

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



Lecture - 08
Multilayer Perceptron and Deep Neural Networks

So, welcome. Today we would be starting up with our next version which is on Multilayer Perceptrons to Deep Neural Networks. And while in the earlier lectures we have all studied about what neural networks are, and we have done a few lab sessions as well on and that was about just understanding neural networks from a classification point of view.

And now where the extension comes down in perspective of visual computing is that, here in contrary to what we were doing in the earlier classes was that, in the earlier classes while we were using feature extractors and feature descriptors which are hard coded functions over there in order to describe an image. And then we extended all the summary or the synopsis coming out of each of these feature descriptors together into a classification of framework using a neural network.

On the contrary today when we are going to do it, here is when a neural network itself has to come down to be an end to end learning framework, which means that input to the neural network itself is an image, while the output from it is still a classification output. So, it can be a classification, output it can be a regression output any of these things which come out. So, where we would start down very specifically is that, here we are going to look into a multilayer perceptron and that is our starting point, and from there eventually we will enter into what is known as the deep neural networks, and then what are their existential criterions and how they are work.

(Refer Slide Time: 01:48)



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

Overview

- Review of a Perceptron
- Multilayer Perceptron Model
- Signal Flow Graph Representation
- Gradient Calculation
- Existential Criteria
- Learning Rule

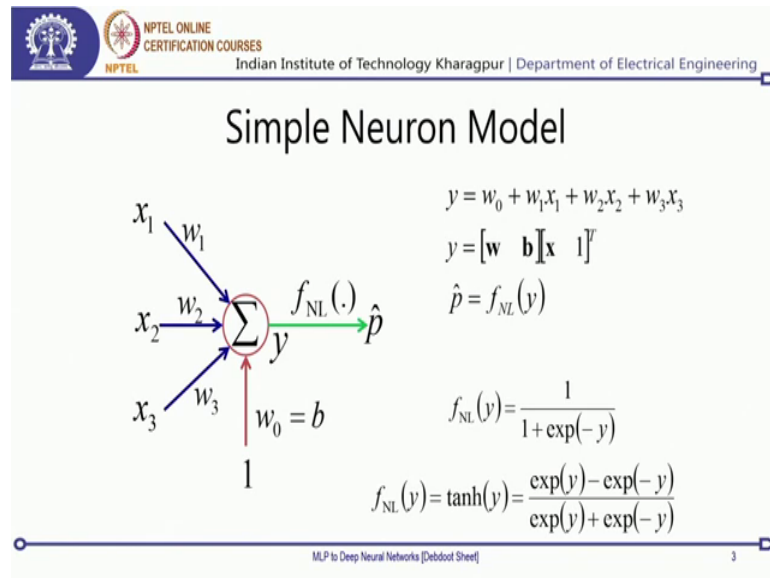
MLP to Deep Neural Networks [Debdoot Sheel] 2

So, effectively we would be doing a basic review of the perceptron model, and the perceptron learning rule once again, and then entering into the multilayer perceptron from there we enter into something called as the signal flow graph representation, and this model of a signal flow graph is how is my input and my output related and what happens during the learning phase.

And this is a quite critical part over here, since in the last lecture and the lab which we had done. So, you were introduced to the concept of error back propagation, and from there we had a gradient descent based learning rule. Now what exactly happens in telling this as a error back propagation and why it happens the way it has it has been named, and what you have seen down in different snippets of code is what we are going to explain you through this signal flow graph representation.

Following that is a very important aspect about gradient calculation, and that is to show down what happens within these functions, and whether the gradient is just for the classification cost function or then or does it need to exist throughout the network. And that is where we will enter into something called as an existential criteria for the network to exist and all other transformations and cost functions also to exist, and then eventually go down to the learning rule. And from there we more or less come down to an end of what happens were done with these kind of deep neural networks.

