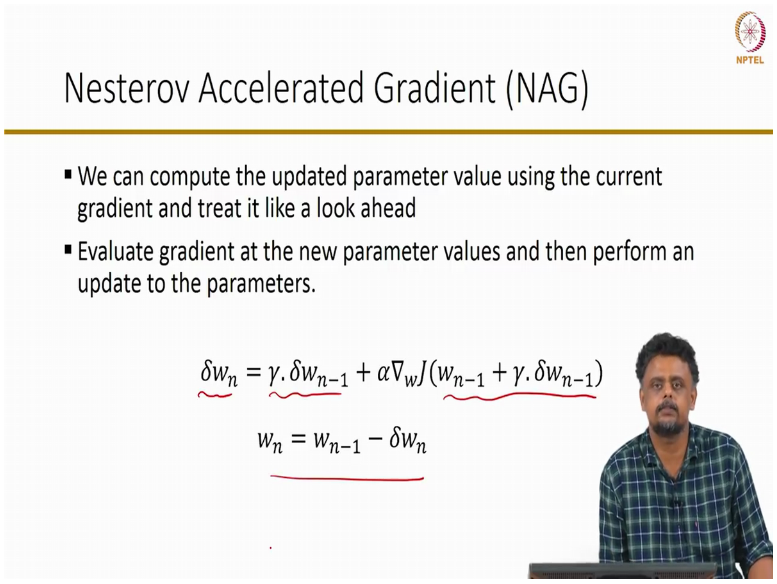
Momentum

- Adds a fraction of the previous update to the current update
- Helps to take larger steps in the relevant direction, preventing oscillations that are common in vanilla SGD

$\gamma = 0.9$

$$\delta w_n = \gamma . \delta w_{n-1} + \underline{\alpha \nabla_w J(w_{n-1})}$$

$$\underline{w_n = w_{n-1} - \delta w_n}$$



Nesterov Accelerated Gradient (NAG)

- We can compute the updated parameter value using the current gradient and treat it like a look ahead
- Evaluate gradient at the new parameter values and then perform an update to the parameters.

$$\underset{\sim}{\delta w_n} = \gamma.\underset{\sim}{\delta w_{n-1}} + \alpha\nabla_w J(\underset{\sim}{w_{n-1} + \gamma.\delta w_{n-1}})$$

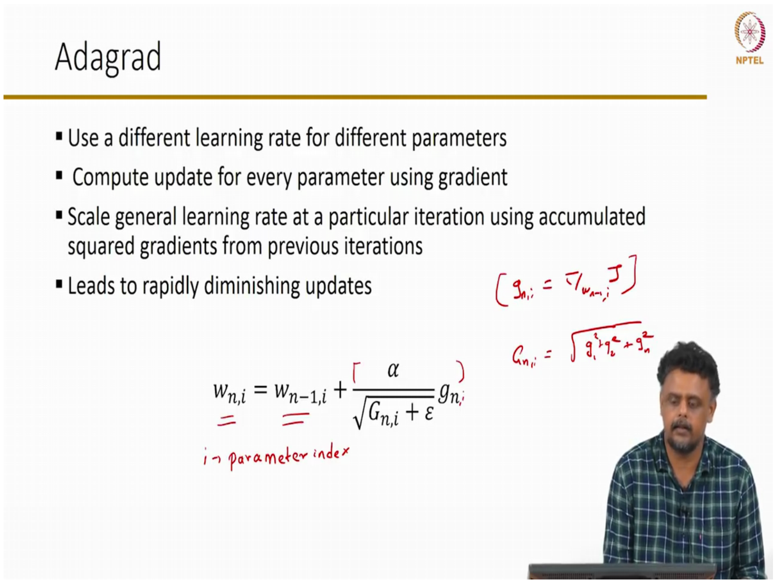
$$\underline{w_n = w_{n-1} - \delta w_n}$$

So these are some of the methods that we will see, momentum-based, which will, then the Nesterov accelerated gradient, Adagrad, AdaDelta, also RMS prop, very similar initially, okay. So we will just start off with the momentum update. So what this variant does is to add a fraction of the previous update to the current update, okay. So which means that it will take a larger step in the relevant direction, so that we preventing oscillations and converging faster. So we saw that, this Delta is the update and it is proportional to, this is update here, this is the update from the previous step.

