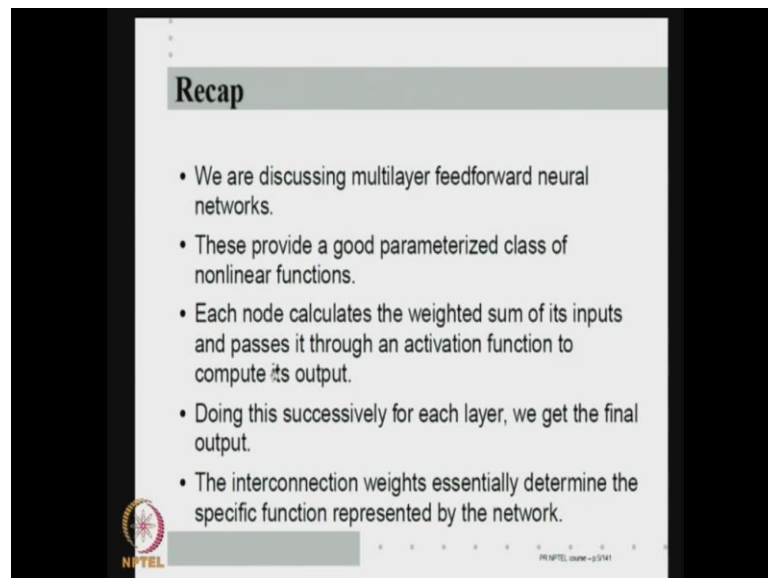


Pattern Recognitions
Prof. P. S. Sastry
Department of Electronics and Communication Engineering
Indian Institute of Science, Bangalore

Lecture - 28
Backpropagation Algorithm; Representational
Abilities of Feedforward Networks

Hello and welcome to this next lecture on pattern technician. As you know we have been discussing the multilayer feedforward networks, we looked at them in general. And in last class we looked specifically at multilayer feedforward networks as good models for representing non-linear functions.

(Refer Slide Time: 00:42)



The slide is titled "Recap" and contains a bulleted list of points. At the bottom left is the NPTEL logo, and at the bottom right is the text "NPTEL course - 31241".

- We are discussing multilayer feedforward neural networks.
- These provide a good parameterized class of nonlinear functions.
- Each node calculates the weighted sum of its inputs and passes it through an activation function to compute its output.
- Doing this successively for each layer, we get the final output.
- The interconnection weights essentially determine the specific function represented by the network.

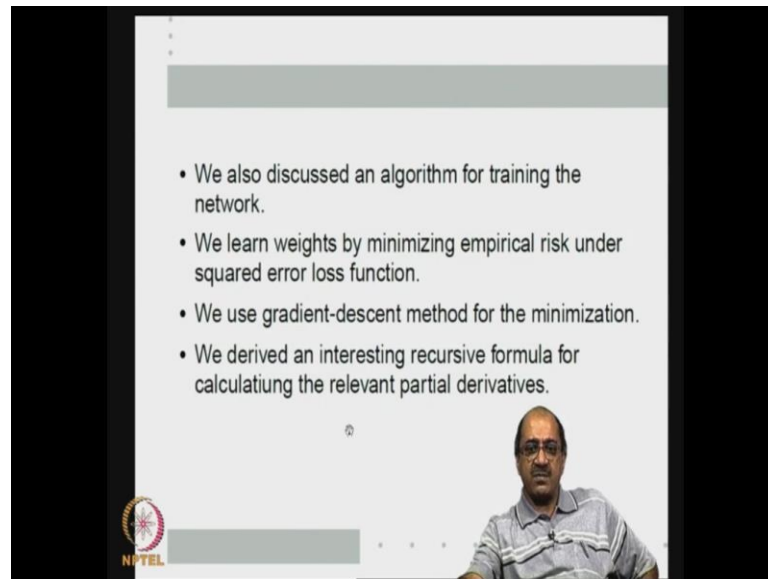
So, as I said last class, our interest in multilayer feedforward neural networks is that they provide a good parameterized class of non-linear functions. We have seen that last time. And basically, the way the network represents a function is through composition of linear sum and sigmoids. So, essentially each node calculates the weighted sum of its input and passes it through a non-linear activation function to compute its output. So, we do it for all nodes in layer and from layer to layer. So, if we do it successively for each layer, we essentially transform the inputs at the layer one to the outputs at the final layer.

So, the actual function that is computed by this is very much dependent on the interconnection weights. Because, given a particular structure the only thing that can change is the inter connection weights. So, by changing the interconnection weights we

change the function that is being represented by the network. So, roughly an architecture represents a class of functions and specific weights represent a specific function.

So, we have to essentially choose the weights to be able to represent this specific function of interest.

(Refer Slide Time: 02:03)



So, we actually discussed an algorithm for training the network. So, like all other classification regression models that we are considering, the training proceeds by having some training samples. So, we are given examples x_i, y_i ; y_i is the target, x_i is feature vector input vector. And we need a function relationship such that, given new x 's, I can predict the y 's properly.

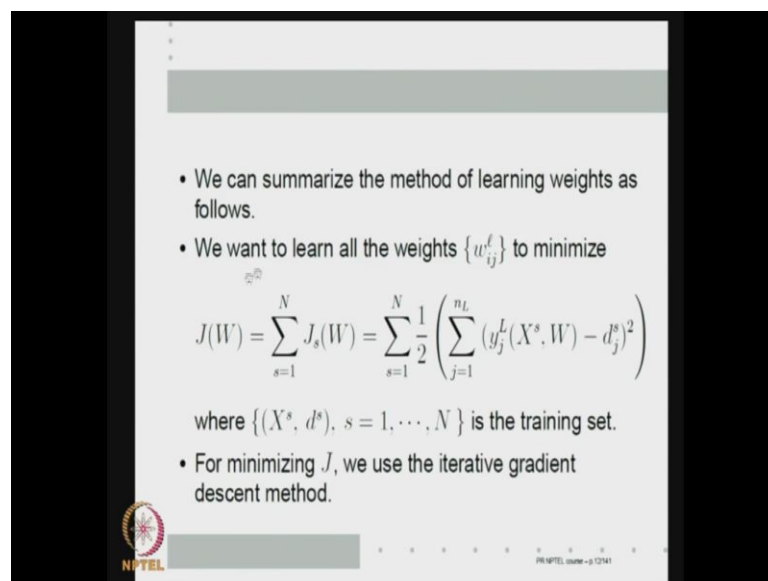
So, given the training examples x_i, y_i , we have to learn the model. So, in this case we have to learn the weights. So, we actually discussed in last class, an algorithm that is used for training the network. What do we actually? We learn the weights by minimizing the empirical risk and squared error loss function. Once, we fix the architecture of a network as we seen as I said earlier, it represents a class of functions. So, we can think of that as the class of functions or the class of classifier h that we are currently considering in the discrimination frame work. Then, we have chosen the squared error loss function.

So, the loss is square of the difference between the output of the network and the desired output. So, we choose the empirical risk of this that is sum this squared error over all the

examples and minimize that over the weights. So, we want to find the weights to minimize the empirical risk and the squared error loss function. And, to do the risk minimization we have chosen the gradient descent method of minimization essentially given the current weights, we find the next weights by from the current weights going into a little bit along the direction of the negative gradient. That is the standard gradient-descent method of optimization.

And for this class of networks, you derived a very interesting recursive formula for calculating the partial derivative. Essentially, when you are deriving gradient descent, new value for a weight w_{ij} is the old value of the weight minus step size into the gradient of the performance index with respect to that weight. Now, we showed that all these partial derivatives, the gradient components can be calculated interesting recursive manner and we derived this recursive formula last class. So, that is what we are going to start discussing today.

(Refer Slide Time: 04:33)



• We can summarize the method of learning weights as follows.

• We want to learn all the weights $\{w_{ij}^L\}$ to minimize

$$J(W) = \sum_{s=1}^N J_s(W) = \sum_{s=1}^N \frac{1}{2} \left(\sum_{j=1}^{n_L} (y_j^L(X^s, W) - d_j^s)^2 \right)$$

where $\{(X^s, d^s), s = 1, \dots, N\}$ is the training set.

• For minimizing J , we use the iterative gradient descent method.

NPTEL

PROF. N. S. RAO, IITM

PROF. N. S. RAO, IITM

So, let us first summarize the method of learning weights. So, what is that we want to do, we want to learn the weights w_{ij}^l . Recall that w_{ij}^l is the weight that connects node i in layer l to node j in layer $l + 1$. We are using layered feedforward networks where, interconnections always connect nodes in one layer to the nodes in next layer. So, nodes in layer l can only connect nodes in layer $l + 1$. So, that is what we have the notation where by w_{ij}^l is the weight that connects node i in layer l to node j in layer $l + 1$.

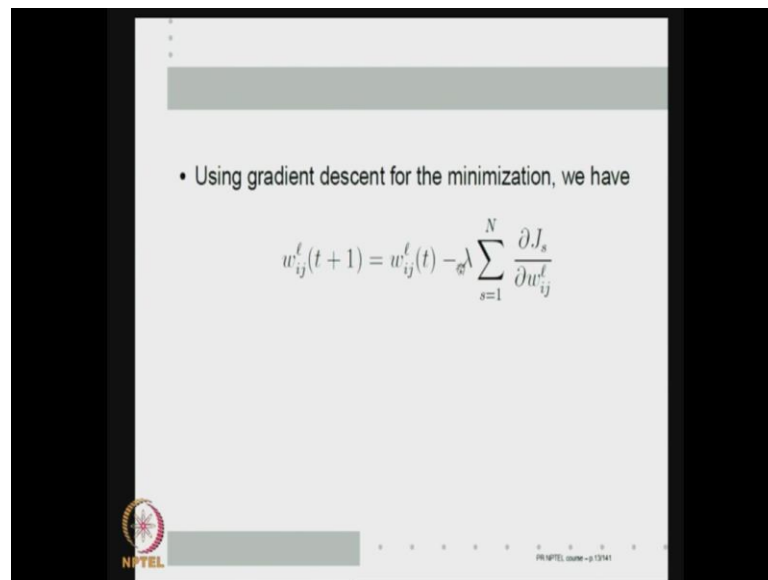
Now, this we want to learn; the weights for all such i, j 's and l that is the old set of weights, we represent by the matrix W . Now, we want to learn this W to minimize a performance index J . Where, J is defined as J of W is summation s equal to 1 to N $J_s W$ where, s is an index that goes over the training samples. The training samples have the form $X_s d_s$ for the input X_s the desired output is d_s . There are N training samples.

So, sum over each of the training samples. So, $J_s W$ is the performance measure error on a single training sample and that is given as capital L is the output layer. So, J is the node index in the output layer. So, y_{jL} is the j -th output of the network which of course, is the function of the input X_s and the weights W . So, this on the input X_s with the current weights W ; this is what my j -th output says. Because d_s is the desired output, d_{sj} is the k -th component of the desired output.

