

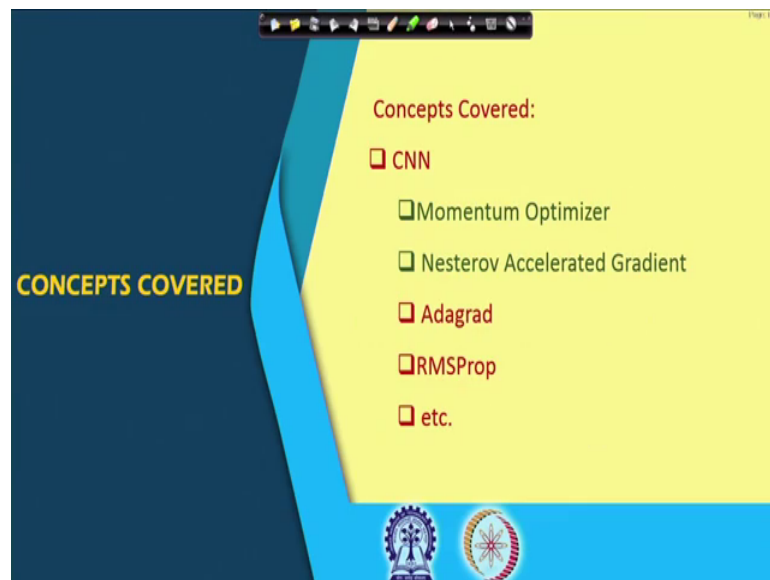
Deep Learning
Prof. Prabir Kumar Biswas
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture – 44
Optimisers: Adagrad Optimiser

Hello, welcome back to the NPTEL online certification course on Deep Learning. So, for last few lectures we are discussing about the optimization procedures of gradient search technique. And as you know that gradient search itself is an optimization algorithm or gradient descent is itself is an optimization algorithm, which tries to minimize the error function by updating the weight vectors in deep neural network. But this gradient descent algorithm faces a kind of challenges like a vanishing gradient problem, then slow learning rate.

So, in order to solve these algorithms, there are different optimization techniques which have been suggested. And the aim of such optimization techniques is to make the gradient descent algorithm or learning algorithm make it faster make it more efficient.

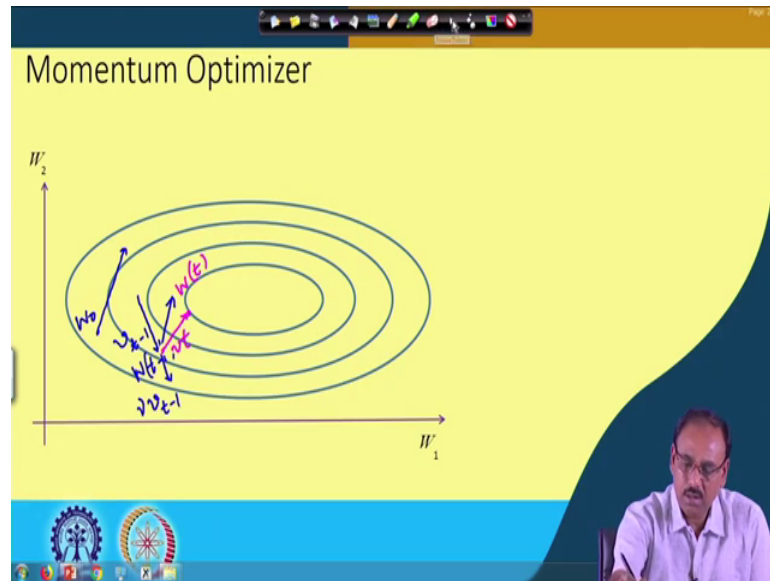
(Refer Slide Time: 01:33)



So, in our previous class, we have talked about two such optimization algorithms, one of them was momentum optimizer and the other one was Nesterov Accelerated Gradient or NAG. In today's lecture, we will try to discuss about what are the drawbacks of momentum optimizer as well as the drawbacks of Nesterov Accelerated Gradient

techniques. And we will see that how those drawbacks can overcome in other algorithms which have been suggested like Adagrad, RMS prop there are other algorithms like adder delta, then adam. So, we will discuss these algorithms one after another and we will try to see that what is the relative performance of these algorithms when you come across different challenging error surfaces.