When you take the fraction of this update, so gamma is typically 0.9 around that value, then you added to the, then you add it to your current update right here. So that way you take our bigger step towards the relevant direction, of course and this is how you would actually update your parameters. So your current update right here has the usual update equations, this is Alpha times the gradient with respect to the current value of the parameters, Wn minus1. And your previous update, the fraction of your previous update is added to it. So this is the moment update equation.

The Nesterov accelerated gradient goes a little bit further, so what it does is to, if you compute the update parameter value, right, so you calculate the update and then you add it to the previous iteration value. And you treat that like a lookahead and you evaluate the gradient at the look at points, okay. So this is the same update equation, so this is the Delta W1 is the update to your parameter. So you can have, just like the moment about it, you have a piece of this previous update, plus, so instead of, so in the previous step, version, momentum update version, you calculated J with respect to Wn minus1, here you calculate J with respect to the lookahead, this is a lookahead parameter, okay.

(Refer Slide Time: 5:25)



Adagrad

- Use a different learning rate for different parameters
- Compute update for every parameter using gradient
- Scale general learning rate at a particular iteration using accumulated squared gradients from previous iterations
- Leads to rapidly diminishing updates

$$\left[g_{n,i} = \nabla_{w_{n-1,i}} J \right]$$

$$G_{n,i} = \sqrt{g_1^2 + g_2^2 + g_n^2}$$

$$\underline{\underline{w_{n,i}}} = \underline{\underline{w_{n-1,i}}} + \left[\frac{\alpha}{\sqrt{G_{n,i} + \varepsilon}} g_{n,i} \right)$$

$i \rightarrow$ parameter index

So and then you perform the usual update right here. So, this we it helps you get better estimates of your parameter updates. So the next version is basically where you try to have different learning rate for different parameters. So, if you notice, in all the momentum as well as the Nesterov accelerated gradient method, you had the same learning rate alpha for all the parameter. It is just that you either took the previous version of the, previous updates as well as the lookahead to get a better estimate for the current update.

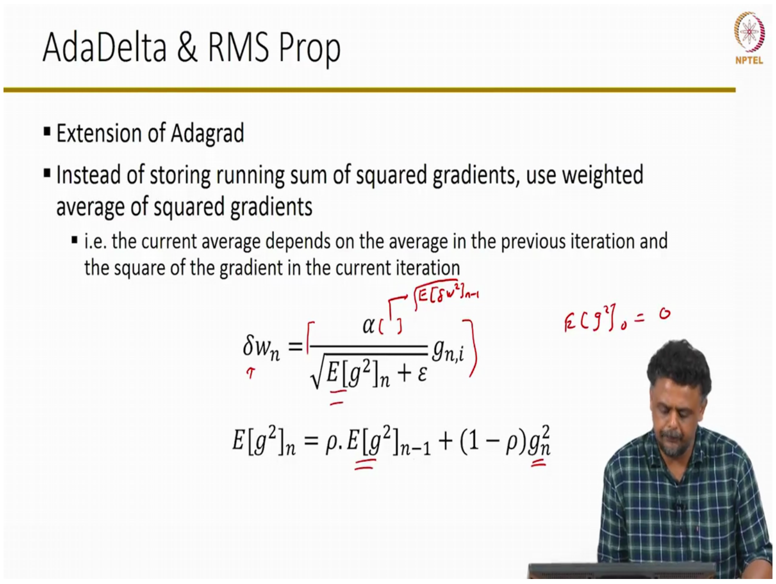
So, here what we do is we will have a different learning rate for different parameters. So this is the equation, so your current parameter is, of course your previous parameter plus this update, I will clarify the indices. So i is your parameter index, right, so you might have hundreds of millions of parameters. So you would consider each parameter at that time, so the current parameter is the previous, from the previous iteration n minus1 plus this update equation.