(Refer Slide Time: 03:14)



So, as with a simple neuron model, just to do a brief recap of what it was; so, say that there were 3 inputs over here, in the earlier case last week when we were doing it. So, these were given down as features say 3 different features, but here now these are no more 3 different features, but these 3 can be 3 pixels. So, you can consider just 3 pixels in an image, and give them as an input to the. So, the pixel in its own way or it can be even say for a given pixel in colored space if you have one particular image in RGB color space. So, each component itself is represented as one independent scalar value. So, your x_1 can be the red value of a pixel, x_2 can be the green value of a pixel x_3 can be the blue value of a pixel and accordingly. So, per pixel basis you can make some sort of a decision coming out as well.



So, let the decision associated with a particular pixel over here be \hat{p} . And now with the simple neuron model what would happen is that, we will have a weighted combination of these inputs going down to a neuron and from there add down a bias, take a summation out over them and this summation is what it has what has this form. So, it's w_0 plus w_1x_1 plus w_2x_2 , plus w_3x_3 , where each of these weights w_1 , w_2 and w_3 are 3 weights associated with each of the 3 values x_1 , x_2 and x_3 and w_0 is what is called as the bias or the w_0 can also be written down as 1 into b where say b is the weight over there and the constant input to this particular edge over there is what is 1 . So, in its linear algebra form which is in its matrix representation, this is a form which

you would be getting now. So, you get y is equal to the inner product or the dot product of 2 matrices.

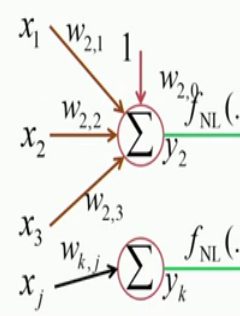
One of them is the weight matrix where you have the weights and the bias, taken together and the other matrix is a column matrix over there. So, that is why its x comma one transpose; where x is this scalar arrangement. So, capital X is basically a matrix arrangement of these 3 scalars which you get down over here. Now having taken all of them together, the next part is to apply some sort of a non-linearity and that is the fNL non-linear function which you get down over here. And these nonlinearities can have multiple different forms and we consider these 2 forms over here. The first form is called as the sigma the second one is called as the tan hyperbolic non-linearity and to do a very basic recap.

So, you remember that in sigma what happens is that as the value of y tends towards plus infinity, this value tends towards plus 1 as the value of y tends towards minus infinity this value tends towards 0. And on the contrary with the tan hyperbolic what happens is as the value of y tends towards plus infinity you get a value which is saturating at plus 1, as the value of y tends towards minus infinity you get a value of this non-linearity which is at minus one. So, taking these 2 together is either one of them you can be using now and based on, whichever you are using your \hat{p} value will appropriately be decided. So, if your \hat{p} has the non-linearity associated as a tan hyperbolic its value will be in the range of minus 1 to plus 1. If it has sigmoid as non-linearity then its value will be in the range of 0 to 1 and this was the simple perceptron model which we have done.

(Refer Slide Time: 06:31)



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

Neural Network Formulation



x_1 $w_{2,1}$ 1 $w_{2,j}$
 x_2 $w_{2,2}$ \sum $f_{NL}(\cdot)$ y_2 \hat{p}_1
 x_3 $w_{2,3}$ $f_{NL}(\cdot)$ y_k \hat{p}_k
 x_j $w_{k,j}$ \sum $f_{NL}(\cdot)$

$y_1 = [\mathbf{w}_1, b_1][\mathbf{x}, 1]^T$ $\hat{p}_1 = f_{NL}(y_1)$
 $y_2 = [\mathbf{w}_2, b_2][\mathbf{x}, 1]^T$ $\hat{p}_2 = f_{NL}(y_2)$
 $y_k = [\mathbf{w}_k, b_k][\mathbf{x}, 1]^T$ $\hat{p}_k = f_{NL}(y_k)$

$\mathbf{y} = [y_1, y_2, y_3, \dots, y_k]^T$
 $\mathbf{b} = [b_1, b_2, b_3, \dots, b_k]^T$

