Introduction to Machine Learning

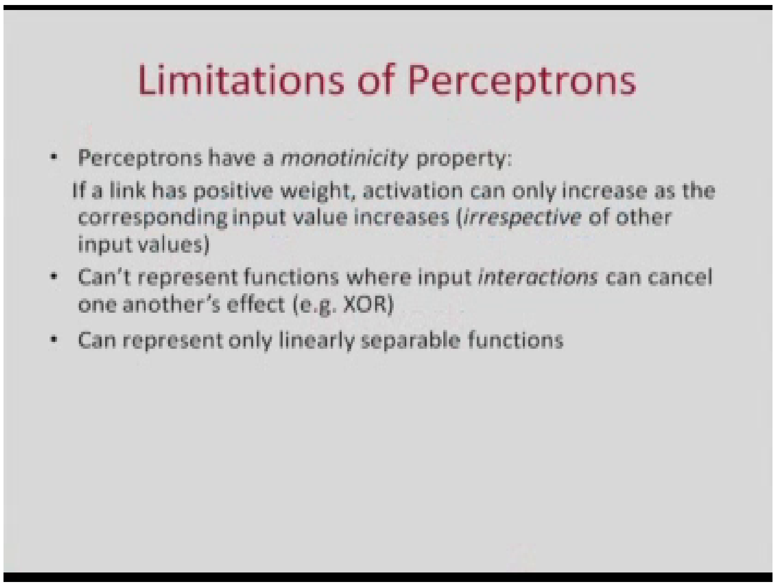
Prof. Sudeshna Sarkar
Department of Computer science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 28
Multilayer Neural Network

Good morning. Today we talk about the second part of lecture in Neural Network, where we will talk about Multilayer Neural Network. We have already looked at individual neural units and discussed that they can represent linear functions, but the main excitement about neural network is because they can represent non-linear functions. And we can represent non-linear functions by stacking layers of perceptrons into different architectures.

So, today we will look at Feed Forward Multilayer Neural Networks which is a particular type of connections in neural network

(Refer Slide Time: 01:13)

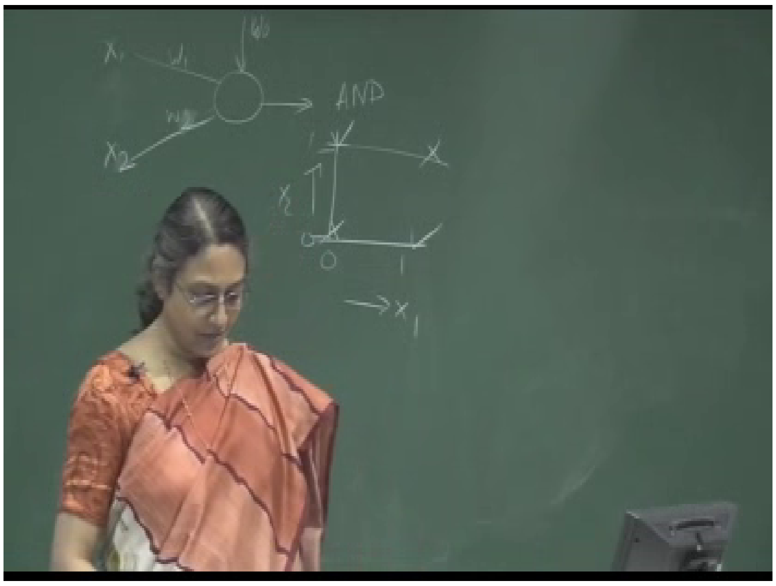


Limitations of Perceptrons

- Perceptrons have a *monotinicity* property:
  If a link has positive weight, activation can only increase as the corresponding input value increases (*irrespective* of other input values)
- Can't represent functions where input *interactions* can cancel one another's effect (e.g. XOR)
- Can represent only linearly separable functions

Now, first of all we will look at what are the limitations of perceptrons. We have seen that the (Refer Time: 01:19) really not discussed, but perceptrons have this monotinicity property.

