

Deep Learning for Visual Computing
Prof. Debdoot Sheet
Department of Electrical Engineering
Indian Institute of Technology, Kharagpur



Lecture - 23
Optimization Techniques and Learning Rules

Welcome. So, today, we would be learning about different kind of learning rules and optimization techniques. So, while we have done a few on the architectures and you have also been upraised on the different kind of training mechanisms. And at some points of time, while we had also played around with cost functions of how to use it, and some of these cost functions were what were typical for classification problems, the others were; what was typical for regression problem. And on the other side, what we were looking down is more of trying to understand how to train these networks. And while we had finished off in the earlier class on vanilla descent gradient or the most simple form of learning out these kind of networks called as a decent gradient rule.

Now, here what I am going to do is at you have seen that while we are writing down the codes for a few of these examples where the data was quite small you could actually do it on the vanilla way. And then for the others where the data set was a bit larger and we want did not want to write down so many codes for training it down we chose to use one of these things called as an optim package. And within the optim package, what we were doing is we could choose down what kind of optimizer we were supposed to use. And in general we have seen that we have been using stochastic gradient descent SGD optimizer as well as ADAM and in some of the cases we were sticking down to ADAM because that is an optimizer which was giving a convergence criteria.

Now, there are different ways of working around with that and how it works out as well as you have seen that we were using another concept called as a batch learning. So, today's lecture is more of going to focus on what is the similarities and the differences between each of them and how they end up trying to solve the same problem overall over there.

(Refer Slide Time: 01:57)



NPTEL ONLINE
CERTIFICATION COURSES
Indian Institute of Technology Kharagpur | Department of Electrical Engineering

Overview

- Vanilla Gradient Descent
- Backpropagation Rules
 - Epoch update
 - Batch update
 - Sample update
- Optimization Techniques
 - Stochastic gradient descent
 - Adaptive momentum

Learning Rules and Optimization Techniques [Debdoot Sheet] 2

So, without much of a delay let us get into how it is organized. So, the first one is where I would be doing a basic revision over the vanilla gradient descent and then eventually would get into the backpropagation rules. Now, for backpropagation over there, we have we typically have three different mechanisms of doing it, one of them is called as the epoch update rule in which you are going to backpropagate only once in an epoch and that is only when the wait gets updated over there. So, your gradient and your backpropagation equation is solved only once per epoch

The next one which is called as an batch update in which you have seen down in the lab exercises that in batch update what we were trying to do predominantly is that we try to update it multiple times in an epoch and there is a fixed number of samples which come down and form down a batch. So, it is not something like epoch update because it is multiple number of times. And the one of the major reasons why we were doing that was we might be limited on the memory capacity of the net of our system which we are implementing.

Now, typically, the problems which we have been taking till now are just a few 100 mb of data and it does not cost so much on the RAM. Most of your system RAMs are greater than 4 GB. So, it is not much of tedious task over there and. In fact, you can load the whole data set on the GPU as well if you have a descent config of GPU available with you. Now, the challenge comes down say if you are handling down a few hundred

gigabytes of data, so which is where you would even saturate and run over your typical RAM of a system. So, today's most of the pc consumer grade PCs would allow you to support up to say 64 GB or even some of the cases maybe 128 GB of RAM support.

Now, there are few server class machines with Intel Xeon process, which would allow you to go up to 768 GB or even more up to 1.5 terabytes of RAM. So, however, the challenge is that all of you are not going to get an access to that one and it costs, it significantly costs I mean those systems are about 10 to 20 times more costlier than most of your typical desktop grade computers. But then that does not restrict you that if you have a desktop grade machine that you will not be able to actually work out any of these bigger large sized data sets say 100 GB is just typically what would be there for any of the video problems which you would deal with.

So, if you are looking into autonomous driving kind of problems and or any kind of video analytics which will come down a bit later on you will have this large kind of thing. And even imagine trying to solve the whole image net from scratch that is also going to take you that much amount of space over there. So, in those kind of cases is where batch update comes of help and what you are seeing now is within your by torch environment batch update was basically a single function call. So, you had to do your data loader and configure it out and that just solves it out.

So, today I am going to get into the whole mechanism as to how this whole mechanism works out. And last but not the least is what is called as a sample update rule and in which like for every single sample which goes into the network you have one backpropagation. Now, the costliest of all is the sample update rule, but you would be seeing, so in the subsequent one where we have the tutorial on each of these being used you would be looking down at while it is the costlier one. But then by virtue of multiple number of updates what comes down, it would technically take down lesser number of epochs to converge as well. However, the dynamics is also very high. So, there is no always, there is not guaranteed that it will always converge.

And on the other side of it, in optimization techniques while we do the plain vanilla gradient descent basically revision over there in optimization techniques, we are going to do around with stochastic gradient descent. And once we are done with stochastic gradient descent, the next one I will come down is adaptive momentum or the Adam

optimizer. Now, these are not just the like these two are not just the only one which are available, so you have conjugate gradient descent you have LBFGS, you have a multiple of those any of those convex optimization techniques which are available over there.

