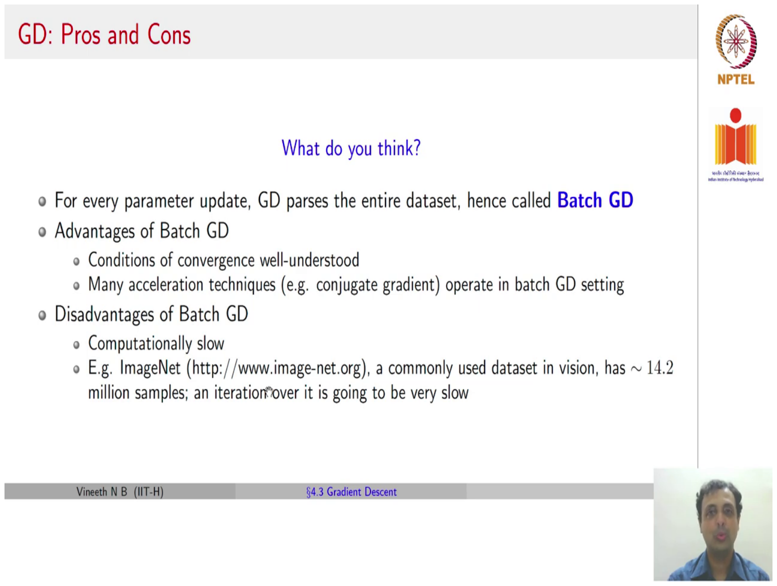
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology Hyderabad
Gradient Descent and Variants – Part 02

Let us now try to analyze, we have seen gradient descent, we have now seen at least one or two methods, momentum based gradient descent and Nesterov momentum based gradient descent, which can help you improve gradient descent training, what are the pros and cons of gradient descent you have seen so far?

(Refer Slide Time: 0:36)



GD: Pros and Cons

What do you think?

- For every parameter update, GD parses the entire dataset, hence called **Batch GD**
- Advantages of Batch GD
  - Conditions of convergence well-understood
  - Many acceleration techniques (e.g. conjugate gradient) operate in batch GD setting
- Disadvantages of Batch GD
  - Computationally slow
  - E.g. ImageNet (http://www.image-net.org), a commonly used dataset in vision, has ~ 14.2 million samples; an iteration over it is going to be very slow

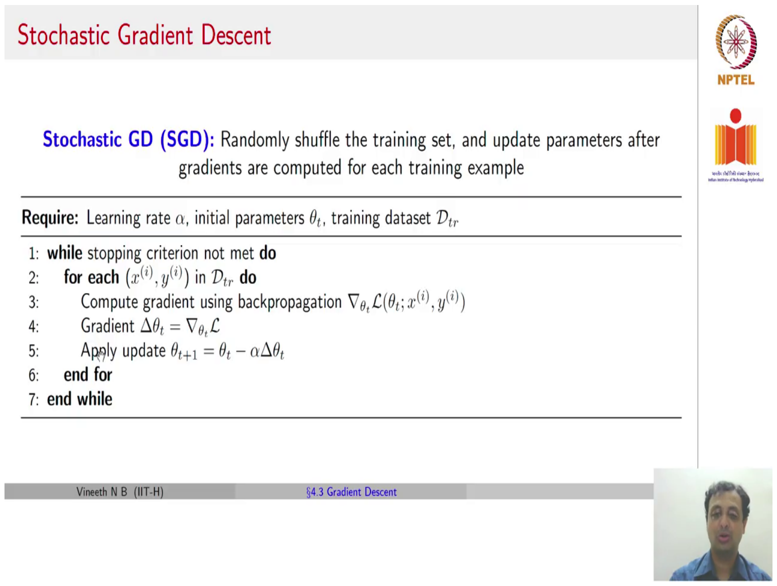
Vineeth N B (IIT-H) §4.3 Gradient Descent

For every parameter update gradient descent passes the entire dataset, we already saw that it computes the gradients, it keeps aggregating the gradients and then takes the average, we called it an epoch. So, gradient descent passes the entire dataset and it is hence called batch gradient descent.

And the advantages of batch gradient descent is, theoretically speaking from an optimization standpoint, the conditions of convergence are well understood and many acceleration techniques, there are improvements from an optimization perspective, such as conjugate gradient operate very well in the batch gradient descent setting, once again it is called batch gradient descent. Because you are waiting for the entire batch of data points to get completed, take all of these gradients, average them and then make an update.

So, the disadvantage of batch gradient descent is, it can be computationally slow for the same reason. Especially if your training dataset had say 10000 data points or 1 million data points, you are going to wait for passing through all of them before making an update. ImageNet which is a computer vision challenge, it is a very commonly used dataset in vision, has today about 14 million samples and one iteration over all of those 14 million samples can be very slow to wait for one parameter update.

(Refer Slide Time: 2:10)



Stochastic Gradient Descent

Stochastic GD (SGD): Randomly shuffle the training set, and update parameters after gradients are computed for each training example

Require: Learning rate α, initial parameters θ_t, training dataset $\mathcal{D}_{tr}$

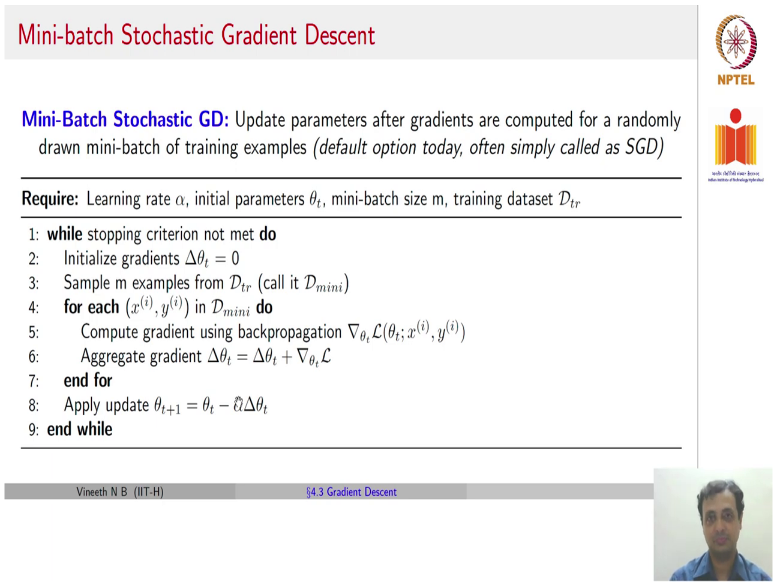
```
1: while stopping criterion not met do
2:   for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ do
3:     Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
4:     Gradient $\Delta\theta_t = \nabla_{\theta_t}\mathcal{L}$
5:     Apply update $\theta_{t+1} = \theta_t - \alpha\Delta\theta_t$
6:   end for
7: end while
```

Vineeth N B (IIT-H) §4.3 Gradient Descent

That brings us to a very popular approach known as stochastic gradient descent. Stochastic gradient descent or SGD stands for, you randomly shuffle your training dataset and update your parameters after gradients are computed for each training example. So, you randomly shuffle and pick one data point, forward propagate it, compute the error, compute the gradient, update all your parameters right away before propagating your next data point. So, here is your algorithmic difference with gradient descent.