(Refer Slide Time: 02:37)



So, before we go to Adagrad, let us try to see that or try to recapitulate what we did in our previous class. So, in the previous class, we have talked about the momentum optimizer technique which obviously at enhancement which is an improvement over simple gradient descent algorithm. So, what is this momentum optimizer?

You find what he said that if we have an error surface whose curvature or the rate of gradient in some dimension is very high, whereas in some other dimension the curvature is very low. Or, in other sense that if I take the gradient in one dimension and the gradient in another dimension in one of the dimension that gradient is very high, whereas in another dimension the gradient is very low. In such case the gradient descent algorithm of the base gradient descent algorithm finds a difficult to navigate to the minimum error point. So, in momentum optimizer that is what is being addressed.

So, let us see what we discussed in previous class the momentum optimizer, let us have a quick recapitulation of that. So, what you have in momentum optimizer is suppose I have initialized the weight vector somewhere over here. So, simple gradient descent algorithm

what I will do is, it will find out the gradient of the error at this initial position let us call it to the initial position W_0 , in the vertical direction or in the direction of W_1 as well as in the direction of W_2 . And as you see over here its gradient direction in the vertical direction will be very high, whereas so, you find that the gradient direction in the vertical direction will be very high, whereas the gradient direction in the horizontal direction will be very low.

As a result what will happen is that gradient descent algorithm will try to adjust or we will try to update the weight vector in such a way that the weight vector moves in this direction, whereas, you find that the minimum error is somewhere over here. So, we are going we are not moving in the right direction to minimize the error. In the same manner in the next iteration, the weight vector will be moving in this direction, then the weight vector will be moving in this direction, then the weight vector will be moving in this direction and so on. So, as a result that you are find out finally you find that you are oscillating around the right path which should have been followed, because over here it would have been correct to move in this direction not to oscillate like this.

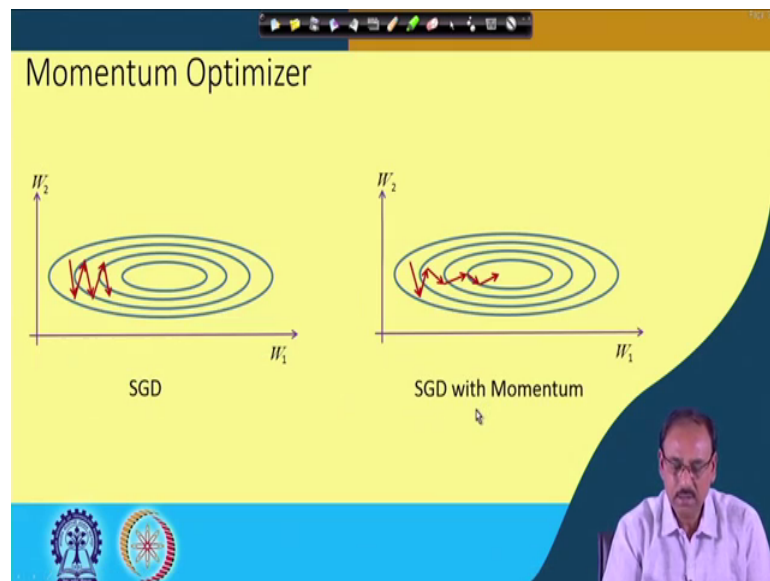
So, this is where the momentum comes to the rescue. So, in case of momentum, what you would do is you again find out the gradient at this location. So, this is my position W_0 at time 0. And suppose at time W_1 the weight vector has come somewhere over here. So, this was the movement at time W_{t-1} , suppose the weight vector has come somewhere over here. So, this is what I have W_{t-1} .