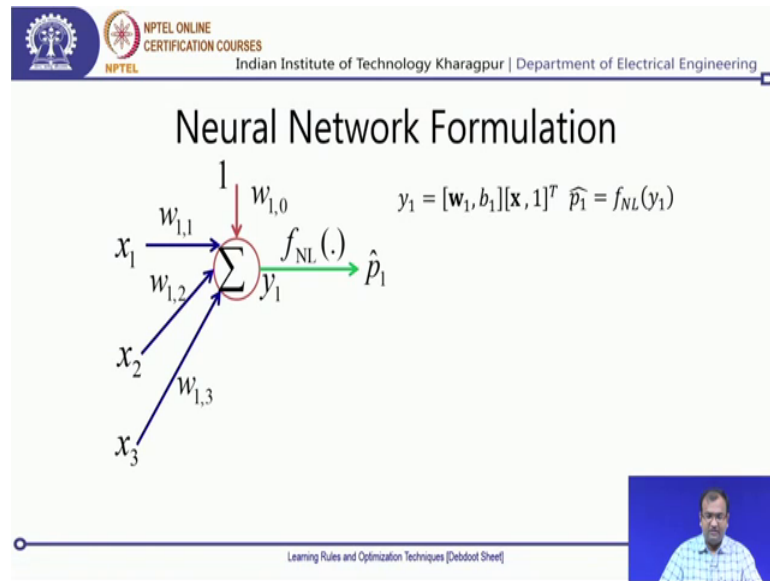
However, given the fact that we are dealing with piecewise convex function which we need to optimize over here within our neural networks. These two are the most commonly used; and commonly used in a sense that they have been found from majority of the problems the community has formed guaranteed convergence with these two as well as the fastest convergence and least amount of variation on the run over there. So, that is the reason why we are sticking to these two mechanisms and explaining you around that.

(Refer Slide Time: 06:26)



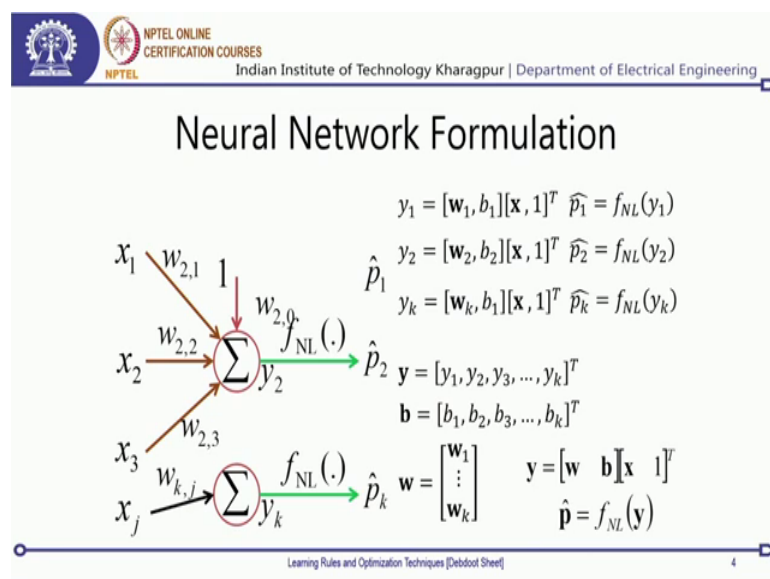
So, let us get into revising this vanilla gradient descent. Now, what comes down is you will start with the basic neural network formulation and how we had come down to the gradient rule over there. This is more of just to get you a revised concept. And we are going to take all of those concepts subsequently down the lectures. And for that it is always good to just have the notions and notations once again clear in your mind.

(Refer Slide Time: 06:49)



Now, say that I just had a three-dimensional input over there, which has three different variables x_1 , x_2 , and x_3 coming down from three different senses that is how I was explaining in the earlier case. And then it is going down and link down to one of these predictive vectors one of the predictions p_1 and since it is a predicted state and not the true state; and for that reason it is called as p_1 hat. Now, over there how it is related down is the first part is just a linear summation over there with weights coming down and then subsequently what you have is a non-linearity imposed on top of it. And now once that part is done so this is for your first prediction.

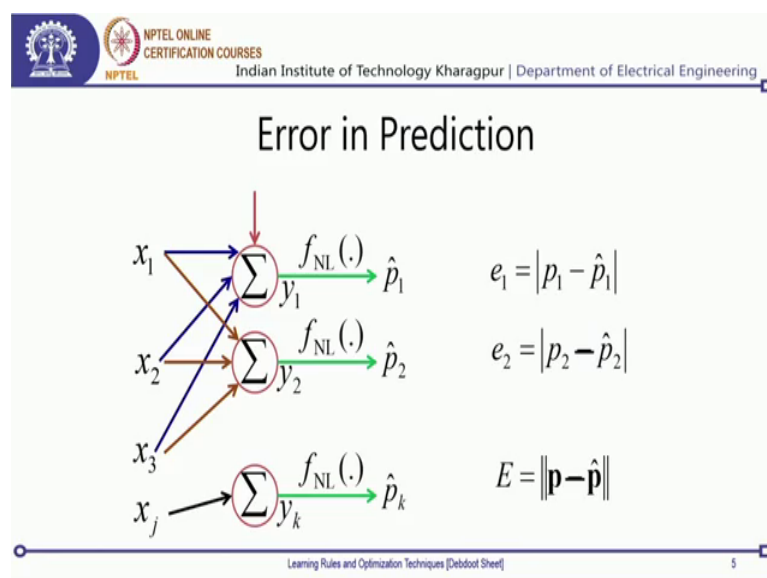
(Refer Slide Time: 07:28)



Now, for the second prediction, it would again keep on going in a similar way. So, in the second prediction, it gives you for p_2 , and then you have your equations coming down for p_2 . And similarly, you can keep on getting it down for any of the k th neuron over there, connect in the j th neuron and then you get your p_k hat. Now, once this set of equations comes down over there, you have your all the piece. Now, you can actually write them down in terms of some sort of a tensor representation where all of your y 's get represented in terms of a tensor, all the biases get represented in terms of a tensor. And then your weights are what were all say row matrices over there you just you place one row matrix below the other row matrix and subsequently you got a 2 d matrix over there of your weights coming down.

Now, with that you have this very simple equation which is just tensor product, inner product over the weight and bias matrix and the input matrix and then you have a non-linearity imposed on top of it. And the since all the y 's are independent of each other, each element of y is independent of each other, and it is tensor over there and the non-linearity is a scalar function. So, you can pretty much apply the non-linearity on the tensor and output is also going to come down as a tensor, so that is clean and clear with us.