So, I square this and sum over J that gives me the error, the square of the error between the output of the network and the desired output. Because, the desired output itself could be a vector. As a matter of fact, we assume it to be a vector of as many components as there nodes in the output layer. So, y_{jL} is the j -th component of the network output; d_{sj} is the j -th component of the desired output.

So, this gives me the square of the error, sum over j that gives me the square of the error between the vector network output and the vector desired output for the input X_s . I sum it over all s 's, that gives me the so to say something proportional to the empirical risk under the squared error loss function over the training samples X_s and d_s . So, essentially we want to find W to minimize J to minimize the squared error and the way we are minimizing is we are using iterative gradient descent.

(Refer slide Time: 07:36)



What will be the iterative gradient descent? $w_{ij}^l(t+1)$, t is the iteration count now. $w_{ij}^l(t+1)$ is $w_{ij}^l(t)$ minus λ times derivative of J with respect to w_{ij}^l . Because, J is sum of J_s 's, the derivative will also be sum of these derivatives.

So, I get $w_{ij}^l(t+1)$ is $w_{ij}^l(t)$ minus λ is the step size for the gradient descent times, s is equal to 1 to N . The partial derivative of each of the J_s with respect to w_{ij}^l . So, look at it each of the J_s is actually the same structure, the only thing that different is, it is the square of the error between the network output of the desired output. But, in the s -th term, I put X_s as the input to the network and then, find the desired output and use the s -th desired output. So, when I find the network output, I put X_s as the input and then find this difference with the s -th desired output. Except for this, each of the J_s are same just that I put different X_s as the input and use the different desired outputs.

So, essentially each of these partial derivatives would have the same structure but, this will be my gradient descent minimization for each i, j and l ; $w_{ij}^l(t+1)$ is $w_{ij}^l(t)$ minus λ times derivative of capital J with respect to w_{ij}^l which is some s is equal to 1 to n partial derivative of J_s with respect to w_{ij}^l .

(Refer Slide Time: 09:15)

• Using gradient descent for the minimization, we have

$$w_{ij}^{\ell}(t+1) = w_{ij}^{\ell}(t) - \lambda \sum_{s=1}^N \frac{\partial J_s}{\partial w_{ij}^{\ell}}$$

• For each training sample (X^s, d^s) , we need to compute the corresponding partial derivatives of J_s .

• Then we can update all the weights.

• This is the batch mode operation.

• We can also have incremental or online learning.

NIPTEL

PH NIPTEL, course - g 17141

So, essentially for each training sample X^s, d^s we will compute the partial derivative with of J_s with respect to w_{ij}^{ℓ} . Once we do this, we can calculate the sum and hence, implement the gradient descent. Then, we can update all the weights. Now, this is what is we called a batch mode operation. In batch mode operation, you first put X^1 and you get the output of the network.

Then, you know the d^1 , using that you find $\partial J^1 / \partial w_{ij}^{\ell}$. Then, you put X^2 , using that you can find the outward network for in response to X^2 . Then, you find the error with respect to d^2 , the desired output for X^2 . Then, once again find this second partial derivative so on, you put all the partial derivatives and sum them up and then, use that sum to change the weight that is the batch mode operation.

In analogy with LMS, we also have a incremental mode operation whereby, at each iteration I will only use one of the training samples. Then, essentially this summation will not be there. Whichever is the training sample I am using in the current iteration, I find gradient only with respect to that training sample. There is not much difference between the 2 things. Because essentially, except which X^s I will put in the input and which d^s I use for finding the outputs, the J_s structures are same and hence, the partial derivatives computational also have similar computations.

(Refer Slide Time: 10:54)

• We have shown that for these networks we get

$$\frac{\partial J_s}{\partial w_{ij}^l} = \frac{\partial J}{\partial \eta_j^{l+1}} \frac{\partial \eta_j^{l+1}}{\partial w_{ij}^l} = \delta_j^{l+1} y_i^l$$

where δ_j^l is called error at that node and is defined by

$$\delta_j^l = \frac{\partial J_s}{\partial \eta_j^l}$$

• The δ_j^l depends on the training sample (X^s, d^s) though our notation does not explicitly show this.

NIPTEL course - p.33/41

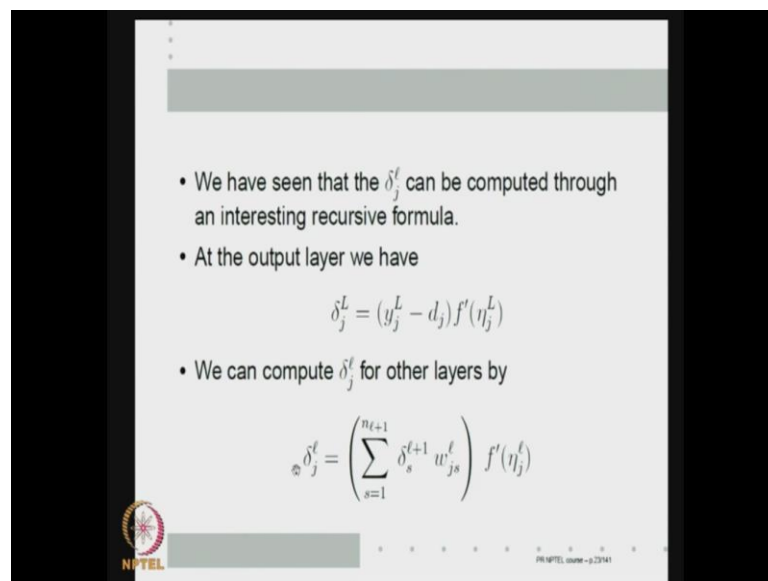
Now, we have shown last class that for these networks there is a very nice structure for the partial derivatives. The partial derivatives I want is $\partial J_s / \partial w_{ij}^l$. I did not put the s here because all the J 's have the same structure, so let us for now forget about the index, that tells me which particular training sample I am looking at. Then, I can roughly write it as ∂J by $\partial \eta_j^{l+1}$, $\partial \eta_j^{l+1}$ by ∂w_{ij}^l because the only way a weight w_{ij}^l will affect the J is by affecting the output of the network. The only way it can output the affect the output of the network is by its affect on the net input into the J -th node layer $l+1$. w_{ij}^l is a weight that is connecting into the J -th node in layer $l+1$.

So, it can affect the total input only into the J -th node in layer $l+1$. So, the only way to w_{ij}^l can have any effect on your final error is by its effects thorough the η_j^{l+1} . So, the partial chain, dual partial differentiation tells me that this derivative can also be written like this. Now, we said that we can write this as ∂J and this one we know because, η_j^{l+1} is a linear combination of all the inputs from layer l and if I take the differentiation with respect to one of the weights, I get that corresponding output from the layer l . And this first term which we called δ_j^{l+1} is called the error at that node and is defined by in general δ_j^l is the derivative of the error with respect to η_j^l , total input to node j layer l . Because, here I want node j layer $l+1$, this becomes δ_j^{l+1} .

This δ_j^l 's are called errors at a node. We will just remember that the δ_j^l actually depends on the particular training sample. Because here, just to keep noticing I dropped that s . But, the δ_j^l is derivative of the square of the error with respect to the net input into that node. But, this depends on which particular training sample I am talking about whether, X 's is, if I am talking about whether this training sample then, that X 's will be input to the network. And the final outputs error is calculated that d_j^L . So, δ_j^l is off course depend on the training sample X 's, d_j^L . But, our notation does not explicitly show, it only gives delta as only the node number as its subscript or super script.

So, even though we do not show it, we mentally remember that, that dependence is there. So, coming back, if I want implement the gradient decent, this is only the partial derivatives I want and this partial derivative is given by this. So, if I know the error at each node, I can implement the partial derivative. So, now the this of course I know, this is the output of the node. So, I just need to know how to calculate δ_j^l .

(Refer Slide Time: 14:20)



- We have seen that the δ_j^L can be computed through an interesting recursive formula.
- At the output layer we have

$$\delta_j^L = (y_j^L - d_j^L) f'(\eta_j^L)$$

- We can compute δ_j^l for other layers by

$$\delta_j^l = \left(\sum_{s=1}^{n_{l+1}} \delta_s^{l+1} w_{js}^{l+1} \right) f'(\eta_j^l)$$

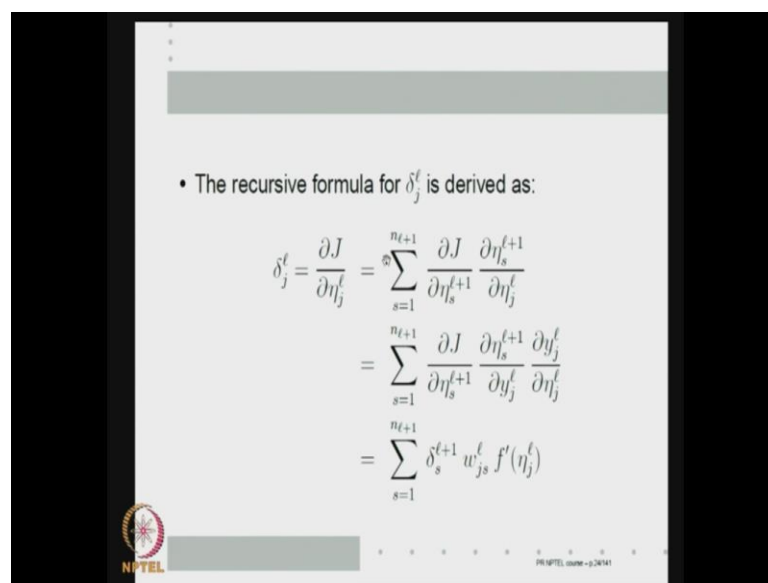
NPTEL logo and course information are visible at the bottom of the slide.

We have seen that δ_j^l can be computed through an interesting recursive formula. Let me first put down the recursive formula. We showed that for the final layer δ_j^L is simply y_j^L minus d_j^L into f' prime η_j^L . y_j^L is the j -th the output of the j -th node in the final layer, d_j^L is the corresponding desired output. This is the actual error into f' prime η_j^L .

l, this happens to be δ_{j-1} by explicitly calculating that partial derivative. And for all other layers the δ_{j-1} satisfies very interesting recursive form like this.

δ_{j-1} can be given in terms of δ_{s-1} . If I know δ_{s-1} for all s then, I can calculate δ_{j-1} for every j. So, if I know the errors for nodes, all the nodes in the layer l + 1 then, I can calculate errors for all the nodes on layer l. So, because I know at the output layer capital L, I can calculate capital L minus 1. Then, I can calculate l minus 2 and so on. So, using this recursing we can calculate the errors at all nodes.