The third, the third step here, sorry, this step 5 here on this algorithm was outside the for loop in gradient descent, you did all, you went through every data point accumulated all your gradients and then made the parameter update, whereas in stochastic gradient descent that step of updating your parameters goes inside the for loop, which means for every data point you are going to make a parameter update, this is going to be faster, let us see a few pros and cons for stochastic gradient descent too.

(Refer Slide Time: 3:26)



Mini-batch Stochastic Gradient Descent

Mini-Batch Stochastic GD: Update parameters after gradients are computed for a randomly drawn mini-batch of training examples *(default option today, often simply called as SGD)*

```
Require: Learning rate $\alpha$, initial parameters $\theta_t$, mini-batch size m, training dataset $\mathcal{D}_{tr}$
1: while stopping criterion not met do
2:     Initialize gradients $\Delta\theta_t = 0$
3:     Sample m examples from $\mathcal{D}_{tr}$ (call it $\mathcal{D}_{mini}$)
4:     for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{mini}$ do
5:         Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
6:         Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
7:     end for
8:     Apply update $\theta_{t+1} = \theta_t - \tilde{\alpha}\Delta\theta_t$
9: end while
```

Vineeth N B (IIT-H) §4.3 Gradient Descent

But before we go there we are going to talk about the setting which lies in between, which is known as mini batch stochastic gradient descent. What is mini batch stochastic gradient descent? Mini batch stochastic gradient descent as the name says is you take a mini batch of training examples, do not just take one, do not take all, take 20 examples from your training dataset, forward propagate all of them, compute all the outputs, compute all the errors, compute all the gradients and then taking average of those 20 gradients alone and update your parameters.

Why do we need this? It is possible that if you keep updating your gradient with every sample, it may give you random directions, one data point will tell you go like this, this is where Himalayas lowest point is there, another data point will tell you another step, you will just be taking different zigzag steps as you traverse through the surface if you listen to every data points gradient.

Averaging gives you a better sense of a gradient direction that is consistent across a few more data points. This mini batch version of stochastic gradient descent is the default option for training neural networks today, often today when people say they use SGD they actually mean mini batch SGD.

Remember mini batch SGD with batch size to be 1 is your SGD itself. But typically people use larger batch sizes 20, 100, so on and so forth to compute these gradients. So, algorithmically speaking the applying update once again went out of the loop but this time you are not going to compute, you are not going to aggregate gradients for the entire training dataset, but only for one mini batch which you denote as B-mini and you are only going to aggregate the gradients and average the gradient only for that mini batch in the algorithm.

(Refer Slide Time: 5:32)

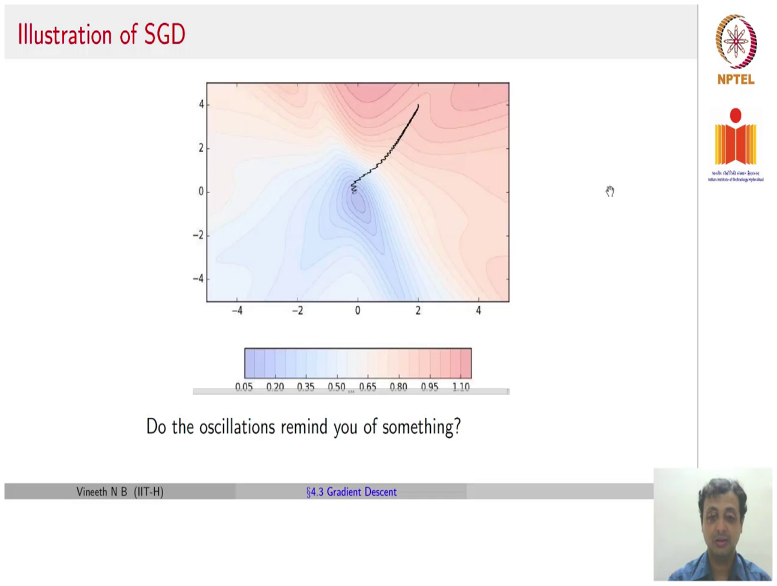


Illustration of SGD

Do the oscillations remind you of something?

Vineeth N B (IIT-H) §4.3 Gradient Descent

Let us try to see an illustration of stochastic gradient descent, this is once again a contour plot of an error surface, let us try to see how stochastic gradient descent works. You can see it progressing towards the minimum, remember the minimum is the blue surface, as it goes closer to the minimum, it seems to oscillate. Do the oscillations remind you of something?

(Refer Slide Time: 6:08)

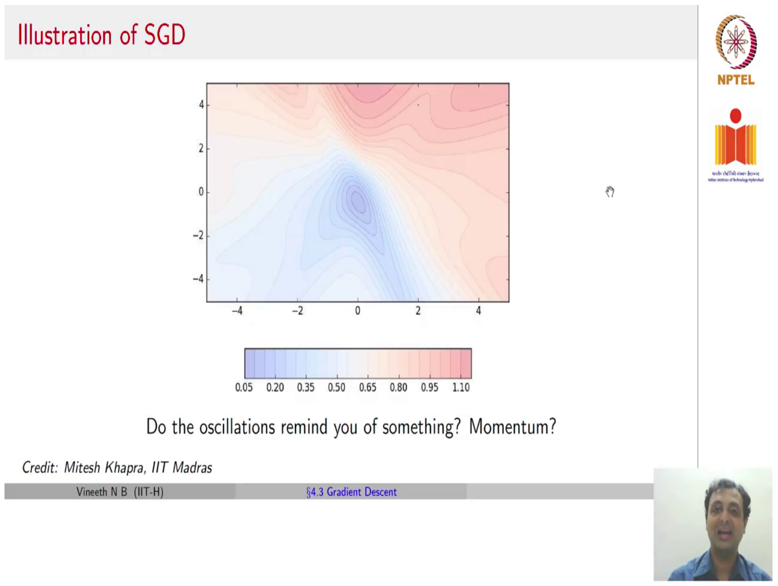


Illustration of SGD

Do the oscillations remind you of something? Momentum?

Credit: Mitesh Khapra, IIT Madras

Momentum may be, momentum also had similar oscillations when it reached the minimum.

(Refer Slide Time: 6:16)