(Refer Slide Time: 08:43)



Now, the point was in terms of error what we could do is we could measure down the error for each of these predictions over there. So, for the first prediction, I have one error;

for the second prediction, I get my error; for the third and then if you want to generalize it out, what we would typically end up getting is a Euclidean distance between the predicted variable and the true state of the variable. So, the true state of the variable which you would like to predict is what is p , and the predicted state of the variable is what is \hat{p} . And then my Euclidean distance is just going to be a scalar quantity over there.

Now, this would bring me to a point where I can have one unified matrix for measuring what is the total performance or what is the total amount of error the whole system mix and that is quite good because now I have a scalar matrix in order to do it and that is what we were doing. So, these errors over here for us in classification you could do a negative log likelihood, you could do a binary cross entropy. And in case of regression, you could do a mean square error you could do a l_1 norm over there. So, there were multiple ways of it which we were doing over there. Now, nonetheless the only imposition, which was imposed is that in some way the function error over there needs to be differentiable. And as well as for the whole network to sustain over there you also need to have the transfer functions and then network all of these relations as well as differentiable.

(Refer Slide Time: 10:01)

Error Backpropagation

| | | | |
|----------|----------|-------------|---|
| x_1 | p_1 | \hat{p}_1 | $J(\mathbf{W}) = \sum_n \ p_n - \hat{p}_n\ $ $\mathbf{W} = \arg \min_{\mathbf{W}} \{J(\mathbf{W})\}$ $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \frac{\partial}{\partial \mathbf{W}^{(k)}} J(\mathbf{W})$ |
| x_2 | p_2 | \hat{p}_2 | |
| x_3 | p_3 | \hat{p}_3 | |
| \vdots | \vdots | \vdots | |
| \vdots | \vdots | \vdots | |
| x_n | p_n | \hat{p}_n | |

Learning Rules and Optimization Techniques [Debabrata Sheel] 6

Now that brings us to the point of error backpropagation. Now, typically what we have during back error propagation is we would be following this standard error back

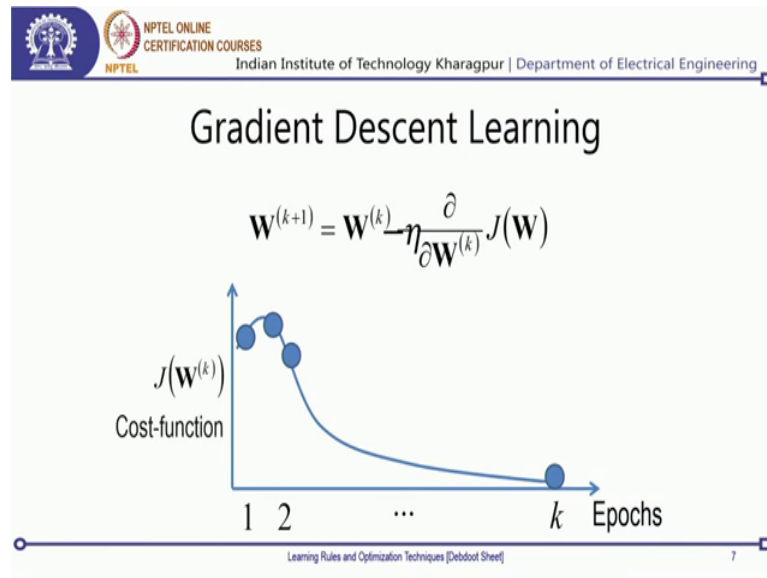
propagation rule in which the future state of my weight W of k plus 1 is what is dependent on the previous state of my weights over there, and then you subtract this extra part of it which is the error. Now, this error is something which is derived from the gradient of the cost function with respect to the weights which are coming down. And since the cost function is scalar quantity, but the weight being tensor over here, this derivative over here also turns out to be a tensor.

Now, how it comes down as it you have your number of samples given down to you. So, say your training set over here has x number of samples, now within a current epoch, so within any given epoch k , when your weights are at a certain value. So, when k is equal to 0, you would initially stand down with p oddly some random values of weights over there. And now with those random values of weights, whenever you set an input of x 1. So, x 1 over here is no more one input from a sensor, but then this is just one timestamp or one instance or sample of those set of inputs coming down.

So, x is a tensor. You see that written in bold. x 1 is first sample, x 2 is the second sample, x 3 is the third sample and this is how it goes down. Now, all these p , p 1, p 2, p 3 these are all the predictions which come down which are expected to be coming down; however, \hat{p} 1, \hat{p} 2, \hat{p} 3 these are the exact predictions which you get out of your neural network. And now based on these true classes or say the true value if we are solving a regression problem a true class if you are solving a classification problem you can actually use this. So, let us look into just the pure regression form of here, which is trying to do a l_2 norm minimization or Euclidean distance minimization.

So, this is how you get your Euclidean distance and the whole objective is that I want to have this weight, particular combination of weight where my sum of Euclidean distances over all the possible combinations of weights is minimum that is ideally what I would like to have. And that can be achieved by using this kind of a gradient descent learning rule. Now, that was my very simple plain gain definition of it.