(Refer Slide Time: 15:36)



• The recursive formula for δ_j^l is derived as:

$$\delta_j^l = \frac{\partial J}{\partial \eta_j^l} = \sum_{s=1}^{n_{l+1}} \frac{\partial J}{\partial \eta_s^{l+1}} \frac{\partial \eta_s^{l+1}}{\partial \eta_j^l}$$

$$= \sum_{s=1}^{n_{l+1}} \frac{\partial J}{\partial \eta_s^{l+1}} \frac{\partial \eta_s^{l+1}}{\partial y_j^l} \frac{\partial y_j^l}{\partial \eta_j^l}$$

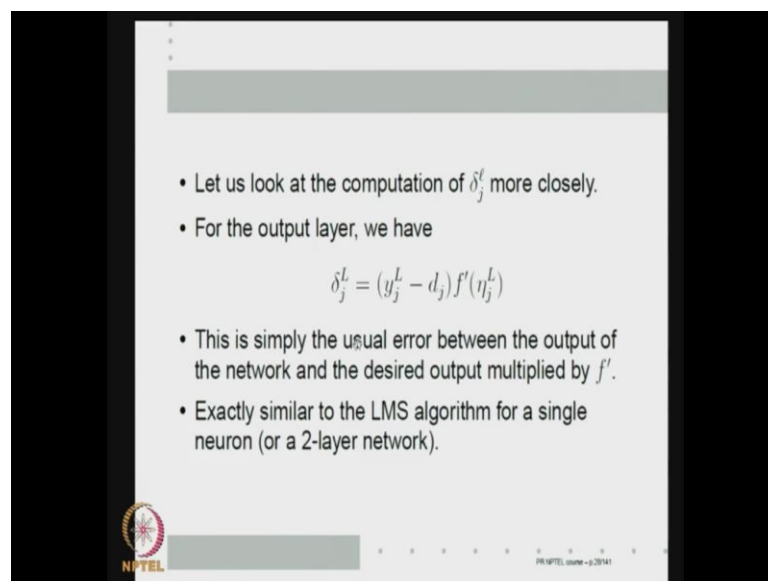
$$= \sum_{s=1}^{n_{l+1}} \delta_s^{l+1} w_{js}^l f'(\eta_j^l)$$

So, this how the recursive formula is derived. This particular recursive formula. Recalls that, δ_{j-1} by definition is $\partial J / \partial \eta_j^l$. So, how does net input into node j layer l effects the final output. Node j layer l, its net input is convert into output and its output is send to all the nodes in layer l + 1. So, some or all nodes in layer l + 1, $\partial J / \partial \eta_s^{l+1}$ by $\partial \eta_s^{l+1} / \partial \eta_j^l$. And this can further be written in terms of the outputs. So, $\partial \eta_s^{l+1} / \partial \eta_j^l$ by $\partial \eta_s^{l+1} / \partial y_j^l$ and $\partial y_j^l / \partial \eta_j^l$. The next thing is, this already has a name δ_{s-1} and because the net input into any node layer l + 1 is a linear combination of the outputs of layer l, I get that weight here and an output of a node 2 is that input is conduct by the activation function. So, this partial derivative gives me the derivative of the activation function. So, this is how we actually derived the previous recursive formula that is our recursive formula. So, now we are almost done. So, we know that this is what we want to implement.

This is my gradient descent that needs me to calculate these partial derivatives. And I know how to calculate this partial derivative; they are in terms of error at that node into the output. So, if I am asking how to update weight that connects node i layer l to node j layer $l + 1$. Then, the output of node i layer l multiplied by the error at node j layer $l + 1$. So, this w_{ij}^l connects node i layer l to node j layer $l + 1$. So, I take at the input end of that arrow, the output at the output end of the arrow the error. So, I multiply the error at node j layer $l + 1$ with input at node i layer l and that product gives me the required partial derivatives. Once the partial derivatives are there of course, I can implement the gradient descent that enhances my learning algorithm.

And the partial derivative is themselves are obtained through an interesting recursive formula. This is the formula for the output layer and for all other layers this is the recursive formula.

(Refer Slide Time: 18:22)



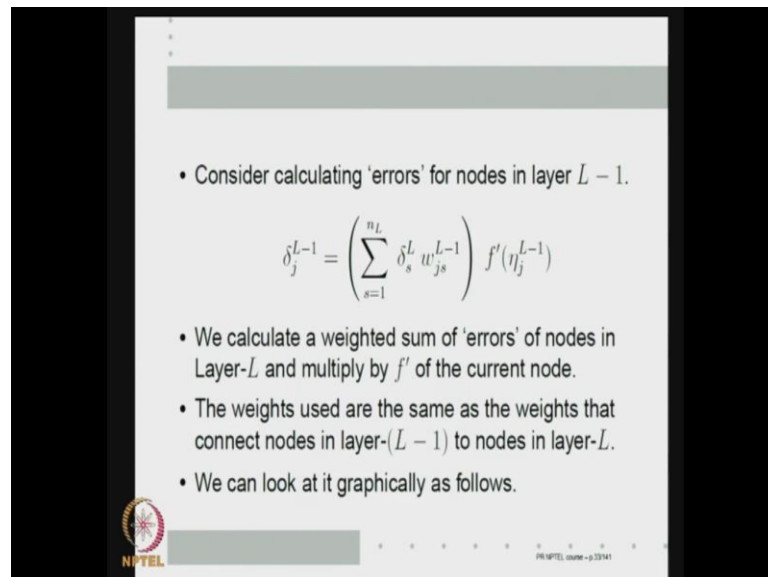
- Let us look at the computation of δ_j^L more closely.
- For the output layer, we have

$$\delta_j^L = (y_j^L - d_j^L) f'(\eta_j^L)$$
- This is simply the usual error between the output of the network and the desired output multiplied by f' .
- Exactly similar to the LMS algorithm for a single neuron (or a 2-layer network).

So, now what we are going to do is to look at what this particular recursive formula means. So, coming back once again, this is the output, this is the error for all nodes in the output layer. This is exactly like the usual error for a LMS algorithm. Where, y_j^L is the actual output of the network, the j -th component; d_j^L is the desired output, j -th component. So, take that, which is the actual error. Multiply by f' because f is the function that connects the net input into that node to its output. So, that is the. For example, if I have only one node in the output and still I use the sigma recursive function

but, I have no hidden notes. Then, this is this is what would be the LMS algorithm. Now, this is very similar to LMS algorithm for a single neuron or a 2 layer network. So, this is understandable. Now, let us consider because we have a recursive formula. Once we know all the error set layer L, we can calculate errors of nodes in layer L minus 1.

(Refer Slide Time: 19:33)



- Consider calculating 'errors' for nodes in layer $L - 1$.

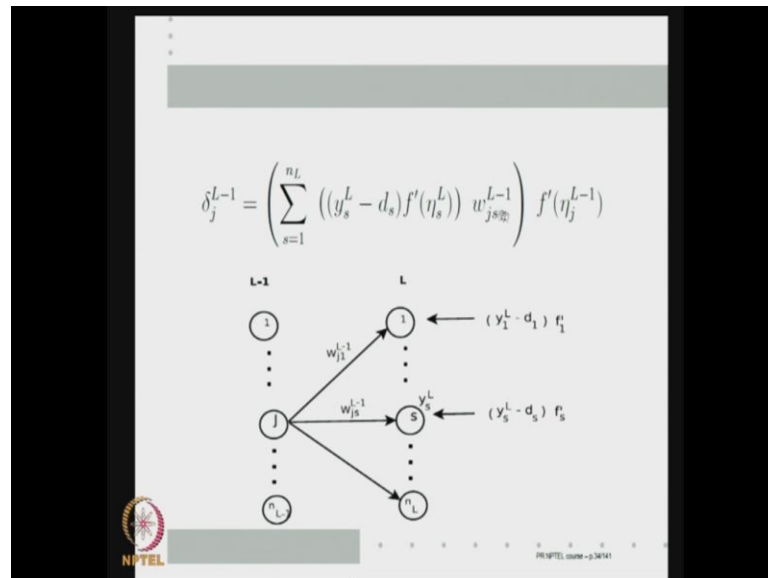
$$\delta_j^{L-1} = \left(\sum_{s=1}^{n_L} \delta_s^L w_{js}^{L-1} \right) f'(\eta_j^{L-1})$$

- We calculate a weighted sum of 'errors' of nodes in Layer- L and multiply by f' of the current node.
- The weights used are the same as the weights that connect nodes in layer- $(L - 1)$ to nodes in layer- L .
- We can look at it graphically as follows.

So, let us look at that recursive formula. If I want error at node j layer L minus 1 then, it is some linear combination of errors of nodes in layer L. Ultimately of course, multiplied by f prime of this particular node so, that does not come re-submission, that will keep separately.

So, if I want error at node j in layer L minus 1, we calculate it as a weighted some of errors of nodes in layer L and multiply by f prime. That is how we get errors for nodes in layer L minus 1. And in the linear combination or a weighted sum, the weights are same as the weights that connect the corresponding nodes in layer in L minus 1 to the nodes in layer L. So, essentially I am taking the errors at nodes in layer L multiplying them by weights. The weights are the same ways between layer L minus 1 and layer L and take a summation.

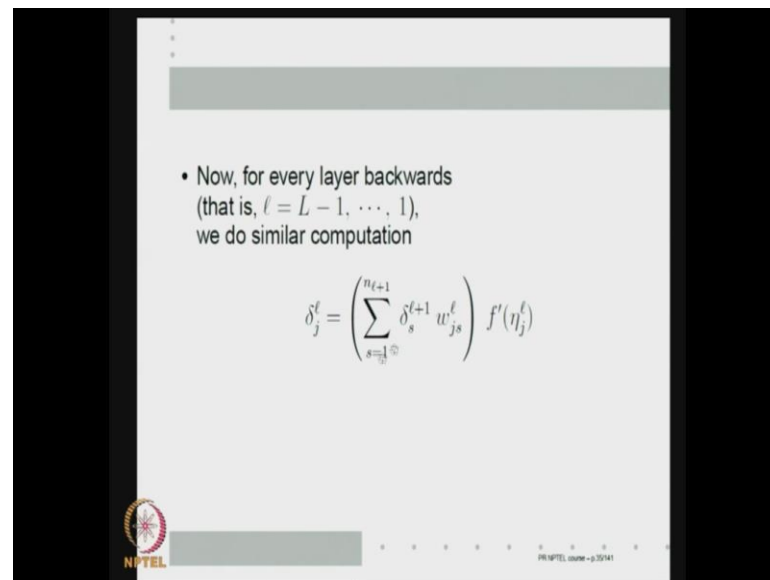
(Refer Slide Time: 20:48)



Let us look at it from the network. So, this is the layer L. Let us say I want to calculate, this layer L minus 1. Let us say I want to calculate error at node J layer L minus 1, that is delta j L minus 1, how do I calculate? For that, I have to do delta j L into w j s L minus 1. Delta j L, I expanded here. We know delta j L is given by this. So, I just expanded that. So, this entire thing is the errors for nodes in the layer L multiply by w j s L minus 1. What is j s L minus 1? That connects node j and node layer L minus 1 to node s in layer L. So, we can think of it as if this is my network. Normally, when I am doing my output calculation, I will take the output of node J multiplied by this weight and goes to this node or multiply this weight and it goes to this node and so on. Instead of that, now we can think of this as, I think that this is the error at this node, not think this is the error at this node y 1 L minus d 1 into f 1 prime, at this node is y s L minus d s into f x prime.