And while coming over here the previous shift or the previous update was in this direction. So, what momentum does is, now momentum computes two terms one is its movement or updated due to the momentum because of the previous movement, and the gradient at location W_{t-1} . So, if you do that you find that in the momentum as before will be somewhere over here suppose the previous upgrade value was v_{t-1} . So, under the influence of this momentum, now this update direction will be say γv_{t-1} that is what is the gradient component. And coming to this is what is the momentum component and coming to the gradient component that gradient will be somewhere in this direction.

So, because of this under the influence of these two, the gradient and the momentum, the net displacement or the net updation of the weight vector will be in this direction which

is actually the sum of the momentum term and the gradient term, and this is what is your v_t . And under influence of this your weight vector is moving over here which is W_t . So, you find that in absence of the momentum term the weight was moving in this direction, whereas with momentum the weight is moving in this direction. So, it has shifted slightly towards the minimum value. So, this is the advantage of momentum optimizer.

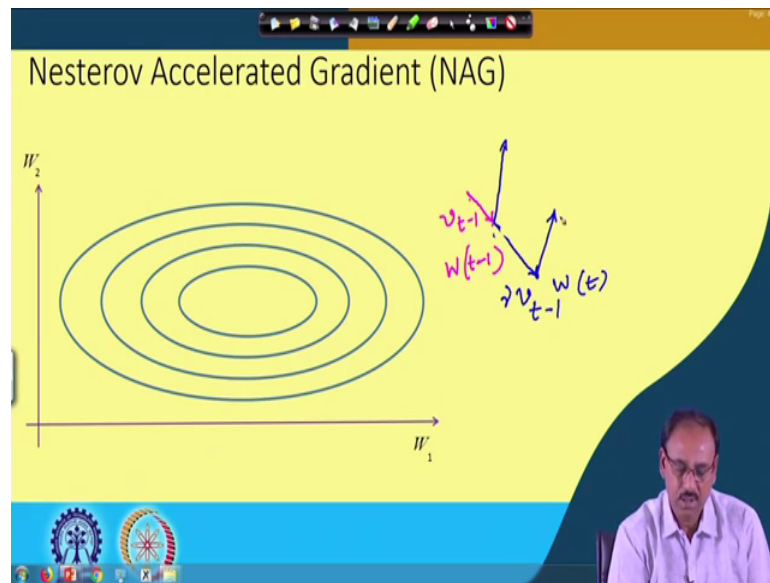
(Refer Slide Time: 08:22)



Now coming to; so under the influence of this if I simply have the stochastic gradient descent optimization algorithm appear stochastic gradient descent gives you the updates which is as given over here. Whereas, if I have the stochastic gradient with momentum, the object moves will be something like this. So, you will find that this momentum term actually is improving the learning rate or the algorithm will learn faster and will move towards the minimum error value for in a faster way.

Now, once we have this momentum then what Nesterov Gradient Accelerated gradient does is, it computes the gradient not at the location of W_t , but at location $W_t - \alpha \cdot v_t$. But at location $t - 1$ based on the momentum it tries to find out what will be its future position or it tries to look ahead, and at that future location it computes the gradient value.

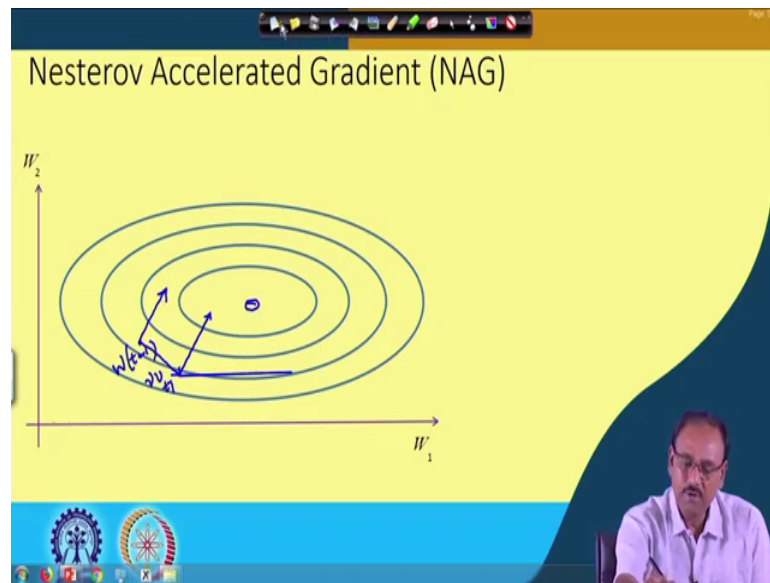
(Refer Slide Time: 09:36)