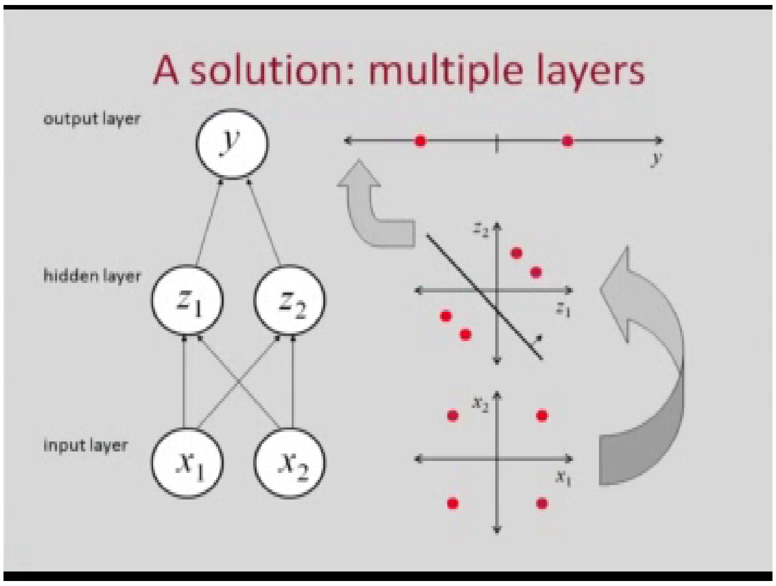
(Refer Slide Time: 01:31)



So in a perceptron what we are doing is that, we have an input and we have multiple outputs and we have weights associated with them. And because of this type of connection perceptrons have a monotonicity property. If a link has positive weight activation can only increase as the input value increases. It cannot represent functions where input interactions can cancel each other. So each input is individually interacting with the neural, so it cannot handle interactions between the (Refer Time: 02:04). For example, perceptrons can be used to represent gates like and suppose there are two inputs x 1 and x 2 you want to pass one if both x 1 and x 2 are true. Then you can set the weights and the thresh hold such that the result will be 1 only when x 1 and x 2 is 1.

However, perceptron cannot represent the XOR function. In XOR function, suppose we have two variables x 1 and x 2 which can take 0 and 1. In XOR is true for in this case it is negative here and here. So, XOR function is not linearly separable and it is not monotonic in the individual inputs and it cannot be represented by a perceptron.

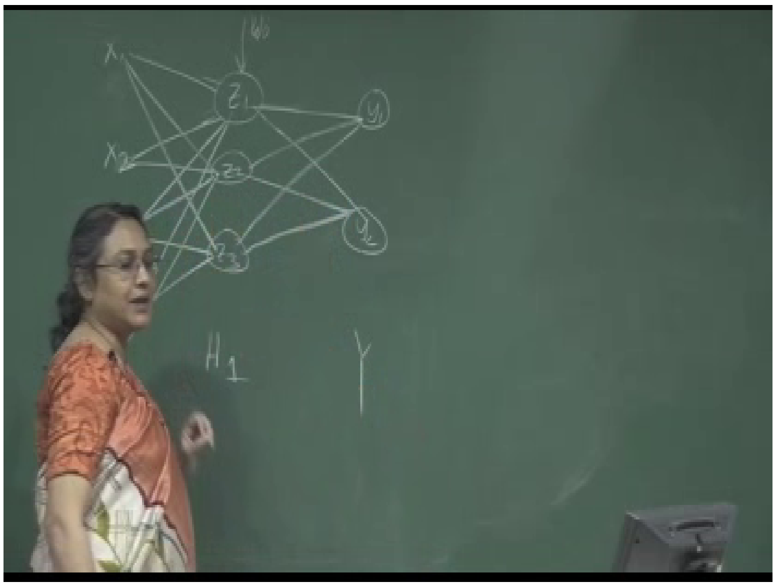
(Refer Slide Time: 01:13)



A solution: multiple layers

So, a solution to this is to have a multilayer neural network. Where we have this units stacked on each other. So, we have these inputs.