$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix}$ $\mathbf{y} = [\mathbf{w} \ \mathbf{b}][\mathbf{x} \ 1]^T$
 $\hat{\mathbf{p}} = f_{NL}(\mathbf{y})$

MLP to Deep Neural Networks [Debdoot Sheel] 4

Now, from taking down a perceptron to getting into a neural network formulation, which is that given I can have multiple kinds of inputs over there it can be different kinds of scalars over there, and I can map it down to again a different group of scalars. So, maybe my first prediction over there is what is called as \hat{p}_1 , and this will be the form of representing everything to my \hat{p}_1 . Now note over here that as we had also discussed in the earlier class is that these weights are no more with just one subscript, but there are 2 subscripts which you see with these weights 1 comma 2, 1 comma 1, 1 comma 3. Now the reasoning behind these weights is that the first subscript is to the target where it is mapping. So, the output over here say I have x_2 which goes to y_1 and that eventually maps down to my target which is called as \hat{p}_1 .

So, my first subscript is going to be the subscript of the target, my second subscript on the weight is the subscript of the source from where it is connecting, and that is the nomenclature which we are following. Now if I arrange all of these weights w_{11} , w_{12} , and w_{13} in a row matrix form then that is what is written down as this bold \mathbf{w}_1 , which is the matrix given down in the equation. Along with that I have my scalar value which is my bias w_{10} or b_1 and accordingly \mathbf{x} is my x_1 transpose is my column vector which comes down and this gets my inner dot product and then my non-linearity applied.

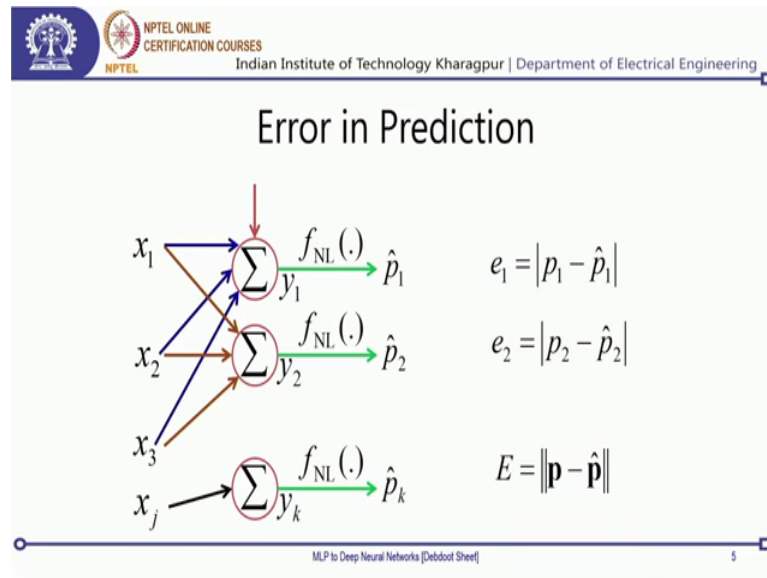
Similarly if I take down my second neuron on the output side of it, and feed it appropriately. So, I would be getting down this second part of the partial network coming down and my group of equations which represent that. Now projecting onto this and going out similarly. So, I can have my x_j th neuron connected down to my y_k th neuron with a weight which is called as w_{kj} , and then put a impose non-linearity on top of it. And then taking all of them together this is a particular form where I get where w_{kj} is a rotate row vector which has a size of $g \times x$. So, the number of connections over there will be basically from 1 to g , which is because that is a total number of x s which you have over here. So, for this combination this is a particular kind of an output relationship which we see.

Now if you look into this matrix of weights and biases which are come together, then you can see down that all of these are outputs which I see y_1, y_2 up to y_k . So, if I take all of these together and just concatenate them. So, I arrange them in a column form in a call in a column major format, which is that it is just has k number of rows and just one column over there; accordingly my bias b that can also be arranged into a column matrix over there.