We think of that as the current values of these nodes, multiply the current values of these nodes with the corresponding weights and sum it up over all these nodes, that is what I am doing. I am taking delta s L, multiply that is the error at this node delta s capital L. And multiplied it by the weight w j L minus 1. And submit to all our node to the layer L that is, this term; the term in this big, big parenthesis. And then, multiply by f prime of net input of this node that gives me the error at this node. Now, this particular style of competition is same at all level.

(Refer Slide Time: 22:50)



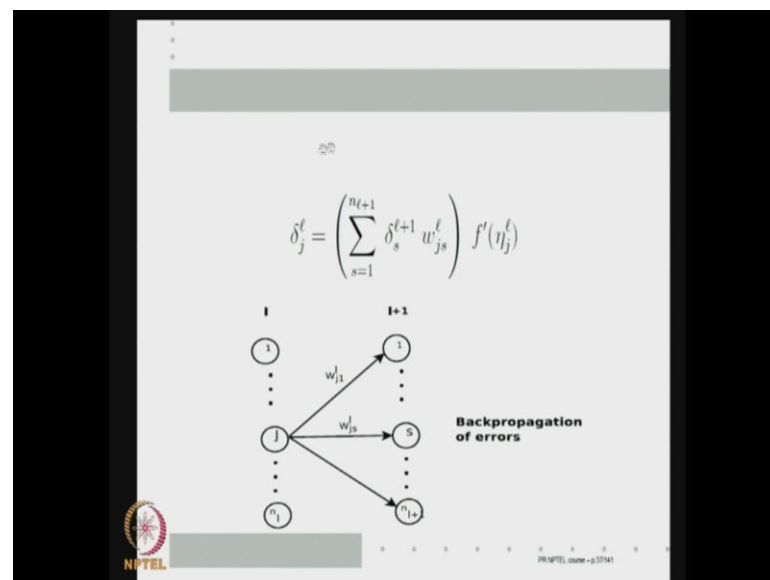
• Now, for every layer backwards
(that is, $\ell = L - 1, \dots, 1$),
we do similar computation

$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} w_{js}^\ell \right) f'(\eta_j^\ell)$$

NPTEL

What do I do for remaining layers? Same thing; delta j l is written as a linear combination of errors of a nodes in layers l plus 1.

(Refer Slide Time: 23:01)



$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} w_{js}^\ell \right) f'(\eta_j^\ell)$$

Backpropagation of errors

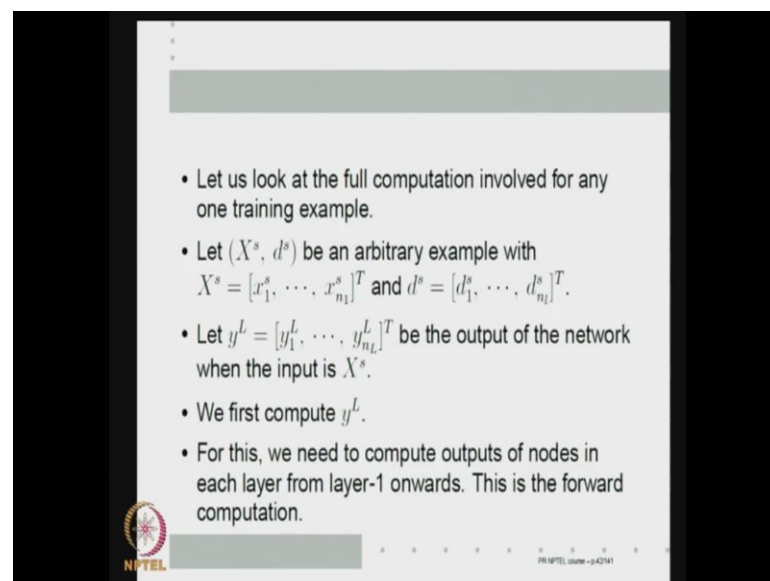
NPTEL

Once again, if little l plus 1 is this layer. So, I take error at this node that is delta 1 l plus 1 multiplied with w j 1 l with this weight. And I take delta s l plus 1 multiplied delta j s l and so on. Add all of them up that gives me this term and multiply by f prime of this node. So, essentially if I calculate weights at each of these points then, I multiply these weights, the errors at the nodes with the weights. Earlier what I am doing, when I am

doing forward competition. I calculate output here, output here, output here and take the output here multiply by this weight, take the output here multiply by this weight, take the output here multiply this weight. That gives me in the net input to this node. Instead of that but, actually getting errors at each of the nodes, I have to take the errors at this node multiply by this weight, error at this node multiply by this weight, error at this node multiply by this weight, add them up.

So, it is structurally the same computation as a layer. So, we can think of the forward computation, computation output as forward propagation of inputs into outputs. I can think of these as the backward propagation of errors. So, that is why this particular way of calculating all the partial derivatives using errors at nodes is called the back propagation algorithm. What it does is, it does back propagation of the errors. Once I know the errors at the output layer I back propagate, so that I get errors layer $L - 1$ then, I back propagate I get layer set L . This is what back propagation is all about.

(Refer Slide Time: 24:47)



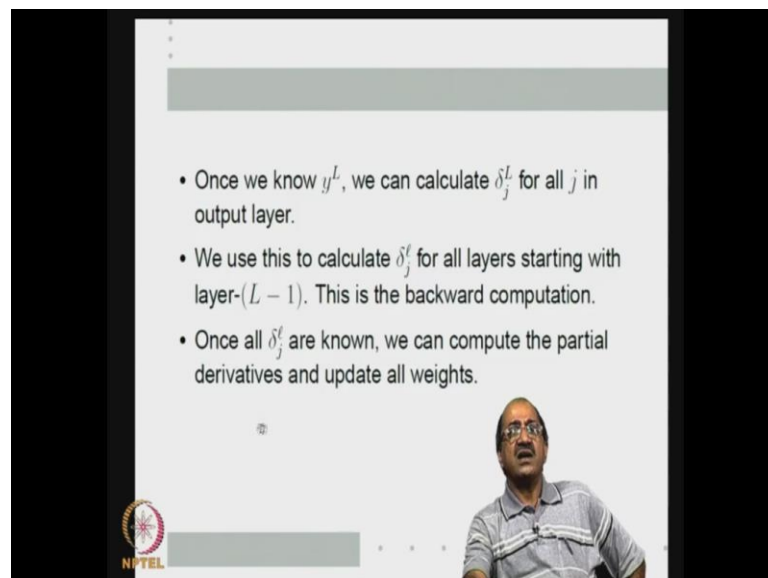
- Let us look at the full computation involved for any one training example.
- Let (X^s, d^s) be an arbitrary example with $X^s = [x_1^s, \dots, x_{n_1}^s]^T$ and $d^s = [d_1^s, \dots, d_{n_l}^s]^T$.
- Let $y^L = [y_1^L, \dots, y_{n_L}^L]^T$ be the output of the network when the input is X^s .
- We first compute y^L .
- For this, we need to compute outputs of nodes in each layer from layer-1 onwards. This is the forward computation.

So, now let us fully understand what is involved in training the network. So, we have to do the same process for each training examples. So, let us look at it any one example. So, let say X^s, d^s be an arbitrary example. X^s is as got n_1 component because, n_1 is the number of nodes in first layer at the input layer according to our notation. And d will have n_l components, n capital L components. Once again, I have notation that is the

desired output. So, essentially learning a function from $n+1$ dimensional space to $n+L$ dimensional space.

So, what do you do? We first have to calculate the output of the network y^L , which will have $n+L$ components. y^L , we have to calculate when the inputs are X s. So, I will put X s as the inputs and successively calculate the outputs. So, this is what you have to first do and we can think of this as the forward computation. So, what do we have to do for forward computation? We have to first put the current input axis as the input layer and then, successively from layer one onwards, keep calculating outputs of the next layer right and then, at the end will have the output of the network. Once you calculate the output of the network.

(Refer Slide Time: 26:09)



The slide contains the following text:

- Once we know y^L , we can calculate δ_j^L for all j in output layer.
- We use this to calculate δ_j^l for all layers starting with layer $(L-1)$. This is the backward computation.
- Once all δ_j^l are known, we can compute the partial derivatives and update all weights.

The NPTEL logo is visible in the bottom left corner of the slide.

Then, we know the desired output and hence, we have to calculate δ_j^L for each node in the output layer. Once you calculate that, we can calculate for layer $L-1$ by back propagation. This is the backward computation. So, starting with layer $L-1$, I can keep back propagating errors and keep calculating errors at each of the nodes backwards. I will first calculate errors at node L then, I calculate errors at node $L-1$ then, I calculate at $L-2$ and so on. So, this is the backward computation. At end of the backward computation, all the errors δ_j^l are known. Now, we can compute all the partial derivatives and update the weights.


(Refer Slide Time: 26:59)

Computing output of network

- For the input layer: $y_i^1 = x_i^s, i = 1, \dots, n_1$.
- For $\ell = 2, \dots, L$, we now compute

$$\eta_j^\ell = \sum_{i=1}^{n_{\ell-1}} w_{ij}^{\ell-1} y_i^{\ell-1}$$
$$y_j^\ell = f(\eta_j^\ell)$$

- Once we have the output of the network, we then need to compute the 'errors' δ_j^ℓ .

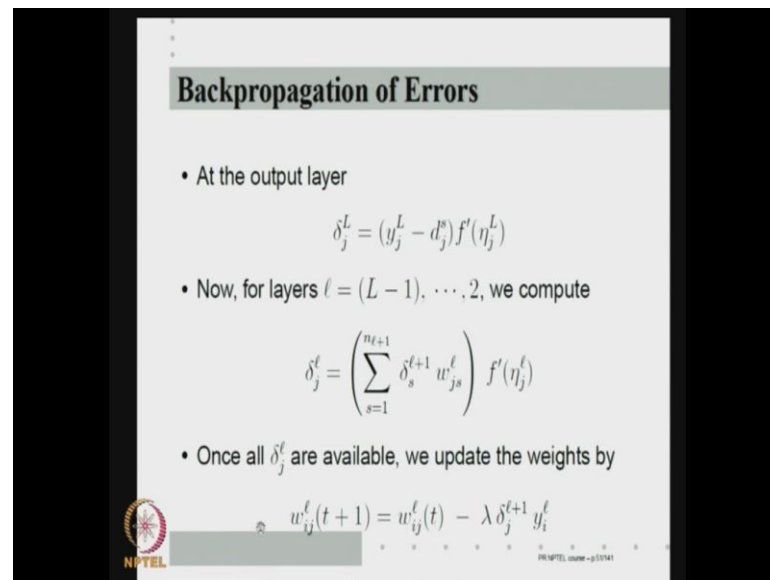
 NPTEL

For NPTEL, course - p-48741

Now, let us go into the details for the forward computation, computing output a network, how do I do? Layer by layer. For the input layer y_i^1 is x_i^s . y_i^1 is in the one super script is in the input layer the i -th node. So, the i -th node simply takes the i -th input x_i^s . For all other layers, layer 2 onwards. I first compute the net input, η_j^ℓ as the weighted sum of the output from the previous layer, $y_i^{\ell-1}$ minus one $w_{ij}^{\ell-1}$ minus summed over i . And then, I pass it through an activation function.