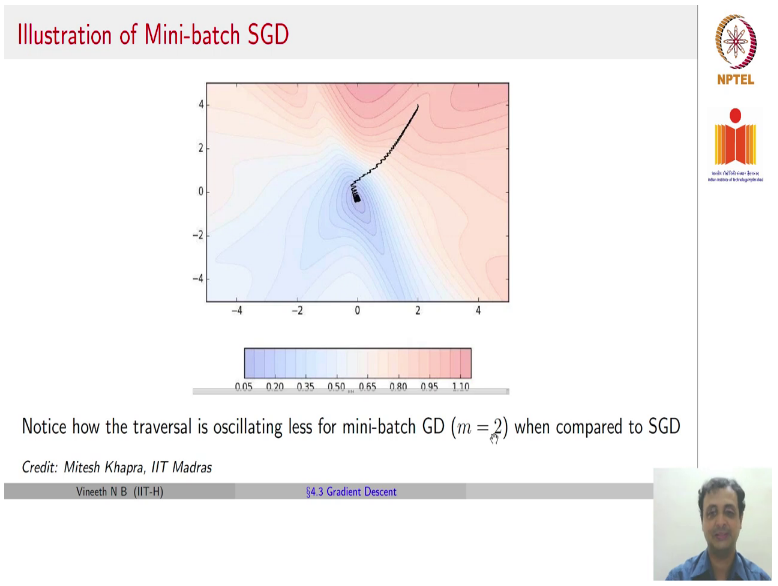


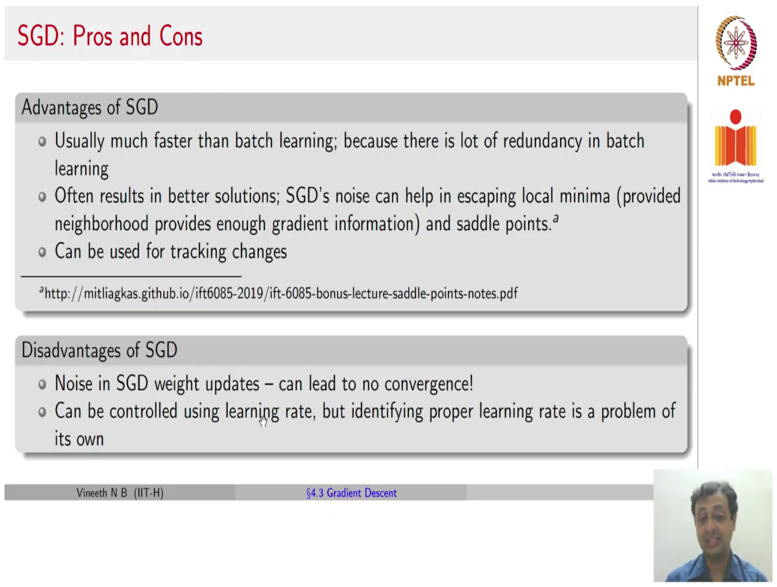
Illustration of Mini-batch SGD

Notice how the traversal is oscillating less for mini-batch GD ($m = 2$) when compared to SGD

Credit: Mitesh Khapra, IIT Madras

Here is an illustration of mini batch SGD, once again the same error surface, let us see if batching data points and computing the gradient helped us a little bit, you do see here the oscillations have mildly reduced for many batch SGD when compared to SGD but the batch size here was just 2, you could probably see lesser oscillations when you keep increasing the batch size, because then you are not going to vacillate based on every data points gradient, but you are going to batch them across a larger set of data points before you update your parameters.

(Refer Slide Time: 7:04)



SGD: Pros and Cons

Advantages of SGD

- Usually much faster than batch learning; because there is lot of redundancy in batch learning
- Often results in better solutions; SGD's noise can help in escaping local minima (provided neighborhood provides enough gradient information) and saddle points.[a]
- Can be used for tracking changes

[a]http://mitliagkas.github.io/ift6085-2019/ift-6085-bonus-lecture-saddle-points-notes.pdf

Disadvantages of SGD

- Noise in SGD weight updates – can lead to no convergence!
- Can be controlled using learning rate, but identifying proper learning rate is a problem of its own

Vineeth N B (IIT-H) §4.3 Gradient Descent

What are the pros and cons of stochastic gradient descent when you compare it to batch gradient descent, let us try to understand this? The advantages of stochastic gradient descent far outnumber its disadvantages, it is usually much faster than batch learning when you train neural networks. Why is that so? It is simply because there is a lot of redundancy in batch learning.

Because if you have one million data points in your dataset, while training your neural network model, then many of these data points may give you similar information, you probably do not need to wait for all of them to give you the gradient and then average, even if you take 100 of them maybe you are going to get some sense of where the gradient is for that particular iteration and move forward in that direction.

So, you are exploiting the redundancy in data to go much faster than standard batch gradient descent, it also often results in better solutions, this may seem a little counterintuitive because you would think that by using the entire batch it could be computationally slow but that is the ideal way to do it, then why does stochastic gradient descent yield better solutions?

The reason is stochastic gradient descent is a noisy version of gradient descent. What do we mean by that? We mean that you are only taking 20 data points, let us assume the mini batch size was 20, you are taking just 20 data points getting the gradient and you keep moving along those directions.

So, it is like asking once again if you are traversing Himalayas, if you ask a thousand people assuming that you had infinite computation and infinite access to people, if you ask a thousand people and they avoid going in a particular direction you probably can rely on that direction. If you asked only two people and asked them what direction to go in, maybe they may get it wrong.

Which means that when you ask a smaller set of people you may oscillate, you may take one direction then somebody else tells you go in another direction, you keep moving around and fluctuating a little bit but you eventually get there because between across all of these people they may point you in the direction, in expectation you would get there. But there may be fluctuations along the way.

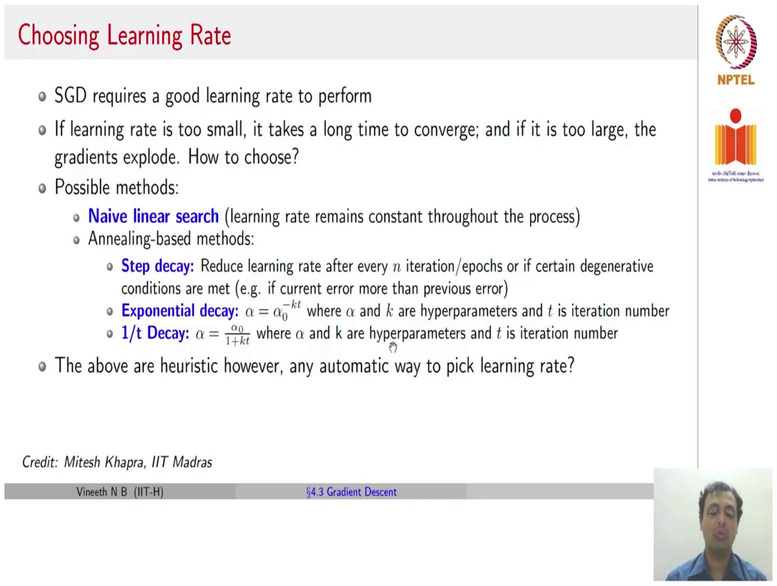
But this noisy updates of stochastic gradient descent is what would actually help you in escaping local minima, bad local minima or saddle points because of these noisy updates you may avoid saddle points and probably get to better local minimum over time, that is what researchers have been able to understand from how SGD has performed in deep learning.

Stochastic gradient descent is also useful for tracking changes. What do we mean? If you have a new set of data points that come in, it is very easy to update with just 20 data points rather than have to run an entire epoch with 1 million plus 20 data points, so remember in batch gradient descent you have to recompute the gradient for all the data points before you make your next move, but in stochastic gradient descent if new data points come that is what we mean by tracking changes here, you can easily update the system and keep moving.

What about disadvantages? Disadvantages of stochastic gradient descent is that the noise in stochastic gradient weight updates can lead to no convergence. To this day sometimes remember as we said that because of the complexity of the error surface where you initialize your weights is very important as to where which local minima you converge to, so often today in practice if in the first few iterations you find too much fluctuations in your error people often simply change the initialization and try again.

Which means noise in SGD weight updates can lead to no convergence but the solution is fairly simple if in the first few iterations things are not going well for you just stop, go back and start at a different location and start all over again, it is fairly a simple solution. This can also be controlled using learning rate, if the gradient you know is noisy do not use a high learning rate, so it can also be controlled using a learning rate but then identifying a proper learning rate can be a problem of its form and that is the problem we are going to talk about next.