So, the operation is something like this. Say for example, if we are at this location at say time $W t$ minus 1, say at time $W t$ minus 1, this was the position of the weight vector. And to come to this position it had made the previous update as $v t$ minus 1. So, under the influence of momentum, now the movement or the future position which can be computed is say somewhere over here. This will be say γ times $v t$ minus 1. And only under the influence of this momentum, the position of the weight vector at time t will be over here which is $W t$.

So, if I simply use the momentum with gradient, gradient descent with momentum what will be computed is the gradient term will be computed over here, and the sum of these two would have been the update in the weight vector. But in Nesterov Accelerated Gradient descent instead of computing the gradient term here that gradient term is computed over here, and your movement or the update is in that direction of gradient at this location. So, you find that when you add the when you have the look ahead gradient as in the accelerated gradient algorithm, you are moving faster to your minimum error position.

(Refer Slide Time: 11:47)



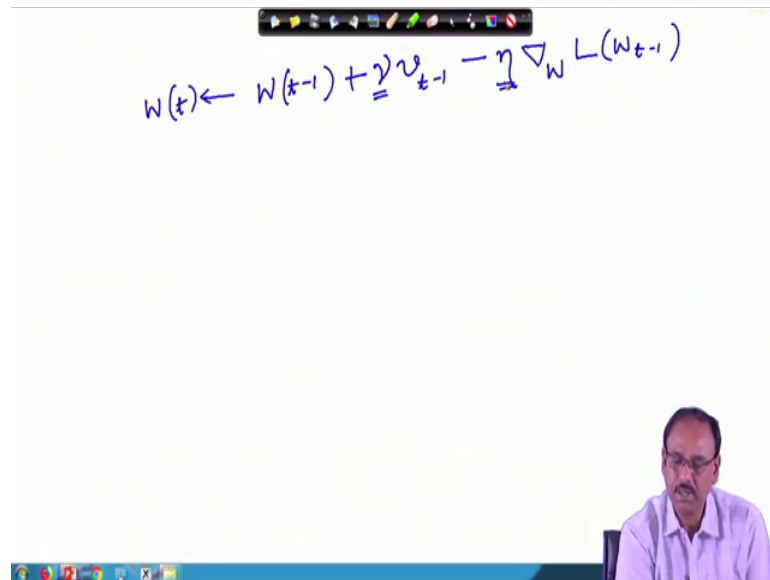
So, now if I illustrate on this diagram, suppose your position at time $W_t - 1$ over here, so this was $W_t - 1$ and this was your the momentum term which is γv_{t-1} , then you are computing the gradient at this location. So, you are moving faster towards the minimum error value. In the previous case, if you are if you do not use this accelerated gradient, the gradient would have been computed over here, and the movement would have gone in this particular direction and that is for how the weight would have been updated. So, this Nesterov Accelerated Gradient further improves the momentum optimization.

(Refer Slide Time: 12:42)

- Both the algorithms require the hyper-parameters to be set manually.
- These hyper-parameters decide the learning rate.
- The algorithm uses same learning rate for all dimensions.
- The high dimensional (mostly) non-convex nature of loss function may lead to different sensitivity on different dimension.
- We may require learning rate be small in some dimension and large in another dimension.