This is the forward computation. See, essentially for the forward computation for every weight in the network, different weights in a different layer but, for every weight in a network have to do one multiplication. And for every node in a network, I have to do one function computation. So, this is my forward computation. Once we have we do it for all the up to layer L , we have the output of the network and now we can start calculating the errors.

(Refer Slide Time: 28:04)



Backpropagation of Errors

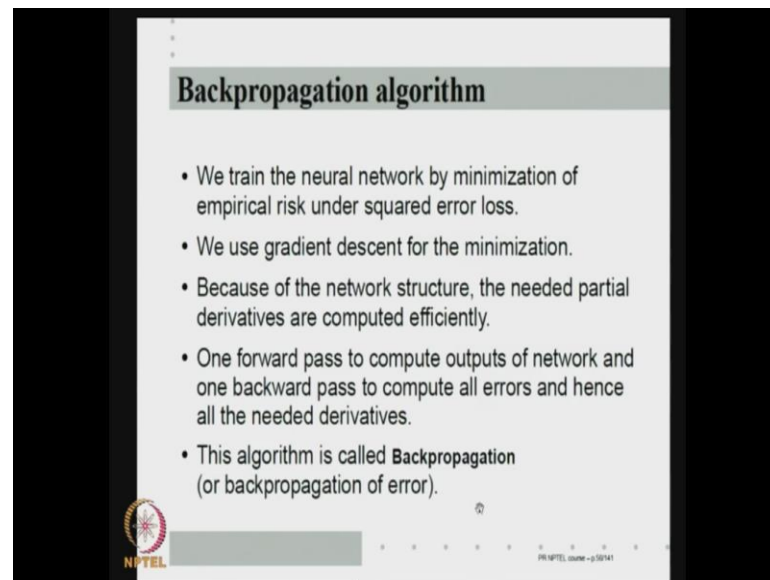
- At the output layer
$$\delta_j^L = (y_j^L - d_j^L) f'(\eta_j^L)$$
- Now, for layers $\ell = (L - 1), \dots, 2$, we compute
$$\delta_j^\ell = \left(\sum_{s=1}^{n_{\ell+1}} \delta_s^{\ell+1} w_{js}^\ell \right) f'(\eta_j^\ell)$$
- Once all δ_j^ℓ are available, we update the weights by
$$w_{ij}^\ell(t+1) = w_{ij}^\ell(t) - \lambda \delta_j^{\ell+1} y_i^\ell$$

NIPTEL course - p.53/61

At the output layer, once I have y_j^L and this is from the training input desired input, I can calculate the errors for all nodes in the outputs layer. Once, I have that. I start going back from layer $L - 1$, $L - 2$, all the way up to layer 2 to compute delta as L . Once I compute all the delta as a L 's, I can update the weights. Of course, I have written it as an incremental version here, of course otherwise, have to do it put a summation here. This is the overall view. So, we have a forward computation.

That is my forward computation. I first get the output of layer 1 nodes simply the inputs and then, from layer 2 all the way up to L I calculate net input and pass through activation successively. So, first I do it for layer 2 then, I get all the outputs from layer 2. Now, I can calculate net inputs into layer 3 and so on. Ultimately, I will get outputs of layer L , which is the network output. And then, I have I will calculate errors, first errors of the output layer. And then, using my recursive back propagation formula, I calculate errors at all the other nodes going back in the layers and once all the delta available, I can update the weights. So, this is the back propagation algorithm for training the network.

(Refer Slide Time: 29:30)



Backpropagation algorithm

- We train the neural network by minimization of empirical risk under squared error loss.
- We use gradient descent for the minimization.
- Because of the network structure, the needed partial derivatives are computed efficiently.
- One forward pass to compute outputs of network and one backward pass to compute all errors and hence all the needed derivatives.
- This algorithm is called **Backpropagation** (or backpropagation of error).

NIPTEL

PH NIPTEL, course - p 50741

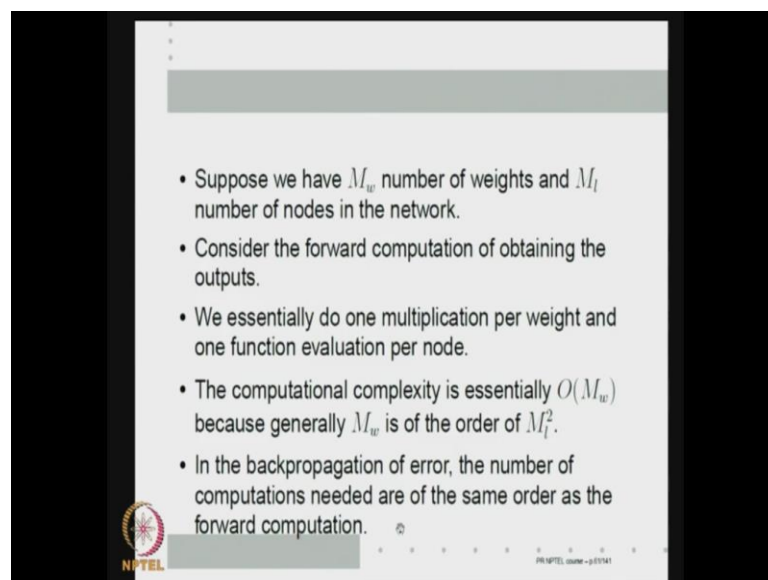
So, let us sum it up. We train the neural network by minimization of empirical risk under this squared error loss function we are given some. We are essentially looking at training the neural network to represent a function, we have given X as $d \times s$. X belongs to some unknown dimensional space and $d \times s$ belongs to some $n \times L$ dimensional space. I am learning a function from $\mathbb{R}^{n \times 1}$ to $\mathbb{R}^{n \times L}$.

So, we train the network by minimizing the empirical risk under squared error loss. For minimizing the empirical risk which is essentially some of squared errors with much like your least squares method. We use the gradient descent for the minimization. Because of the network structure, the needed partial derivatives are computed very efficiently. What is the process involved? One forward pass to compute outputs of the network and one backward pass to compute all the errors and hence all the derivatives.

So, first there is a forward way of pushing inputs in the network to the outputs as network, which involves of course weighted sums and function computations and once I have the output network, by comparing desired output I can calculate errors at the output node. Now, I do back propagation which once again simply involves weighted sums and functional computations. So, one forward pass to compute the output network and one backward pass to compute all the errors and hence, all the linear derivatives. And then, I can update the weights and keep going on.

This iterative algorithm for learning weights in your network is called the back propagation algorithm, is short for back propagation of errors. As a matter of fact, back propagation algorithm has been historically so successful in neural networks. For a large majority of neural network model users, neural used feed for neural network models is synonymous with the back propagation. This course as I said, we using back propagation for training feedforward neural networks, is a very good algorithm for feedforward neural networks. But, will see that there are other methods of training neural networks. But, in spite of that, for neural networks where we have this kind of sigma activation function, back propagation is the most famous algorithm for training the network.

(Refer Slide Time: 31:45)



Now, let us look at a few issues about back propagation. Suppose, we have M_w and M_l number of weights, total number of weights in network between all layer counting weights between all layers, let us say M_w number of weights in network and there are M_l number of nodes in the network. Now, the forward computation, what we have to do? I have to do weighted sum. In the weighted sum each weight is used only once exactly.

So, I will do in calculating the net inputs of all nodes in the network put to be the through all the layers, I would do as many multiplications as there are weights. So, M_w number of multiplications I have to do. We can assume that each function computation is some constant time say, multiplication cost wise, I do as many function computations as

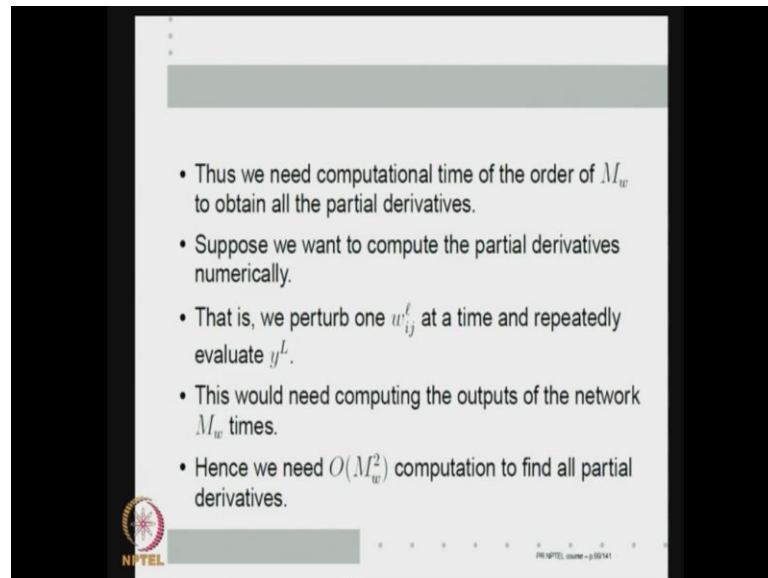
there are nodes M_1 . So, totally my computation is M_w multiplication plus some constant times M_1 multiplications or M_1 function $(())$. So, we do one multiplication per weight, one function evaluation per node. So, the total computational complexity is should have been M_w plus M_1 .

M_w multiplication plus k times M_1 so, is an order notation. So, k does not matter. But, often M_w is the order of k_1 square because a weight connects every pair of nodes. So, if I got 10 input nodes, 10 output, 10 hidden layer nodes. There 20 nodes. But, I will have 400, I have 100 weights. Because, each of the 10 input nodes are to be connected to each of the 10 hidden nodes.

So, normally the number of weights is of the order of square of the number of nodes. So, M_w plus M_1 is the same order as M_w because, M_w goes as M_1 square. So, the forward computations complexity essentially order M_w , I will do of the same order of multiplications as the number of weights in the network and very reasonable because you compute outputs, each weight participates in a weighted sum computation once. So, total number of multiplications is of the order of the number of weights. Of course, of the number of weights plus number of nodes but, we left out number of nodes because, number of weights is an order higher than number of nodes.

What about the back propagation? Back propagation, the nature of computation is same. Essentially, you think of the errors at the output nodes like the inputs at the input layer and then, do the same kind of weighted sum calculation. So, for the back propagation error, the number of computation needed is of the same as the order as the forward computation. In the forward computation, I am calculating f ; backward computation calculating f' . So, is not particularly different. So, in the back propagation of error, the number of computations needed out of the same order as the forward computation. Now, one back propagation one pass of back propagation gives me all the partial derivatives respect to all the weights which means, we need computation time of the order of M_w to obtain all the partial derivatives.

(Refer Slide Time: 34:50)



- Thus we need computational time of the order of M_w to obtain all the partial derivatives.
- Suppose we want to compute the partial derivatives numerically.
- That is, we perturb one w_{ij}^L at a time and repeatedly evaluate y^L .
- This would need computing the outputs of the network M_w times.
- Hence we need $O(M_w^2)$ computation to find all partial derivatives.