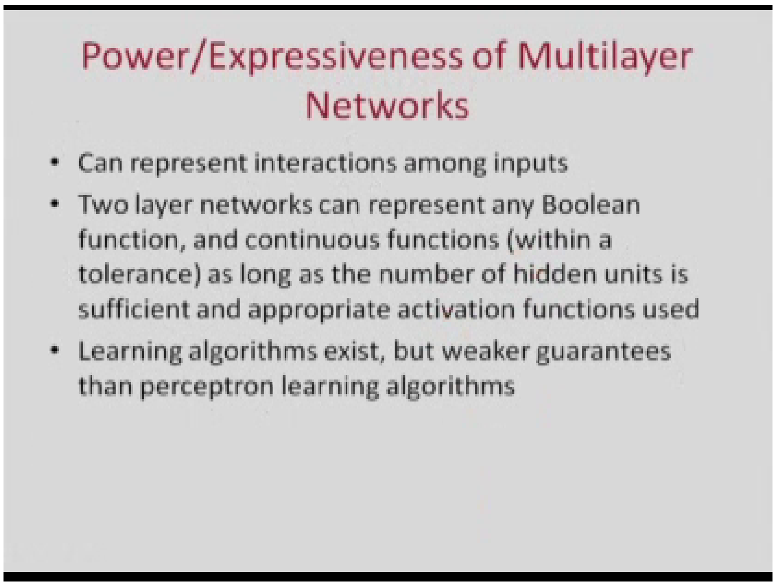
(Refer Slide Time: 03:40)



So, we have x 1, x 2, x 3, x n as the input and we have let us say 1, this is the first layer we call it Hidden layer 1 and it has units let us say z 1, z 2, z 3, z l and then there is the output layer where we have units y 1, y 2. Now, we have inputs feeding through from input to this layer and etcetera, and then from here to this layer. We can have a multilayer network where we have the input layer and the output layer and between the input and the output we can have other units which are called the Hidden Units. Why Hidden? Because, in the training examples they are not observed, we have the input and the output these are called the hidden units. And through these hidden units we can represent many non-linear functions.

For example, if you look at this picture we have x 1 and x 2 and each of z 1 and z 2 is a linear function of x 1, x 2. And then y is a linear function you know does a linear separation between x 1 and x 2, but y does a y is a function of z 1 and z 2 and if we use suitable non-linear activation functions then this sort of connection can represent XOR or other non-linear function.

(Refer Slide Time: 05:30)



Power/Expressiveness of Multilayer Networks

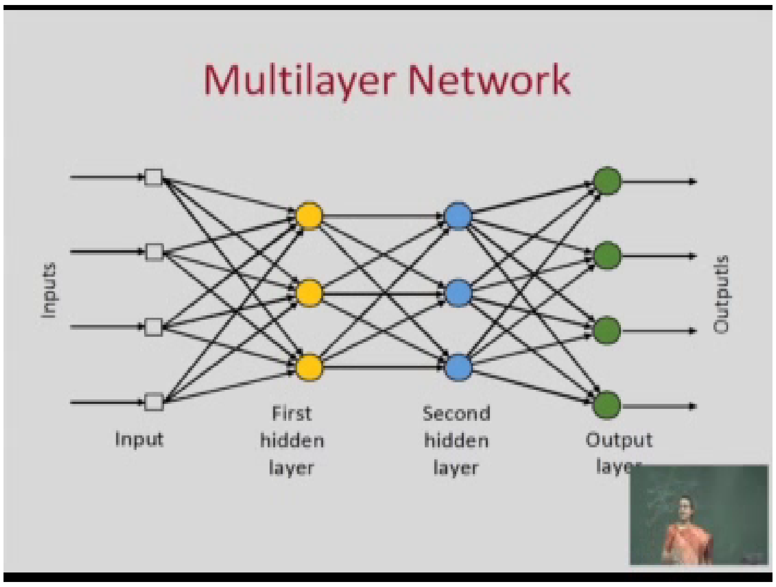
- Can represent interactions among inputs
- Two layer networks can represent any Boolean function, and continuous functions (within a tolerance) as long as the number of hidden units is sufficient and appropriate activation functions used
- Learning algorithms exist, but weaker guarantees than perceptron learning algorithms

So, we can look at this expression of what multilayer networks can express. We have seen that single layer networks can represent linearly separable function. Multilayer networks can express interactions among the input. In particular a 2 layer network means this is a 2 layer network where you have 1 hidden layer and 1 output layer, this is the input this is the hidden computing layer this is the output computing layer. And these 2 layer neural network can represent any Boolean function.