(Refer Slide Time: 11:54)



Choosing Learning Rate

- SGD requires a good learning rate to perform
- If learning rate is too small, it takes a long time to converge; and if it is too large, the gradients explode. How to choose?
- Possible methods:
  - **Naive linear search** (learning rate remains constant throughout the process)
  - Annealing-based methods:
    - **Step decay:** Reduce learning rate after every n iteration/epochs or if certain degenerative conditions are met (e.g. if current error more than previous error)
    - **Exponential decay:** $\alpha = \alpha_0^{-kt}$ where α and k are hyperparameters and t is iteration number
    - **1/t Decay:** $\alpha = \frac{\alpha_0}{1+kt}$ where α and k are hyperparameters and t is iteration number
- The above are heuristic however, any automatic way to pick learning rate?

Credit: Mitesh Khapra, IIT Madras

Vineeth N B (IIT-H) §4.3 Gradient Descent

How do you choose a learning rate? We said that momentum was one way to artificially inject a learning rate into the system because the momentum in the previous step and the current gradient aligned it is almost like you had a high learning rate in those directions. In general, though if the learning rate is too small it can take a long time to converge, if the learning rate is too large the gradients can explore.

Now, how do you choose a learning rate in practice, assuming we do not use momentum, how do you choose a learning rate in practice? One simple option is naive linear search, you just keep your learning rate to be constant throughout the process, just fix it, just use the same learning rate across all your iterations until you reach convergence.

Or you could use annealing based methods, annealing based methods are methods where you gradually reduce the learning rate over time using some formula that you come up with, let us see a couple of them now. One of them is known as step decay, where you reduce the learning rate after every n iterations or if certain degenerative conditions are met.

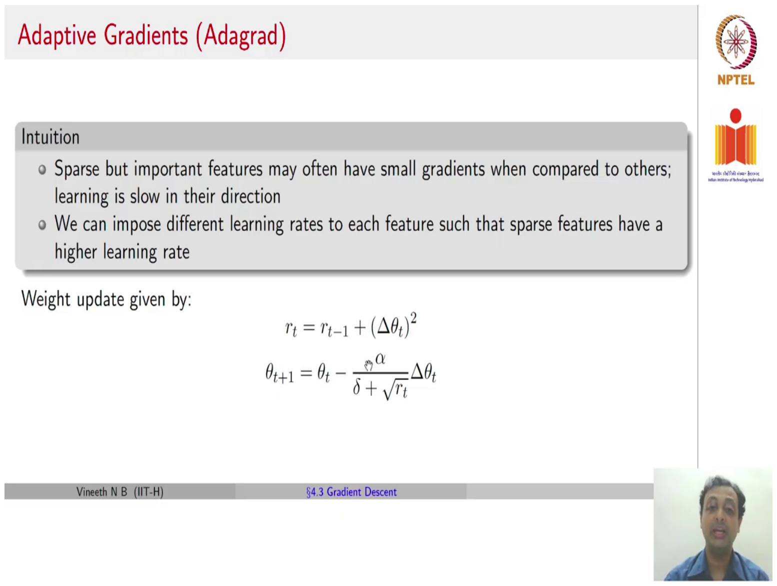
For example, if the current error is more than the previous error which should not happen and you do gradient descent, you expect that the current error should always be lesser than the previous error but when you do stochastic gradient descent that may not be guaranteed because you are using only 20 data points to design your gradient.

So, what you can do is keep reducing your learning rate after every n iterations. So, that is one simple approach. The idea here is that as you keep training over iterations in epochs you are probably getting closer to your minimum and when you get very close to your minima you want to take small steps so that you reach your minima and do not over shoot it, that is the reason for reducing the learning rate over time, that is called step decay.

Another popular option is known as exponential decay. In exponential decay you set an initial learning rate called α_0 and then $\alpha = \alpha_0^{-kt}$ where α_0 and k are hyper parameters and t is an iteration number. So, you start with the learning rate α_0 and keep progressively reducing it exponentially in this particular form over every few iterations, because t is an iteration number here, as the iteration number increases the learning rate automatically starts falling down.

You can also use a simple 1 by t decay formula, where you start with an α_0 and α at a particular time step can be given by $\alpha = \alpha_0/(1 + kt)$, where α_0 and k are hyper parameters and t is an iteration number. All of these as you can see are heuristics, we are just saying let us just do one of these and maybe it will work. Can we do something better, can we be more aware of what a neural network is doing and accordingly choose a learning rate rather than have these global rules that we fix irrespective of what the network is doing? Is there a way to do this?

(Refer Slide Time: 15:29)



Adaptive Gradients (Adagrad)

Intuition

- Sparse but important features may often have small gradients when compared to others; learning is slow in their direction
- We can impose different learning rates to each feature such that sparse features have a higher learning rate

Weight update given by:

$$r_t = r_{t-1} + (\Delta\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_t}}\Delta\theta_t$$

Vineeth N B (IIT-H) §4.3 Gradient Descent

Yes, there are a few methods which we are going to discuss now. One of the earliest methods in this context was known as adaptive gradient or adagrad. Adagrad has a simple intuition as to the fact that sparse but important features may often have small gradients when compared to others and learning could be slow in that direction.

In our case features means dimensions of weights rather a subset of weights, so you have your entire space of weights, a subset of weights which change only once a while, so let us assume that there are certain weights that to keep do not change with every mini batch, in mini batch SGD, but change only once a while.

They may change by a small amount but that amount is important because they only change once in a while. Sparse features could be important and learning could be slow in this direction because the gradients could be small in that direction. So, a simple thought then is why should the learning rate be the same for every gradient in your neural network?

Once again remember that the gradient is a vector of the derivatives of the loss with respect to every weight in your neural network. Why should you use the same learning rate for all of them? Why can't you use a different learning rate for each of these weights? So, what you do in Adagrad is you accumulate the squared gradients as a running sum, rt is a quantity that holds a running sum of squared gradients, once again keep in mind that the gradient is in $\Delta\theta_t$ is simply the gradient that is the parameter update and that is a vector of the gradient with respect to every weight in the neural network.

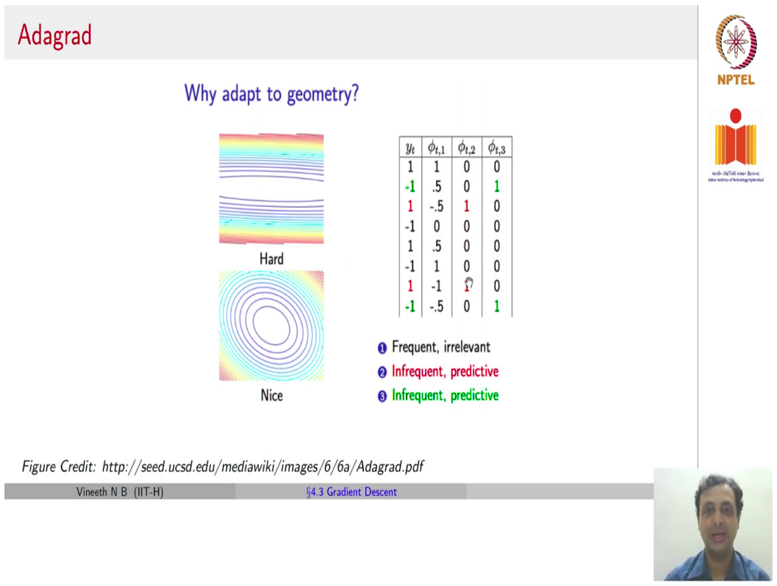
So, for each of those weights you maintain a running sum of the gradient squares and then your parameter update states $\theta_{t+1} = \theta_t - \alpha/(\delta + \sqrt{r_t}) \; * \; \Delta\theta_t$, $\Delta\theta_t$ is your gradient in your current time step.