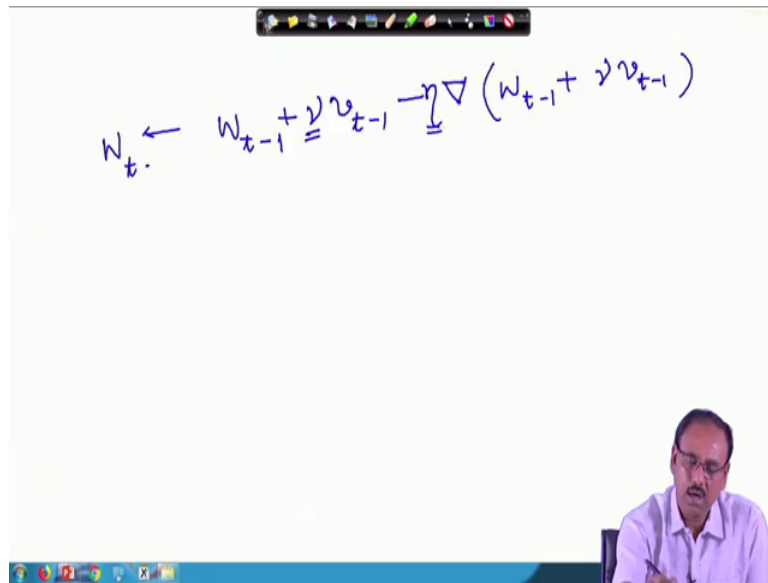
However, both of these momentum optimizer and NAG optimizer suffers from certain problem. So, what are the problems the problems are in case of both the algorithms, we required in the hyper-parameters, and this hyper-parameters are to be set manually. So, what are these hyper-parameters let us try to see that what these hyper-parameters are.

(Refer Slide Time: 13:05)


$$W(t) \leftarrow W(t-1) + \gamma v_{t-1} - \eta \nabla_W L(W_{t-1})$$

In case of momentum optimization, how we have optimized the weight your W_{t+1} or let me put it as W_t . W_t was W_{t-1} plus we had the momentum term, momentum term was γv_{t-1} minus we had this gradient term which is some η times gradient of my loss function L , where the law gradient is computed with respect to weight W and it is computed at location at W_{t-1} . So, I have two hyper-parameters, one is this γ and the other one is η . So, these two hyper-parameters are to be set manually in case of momentum optimizer.

(Refer Slide Time: 14:20)



The whiteboard contains the following handwritten equation:

$$W_t \leftarrow W_{t-1} + \gamma v_{t-1} - \gamma \nabla (W_{t-1} + \gamma v_{t-1})$$

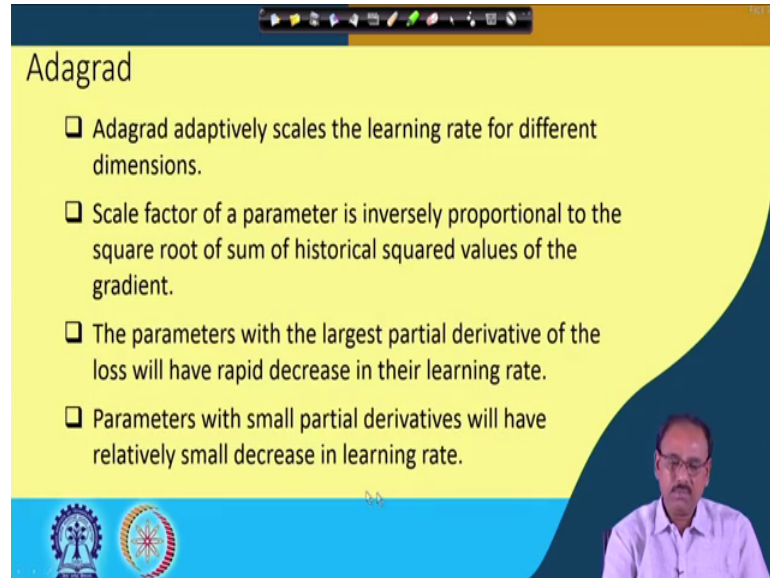
And what do you get in case of nesterov gradient descent, in case of nesterov gradient descent the W_t minus or W_t gets, W_{t-1} plus we have the same momentum term γv_{t-1} minus now you take the gradient, but the gradient not at location W_{t-1} , but you are taking gradient at $W_{t-1} + \gamma v_{t-1}$, and I have the rate of convergence which is η . So, here again I have these two hyper-parameters which are preset manually. So, this is one of the drawbacks of both the momentum optimizer as well as Nesterov Accelerated Gradient algorithm.