We know forward pass is of the order of M_w , backward pass and forward pass are about the same computation. And one backward pass gives me all the partial derivatives. So, to get all the partial derivatives with respect to all the weights, all the M_w weights, I need computation only of the order of M_w . This is a remarkable efficiency. I am calculating M_w partial derivatives. If I am calculating M_w partial derivatives, I can do anything less than order M_w computation.

Obviously, I am calculating M_w numbers so that my order of computation should be of the order of M_w that gives only linear as M_w which is really nice. To appreciate this, let us say we want to compute the partial derivatives numerically of course, you have to do the partial derivatives computation with us for a particular input. So, I you know if I do analytical expressions then, I will get a huge expression, as big as the entire input output mapping of the network for each (()).

So, that may not be as nice. But, suppose you want to compute partial derivatives numerically, what does that mean? For each weight, I have to perturb the weight and calculate the output. I need the derivative, the output with respect to the weight. So, I put the current value of the weight calculate the output that I have already done. Now, perturb one weight only one weight keeping everything else same as earlier and now, find the output. And then, change in output by change in weight will give me the partial derivatives with respect to weight. Now, if I do this, what it means is; because each weight has to be

perturbed once, I have to calculate the output of the network as many times as there are weights.

Which means, I need to calculate the outputs networks $M \times W$ times, each computation of the output will be of the order of $M \times W$ multiplications. To find all the partial derivatives, I need $M \times W$ square computations $r \times W$ square. Because, if I want to do numerically like this, for each weight I have to calculate one output. Calculations of output take $r \times M \times W$ computations.

So, there are $M \times W$ weights. So, I take $M \times W$ square computations. Even, if I am using my equations will be the same because, for each weight the amount of computation is equation would be of the same order of as calculating the output of the entire network starting from the input. So, if I am doing blindly or in a simple minded manner finding all the partial derivatives either by, brute force crunching of expressions or through numerical estimation, I would do order $M \times W$ square computation. Whereas, that perturbation allows me to do it in order $M \times W$.

(Refer Slide Time: 37:55)

• In deriving the backpropagation algorithm we used

$$\frac{\partial J}{\partial w_{ij}^l} = \frac{\partial J}{\partial \eta_j^{l+1}} \frac{\partial \eta_j^{l+1}}{\partial w_{ij}^l}$$

• We can use this also in obtaining partial derivatives numerically.

• That is, we perturb one η_j^l at a time.

• Sometimes called *node perturbation method*.

NPTEL

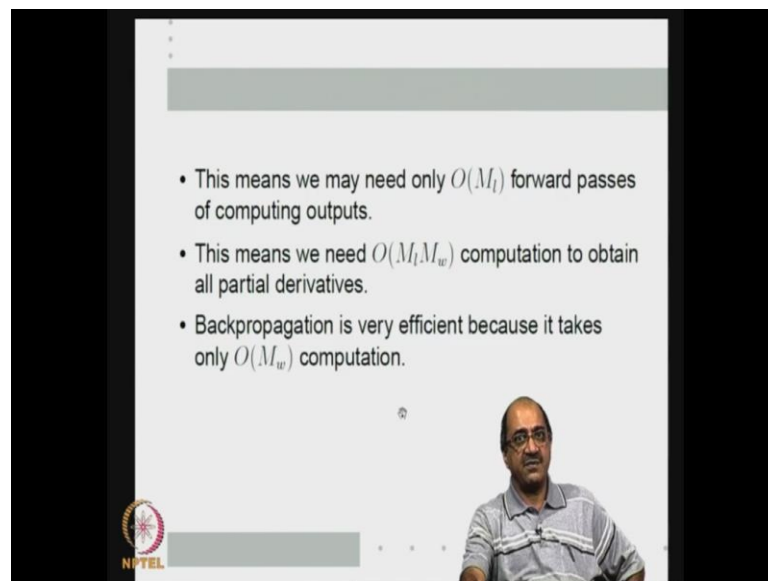
NPTEL course - 3-73141

Of course, one the reason is in back propagation we use a lot of interesting properties of the partial derivatives. For example, we show that the partial derivative this, can be written after the partial derivatives. That is how the error at an over came. Multiplied by the output of the previous node and because this is known, I only need to calculated these partial derivatives, to know these partial derivatives.

See, these are weights partial derivative. So, there are as many partial derivatives as weights here, there is any many partial derivatives as nodes. So, this will certainly save me even if I am doing numerical. We can use the same idea in partial derivatives numerically also. What it means? Instead of finding this partial derivative numerically, I will only find this partial derivative numerically. What does that mean? Instead of perturbing one w_{ij} at a time, we perturb one n_j at a time.

So, how many n_j 's are there? As many as there are nodes. There are M_l nodes. So, for each node I have to perturb once for which means, each node I have to do one forward pass of computing the output of the network. So, how many times you have to compute the output now? Earlier, I am computing M_w times the output of the network, I need not how to do it for M_w times, I need to do only M_l times. So, my order of computation by perturbing a node at a time is sometime called node perturbation method of numerically evaluated in the partial derivatives.

(Refer Slide Time: 39:26)



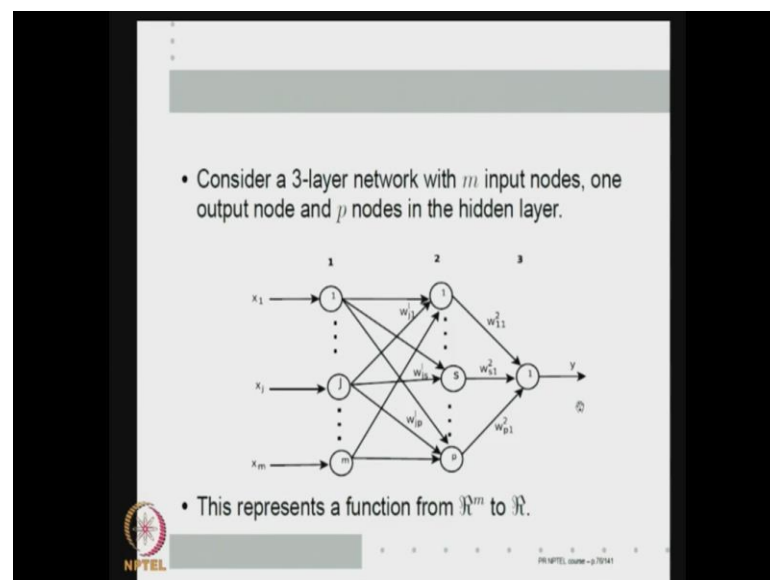
So, which means we need only to calculate order M_l forward passes. Because, there are only M_l nodes for each node, I want to find the output once. Because, I have found output with n_j already, I perturbed only that particular that input keeping all others constant and find the output so that I can get partial derivative with respect to that input.

So, ultimately I need to calculate the output of the network M_l times. So, I need order M_l forward passes, each forward pass has a order M_w computations. So, I need order M_l ,

Mw computations. Given earlier that, Mw is M^2 kind of thing. M^2 , Mw would be like $Mw \log Mw$. M^2 will be of the order of $\log Mw$. So, we need $Mw \log Mw$ computations, if we do no perturbation.

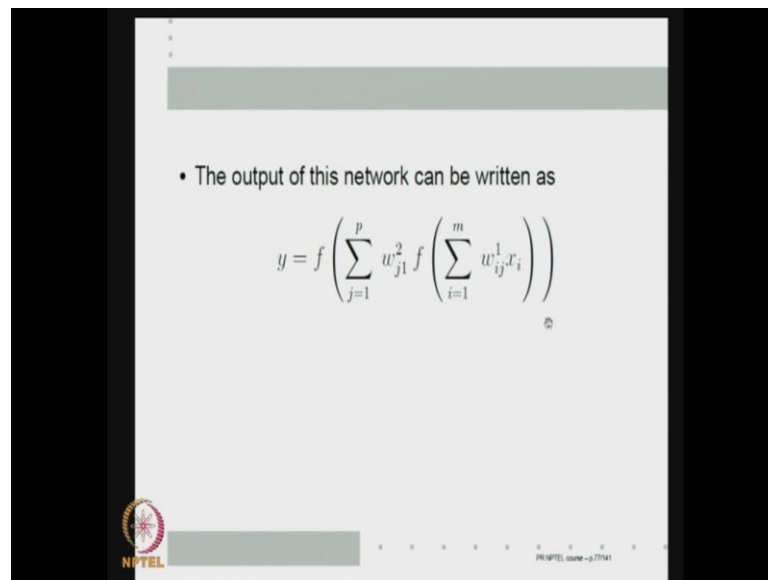
But back propagation is very efficient because it takes only order Mw computation to calculate all the Mw partial derivative. I cannot do any better than this. So, this is one of the reasons why back propagation is a very popular algorithm for doing gradient descent, gradient descent minimization of the squared error empirical risk on for learning weights in a new electron.

(Refer Slide Time: 40:43)



So, let us go little further to ask how these networks are. Now, let us consider 3 layer networks again. Let us say, there are m input nodes. Let us say, p nodes in the hidden layer and one node in the output layer. As we seen, layer 1 is always input layer; layer 3 is the last layer is always output layer. Everything in between are hidden layers in node because, I have a 3 layer network, there is only 1 hidden layer. So, suppose there are p nodes in the hidden layer, m nodes in the input layer and 1 node in the output layer. So, such a network will represent a function from \mathbb{R}^m to \mathbb{R} . So, let us write an explicit expression for this function. What will be the function? This output is, this weight into this output plus this weight into this output, this weight into this output, all added have and passed it through the n function of this.

(Refer Slide Time: 41:42)



• The output of this network can be written as

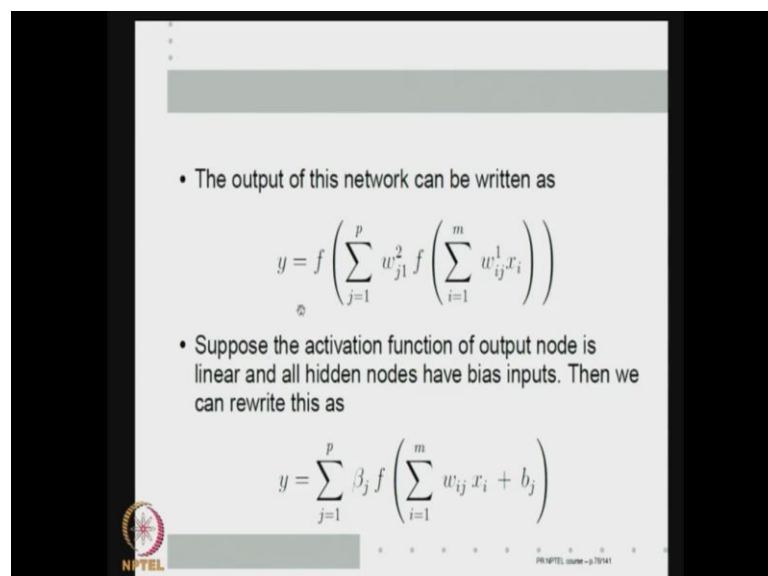
$$y = f \left(\sum_{j=1}^p w_{j1}^2 f \left(\sum_{i=1}^m w_{ij}^1 x_i \right) \right)$$

NIPTEL logo and navigation icons are visible at the bottom of the slide.