Now, why does this address this intuition? Delta here is simply used for numerical stability, you just set it to be a value like 10^{-6} or 10^{-7} so that you do not face a divide by zero error, in case r_t is zero you do not want to divide by zero error when you compute so that is the reason you have a δ there.

But otherwise what this is doing is, when r_t is small which is going to be a very small value, this is going to become large or rather the learning rate for that weight is going to become large, when r_t is large the learning rate will become small. We are saying that for one of your weights if your gradient was large give it a small learning rate and for one of your weights if your gradient is small then give it a larger learning rate.

This seems to fit with the intuition that we had so far and the nice part now is this kind of an approach because r_t is maintained for each gradient or each weight individually. This automatically takes care of changing the learning rate for each weight differently.

(Refer Slide Time: 19:08)



Adagrad

Why adapt to geometry?

y_t	$\phi_{t,1}$	$\phi_{t,2}$	$\phi_{t,3}$
1	1	0	0
-1	.5	0	1
1	-.5	1	0
-1	0	0	0
1	.5	0	0
-1	1	0	0
1	-1	1	0
-1	-.5	0	1

1. Frequent, irrelevant
2. Infrequent, predictive
3. Infrequent, predictive

Figure Credit: http://seed.ucsd.edu/mediawiki/images/6/6a/Adagrad.pdf

Vineeth N B (IIT-H) §4.3 Gradient Descent

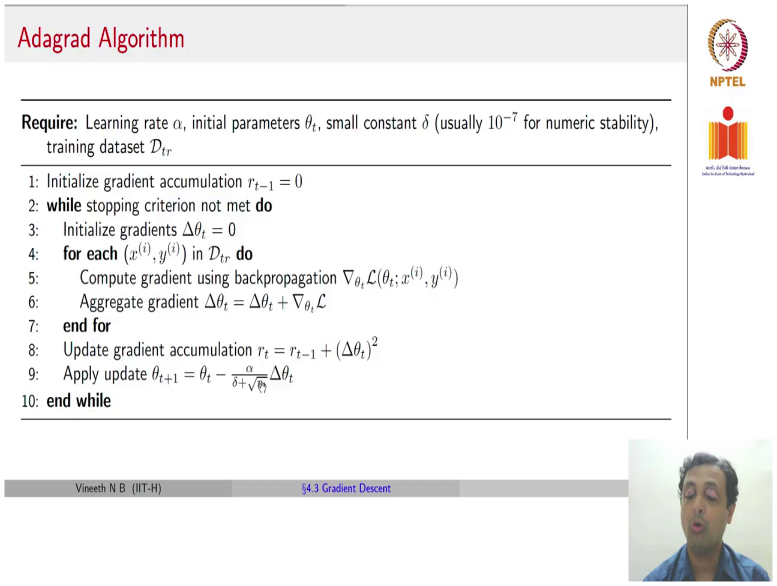
Let us try to understand this from a simple example. So, we do understand that your error surface, these are contour plots, when your contour plots are spherical, things are easier, it is easier to converge, when your contour plots get elliptical is where your oscillations can occur. Another important property of elliptical contour plots is that there are certain dimensions where the change can be rapid, and certain dimensions where the change could be rather small, which is what elliptical basically means.

So, if you see this example here which denotes, let us assume the gradient and the corresponding y_t value, they are now saying here that if you look at one of these, let us look the black frequent and predict irrelevant which simply states that all those values keep changing regularly but they do not seem to be connected to the class output, these are some values and they do not seem to be connected to the class output in this particular scenario or the output of whatever function you are modeling.

While there could be other values such as red and green which change only once a while so you can see in $\phi_{t,2}$ it does not matter here as to what these quantities are as this is only an example, so when $\phi_{t,2}$ is 1 we see that y_t definitely becomes 1, it does not matter when it is 0 but when it is 1 the output is definitely 1.

Similarly, for $\phi_{t,3}$ you see that when it is 1 the output is definitely minus 1. So, it looks like this particular parameter changes only once a while but when it changes it makes an impact on the final decision. So, this is a scenario that says that even if the gradient is small but if it happens only once a while, let us give more weight to changes in those directions, in those weights and that is what this algorithm does.

(Refer Slide Time: 21:18)



Adagrad Algorithm

```
Require: Learning rate $\alpha$, initial parameters $\theta_t$, small constant $\delta$ (usually $10^{-7}$ for numeric stability),
    training dataset $\mathcal{D}_{tr}$
1: Initialize gradient accumulation $r_{t-1} = 0$
2: while stopping criterion not met do
3:   Initialize gradients $\Delta\theta_t = 0$
4:   for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ do
5:     Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
6:     Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
7:   end for
8:   Update gradient accumulation $r_t = r_{t-1} + (\Delta\theta_t)^2$
9:   Apply update $\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_t}}\Delta\theta_t$
10: end while
```

Vineeth N B (IIT-H) §4.3 Gradient Descent

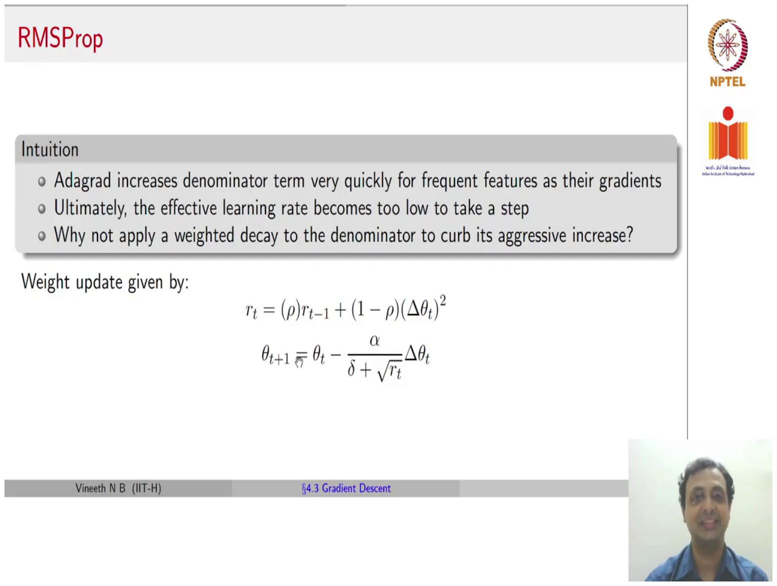
So, here is the algorithm, so you aggregate the gradient, you update your gradient accumulation, so you initialize your gradient accumulation in a parameter called $r_{t-1} = 0$ and you update your gradient accumulation to be $r_t = r_{t-1} + (\Delta\theta)^2$, you keep accumulating the square gradients and then this is the formula.

One small observation here which is probably not clear from this algorithm is this particular quantity on step nine is an element-wise product. So, which means, remember in this case that while alpha was a scalar so far, r_t is a vector, $\sqrt{r_t}$ will also be a vector, so this entire $\alpha/(\delta + \sqrt{r_t})$ will become a vector, it is a vector of learning rates multiplied by a vector of gradients element-wise multiplication, it is sometimes also called Hadamard product. Hadamard product is nothing but element-wise multiplication, remember it is not like dot product, in a dot product you take element-wise multiplication and then add them all up and get a scalar.