So, these are the 2 matrices which we see over here, and then my w each of these w_1, w_2 up to w_k they can also be stacked one on top of the other, because each is independent of the other one and now that would give me some sort of a rectangular matrix. Now if I clearly look into it then my total transformation equation over here can be written down in terms of just a matrix multiplication. So, this will be a matrix multiplication of my weights w and b bias these 2 kinds of matrix with the input over there, which is erased as a matrix and then that gives me an output matrix over there.

And this output matrix is a column matrix; my input matrix is also a column matrix. So, that is my y and then I have a non-linearity applied on a matrix, which means that each element of the matrix is appropriately subjected to the non-linearity over there, and then taking all of them together is what I get down as my target output. So, this was my very basic understanding of how a neural network works down as such, and then this was what we had done with multilayer perceptron in the last class itself.

(Refer Slide Time: 10:44)



Now, again going down a bit more into the revision part over there; so my error in prediction how it was different was that if I have one of these predictors p_1 , then I get down one value of a scalar for another predicted each p_2 , I get down an others error which is e_2 .

Now if I have an array of these predicted variables over there, then I cannot keep on calculating each and every error singly because in that case I do not get a consolidated knowledge about the total network as such. So, in order to do that, what we do is we find out what is the Euclidean error over there. So, a Euclidean error or the total error of the network is basically a scalar value which is the Euclidean norm or the l_2 norm of all my predictors. So, whatever is my actual ground truth which is \mathbf{p} , and my predicted value $\hat{\mathbf{p}}$. These 2 matrices are subtracted and then you take the amplitude of that or the l_2 norm of these 2 subtractions.

(Refer Slide Time: 11:42)

Error Backpropagation

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

MLP to Deep Neural Networks [Deeboot Sheet] 6

So, and given that the next part is that you will be doing a back propagation in order to learn down your algorithm. So, the idea is that you have these successive bunches of observations and predictions, and what is the ground truth.

So, if x is a matrix is a of all of these scalar x s over there, and one of these samples is x subscript 1. Then the ground truth corresponding to that is what is p subscript 1 and at any given point of time, when you feed the whole data x subscript one through your network you would be getting down a predicted value which is p hat subscript 1. So, similarly I take my input sample as x subscript 2, and I feed it forward through my network I get a p hat subscript 2, while the actual ground truth over there is p 2. So, I keep on doing this together and then for my n -th sample which is my last sample in my training data over there.

As I feed my last sample through my network over there my output is p n hat and that my actual ground with corresponding to that is p n. So, together if I have all of this. So, what you would see is that, there comes an error which is there for each sample. So, for my first sample, second sample, third sample till my n -th sample I will be getting a different value of error, but can we give some sort of a consolidated error for the network in terms of its performance across all of these training examples which we are taking now.

And for that reason we devised another metric which is called as the cost function of the whole network in terms of its weights. Now that is what is defined as j w over here, and

that is a summation of the Euclidean distance between these predicted \hat{p} and the actual observed and the actual ground truth of these p s we just supposed to be there.

Now if you look into this cross function over there what we said is that $J(w)$ is your cost function where these varies in terms of your weights which are w . Now what comes down definitely in some bodies mind is that, why is it varying with respect to something called as a w ? And the reason is that this w s are weights which are the only thing which now would be guiding down and accordingly manipulating what happens to your predictions over there.

So, because there is not anything else on which it can change. See my input x is constant. So, that will be different number of samples and across samples and across. So, between 2 samples it will be a different value that is always known, but when I am training across an epoch learning, which we had done studies in the last classes as well.

So, when you are training across epochs what happens is every epoch you are going to send the same sample over there the only reason why the prediction value bit of putting down the same sample. So, say at my first epoch which is my epoch number 0 over there, I put down x_1 as my sample and I get down a predicted value \hat{p}_1 . I do all my updates and everything and then it comes to my second epoch which is epoch one in my epoch one I put down x_1 it would be getting known a different value which is \hat{p}_1 , but \hat{p}_1 at the epoch one is very different from \hat{p}_1 at epoch 0.