And continuous functions within a tolerance provided of course you have the requisite number of hidden units and you would use appropriate activation functions then all Boolean functions and all continuous functions within a certain tolerance can be represented using 2 layer neural networks. If you have 3 layer neural networks, then you can represent all computable functions. These functions can be represented using 2 layer and 3 layer neural networks. So, they have very good representation capacity.

But the next question is, is it learnable? Just because a presentation exists to represent a function does not immediately mean that you can learn the function well. But, for neural networks like this learning algorithms do exist, but they have weaker guarantees. In perceptron learning rule we said that, if a function exists then this procedure will converge. So, for multilayer neural networks we cannot give such strong guarantees, but algorithms exist and people are working on different very exciting types of algorithms.

(Refer Slide Time: 07:45)

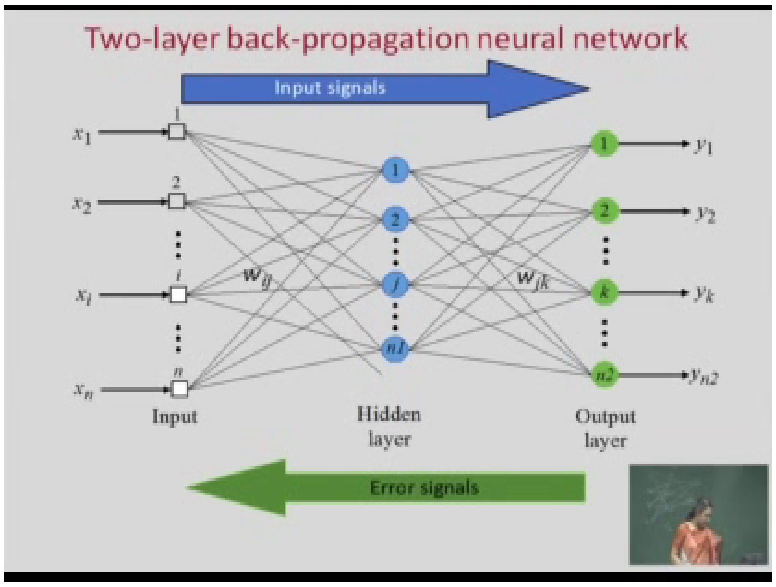


Multilayer Network

So, let us look at a general structure of a multilayer network. This is a 3 layer network where there is the input, and then we have the first hidden layer, the second hidden layer, and the output. This is an example of a layered feed forward neural network. This is a feed forward neural network because the inputs, these connections that we have drawn are single connectional. Input to first hidden layer, first hidden layer to second hidden layer, second hidden layer to output, all the edges are single directional and it is going forward from the input to output there is no back link.

So, this is why is called feed forward neural network. This is called a layered network because we have organized the neurons into layers and layer i is connected to layer i plus 1. Also this particular diagram shows a fully connected layered feed forward network, where there are 2 hidden layers, 1 output layer and of course the input layer is there.

(Refer Slide Time: 08:57)


Two-layer back-propagation neural network

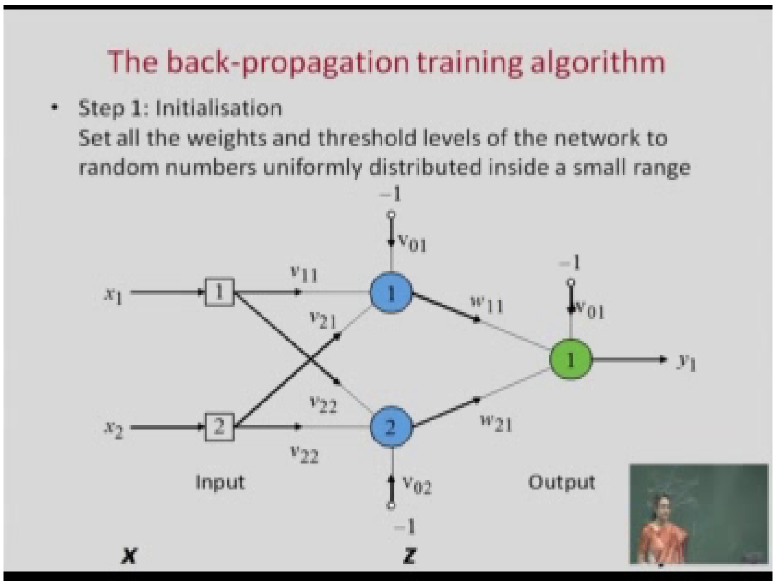
So, in this particular type of feed forward neural network the input will be going from feed forward from input to the output through the hidden layer. Now in the while talking about perceptron training we said that based on the error in the output we change, if we observe there is an error in the output between what should be the ideal value and what is computed then we change the weights of this connections so that that error is made smaller. So that is what we looked at in perceptron training.