The these hyper-parameters actually decide what is your learning rate. If the hyper-parameters are too high, the learning rate will be fast; and if the hydrometers are too low, the learning rate will be slow. And it is the same learning rate both in case of momentum as well as in case of Nesterov Accelerated Gradient is the same learning rate which is used in all the dimensions. But in actual scenario in high dimension and non-convex nature of loss function, it is quite possible that your loss function will be more sensitive in certain direction, and it will not be so sensitive in some other dimension. Or the gradient in some dimension will be quite high and the gradient in some other dimension may be very low.

So, as a result your algorithm may require that learning rate be small in the direction of higher gradient, and it may be high in the direction of lower gradient. So, these are the different problems of momentum as well as Nésterov Accelerated Gradient technique.

And such problems are actually addressed in allow another algorithm which is known as Adagrad. So, what is this Adagrad algorithm?

(Refer Slide Time: 16:47)



Adagrad

- ❑ Adagrad adaptively scales the learning rate for different dimensions.
- ❑ Scale factor of a parameter is inversely proportional to the square root of sum of historical squared values of the gradient.
- ❑ The parameters with the largest partial derivative of the loss will have rapid decrease in their learning rate.
- ❑ Parameters with small partial derivatives will have relatively small decrease in learning rate.

The Adagrad algorithm it tries to adaptively scale the learning rate in different dimensions. So, now the learning rate is not same in all the dimensions, the learning rate in the dimension say in the direction in the dimension of W_1 . We have seen earlier that in the vertical direction the gradient was very high when we were explaining the momentum optimizer technique. We had seen that that gradient in the vertical direction is very high and that gradient in the horizontal direction was very low, as a result your learning rate in the vertical direction was high and the learning rate in the horizontal direction was low. And that is the reason when you update the weight vectors, this updation technique was I mean the updated weight vectors was actually oscillating around the correct path that should have been taken.

So, in case of Adagrad, it can adaptively tune the learning rate in different directions. So, in the vertical direction, it will try to reduce the learning rate, whereas in the horizontal, in the horizontal direction, it will try to enhance the learning rate. And the scale parameter or the scale factor to scale the learning rates in different dimensions, they are actually proportional to the square root of sum of historical squared values of the gradient. We come to see come to come to this when we explain this algorithm with the help of what are the mathematical equations that we have.

And as a result the parameters which have the largest partial derivative, because it is being scaled with the sum of squares of historical gradient values, so the parameters that have very large partial derivatives of the loss function along those directions the learning rate will decrease rapidly. And the parameters with small partial derivatives along those directions, the learning rate will the rate of decrease of learning rate will be very small, so that is how adaptively the Adagrad algorithm can scale the learning rate in different dimensions.

(Refer Slide Time: 19:13)

Adagrad

$$g_t = \frac{1}{n} \sum_{X \in \text{Minibatch}} \nabla_W L(W_t, X) \quad r_t = \sum_{\tau=1}^t g_\tau \circ g_\tau$$

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\epsilon + r_t}} \circ g_t$$

$\circ \rightarrow$ element-wise product

So, let us see what this Adagrad algorithm is. So, again in Adagrad algorithm, you have to compute the gradient at location W_t , where X is your input vector. And because we are talking about the batch gradient descent algorithm, we are trying to improvise upon batch gradient descent algorithm. So, whatever we are computing, all these computations we are with respect to the batch of training samples that you have.

So, if I have say n number of samples in a mini batch used for training the neural network, the deep neural network, the gradient at time instant t when my weight vector is at time W_t . So, I have to compute what is the loss and I have to take the gradient of this loss with respect to weight vector and this gradient has to be computed over all the training samples in the mini batch, and you have to take the average of those gradients which are computed over all the training samples with respect to the weight at time W_t .

So, this is the gradient you are computing and this gradient is say g_t , whereas I said that g_t is computed over the mini batch at time instant t or with weight vectors as W_t . And then you compute the square of the different components of this gradient and then you sum them up. And this summation has to be done over τ is equal to one to the current time instant, which is t . This particular computation, so g_τ indicates of the variable t the subscript t has been replaced by τ indicating that it is my index over which the summation has to be computed. And this particular notation g_τ , then the small g_τ , and then this small circle over here, the small circle indicates the component wise multiplication.