And the only reason why this was changing is a within the network, the only variable component is weight which changes. So, that is the reason why we would write down this cost function in terms of our weights itself. Now that I have my cost function written down in terms of my weights, my final point is that we need to come down to a point where to a point in the weight space, such that my argument of these cost functions is minimum or as I keep on it says the point is that, if we keep on changing the weights there will be one particular combination of these weights such that my error is minimum, and that is the exact one which I would like to achieve.

Now and how that achieved is through something called as the gradient descent learning rule. So, in this gradient descent learning rule what we do is basically that its an iterative process, in which what you do is you start with some randomized assumption of weights

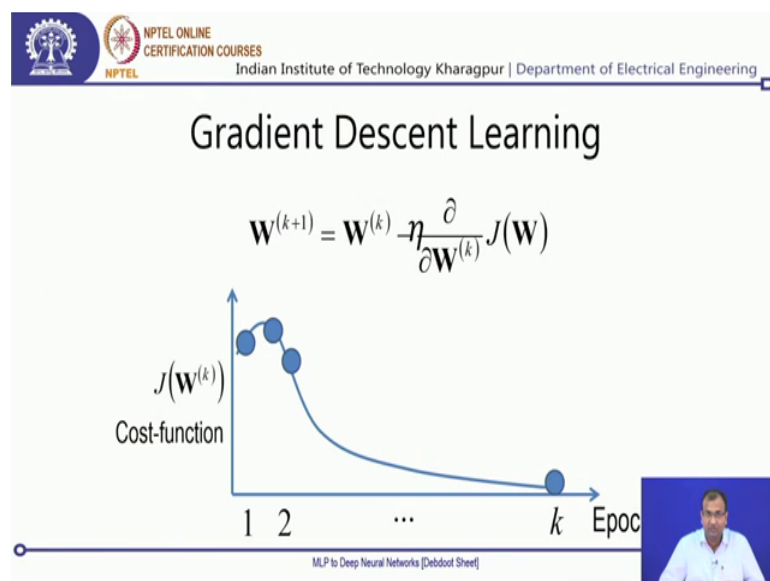
in the first epoch, and then see that is w^k and then you compute or whatever is your gradient of the weight space over there in terms of your cost function.

And that is $\frac{\partial}{\partial w} J(w)$, and then you weigh it by a factor an empirical factor which is called as η also known as a learning rate. So, what this controls is that, your gradient of this error function over the $\frac{\partial}{\partial w} J(w)$ that can have any range of a value. Now if the ranges of say these w weights are in a range of 0 to 1, and then say my $\frac{\partial}{\partial w} J(w)$ is in a range of 10^{-9} .

So, the rate at which it would be updating or impacting the value of w is going to be very less, and in that case this η factor over here comes to your rescue; because what you can do is you can set an η factor say 10^6 . So, if I multiply your value in 10^{-9} to a value of 10^6 that will put me a give me a value which is in the range of 10^{-3} . And that value is something which will actually be impacting significantly how the value of w is changing over there.

So, this learning rate basically is a fact way of mathematically modulating the gradient over there, such that we have a value of this error and the gradient coming down which will be in some way significantly impacting the change in w , and that is how my w^{k+1} will be revising at a much better rate, then w^k would have if we did not have this learning rate called as η .