Here also we need to do the same thing. However, here there is one difficulty. We know what should be the ideal output here and the ideal output here, so based on that we can change these weights. But, at the hidden node the ideal output is not told to us, it is not known to us directly, so on what basis do we compute these weights. We know the ideal output here, we can compute the error directly and on that basis we can update these weights. But we need to know what is the error here.

So, what is done is that the error that is observed here is propagated backwards. The assumption is the error here is a result of error here, here, and here, because these are the three nodes that are feeding inputs. This error here is due to errors here, and therefore what we do is that this error we now back propagate to these nodes. The error here also we back propagate to this nodes based on these each of the nodes we know what is the notional error, based on that error we update this weights. If there were more layers here, the error here will again be back propagated to those other nodes.

So, the error signal flows backward. The computation in the network is forward, but the error signals flow backward based on this computation of the error signals we figure out how the weights have to be changed. That is why the method for updating weights in such multilayered neural network is called back propagation. So the input is going forward and the error signal is going backward.

(Refer Slide Time: 11:49)

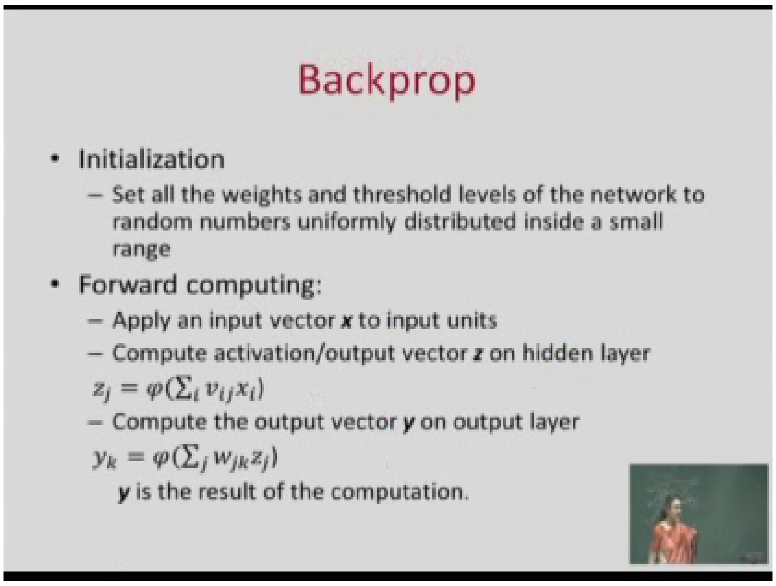


The back-propagation training algorithm

- Step 1: Initialisation
  Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range

So, these are the steps in the back propagation training algorithm. First step is initialization, we give the structure of the network and we initialize the values of these weights and usually we give these weights small random values. We give them small random values to the weights at different biases in the network.

(Refer Slide Time: 12:21)



Backprop

- Initialization
  - Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range
- Forward computing:
  - Apply an input vector ***x*** to input units
  - Compute activation/output vector ***z*** on hidden layer