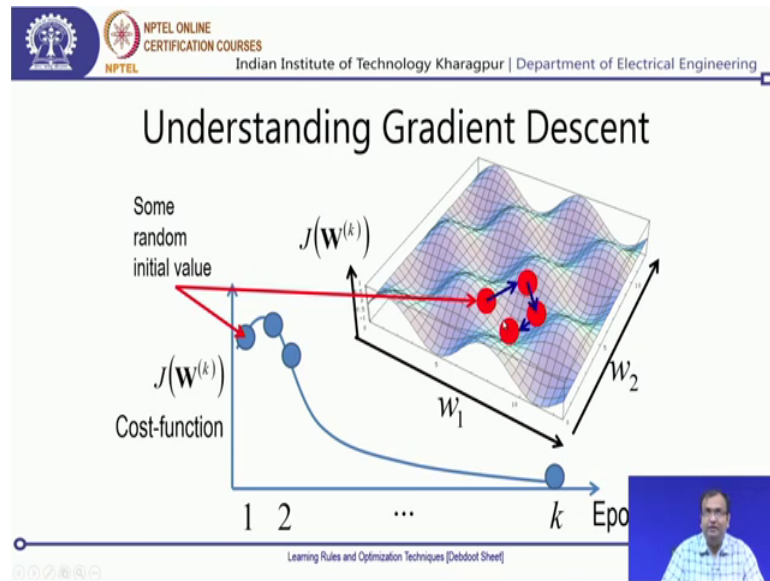
(Refer Slide Time: 12:26)



Now, based on that what we have realised is that we can now formulate this whole learning mechanism in terms of something called as a gradient descent learning rule. In which what comes down is that you would start with one of these assumptions on the weight over there; and then based on that you get down your J W which is the error and then based on that you can get down your $\frac{\partial}{\partial W}$ or J W or the derivative of the cost. And then using that you can update it can get down W of k plus 1.

So, you start with the first epoch, you get an estimate of the second epoch, you replace that. Now, with that estimate, you get an estimate of the third epoch, and you keep on replacing. And then subsequently what you could see is that as you keep on replacing and then it keeps on going. Since, this derivative has to be minimized you would see that this error also over here sorry since this error has to minimize by solving it out, so you see that this error would keep on going subsequently down that is what is happening if I am trying to look across epochs over here.

(Refer Slide Time: 13:17)



Now, the point is that if I try to visualize it on a 2D space as we were trying to do earlier and say that there were just two different weights over there w_1 and w_2 and such that this w_1 and w_2 can vary over a large range. Now, what would happen down is that you would start with some random value over here. Now, once you start with a random value you have a particular value of the cost function random value of w_1 and w_2 are the weights over there. You have random value of the cost function coming down. Now, based on that you have your error computed which is $J(\mathbf{W})$. Now, based on $J(\mathbf{W})$, you can take a derivative of $J(\mathbf{W})$ now using that you update you get the next estimate of w .

Now this next estimate of w is because the w or the weight over here has changed. So, this has shifted over to some other point over here, so that sets my second epoch. Now, it does not necessitate that at the subsequent epoch that your error has to always go down now that is something you are trying to achieve, but it does not necessitate that it will always go down. Now, if the error is a convex function it would always so convex function is always like this kind of a slope where you have one minima point over there. Now, if that is the condition then; obviously, it is going to come down; however, what we see is that it keeps on changing. And then, these weights also keeps on changing and subsequently what happens is that it comes down to a minima point.

And now if you had any mechanism in which you could like really play around with all the possible values of w 's over here then you would get down that this $J(\mathbf{W})$ would be

some sort of a surface over there. And this part of it if we are restricting our w 's within this small part over here then obviously, this minima there is one unique minima point over there and that is a cost function. However, if you are if you want to play around with this whole part then that is not a convex function because there are multiple points which have minima and maxima coming down? And sort of like all minima locations over here are equivocal minima and that is what something creates a major challenge of us.

However, with plain j and or the vanilla gradient descent the very simple mechanism is that you start with a point and then assume that there is a local minima or which is absolute minima for you and that is somewhere in the neighbourhood. And I will be able to come down to that position that is what we are trying to do.

(Refer Slide Time: 15:33)

Gradient Computation

$$\begin{array}{c}
 x_1 \xrightarrow{w_1} \\
 x_2 \xrightarrow{w_2} \\
 x_3 \xrightarrow{w_3} \\
 1 \xrightarrow{w_0 = b}
 \end{array}
 \rightarrow \sum \rightarrow y \xrightarrow{f_{NL}(\cdot)} \hat{p}$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial y} \frac{\partial y}{\partial \mathbf{w}}$$

Derivative of cost function

Derivative of the linear network


Derivative of non-linear transfer function

Learning Rules and Optimization Techniques [Debbot Sheet]

Now, what we were essentially doing is that in order to solve all of this we wanted we needed to do the gradient computation. And for that the whole idea was that if I have a very simple perceptron model over here, such that I am predicting out only one value \hat{p} using three different inputs three different scalars x_1 , x_2 and x_3 then this is what this whole equation would break it down to. So, your $\frac{\partial J}{\partial \mathbf{W}}$ would actually be product of three partial derivatives because your J is something that the cost over there the cost function does not have w in itself. So, it just has the prediction over there. So, it is a $\frac{\partial J}{\partial \mathbf{p}}$ of J .


Next, is that the j prediction over here this p is dependent on this y over here in terms of the non-linearity. So, that is the next part over it. And then finally, this y is dependent on all of these w 's, so you have a $\frac{\partial J}{\partial W}$ of y . Such that the first part is what is called as the derivative of the cost function, the second part is called as the derivative of the non-linear transfer function, and the third one is the derivative of the linear network. And you need all the three derivatives to exist in order for the gradient to be computed.

(Refer Slide Time: 16:44)

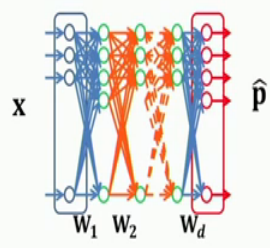


NPTEL ONLINE
CERTIFICATION COURSES

Indian Institute of Technology Kharagpur | Department of Electrical Engineering




Gradient Calculation



$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{y}_d} \frac{\partial \mathbf{y}_d}{\partial \mathbf{w}_d}$$

$$\frac{\partial \mathbf{y}_d}{\partial \mathbf{w}_d} = \frac{\partial \mathbf{z}_{d-1}}{\partial \mathbf{y}_{d-1}} \frac{\partial \mathbf{y}_{d-1}}{\partial \mathbf{w}_{d-1}}$$

$$\frac{\partial \mathbf{y}_1}{\partial \mathbf{w}_1} = \mathbf{x}$$



Learning Rules and Optimization Techniques [Debdoot Sheet]

Now, for the multi linear perceptron, the challenge was that here it was pretty simple, you had one output and just a simple connection, but then here you have subsequent number of them. And you do not know what is the state as the output of each of these. If you knew what is the ground state or what is the ideal expected value at the output of each of these hidden layers, then it becomes easy because then you can optimize one layer at a time. However, this is not what you have you have it for the whole set coming down over here. So, the idea over here is trying to break it down in a layer wise fashion such that if for a d th layer, I am trying to compute out then I would be breaking it down over there and for there. And then finally, for the first layer I will get down my $\frac{\partial J}{\partial W}$ as x , and this is what we were doing.