(Refer Slide Time: 21:42)

$$\begin{matrix} A \\ \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \end{bmatrix} \end{matrix} \quad \begin{matrix} B \\ \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_d \end{bmatrix} \end{matrix} \quad \begin{matrix} \circ \\ = \end{matrix} \quad \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_d b_d \end{bmatrix}$$

Or in other words what I mean by that is say if I have two vectors A and vector B where vector A has components the $a_1 a_2$ up to say a_d these are the components, and vector B also has components $a b_1 b_2$ up to say b_d , where d is the dimensionality of the vectors, then this operation component wise multiplication it is nothing, but $a_1 b_1 a_2 b_2 a_d b_d$. So, this is what this notation indicates.

So, given this now let us go back. So, this r_t it actually accumulates the sum of the squares of component wise multiplications of the products, and it is accumulated over time starting from t equal to 0 to t equal to t . And when you are updating the weight vector, your updation equation is W_{t+1} is equal to W_t minus η times g_t upon $\epsilon I + r_t$. So, this I is a vector where all the components of the vector is equal to

one and this r_t is the vector which is given by this. So, again when I am in showing this particular expression that square root of sigma or square root of epsilon I plus r_t , this actually means square root of epsilon plus r_1 for t equal to 1, it means square root of epsilon plus r_2 and so on.

(Refer Slide Time: 23:45)

Adagrad

$$\begin{bmatrix} W_{t+1}^{(1)} \\ W_{t+1}^{(2)} \\ \vdots \\ W_{t+1}^{(d)} \end{bmatrix} = \begin{bmatrix} W_t^{(1)} \\ W_t^{(2)} \\ \vdots \\ W_t^{(d)} \end{bmatrix} - \begin{bmatrix} \frac{\eta}{\sqrt{\epsilon + r_t^{(1)}}} g_t^{(1)} \\ \frac{\eta}{\sqrt{\epsilon + r_t^{(2)}}} g_t^{(2)} \\ \vdots \\ \frac{\eta}{\sqrt{\epsilon + r_t^{(d)}}} g_t^{(d)} \end{bmatrix}$$

Handwritten notes on the right side of the slide:

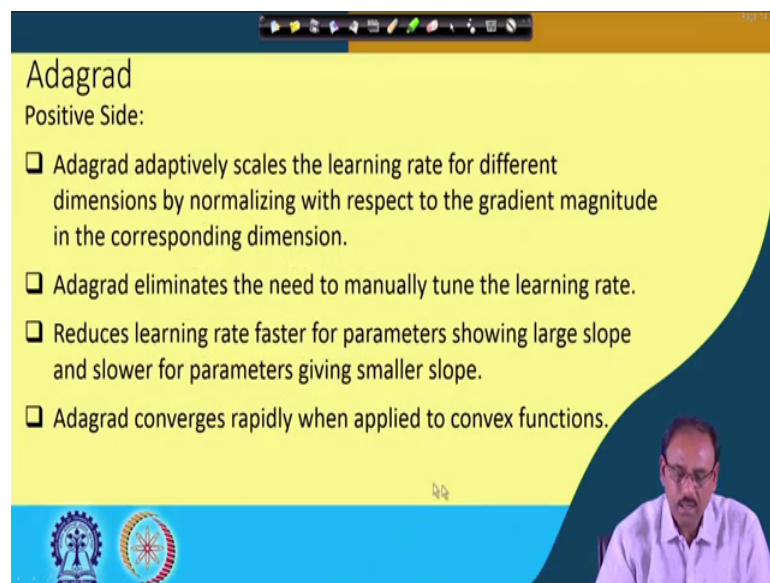
- $W_{t+1}^{(i)}$ with an arrow pointing to the first component of the update vector.
- $W_t^{(i)} - \frac{\eta}{\sqrt{\epsilon + r_t^{(i)}}} g_t^{(i)}$ with a bracket under the fraction.