In a Hadamard product, you simply take element wise multiplication and your output is still a vector, you do not add them all, that is the main difference and this operation here in step nine is actually a Hadamard product, element wise multiplication between two vectors. There is one problem with the Adagrad method, can you try this for the problem?

The problem is that this gradient accumulation term r_t is a running sum that simply keeps accumulating over iterations, which means as you run more and more iterations, it will only keep increasing and get higher and higher and higher over time which means this denominator is definitely going to get high over time for all the weights.

(Refer Slide Time: 23:22)



RMSProp

Intuition

- Adagrad increases denominator term very quickly for frequent features as their gradients
- Ultimately, the effective learning rate becomes too low to take a step
- Why not apply a weighted decay to the denominator to curb its aggressive increase?

Weight update given by:

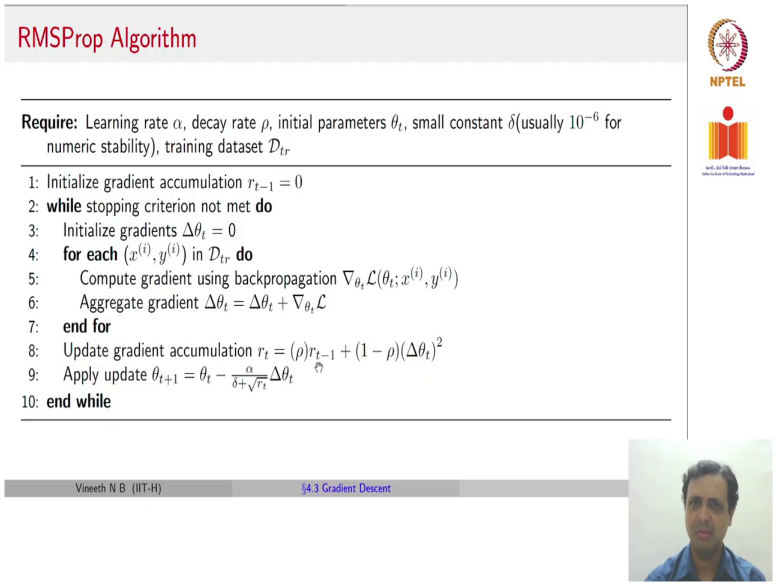
$$r_t = (\rho) r_{t-1} + (1-\rho)(\Delta\theta_t)^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\delta + \sqrt{r_t}} \Delta\theta_t$$

Vineeth N B (IIT-H) §4.3 Gradient Descent

In particular, there are some of the weights for which gradients keep occurring in every iteration where the denominator term can very quickly increase, this will lead to a lower learning rate for those dimensions in the weights and learning may not happen along those directions. RMSProp which was a method that came along the same time as Adagrad proposed a slightly different approach to this problem and said let $r_t = \rho r_{t-1} + (1 - \rho)(\Delta\theta_t)^2$, it is a linear combination where the two coefficients add up to 1.

This now ensures that because of the ρ and 1 − ρ, the sum cannot keep exploding and is now controlled. Think about it for a while as to why and you will get why this will not allow you to explode. And now your θ_{t+1} and θ_t, the entire expression is exactly the same as Adagrad.

(Refer Slide Time: 24:42)



RMSProp Algorithm

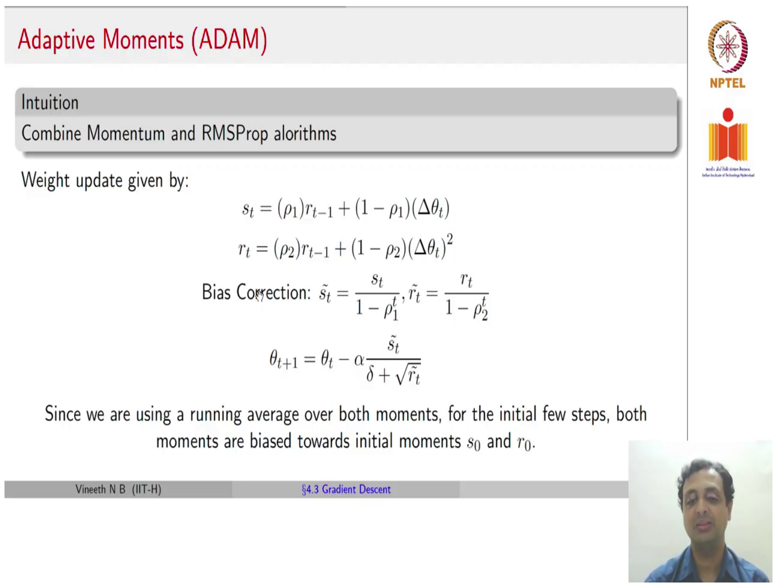
Require: Learning rate α, decay rate ρ, initial parameters θ_t, small constant δ(usually 10^{-6} for numeric stability), training dataset $\mathcal{D}_{tr}$

```
1: Initialize gradient accumulation $r_{t-1} = 0$
2: while stopping criterion not met do
3:     Initialize gradients $\Delta\theta_t = 0$
4:     for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ do
5:         Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
6:         Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
7:     end for
8:     Update gradient accumulation $r_t = (\rho)r_{t-1} + (1-\rho)(\Delta\theta_t)^2$
9:     Apply update $\theta_{t+1} = \theta_t - \frac{\alpha}{\delta+\sqrt{r_t}}\Delta\theta_t$
10: end while
```

Vineeth N B (IIT-H) §4.3 Gradient Descent

Here is the algorithm for RMSProp with only that change in the step where you have now the running sum of the accumulated squared gradients to be a convex combination monitored by a quantity called ρ which is known as the decay rate. The decay rate decides how much you want to consider the previous squared gradient while you get the current squared gradient, very similar to momentum but in a different context.

(Refer Slide Time: 25:12)



Adaptive Moments (ADAM)

Intuition

Combine Momentum and RMSProp alorithms

Weight update given by:

$$s_t = (\rho_1) r_{t-1} + (1 - \rho_1)(\Delta\theta_t)$$

$$r_t = (\rho_2) r_{t-1} + (1 - \rho_2)(\Delta\theta_t)^2$$

$$\text{Bias Correction: } \tilde{s}_t = \frac{s_t}{1 - \rho_1^t}, \tilde{r}_t = \frac{r_t}{1 - \rho_2^t}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\tilde{s}_t}{\delta + \sqrt{\tilde{r}_t}}$$

Since we are using a running average over both moments, for the initial few steps, both moments are biased towards initial moments s_0 and r_0.

Vineeth N B (IIT-H) §4.3 Gradient Descent

The most popular algorithm that is used for training neural networks today for adapting learning rates is known as ADAM or adaptive moments. Adaptive moments uses a simple intuition of combining ideas of RMSProp and momentum in a way, let us see how that is done. While Adagrad and RMSProp accumulate the squared gradients ADAM accumulates both the gradient and the square gradients, s_t accumulates the gradient, r_t accumulates the squared gradient just like RMSProp. Why?

You look at the final update equation here, the update says that $\theta_{t+1} = \theta_t - \alpha\tilde{s}_t/(\delta + \sqrt{\tilde{r}_t})$. I will explain the tilde part in a moment but let us first assume that there is no tilde let us just assume s_t and r_t. So, what is this doing $\alpha s_t/(\delta + \sqrt{r_t})$ is exactly the same that we saw in RMSProp and Adagrad, s_t would have been the current gradient in Adagrad and RMSProp but now s_t is the running sum of the gradient. And what is the running sum of the gradient?