(Refer Slide Time: 17:26)

NPTEL ONLINE
CERTIFICATION COURSES
Indian Institute of Technology Kharagpur | Department of Electrical Engineering

Gradient Calculation

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{\partial J(\mathbf{W})}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{y}_d} \left(\frac{\partial \mathbf{z}_{d-1}}{\partial \mathbf{y}_{d-1}} \frac{\partial \mathbf{y}_{d-1}}{\partial \mathbf{w}_{d-1}} \right) \dots \left(\frac{\partial \mathbf{z}_2}{\partial \mathbf{y}_2} \frac{\partial \mathbf{y}_2}{\partial \mathbf{w}_2} \right) \mathbf{x}$$

Derivative of cost function
 $\nabla J(\mathbf{W})$


Derivative of non-linear transfer function

Derivative of the perceptron

Input to the network

$\nabla(\mathit{net})$

Learning Rules and Optimization Techniques [Debdoot Sheet]

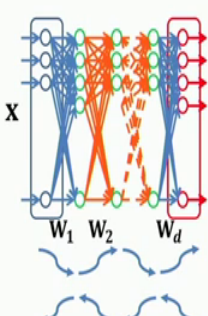


Now, on this whole thing, if you write it down cumulatively as one single update rule then this is the gradient which comes down, where the first part is what is called as the derivative cost function or the nable of j w . The next few bracketed parts over there is what belongs to the derivative of the non-linear function as well as you have the derivative of the perceptron at each layer coming down. And then this together is what is called the derivative of the network or nabla net and then you have your input to the network coming down over there. Now, this is what your whole gradient was looking like.

(Refer Slide Time: 17:57)

NPTEL ONLINE
CERTIFICATION COURSES
Indian Institute of Technology Kharagpur | Department of Electrical Engineering

Learning Rule



Step1: Forward \mathbf{x} to obtain $\hat{\mathbf{p}}$ net.forward()

Step2: Compute $J(\cdot)$ criterion.forward()

Step3: Compute $\nabla J(\cdot)$ criterion.backward()


Step4: Compute $\nabla \mathit{net}(\cdot)$ net.backward()

Step5: Update \mathbf{W} updateParameters()

Step6: Go to Step1 if $J(\cdot) > \epsilon$

Step7: End

Learning Rules and Optimization Techniques [Debdoot Sheet]



Now, in my learning rule, what I was doing is that for any given epoch what I would do is I have my weights available to me. So, I would do a forward of x and get down my prediction over there, then I would compute my error, and then I would compute my derivative of my error. Now, once that is available I compute the derivative of the network, and then I update my w . And this update goes down one-step at a time and that is the reason why it is a back propagation. So, from output stage to the input stage, it goes down. And then you get back to step number one and if until your error is below a certain limit you just keep on updating this one. So, it keeps, it is an iterative process which keeps on going over there.

Now, once it is over there you can end it out. Now, these were a few things on your codes which you were doing down typically in terms of what is a forward backward and parameter update over there.

(Refer Slide Time: 18:49)



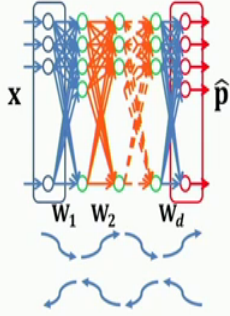
The image shows a slide from an NPTEL online certification course. The slide has a white background with a blue header and footer. The header contains the NPTEL logo, the text 'NPTEL ONLINE CERTIFICATION COURSES', and 'Indian Institute of Technology Kharagpur | Department of Electrical Engineering'. The main content of the slide is the title 'BACKPROPAGATION RULES' in large, bold, black capital letters. In the bottom right corner, there is a small video inset showing a man with glasses and a blue shirt speaking. The footer contains the text 'Learning Rules and Optimization Techniques [Debdoot Sheet]'.

Now, the point is we wanted to get into what were these different kinds of back propagation rules.

(Refer Slide Time: 18:53)

NPTEL ONLINE
CERTIFICATION COURSES
Indian Institute of Technology Kharagpur | Department of Electrical Engineering

Epoch Update



| | | | |
|----------|----------|-------------|---|
| x_1 | p_1 | \hat{p}_1 | $J(\mathbf{W}^{(k)})$ |
| x_2 | p_2 | \hat{p}_2 | $\nabla J(\mathbf{W}^{(k)}): x_n$ |
| x_3 | p_3 | \hat{p}_3 | $\nabla_{net}^{(k)}: x_n$ |
| \vdots | \vdots | \vdots | x_n |
| \vdots | \vdots | \vdots | $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}} = E[\nabla J(\mathbf{W}) \nabla_{net} x_n]$ |
| x_n | p_n | \hat{p}_n | $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}}$ |

Learning Rules and Optimization Techniques [Debdoot Sheet] 15

So, one of these very famous one is what is called as an epoch update rule. So, over here what happens is that you do a forward pass and then you keep on accumulating all the errors and then in one shot you just update the weight over there. So, typically what you have is say I have a number of samples in my training sets such that within any given epoch I am going to take down these n number of samples as my input over there. Now, for so that that is a pretty straight forward because you will do one sample one sample at a time and then you keep getting all of this. And then you get down your basic amount of errors. So, this error is what you compute at the kth epoch. So, if my weight at the kth epoch is \mathbf{W}^k then I get my J of \mathbf{W}^k . So, I just chose to remodify these variables a bit and write.