So, I can write that as, f of summation j is equal to 1 to p as many as hidden nodes. w_{j1}^2 this is the weight that connect nodes j in layer 1 to node j in layer 2 to node 1 in layer 3 into f into output of node j in layer 2.

What is output of node j in layer 2? If I want any node output, I have to find the weighted sum of all the inputs. inputs are The outputs of these layers is access. So, I make the corresponding input multiplied by the weight and pass it through activation function that gives me the output here.

(Refer Slide Time: 42:29)



• The output of this network can be written as

$$y = f \left(\sum_{j=1}^p w_{j1}^2 f \left(\sum_{i=1}^m w_{ij}^1 x_i \right) \right)$$

• Suppose the activation function of output node is linear and all hidden nodes have bias inputs. Then we can rewrite this as

$$y = \sum_{j=1}^p \beta_j f \left(\sum_{i=1}^m w_{ij} x_i + b_j \right)$$

NIPTEL logo and navigation icons are visible at the bottom of the slide.

So, it will be that. Well, suppose the network uses the activation function, this is the activation function with the output node; let us say we use linear activation functions. So, the output node is linear. So, this weight only depends on the index in the hidden layer. So, that 1 and 2 are constant, let me write it as, B_{ij} ; j goes from 1 to p . Then, I do not need this 1 here, w 's are only here. So, I can write $w_j x_i$ plus here, I did not put the bias, I told you earlier each node can have bias.

So, I will put a bias. So, if I have a one hidden layer network with p nodes in the hidden layer. Mind you; whenever, I am representing a function the number of nodes in the input and output layers are fixed based on the dimension of the domain and range of the function. So, if I am running a function from \mathbb{R}^m to \mathbb{R} , there should be m input nodes, one output node that is fixed.

So, the only thing I do not know is, if I am using one hidden layer, I do not know how many hidden nodes I can have. Let us say they are p then, this is a general expression for a function represented by a neural network with one hidden lower layer with p hidden nodes. B_{ij} represents the weight between the hidden node j and the output node; there is only one output node. w_{ij} represents the weight between the i -th input node and j -th hidden node; x_i 's are the input and b_j is the biases for the hidden nodes. I am not putting the bias for the output node. So, this is the form of the function represented by this 3 layer network.

(Refer Slide Time: 44:01)

Representational abilities

- Now we can ask what kind of functions can be represented like this?
- We now state a theorem that says that if p is 'sufficient', then this 3-layer network can well approximate any continuous function over a compact set in \mathbb{R}^m .
- **Theorem** Let $\phi : \mathbb{R} \rightarrow \mathbb{R}$ be a bounded, strictly monotonically increasing continuous function. Let $C(I_m)$ be set of all continuous real valued functions from $I_m = [0, 1]^m$ to \mathbb{R} .

NITEL

PRUAPTEL course - p 00141

Now, we can ask the question, what kind of functions can be represented by like this? I am asking;

(Refer Slide Time: 42:29)

If I consider function from \mathbb{R}^m to \mathbb{R} , some functions can be written like this, some functions may not be written like this, depending on what type I am choosing. So, I am asking, for what all functions will there be some β_j 's, w_i 's and b_j 's such that, they can be written like this. That will tell me what kind of functions my network can represent. So, the question is, what kind of functions can be represented like this. Now, I state a theorem says that if p is sufficient then, this 3 layer network can approximately represent any continuous function over compact set in \mathbb{R}^n . Given any compact set in \mathbb{R}^m ; if those of you do not know what compact set is, simply take a task closed is like a cylindrical set of closed n terminals.

So, think of it as the closed bounded set in \mathbb{R}^m . So, given any compact set \mathbb{R}^m , we can approximate any continuous function by such a network. We will formally state the theorem now. So, for simply the notation for compact set we can simply think of them as some closed interval to the \mathbb{R} power m that is what we are going to do. So, here is the theorem: Let ϕ be some bounded strictly monotonically increasing continuous function. This will serve the purpose of our activation function.

So, given some function ϕ which is bounded, strictly monotonically increasing and continuous, which is all what our sigmoids and our tan hyperbolic everything satisfies. Let us script C of I^m with the set of all continuous real valued functions on I^m ; where, I^m is the cylindrical set in is the set of all is the $[0, 1]^m$. I^m is the unit hypercube in m dimensions. The interval $[0, 1]^m$ is the Cartesian product of the interval $[0, 1]$ m times.

So, $[0, 1]^m$ is the hypercube of side 1 in \mathbb{R}^m . One of its corners is at the origin. If you call that as I^m , this is a simple compact set you have taken \mathbb{R}^m . C of I^m is the continuous real valued function that map I^m to \mathbb{R} . So, any continuous function continuous real valued function that maps elements of I^m to \mathbb{R} in $C(I^m)$. If we taking I^m because as I said we only want to consider any compact set in \mathbb{R}^m and this is as good as any other compact set in \mathbb{R}^m . So, let us see I^m be the set of all continuous real valued

functions and we have asking, can I represent every function in $C(I_m)$ using a network and we can.

(Refer Slide Time: 47:02)

Then, given any $h \in C(I_m)$ and $\epsilon > 0$, there exists a p and real numbers $\beta_j, b_j, w_{ij}, i = 1, \dots, m, j = 1, \dots, p$, such that the function

$$\hat{h}(X) = \sum_{j=1}^p \beta_j \phi \left(\sum_{i=1}^m w_{ij} x_i + b_j \right)$$

satisfies

$$\sup_{X \in I_m} |h(X) - \hat{h}(X)| \leq \epsilon$$

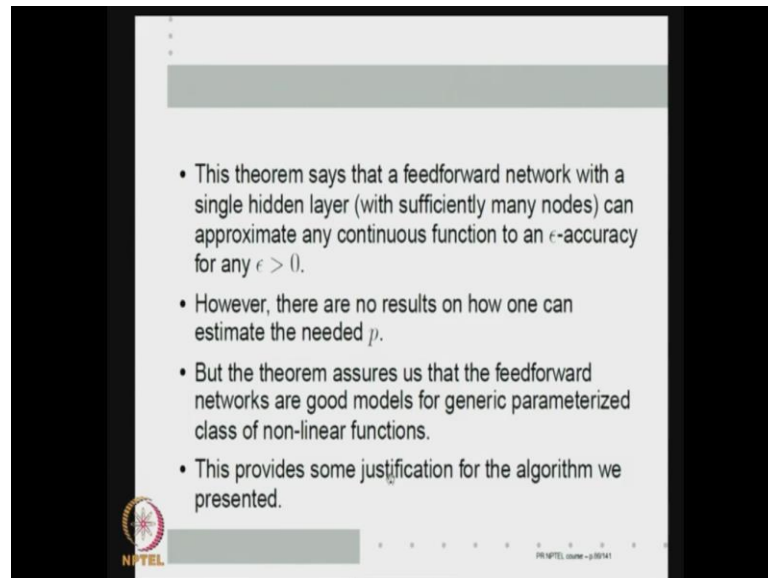
NPTEL

NPTEL, course - p32141

Then, given any function in C_m , given any continuous function, continuous real valued function h and any epsilon. I want h represented in a epsilon accuracy then, there exist a p some number of hidden nodes and real numbers β_j, b_j, w_{ij} that means the weights. Such that, if I calculate a function \hat{h} , which is actually was this through a network represents with p as the number of hidden nodes and this β, b and w_{ij} as the corresponding weights. If I calculate \hat{h} like this then, the maximum difference between h and \hat{h} , maximum taken over all points in I_m is less than epsilon. That means, given any epsilon and given any function h , I can always find a 3 layer network with some number of hidden nodes and some weights such that, the output of the network does not differ from the actual value of the function by more than epsilon at for any input.

It is a very remarkable theorem. This tells me that my 3 layered network is good enough to represent any continuous function to an arbitrary accuracy. Of course, I do not know what really network I want. What do you mean by what 3 layer network? The only thing that is not known 3 layered network is p , how many hidden nodes I should have. This theorem only tells me if a p exists. I do not know what the value of p is.

(Refer Slide Time: 48:36)



So, what the theorem says is that a feedforward network with a single hidden layer but, with sufficiently many nodes in the hidden layer can approximate any continuous function to an epsilon accuracy for any epsilon. So, essentially if I put sufficient nodes in my hidden layer, I can represent any continuous function. So, it is a very good parametric class of function because I do not need, I can represent all continuous functions is good enough for me.

Now, see in this representation I am using phi as the activation function the hidden nodes that is why I chose phi to be bounded strictly want to turn increasing continuous and so on. The only catch is of course that I do not know how to estimate this p . So, given some samples nobody will tell me how I can choose a particular 3 layered network. A particular 3 layer network By a particular 3 layer network we mean, how many nodes in the hidden layer should I choose. But, such as it is, if I chose sufficient number of hidden nodes then, the theorem guarantees that I can find a representation.

Now, because the supremum difference between $h(x)$ and $\hat{h}(x)$ is bounded essentially, I can find the corresponding weights for a given p by minimizing this sphere error. So, if I can actually find the global minimizer of this squared error then, you know I can arbitrarily I can get a good approximation to any function. So, in this sense, the algorithm that we have given is reasonable. This theorem gives me some justification for the algorithm essentially, even if I use only one hidden layer, if I put sufficient nodes in

the hidden layer and if my algorithm can get me the weights that are global minimizers of the squared error loss then, I can approximate any continuous function.

So, the theorem assures that the feedforward networks are good networks for generic parameterized class of non-linear functions and provide some justification for the algorithm we have presented. Of course, our algorithm is gradient descent and hence it stuck in local minimum. But, if I can go to global minima, I get weights and if I start up with sufficient of hidden nodes, I can represent. So, essentially we set out to have a good parameterized class of non-linear functions and the models we considered all good parameterized class of non-linear functions the theorem assures us that. And we can certainly find the weights by minimizing the empirical risk under squared error loss.

(Refer Slide Time: 51:28)

- The three layer network represents a function
$$h(X) = \sum_{j=1}^p \beta_j f \left(\sum_{i=1}^m w_{ij} x_i + b_j \right)$$
- Earlier we have considered linear regression models of the form
$$h(X) = \sum_{j=1}^p \beta_j \phi_j(X)$$
- What is the difference?