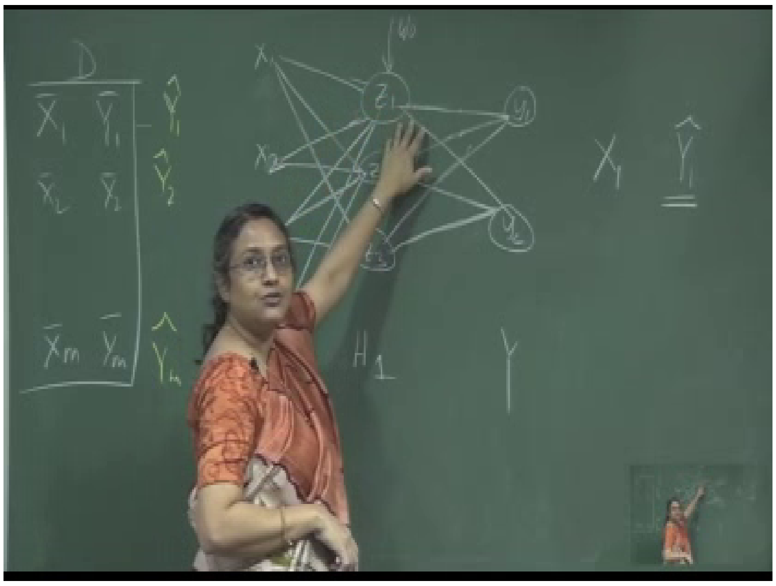
    $z_j = \varphi(\sum_i v_{ij} x_i)$
  - Compute the output vector ***y*** on output layer

    $y_k = \varphi(\sum_j w_{jk} z_j)$

    y is the result of the computation.

Now after initialization we do the forward computing.

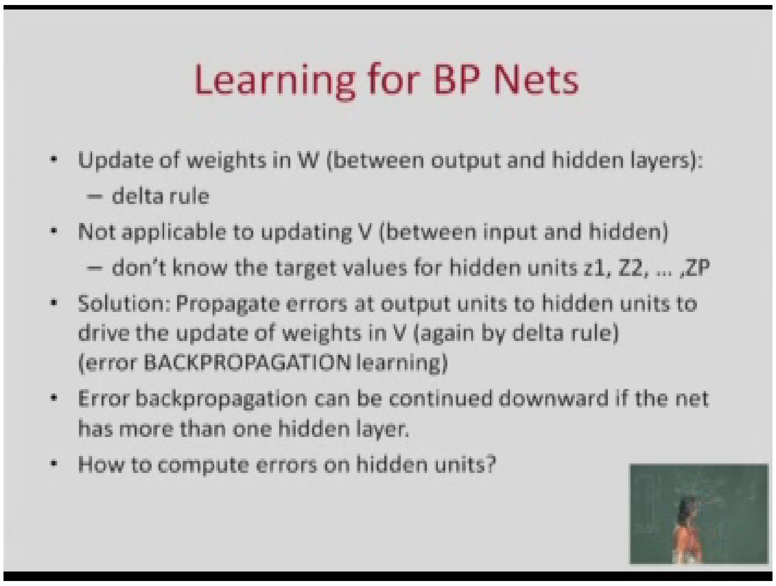
(Refer Slide Time: 12:37)



Now what we do is that we take our training example which comprises x 1, y 1, x 2, y 2, x m, y m. These are our training examples we take the examples one by one and apply them to the network. We apply x 1 to the network we get some output y 1 hat. So, given x 1 we get y 1 hat, so this is the output that we get. And the ideal output is y 1 hat, and the outlet let me draw it here. So, the output that we get is y 1 hat, here we get y 2 hat, here we get y m hat for a particular configuration of the network. And if this y 1 hat and y 1 are different this is different then we look at the error and based on this we change the weights.

So, in the format computing we apply the input first to this layer here we first take the summation followed by the activation function we get the output at this layer; we get the output z 1, z 2, z 3, and then based on that we compute y 1. y 2 etcetera as this is computed as the activation function applied to sigma w i z i right here, it is activation function applied to sigma x i w i. So, we get the do the forward computing.

(Refer Slide Time: 14:07)