Momentum, remember momentum was what you use the previous gradient, the previous gradient so on and so forth using momentum term ρ_1 is the momentum coefficient for us in this particular context, that is why ADAM can be looked at as a combination of RMSProp and momentum.

Now, coming to why this is tilde here, why not just s_t and r_t is for a very simple reason of doing what is known as bias correction in the initial iterations. In the initial iterations if ρ_1 is set to a high value which is what is typically done, ρ_1 just like momentum ρ_1 could be set to something like 0.9 or something like that.

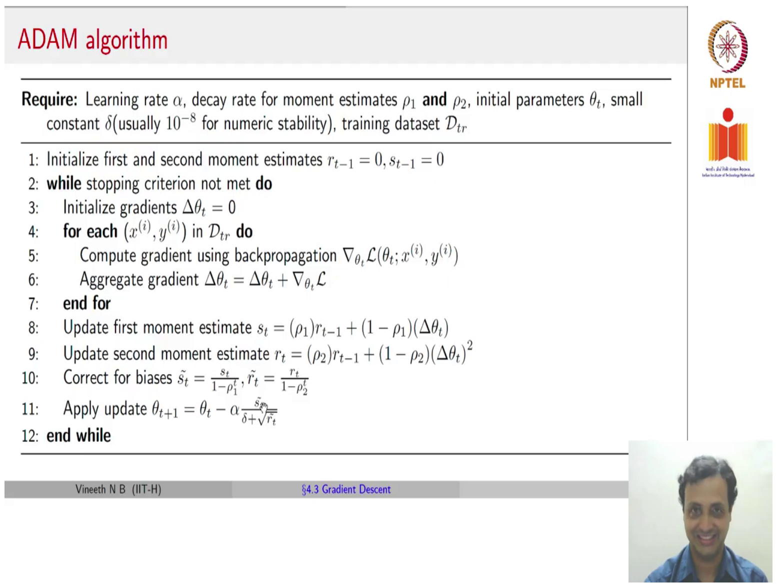
In your initial step r_{t-1} would be initialized to 0 that is where r_{t-1} will start, 1 minus ρ_1 will make it a very small value because ρ_1 is close to 1, so which means the gradients will not have any impact on s_t in the initial iterations until they add up, so there may not be much training that happens in the initial iterations.

So, what do we do for that? We say let $\tilde{s}_t = s_t/(1-\rho_1^t)$ similarly $\tilde{r}_t = r_t/(1-\rho_1^t)$. What does this do? If ρ_1 is 0.9, remember 1 minus 0.9 is 0.1, so s_t by 0.1 will now increase it 10 fold, so by whatever fold it was reduced here, it is now going to increase it by 10 fold in the initial iteration.

What happens in later iterations? We have t here, t is the number of the iteration that you are talking about. What happens here? Remember ρ_1 is less than 1 it is chosen to be 0.9 or something like that, so ρ_1^t will keep reducing all the iterations, 0.9 into 0.9 is 0.81, so it keeps as long as you have a value less than 1 as you raise it to the power t it is going to keep reducing which means this quantity the denominator will keep reducing over iterations and sorry this is not the quantitative denominator that ρ_1^t will keep reducing to 0 and $1 - \rho_1^t$ will get closer to 1 over the iterations.

Which means after a few iterations $\tilde{s}_t$ will become s_t, $\tilde{r}_t$ will become r_t. So, the step of bias correction was only to handle the initial epochs where the gradient may not have a significant impact.

(Refer Slide Time: 29:27)



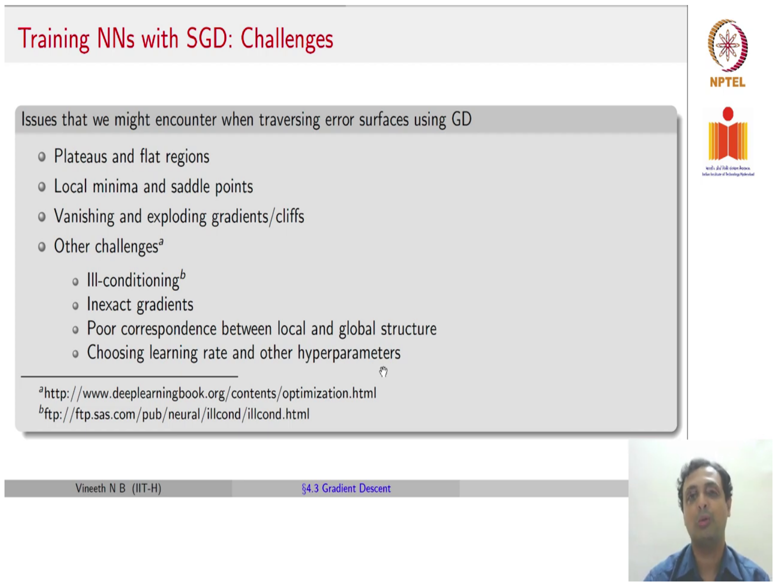
ADAM algorithm

```
Require: Learning rate $\alpha$, decay rate for moment estimates $\rho_1$ and $\rho_2$, initial parameters $\theta_t$, small
    constant $\delta$(usually $10^{-8}$ for numeric stability), training dataset $\mathcal{D}_{tr}$
1: Initialize first and second moment estimates $r_{t-1} = 0, s_{t-1} = 0$
2: while stopping criterion not met do
3:     Initialize gradients $\Delta\theta_t = 0$
4:     for each $(x^{(i)}, y^{(i)})$ in $\mathcal{D}_{tr}$ do
5:         Compute gradient using backpropagation $\nabla_{\theta_t}\mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
6:         Aggregate gradient $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t}\mathcal{L}$
7:     end for
8:     Update first moment estimate $s_t = (\rho_1)r_{t-1} + (1-\rho_1)(\Delta\theta_t)$
9:     Update second moment estimate $r_t = (\rho_2)r_{t-1} + (1-\rho_2)(\Delta\theta_t)^2$
10:    Correct for biases $\tilde{s}_t = \frac{s_t}{1-\rho_1^t}, \tilde{r}_t = \frac{r_t}{1-\rho_2^t}$
11:    Apply update $\theta_{t+1} = \theta_t - \alpha\frac{\tilde{s}_t}{\delta+\sqrt{\tilde{r}_t}}$
12: end while
```

Vineeth N B (IIT-H) §4.3 Gradient Descent

This is your ADAM algorithm, it is simply a summary of what we just said. These are the two steps, you update a first moment estimate which is the gradient, second moment estimate which is the square gradient you maintain a running sum of both, correct for biases and do your update. The key thing in the update is you do not have the gradient here because the gradient is subsumed into s_t.