(Refer Slide Time: 17:29)

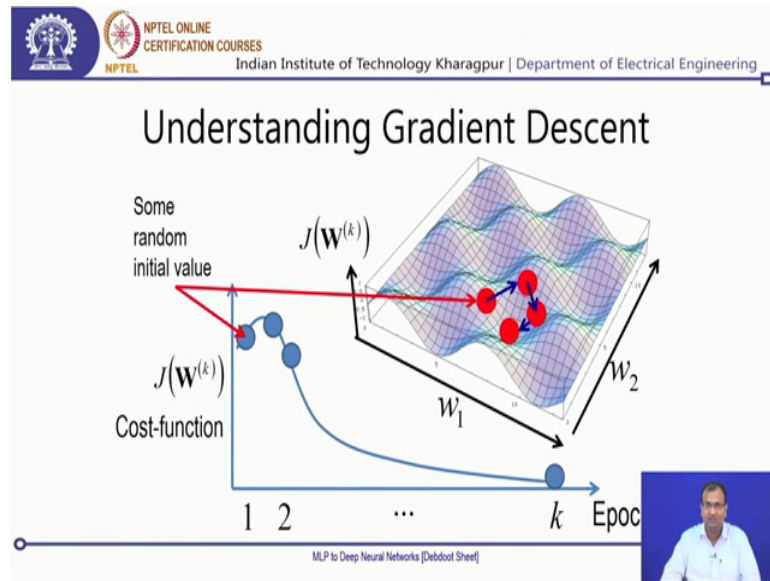


Now given that we have all of this done, so, what typically happens within a gradient based learning rule is something like this, that you would be starting down on your first epoch and then as you change these cases. So, you will be getting your first value of J_w , based on your J_w you would be calculating your gradient, multiplying that with your empirical constant η or the learning rate and then you update your w to w of k plus 1. As you update it to w of k plus 1 you would be getting a different factor of J_w , which is J_w of k plus 1 and accordingly these keep on changing and so on and so forth till you are at the final conclusive step over there.

Now this was one way of trying to visualize our learning in terms of its cost function versus epochs and this can be a typical graph. So, you would often be seeing that you start with a particular error and your error increases and then keeps on decreasing, or it may so happen that it increases then keeps on decreasing and suddenly again it's a local maxima it increases slightly then goes down, it may keep on jittering. And these are all aspects about what is happening within the learning itself.

So, if your value of η I mean you can keep your value of η very very less in that case what will happen is it will come down very slowly, but it will be a very smooth transition which it will be getting. If your value of η is very high then it can start oscillating and jittering over there and; that means, that it's basically overshooting the local minimum point at every time, when it's coming down somewhere closer to the local minima and these are different issues which we would be tackling down through experimental processes and some more learning experiences subsequently.

(Refer Slide Time: 19:05)



Now, looking at the same part of gradient descent again in the weight space, what we had learnt was that say I start with some random value over there, and this is my point and if I. So, here what I am doing is typically I am looking into 2 different plots. So, one is my plot of epoch versus cost function, the other is my plot of weight space versus cost function and these are 2 different aspects as such.

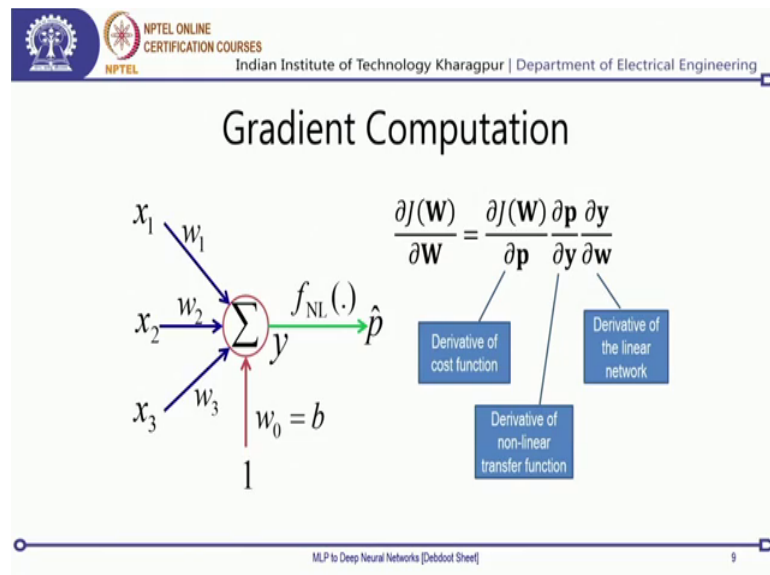
So, the second epoch when I update my weight; so it shifts to a different weight vector coordinate over there, and subsequently I have a different cost function value also calculated through it, and subsequently it goes to the next one then to the next one and finally, to my convergence. Now if you look into this part of the plot what you would see is that, for any kind of a perceptron model you would be seeing that the error function over there form some sort of a very structure, which quite mimics a cascade like design where you have multiple number of crests and troughs present over there.