Now, when I am computing my nabla of J of \mathbf{W}^k , which is the derivative of J of \mathbf{W}^k then that is what is corresponding to each of these samples x_n . So, for each x_n , I will get down one-one derivative ok. Next for my network also because for each single sample input over there, there is a different value of my derivative coming down, so I keep each of them. Now, the point is when I want to do a weight update in an epoch, so I can just do it only once I cannot do it for each sample because that that just defies my whole constant. So, what I need to do is somehow I need to map down this complete amount of derivative onto something coming. Now, the point is earlier one we had looked that my $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}}$ is something equal to nabla of J of \mathbf{W} into nabla of net into

x n right. So, derivative of my cost function into derivative of my network into my actual input which was given down over there.

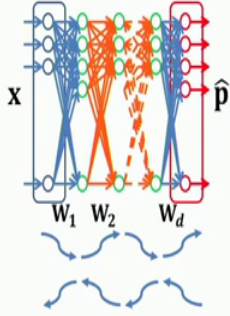
Now, since here I have these computed for each sample over there, what I would technically do is I will take an average over all of them. So, my simple point is my derivative of the cost function over the weight space at a given epoch is an ensemble or is an average over all the possibilities which were there at each of these samples. So, for some of the samples, it might be a zero gradient that is for some of the samples is a very high gradient, but I take this average value over there and solve my purpose. Now, that makes it easier because my update rule is pretty simple that I need to update it only once. So, these are what will get calculated for every sample which goes into it, but then I keep all of these values stored down, I take a average over all the values and then I do one update at a time, so that is pretty simple straight forward going down over there. Now, that is what is called as an epoch update rule.

Now, comes the challenge. Now, say that I give you a dataset which is about 100 GB in size. Now, for most of your systems, you will not have 100 GB of RAMs available to you. Now, the only option is that you keep on storing all of this as a swap file into your swap partition if you window system the; you would be doing a paging over there and just storing it as a page. Now, hard drives ios are what are too time consuming theoretically it looks pretty good I mean I have hard drive over there on which I can do, but then the data storage dates are a few hundred times slower than the data storage rates and acquire rates on a standard RAM. Now, that is the challenge that is the challenge which we have over here, and which is what got solved out with the batch update rule quite easily.

(Refer Slide Time: 22:19)

NPTEL ONLINE
CERTIFICATION COURSES
Indian Institute of Technology Kharagpur | Department of Electrical Engineering

Batch Update



| | | | |
|----------|----------|-------------|---|
| x_1 | p_1 | \hat{p}_1 | $J(\mathbf{W}^{(k)})$ |
| x_2 | p_2 | \hat{p}_2 | $\nabla J(\mathbf{W}^{(k)}): x_n$ |
| x_3 | p_3 | \hat{p}_3 | $\nabla_{net}^{(k)}: x_n$ |
| \vdots | \vdots | \vdots | x_n |
| \vdots | \vdots | \vdots | $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}} = E[\nabla J(\mathbf{W}) \nabla_{net} x_n]$ |
| x_n | p_n | \hat{p}_n | $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}}$ |

Learning Rules and Optimization Techniques [Debdoot Sheet] 16

Now, in a batch update rule what happens is that say that I am given down this n number of samples over here, and this n is something in the order of a few billion such that this is something say tens of billions and then your dataset is quite above 100 GB. Now, instead of taking all the samples together, what I would do is I would group it down. So, I would say over here I take just two samples. I can take multiple samples over there as well. So, typically like 100 samples, 16 samples, it depends on how much goes onto your memory and what your system can combine over there.


Now, for a given batch of samples or given small group of samples within an epoch, what I would do is I would find out these parameters over there. Now, this k is no more called as a epoch parameter when you are doing a batch update, but this is more of a batch iterative number. So, you can have multiple number of batches within an epoch. So, this k will be varying based on that. Now, in the next epoch this k would be a k plus 1 like that. So, what essentially means is that within every epoch, you will do multiple number of times.

Now, take a very simple example that say you have 1000 sample points over there. So, n is equal to 1000, and then your batch size is 10. So, you will get down basically 100 such small batches of 10, 10 samples each which means that within an epoch. If you are solving these equations then you are solving it 100 times, now obviously, it is computationally more expensive than the earlier mechanism of one single shot update of

an epoch because there it was updating only once anyways these gradients and everything is what you are computing down per sample case so that that computation remains constant. However, here what it was happening is over there you were doing it only once here you are going to do it hundred times. Now, that is a change which comes in because these weights are not a small number, they are a large matrix which you have to keep on modifying.


The next challenge which comes down is when you have this kind of a total epoch update rule, the major problem is that you will have to keep on storing all of these and you will have to load all the data in one single shot. Now, that is not something which is feasible. Now, for that reason we have this batch update rule, which is quite a useful one. Now, down the line when I enter into the optimizations and come down to stochastic gradient descent I would show you why batch update is quite good and what is the coolest thing which they use within a batch update in stochastic gradient descent. So, this is what happens for the first batch.