So, going by this your actual updation when I expand this equation the actual updation equation is given by this. So, you find that $W_{t+1}^{(1)}$ is actually being updated as $W_t^{(1)} - \eta \cdot g_t^{(1)} / \sqrt{\epsilon + r_t^{(1)}}$. So, it is only the first component of your weight vector which has been updated and the scale factor of the learning rate or the first component of the squared sum of the gradients is being scaled by square root of epsilon plus $r_t^{(1)}$ ok, where this $r_t^{(1)}$ is the sum of squares of the gradients and you take the first component of it.

So, your gradient of; the gradient of the first component is actually scaled by epsilon plus sum of the squares of the gradients of the first component and you take the square root of this. So, similarly if I go for updation of the i th component say $W_{t+1}^{(i)}$. So, if I write it this way $W_{t+1}^{(i)}$, the i th component this will be updated as $W_t^{(i)} - \eta \cdot g_t^{(i)} / \sqrt{\epsilon + r_t^{(i)}}$. So, you are scaling by epsilon plus $r_t^{(i)}$, where $r_t^{(i)}$ is nothing but sum of squares of the i th component and you are taking the sum starting from the beginning.

So, this is actually the cumulative square, and that is how you find that every update component every component every update component is being scaled by the square root of sum of squares of the corresponding gradient value. And the purpose of epsilon putting epsilon is epsilon is a very small positive number positive quantity. So, this epsilon is put for stability of the division that in case your sum of squares of the gradient values become 0, I do not face a situation of a division by 0. So, this is the purpose that you are putting the epsilon. So, this is what is your Adagrad algorithm.

(Refer Slide Time: 26:42)



Adagrad
Positive Side:

- Adagrad adaptively scales the learning rate for different dimensions by normalizing with respect to the gradient magnitude in the corresponding dimension.
- Adagrad eliminates the need to manually tune the learning rate.
- Reduces learning rate faster for parameters showing large slope and slower for parameters giving smaller slope.
- Adagrad converges rapidly when applied to convex functions.

So, as we have said before that this Adagrad algorithm has got certain positive points that Adagrad is trying to adaptively scale the learning rate of different dimensions by normalizing with respect to the gradient magnitude in the corresponding dimension or square root of the sum of squares of the gradient values in the corresponding dimension. And the eta that we have put in the expression, this eta is actually the initial learning rate, and I do not need to tune it manually. And this makes your learning faster, because always the Adagrad is trying to push the updates in the right direction in the direction of the minimum error value. And as a result the Adagrad algorithm converges very rapidly when your error function is actually a convex function.

(Refer Slide Time: 27:41)

Adagrad

Negative side:

- ❑ If the function is non-convex:- trajectory may pass through many complex terrains eventually arriving at a locally region.
- ❑ By then learning rate may become too small due to the accumulation of gradients from the beginning of training.
- ❑ So at some point the model may stop learning.

But Adagrad also has certain negative point that is if the function is non-convex, then the trajectory may pass through many complex terrains and eventually it may find a locally convex region. And what we try to do is and what should be our aim is that once you get a locally convex region, the Adagrad algorithm or your learning algorithm should quickly converge at the minimum of that local convex region. But the problem is because in case of Adagrad algorithm your scale factor is the accumulation of the squares of the gradients and because the square of the gradient is always a positive term, so as t increases or the number of iterations increases your scale factor goes on increasing.

And because it goes on increasing monotonically, so as t becomes large your scaling factor that is $\frac{1}{\sqrt{\epsilon + \sum_{\tau=0}^t \|\mathbf{g}_\tau\|^2}}$ that term may be almost 0, it may be very very small. And when it becomes very small, your training or the learning rate becomes almost a 0, it is very small or vanishes. So, by the time Adagrad algorithm comes to that local convex the learning rate may be very, very slow, and as a result your overall learning of the convergence may take large time. So, at that point your model may eventually stop learning. So, this is a problem of Adagrad algorithm though it has many positive points.

So, in our next lecture, we will try to see other algorithms. And we will say we will try to analyze that how the problem being faced by the Adagrad algorithm can be addressed in those algorithms.

Thank you.