(Refer Slide Time: 29:54)



Training NNs with SGD: Challenges

Issues that we might encounter when traversing error surfaces using GD

- Plateaus and flat regions
- Local minima and saddle points
- Vanishing and exploding gradients/cliffs
- Other challenges[a]
  - Ill-conditioning[b]
  - Inexact gradients
  - Poor correspondence between local and global structure
  - Choosing learning rate and other hyperparameters

[a]http://www.deeplearningbook.org/contents/optimization.html
[b]ftp://ftp.sas.com/pub/neural/illcond/illcond.html

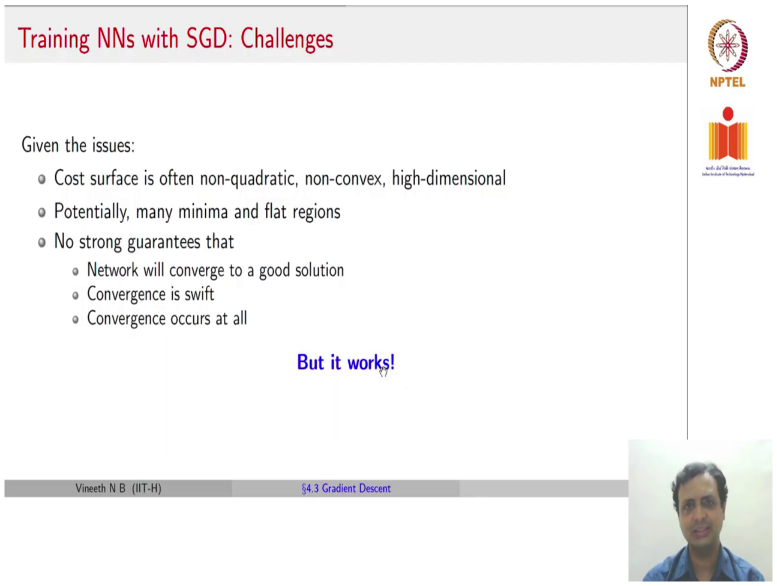
Vineeth N B (IIT-H) §4.3 Gradient Descent

That summarizes the various methods of improving gradient descent while training neural networks. So, some of the issues that we saw are plateaus and flat regions, local minima and saddle points, other issues that we will see as we go forward are vanishing and exploding gradients, as neural networks become deeper and deeper the gradients that propagate from the last layer to the first layer can keep vanishing or exploding, that can cause what are known as cliffs in the error surface, we will see this a bit later.

Other challenges that could happen is sheer ill conditioning of the matrix of weights or the loss functions or the gradients that you are dealing with, please look at this link here to understand what this means, you could face from a (prob), you could face a problem of inexact gradients, the gradients numerically may not be computed properly, for example, finding if a gradient is 0 is not trivial because numerically attaining 0 while training is not possible, you have to ensure that you manage it in some way.

But in general numerical issues and gradients could also cause problems. There could be poor correspondence between local and global structure, which means you have a global error surface and you want to get to the global minimum but the local region in which you are in may not really correspond with that global structure, you will be stuck with trying to handle the undulations in that local structure. Choosing learning rate and other hyper parameters like the momentum hyper parameter or the decay rate are also problems to deal with.

(Refer Slide Time: 31:34)



Training NNs with SGD: Challenges

Given the issues:

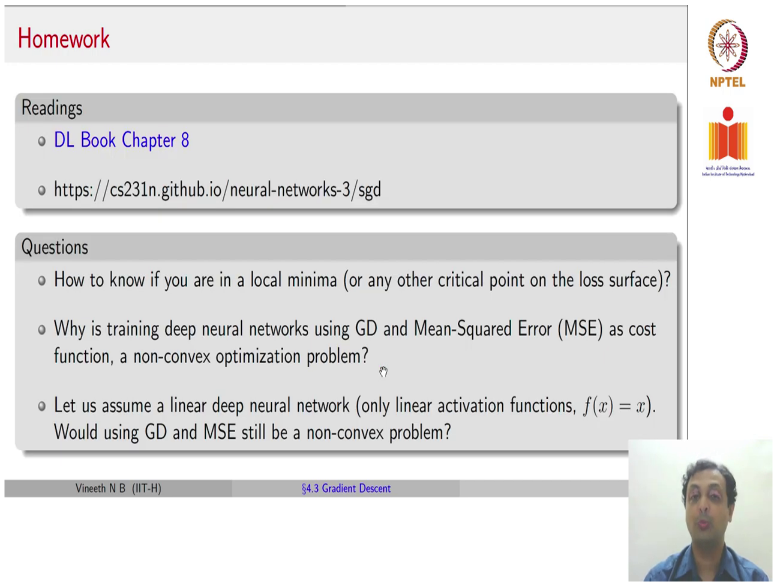
- Cost surface is often non-quadratic, non-convex, high-dimensional
- Potentially, many minima and flat regions
- No strong guarantees that
  - Network will converge to a good solution
  - Convergence is swift
  - Convergence occurs at all

But it works!

Vineeth N B (IIT-H) §4.3 Gradient Descent

To summarize, the error surface or cost surface in neural networks is often non-quadratic, non-convex, high dimensional. Potentially in many minima and flat regions, there are no strong guarantees that the network will converge to a good solution, the convergence is swift or that convergence occurs at all, but it works. This is an issue that is being investigated today, but it is fabulous that training neural networks with stochastic gradient descent works and is being used around the world in several applications.

(Refer Slide Time: 32:15)



Homework

Readings

- DL Book Chapter 8
- https://cs231n.github.io/neural-networks-3/sgd

Questions

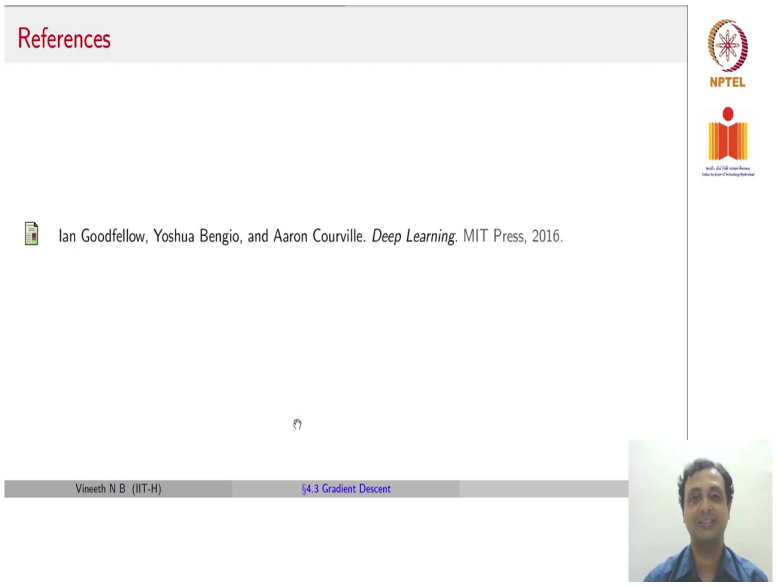
- How to know if you are in a local minima (or any other critical point on the loss surface)?
- Why is training deep neural networks using GD and Mean-Squared Error (MSE) as cost function, a non-convex optimization problem?
- Let us assume a linear deep neural network (only linear activation functions, $f(x) = x$). Would using GD and MSE still be a non-convex problem?

Vineeth N B (IIT-H) §4.3 Gradient Descent

Here is your homework, read up on Deep Learning book chapter 8, at least the relevant sections in that, as well as you can also go through this particular lecture on SGD. A few questions for you to take away is how to know if you are in a local minima or any other critical point on the loss surface, I did mention a few times that finding that or finding the convergence criterion is not trivial. Let us ask that question for you to find before we answer.

Why is training deep neural networks using gradient descent and mean square error a non-convex optimization problem? This is the first question that we started the lecture with, this is something for you to ponder upon. And finally if we assume a deep linear neural network, no activation functions at all, would using gradient descent and mean square error still be a non-convex optimization problem when you train deep neural networks? These are some questions to think about.

(Refer Slide Time: 33:18)



References

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Vineeth N B (IIT-H) §4.3 Gradient Descent

And here are references.