The alpha is your general base learning rate and for each parameter you will have a specific, so here again I missed out this index here, implicit Gni. So what is this term in the square root and what is the Gni. So, Gni is your typical, that is your gradient, okay, it is your gradient, Gni is the gradient at the current step, so n. And the square root term, this Gni is the sum of squares of the gradient. So, this is G1 square plus G2 square plus G3 square plus G n square, okay, for that particular, of course I've left out the index i in this, but for that particular parameter.

So you take the gradient with respect to that specific parameter index i and you accumulate the gradient, the squares of the gradient and take the square roots. And then to normalise your base at learning rate alpha by that term. So, what it does is that if you have some parameter which is getting updated frequently, some parameters will have very small or negligible updates, so some parameters will have large and frequent updates. So, then we just make sure it is just it is normalised, okay, the updates are normalised with this particular running average.

So the disadvantage is that as soon as the number of iterations increase, the denominator will go really large and of course your updates will become very small, okay. So G is just a shorthand for your usual gradient, this is gradient with respect to the current value of the parameters. The capital G is the sum of the squares of the gradient from the previous iterations up to the current iteration. So that is the Adagrad updates.

(Refer Slide Time: 8:22)



AdaDelta & RMS Prop

- Extension of Adagrad
- Instead of storing running sum of squared gradients, use weighted average of squared gradients
  - i.e. the current average depends on the average in the previous iteration and the square of the gradient in the current iteration

$$\delta w_n = \left[\frac{\alpha\,(\overset{\sqrt{E[\delta w^2]_{n-1}}}{\,})}{\sqrt{E[g^2]_n + \varepsilon}}\, g_{n,i}\right)$$

$E[g^2]_0 = 0$

$$E[g^2]_n = \rho . E[g^2]_{n-1} + (1-\rho) g_n^2$$

And AdaDelta and RMS prop, the basic version is still the same, except that instead of taking the running some of the gradients, you took a weighted sum or exponentially decaying sums, so that it doesn't, other than not it doesn't blow. So it is, this is a similar formula, the update is very similar to what we saw earlier, except that for this term, instead of the capital G, which is just a running sum of the gradients, you will have a weighted running sum, okay.

So this is the gn square is the current gradient, gradient at the current step and of course it is weighted by this factor 1 minus rho, rho is something between 0 and 1, plus the weighted sum of the gradients from the till the previous. So, you can start from E of g square, at the 1st step this should be 0, right. So, you would not complete the gradients but then you would weight it every time with this rho. So this way you won't have the problem we had earlier with Adagrad wherein we had the running sums of variants accumulating, becoming very large number and the parameter updates become very small.

Here you have a weighted sum, in this case you have a weight for the current square gradient and then you have a weight for the previous, the sum of the gradient in the previous step. So you can weight it like that and that way you will have a very decaying sums, and exponentially decaying weighting average. So this method is again, 2 methods, AdaDelta and RMS prop, they're very similar, okay, except that AdaDelta also have, so another factor in the numerator, which is similar to this.

Instead of the gradient, so here you will have the expectations of the or the weighted average of the updates, right. So this will, this is done to make sure that, of course the square root, I missed out on the square root, this is done to make sure that the units match. So if you look at this particular equation, the let us say that the W's have a particular unit, right, some dimensions, and in this case they do not match, right. Because the gradient dimension and here are the square root of the gradient square, they will cancel out and alpha is there some constant.

And so in 2 for the units to match, you will have this learning average, so it helps match that, that is one application. RMS prop does not have this, so that is the difference, okay. So, there are many other techniques also, very similar comedy something called Add M, which I have not described here, there are lots of references online, you can look them up. So all of these are quite very popular choices for optimising deep neural networks as well as general CNN and things like that.

Many of them are available as blackbox implementations in many of the software platforms like tense flow, pie torch, or even MATLAB, so you are welcome to go try them out when you begin to coding your own deep neural networks. So, this includes our session on gradient descent variants, okay thank you.