(Refer Slide Time: 24:50)

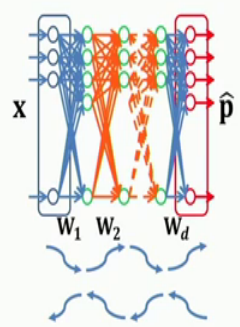


NPTEL ONLINE
CERTIFICATION COURSES

Indian Institute of Technology Kharagpur | Department of Electrical Engineering



Batch Update




| | | | |
|----------|----------|-------------|-----------------------------------|
| x_1 | p_1 | \hat{p}_1 | $J(\mathbf{W}^{(k)})$ |
| x_2 | p_2 | \hat{p}_2 | $\nabla J(\mathbf{W}^{(k)}): x_n$ |
| x_3 | p_3 | \hat{p}_3 | $\nabla_{net}^{(k)}: x_n$ |
| \vdots | \vdots | \vdots | x_n |
| \vdots | \vdots | \vdots | |
| x_n | p_n | \hat{p}_n | |

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}} = E[\nabla J(\mathbf{W}) \nabla_{net} x_n]$$

$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}}$$

Learning Rules and Optimization Techniques [Debdoot Sheet]

16



Next once that is done, so with the weights which was updated in the previous, one you would again compute and get this one for the second batch and then you solve it out. And then you keep on doing and so on and so forth such that you end up the last batch and then this is how it keeps on going. So, here it is no more called as a batch and a epoch, but more of these k is what is called as the iteration factor. And there are crazy different

ways, so some people what they do is they keep on sub sampling and taking multiple iterate's as well, so that that is also an interesting way of doing it out.

(Refer Slide Time: 25:18)

The slide is titled "(per) Sample Update" and features a diagram of a neural network on the left. The input vector \mathbf{x} is processed through layers with weights $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_d$ to produce an output vector $\hat{\mathbf{p}}$. To the right of the diagram, a list of samples is shown: $\mathbf{x}_1, \mathbf{p}_1, \hat{\mathbf{p}}_1$; $\mathbf{x}_2, \mathbf{p}_2, \hat{\mathbf{p}}_2$; $\mathbf{x}_3, \mathbf{p}_3, \hat{\mathbf{p}}_3$; and $\mathbf{x}_n, \mathbf{p}_n, \hat{\mathbf{p}}_n$. The n -th sample row is highlighted in orange. Below this list, the following equations are presented:

$$J(\mathbf{W}^{(nk)})$$

$$\nabla J(\mathbf{W}^{(nk)}): \mathbf{x}_n$$

$$\nabla_{net}^{(nk)}: \mathbf{x}_n$$

$$\mathbf{x}_n$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(nk)}} = \nabla J(\mathbf{W}) \nabla_{net} \mathbf{x}_n$$

$$\mathbf{W}^{(nk+1)} = \mathbf{W}^{(nk)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(nk)}}$$

The slide also includes the NPTEL logo and text: "NPTEL ONLINE CERTIFICATION COURSES", "Indian Institute of Technology Kharagpur | Department of Electrical Engineering", and "Learning Rules and Optimization Techniques [Debbod Sheet]" with the number "17" in the bottom right corner.

The next one is what is called as the per sample update rule or also a sample update rule. Now, this is something similar to the vanilla gradient descent in which what you try to do is that for every single sample, you will be updating. Now, the example comes down pretty simple, and you can actually look into your batch update rule and from there find out like how it works out. So, the idea is that if your batch size becomes one, so what you essentially have is one single sample going through it. Now, that is what we essentially do. So, you pass down one single sample now for this sample you compute out your factors over there.

Now, if you look over here what I did is I have changed this iterator factor to $n k$ now because n is the sample number over there, and k is the number of epochs on which you are going down. So, it makes it easier for me to map it. However, now since it is just for one sample, so you do not need to have an estimator running to get it down, so that is the change which comes down over here and then you have an update. Now, you do it for the first sample, you have the weight updated, now you do it. So, this is when the backprop happens, and the weight update goes down, then you do it for the second sample, you do a forward pass, you compute all of your parameters over here, then you get your gradient

and then you do a weight update. Now, once that is over you repeat the same thing for the third and then you keep on doing and so on and so forth till you reach the last sample.

Now, what you would realise over here is it is a good way because you are doing more number of updates. And in fact, what you would realize is that if you are plotting it down in terms of a number of epochs taken down, then in the next lecture where we do the labs you would realize that it would take you less number of epochs really less number of epochs to do it. However, each epoch is going to be very time consuming. So, the total time consumed way if you look into it you would find out that this is not something that is economically feasible to do. Now, I leave that part to the next lecture where I would be showing it with an exercise, we will be solving the same network optimization using three different ways and then find out like which is something which is suggested typically to do. So, I will leave that as an interesting point for the next lecture; now, having said that these were the different kind of update mechanisms.

(Refer Slide Time: 27:25)



The next point which you have is trying to choose an optimization technique. Now, over here one very popular one is what is called as a gradient descent technique.

(Refer Slide Time: 27:30)

Stochastic Gradient Descent

| | | |
|----------|----------|-------------|
| x_1 | p_1 | \hat{p}_1 |
| x_2 | p_2 | \hat{p}_2 |
| x_3 | p_3 | \hat{p}_3 |
| \vdots | \vdots | \vdots |
| \cdot | \cdot | \cdot |
| x_n | p_n | \hat{p}_n |

$J(\mathbf{W}^{(k)})$
 $\nabla J(\mathbf{W}^{(k)}): x_n$
 $\nabla_{net}^{(k)}: x_n$
 x_n

| | | |
|----------|----------|-------------|
| x_1 | p_1 | \hat{p}_1 |
| x_3 | p_3 | \hat{p}_3 |
| x_2 | p_2 | \hat{p}_2 |
| \vdots | \vdots | \vdots |
| \cdot | \cdot | \cdot |
| x_n | p_n | \hat{p}_n |

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}} = E[\nabla J(\mathbf{W}) \nabla_{net} x_n]$$

$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}}$$

Epoch 1
Epoch 2

Learning Rules and Optimization Techniques [Debdoot Sheet] 19

So, I will show you one epoch, so just between two epochs what is the difference which comes down. So, I will start with the plain simple batch mechanism of update. So, see when batch number becomes one, it basically boils down to the per sample update and that is something which is similar to the exact vanilla gradient descent. So, there is no change over there. Now, for stochastic gradient descent, you will either have to use a batch or you will have to have an epoch, but then for epoch it also it does not make a change. When we go through this, batching thing you will realise why it does not change for the epoch.