So, as these points of my weights they keep on moving down, they would always be encircling and coming down to my local minimum point as soon as possible, and the way it comes down to this local minima is what is my learning which is happening. And also from the last lecture about introduction to deep learning and what happens within this multilayer perceptron line challenges, you did understand that one of the major points is that we can actually initialize a network at any random point over there, and based on that it can start converging and oscillating around any of these trucks and that definitely

means that where it is going to converge is now some sort of dependent on where I started.

If I started down in the neighboring one, then it will be in the rough of the neighboring if say there is no global minima, but everything is equivocal point over that there is no unique global minima in that case now. If we have unique global minima then the challenge is obviously that you do not lock into any of these non global minima positions, but rather somehow escape into this from this small trough like regions and exactly converge onto your global minima position.

(Refer Slide Time: 21:14)



Now, having said all of this, what comes down to our mind is something interesting. So, you have seen that there is for any kind of a given network, if I have 3 different scalar values over there then I can take in these scalar values, and I can predict out one of these predicted outputs over there. And for a simple perceptron how it goes down is something of this sort that I have my weights w_1 , w_2 and w_3 , I take all of these weights together and then I sum them up and accordingly I get now my bias also coming into play and then I am wrapping down to my output over there ok.

The question is that now that I need to find out my $\frac{\partial J}{\partial w}$, then what would I be doing. So, in order to solve it out, the best possible way is basically trying to look into something which is called as the chain rule of differentiation which you have done in

your high school mathematics itself. So, what goes down by the chain rule is that, we would try to break down all of these into its constituent components.

So, the simplest way of doing this is that let us bring down this derivative product which is partial derivative of J with respect to w , because we cannot directly compute. So, what we will be doing is. So, you know that the output of this J was a cost function and that was for our case a Euclidean distance of the predicted output with respect to the ground truth. So, I do not have any component of x as such directly visible neither my weights directly visible over there. But what I have is definitely my \hat{p} or p over here which is my predicted output state.

So, I can take a derivative of the cost function with respect to this output, next is my output is dependent on y through our non-linear function. So; that means, that I cannot take this partial derivative of p with respect to y . Now if I look till the first 2 partial fractions on my right hand side over there. So, you can see that $\frac{\partial J}{\partial w}$ can now be represented as a product of $\frac{\partial J}{\partial p}$ and $\frac{\partial p}{\partial y}$.

So, together these 2 first 2 parts of other will give me $\frac{\partial J}{\partial y}$ coming down. Now the next part is now that I have a $\frac{\partial J}{\partial y}$ I should be getting on another part of the partial fraction which is partially a partial fraction of these derivatives, which is my $\frac{\partial J}{\partial w}$ of y which is my output of this summation block in my neurons. So, together this is what will help me in getting down my total gradient computation.

Now, the first part of this gradient which is the gradient which is some sort of a derivative of the cost functions. If you look into the second part of the gradient, then you see that its a derivative or the non-linear transfer function. And if you look into the third part of the derivative that is a derivative of the linear network itself. And these 3 things together are what will be helping me in finding out the gradient part for my whole network in order to learn.

So, with this I will end up our lecture for today over here and in the subsequent lecture we would be starting up with this point on gradient computation, and subsequently going down to how this can be extended for a multilayer perceptron and then enter into eventually the deep learning, and how to train down these deep neural networks and then what will be the exist engine criteria.

With that, thank you and stay tuned for the next lecture.