NPTEL course - 909041

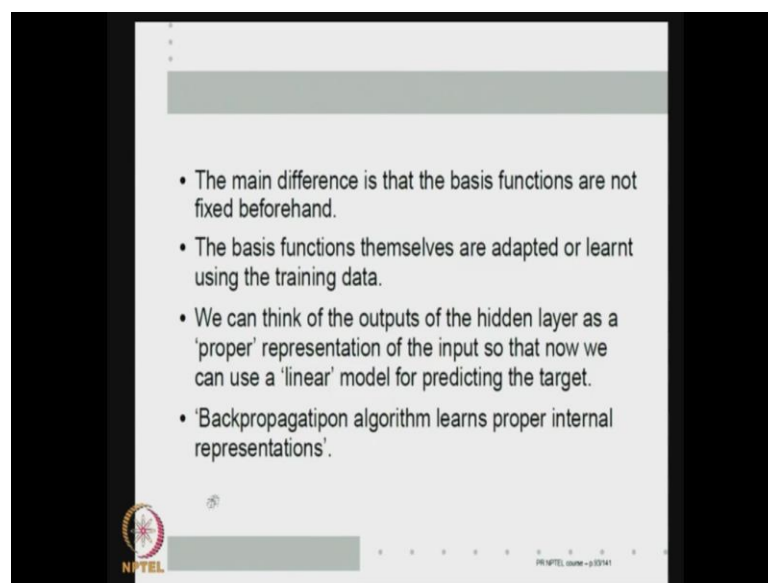
Let us look at it a little more carefully. So, this is what it 3 layer network represents: D is equal to $\sum_{j=1}^p \beta_j f \left(\sum_{i=1}^m w_{ij} x_i + b_j \right)$. When we did our linear least square regression, we said that any function in the linear regression model can be written as $\sum_{j=1}^p \beta_j \phi_j(X)$. So, if I think of all this part $f \left(\sum_{i=1}^m w_{ij} x_i + b_j \right)$ this is something that depends on only on j because summed over all the j of course, it depends also on the X 's.

So, I call it $\phi_j(X)$. So, this structure is same as this structure. We already know how to learn this using our linear regression models, what is the difference. So, we said that the linear regression models linear least squares can do this. So, what is extra that it is

giving? Difference is the linear least squares these ϕ_j 's have to be fixed beforehand, they are not they cannot be chosen based on the data.

Whereas here, ϕ_j 's are this, if ϕ_j 's involve this w_{ij} and b_j and w_{ij} and b_j are learnt using the data. So, if ϕ is the functions of not prefix, they are learnt using the data. In the linear regression models this ϕ is a functions of prefix like x , x square x cube so on. Whereas here, I am choosing the w_{ij} 's and b_j 's are learnt from the data. So, it is effectively as if I am learning the ϕ_j 's needed using the data.

(Refer Slide Time: 53:09)



The difference is that, the basis functions are not fixed beforehand. Basis functions themselves are adopted or learnt using the training data. These ϕ_j 's actually come from here, they contain adaptable adjustable parameters. So, when I am not using fixed basis function, if you remember again and again when you talked about linear regression we kept on saying as long as the basis functions are fixed. So, that is the issue. If basis functions themselves are adopted then, I can learn non-linear functions. So, we can think of the output of the hidden layer.

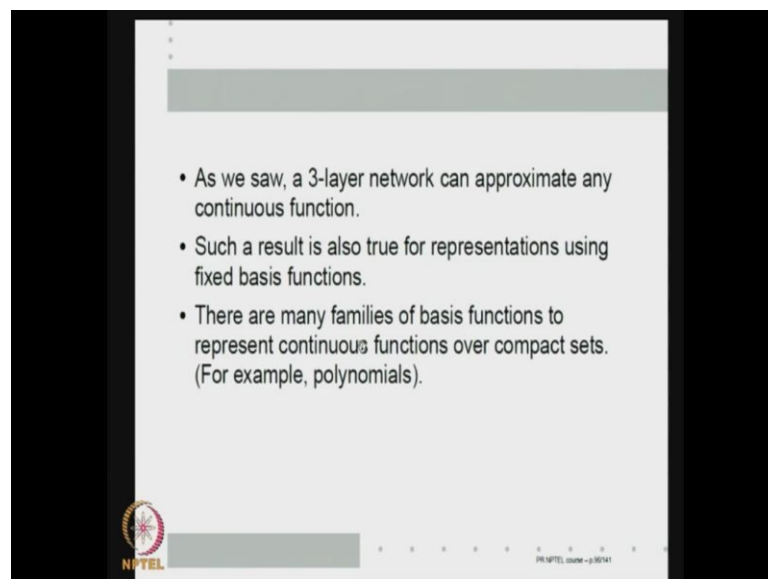
Now, if I go back to my 3 layered network. If I just consider hidden through output layer, this is like that b_j is a ϕ_j because the outputs of this f_{ij} , once the weights are all fixed, ϕ_j 's are fixed. So, this is like a fixed basis representation. So, we can think of what we are doing using back propagation is learning this weights so that instead of representing input as x_1 to x_m , we representing as ϕ_1 , ϕ_2 , ϕ_p . The ϕ_1 ,

$\phi_1(x)$, $\phi_2(x)$, ..., $\phi_p(x)$ is good representation because now a linear model can represent the function you want. So, the entire issue of learning these weights is like transform this representation x_1 to x_m representation to $\phi_1(x)$, $\phi_2(x)$, ..., $\phi_p(x)$ representation.

So, we can think of back propagation as learning a proper internal representation. It is nothing to do with only one hidden layer, there can be any number of hidden layers. So, essentially from the input up to the last but one layer is a representation of the output. So, back propagation allows you to learn the right representation so that now under that representation a simple linear model can learn. That is what my back propagation is doing.

So, you can think of the output of the hidden layer as a proper representation input so that now we can use a linear model for predicting the target. This is the reason why the activation algorithm learns proper internal representation that is how it is termed. So, one often says that, use back propagation to learn proper internal representations.

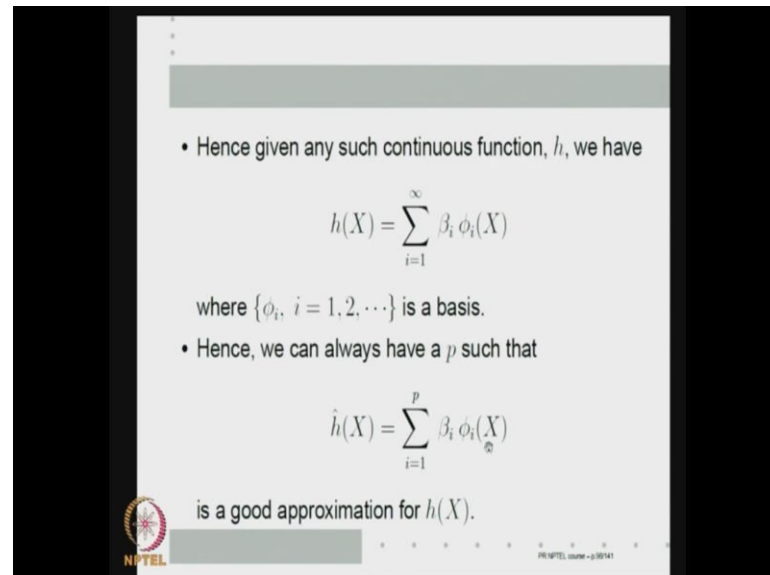
(Refer Slide Time: 55:40)



There is one more issue we should consider, we said that the 3 layer network can approximate any continuous function hence, very nice; so, it is a very good model for non-linear functions that is the reason we are studying it and so on so forth. But then, there may be many other ways of representing all continuous function. This result is also true for many representations; for examples I can use polynomials, I can use you know fourier functions. There are many families of basis function to represent all

continuous functions. Continuous functions are compact sets, it forms a vector space, so there will be some spaces for it and in terms of basis I can represent anything.

(Refer Slide Time: 56:21)



• Hence given any such continuous function, h , we have

$$h(X) = \sum_{i=1}^{\infty} \beta_i \phi_i(X)$$

where $\{\phi_i, i = 1, 2, \dots\}$ is a basis.

• Hence, we can always have a p such that

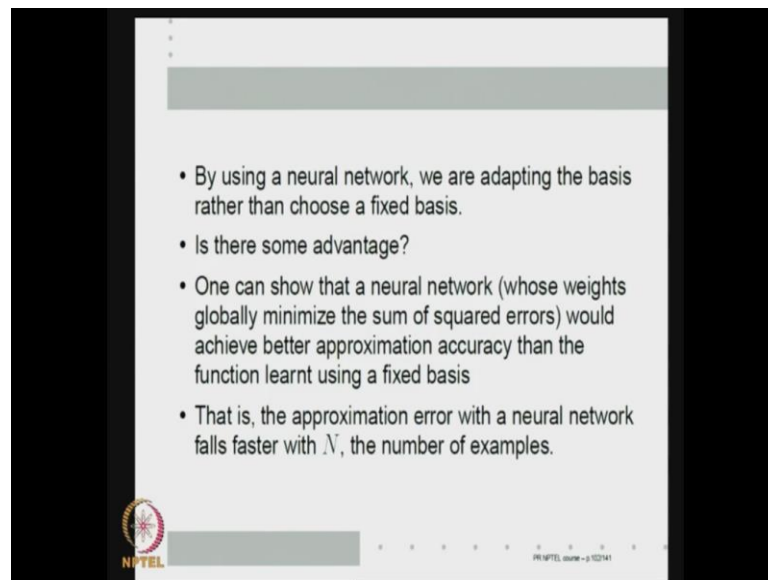
$$\hat{h}(X) = \sum_{i=1}^p \beta_i \phi_i(X)$$

is a good approximation for $h(X)$.

What does that mean? Given any continuous function h , we can always represent h as equal to $\sum_{i=1}^{\infty} \beta_i \phi_i(X)$ because this function space may have infinite dimensional. $\beta_i \phi_i(X)$ where; ϕ_1, ϕ_2 and so on is a basis for a vector space. So, such a representation always exists. And polynomial is one particular thing we already know for example, in an \mathbb{R} to \mathbb{R} very function can be represented given the basis $1, x, x^2, x^3$ so on.

All continuous functions of compact sense can be represented as polynomial. So, there are many such basis functions ϕ_i and I can represent any continuous function as an infinite sequence infinite summation basis function like this. Once I know that this infinite sequence converges to $h(X)$ there would always exist a number m such that, this sum will be equal to $h(X)$ within an epsilon of $h(X)$. So, I can always have a p such that, if I take the sum up to p then, that $\hat{h}(X)$ is a good approximation to $h(X)$. So, what is the big deal again, we have the 3 layer network also continuous function, this also represents all continuous functions. In this case I am using fixed basis. In 3 layered network I am not using fixed basis.

(Refer Slide Time: 57:35)



What is the difference? By using neural network, I am adopting the basis function rather than choose a fixed basis. Is there some advantage? Yes, one can choose the neural network whose weights globally minimize the sum of squared error would achieve better approximation accuracy than the function learning using fixed. So, given a fixed number of examples, a neural network will give me better accuracy than any of the fixed basis functions. Essentially, that is what is meant by I can achieve better approximation.

So, even though I can use fixed basis and still get the result that any continuous function can be represented. By using a neural network, I can get better approximation. That is for the same number of examples I get better approximation. So, this is what the neural networks where the basis functions themselves are adopted are giving me as an advantage. There are universal approximators of continuous functions and I can learn it with better accuracy than using a fixed basis. So, there we stop for today. Next class we will look at a few more issues of back propagation.

Thank you.