Now, over here in stochastic gradient descent what we typically do is that you have your training sample. So, you find out one of the batches over here. So, if this is your batch then for one of these batches this is the set of equations which you will be solving out. And then you keep on repeating over the different batches; for the second batch, you again have this one and then this is what is solved down and you end up coming to the last batch. So, this is (Refer Time: 28:29) for my first epoch what I have done great.

Now, the point is in my standard batching technique whatever I was telling is you will keep on doing the same thing. So, again in the next time, you will get your x_1 and x_2 in your batch and then x_3 and x_4 in your batch and then subsequently x_{n-1} and x_n in your batch and do it, but then in stochastic gradient this is not what we do. So, what we do is within every epoch we would randomize the samples over there. Such that



say in the second epoch you have a different kind of a batch formation. So, you take the first sample, you put it down in the second. The second sample now enters in the third position and then the third sample enters into the second position such that your first batch now in the second epoch is something which is made out of x_1 and x_3 . So, the first sample and the third sample, no more the first and the second sample.

Now, what this would change is that because of this change over here, so obviously, the nature of these expectations as well changes, and that is the reason why you get down a good amount of variance by just twindling out the samples. So, this will reduce any kind of a local lock in condition. So, sometimes it might happen that two or three samples always whenever they come down in a pair or certain way then this expectation over there. So, individually one might have a positive gradient, another might have negative gradient, but both of the same magnitude. So, this expectation would end up becoming zero and essentially there is no update going down.

Whereas, the total error over there is usually high it was just because of this expectation over there or the average value that the total error was coming down as 0, and there was no update happening over there. But when the moment you keep on changing this per epoch, so maybe in the next epoch it just shuffles out and just because of that shuffle though the weight was not updated in the earlier case, but now because of that shuffling you have a higher error coming in and that is what will reduce this lock in. So, this randomization is what is present in a stochastic gradient descent.

So, the stochastic gradient comes from the concept that instead of estimating the ideal value or the absolute value of the gradient in one epoch, we are just estimating it in terms of a fixed number of samples in that epoch and that number of samples is what is called within the batch that is the only difference. Now, this one great way of reducing local lock in conditions; however, there is another very powerful mechanism, which is called as adaptive momentum technique.

(Refer Slide Time: 30:59)




NPTEL ONLINE
CERTIFICATION COURSES
Indian Institute of Technology Kharagpur | Department of Electrical Engineering

Adaptive Momentum (Adam)

$$\mathbf{m}^{(k+1)} = \beta_1 \mathbf{m}^{(k)} + (1 - \beta_1) \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}} \quad \hat{\mathbf{m}} = \frac{\mathbf{m}^{(k+1)}}{(1 - \beta_1)}$$
$$\mathbf{v}^{(k+1)} = \beta_2 \mathbf{m}^{(k)} + (1 - \beta_2) \left(\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}^{(k)}} \right)^2 \quad \hat{\mathbf{v}} = \frac{\mathbf{v}^{(k+1)}}{(1 - \beta_2)}$$
$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}} + \epsilon}}$$

Learning Rules and Optimization Techniques [Debdoot Sheet]



Now, what happens within an adaptive momentum technique is something of this sort. So, in an Adam, you change the set of equations over there totally, the weight update no more looks the same way. You make use of two different quantities, one is called as the momentum, the second is called as the velocity. Now, let us look into how this momentum is defined. So, you have one variable which is called as \mathbf{m} and this is also a tensor. Now, we will get down to why this exactly is a tensor. In general, because you have this derivative coming down, so obviously, this is a tensor and anything derived on top of it is a tensor.

Now, what I would do is I would start with some random value of momentum at π zero at epoch and then keep on following this update rule in order to keep on updating. So, at my next one, I will have this updated over here. Now, this is what would come down as a update mechanism on the momentum. Now, my $\frac{\partial \partial J(\mathbf{W})}{\partial \mathbf{W}}$ of \mathbf{W} is the same. So, if I am doing a standard say a batch wise mechanism over there then this is what will come down and in fact in your adaptive momentum as well your shuffling of samples between epochs for creating batches is also pretty much valued.

Now, once I have my momentum coming down, I find out an estimator of this momentum which is called as $\hat{\mathbf{m}}$ at any particular given epoch key epoch or batch whatever you can call it down. Now, this β_1 which is one of this relaxation factors, there are two ways of it either you can have a very static β_1 which is more

preferred practice, but you can also have a beta one which is varying over epoch, so that can also be put into place. But for most of the practical implementations we have beta one which is a constant value.

Now, similarly you find out the second quantity which is called as the velocity v that is velocity is something which is dependent on the second order of the square of the derivative over there. So, this is not the second derivative, but it is square of the derivative a much sharper function. So, you have one which is depending just on the derivative, and the other one which is dependent on the square of the derivative. And then similarly you come down to an estimate or normalized estimator as well for v ; and then finally, my update is something which is defined as over here. So, in the earlier case, you just had a η times of $\frac{\partial}{\partial} W$ of j W , but here I take then whole ratio between these two.

Now, the whole point of getting down a square root over here is basically because this is a squared order this is not in the same order as that of $\frac{\partial}{\partial} W$ of J W . So, for that reason we just take a square root over this one and then this is what comes down and constitutes a very simple mechanism called as Adam. Now, this epsilon is also a constant number, which is typically given down such that this denominator does not become zero; or else otherwise if there is a chance that v can become down zero. So, b can become zero at any given point of time. And then if say this is 0, and then oh sorry this would be v this is not m , and this v value over here also would become 0, then comes down the problems. So, we do not just want to make that as an issue and for that reason we put down this extra epsilon offset over here.

Now, these are typical ways in which you can do different learning rules and optimization techniques and that is what so stay tuned onto for the next lecture, where we do a hands on with this whole thing, and then you can keep on enjoying further based on that.

With that, thanks.