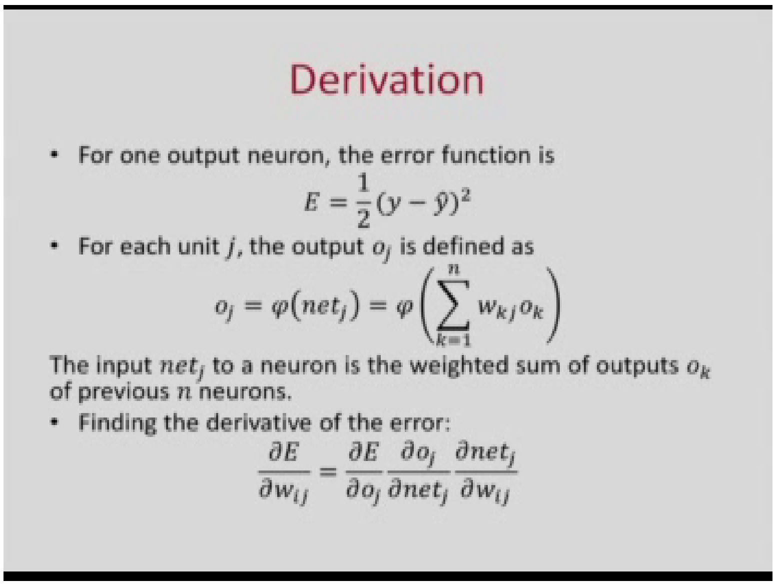
Learning for BP Nets

- Update of weights in W (between output and hidden layers):
  - delta rule
- Not applicable to updating V (between input and hidden)
  - don't know the target values for hidden units z1, Z2, ... ,ZP
- Solution: Propagate errors at output units to hidden units to drive the update of weights in V (again by delta rule) (error BACKPROPAGATION learning)
- Error backpropagation can be continued downward if the net has more than one hidden layer.
- How to compute errors on hidden units?

After doing the forward computing, now we have to update the weights. As i said that we can update the weight at the output layer in a similar fashion as we did for single layer units. If you think of you have single layer sigmoid units we have already worked out how to update the weights. These weights, suppose let us say this set of weights we call them w and this set of weights we call them v. So, w is updated using the sigmoid layer training rule which we discussed in the last class.

But, how to update v; As we said that we do not know the target values of z 1, z 2, z 3, so what we do is that we back propagate the error here to the error here. So, we propagate errors which are visible at the output units to this hidden units and then based on that error we do the similar training rule. And if there were more layers between this and the input so these errors can be further propagated in this direction, further propagated downwards. So, error back propagation can be continued if the net has more than hidden layers.

(Refer Slide Time: 15:44)



Derivation

- For one output neuron, the error function is

$$E = \frac{1}{2}(y - \hat{y})^2$$

- For each unit j, the output o_j is defined as

$$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right)$$

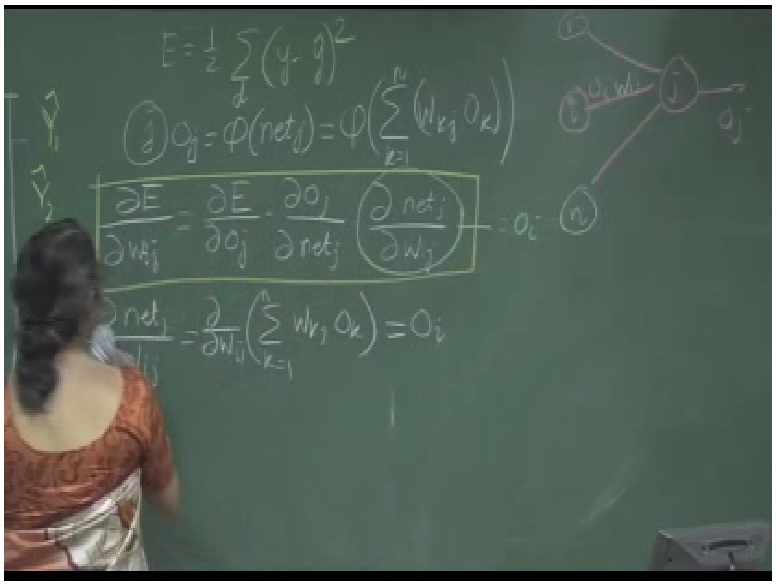
The input net_j to a neuron is the weighted sum of outputs o_k of previous n neurons.

- Finding the derivative of the error:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Now, we will have to see how to compute the errors. And for that let us try to do the derivation of this computation.

(Refer Slide Time: 16:08)



$E = \frac{1}{2}\sum_d (y - \hat{y})^2$
$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} (w_{kj} o_k)\right)$
$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} = o_i$
$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = o_i$
$\hat{y}_1$
$\hat{y}_2$
$o_i w_{ij}$
o_j

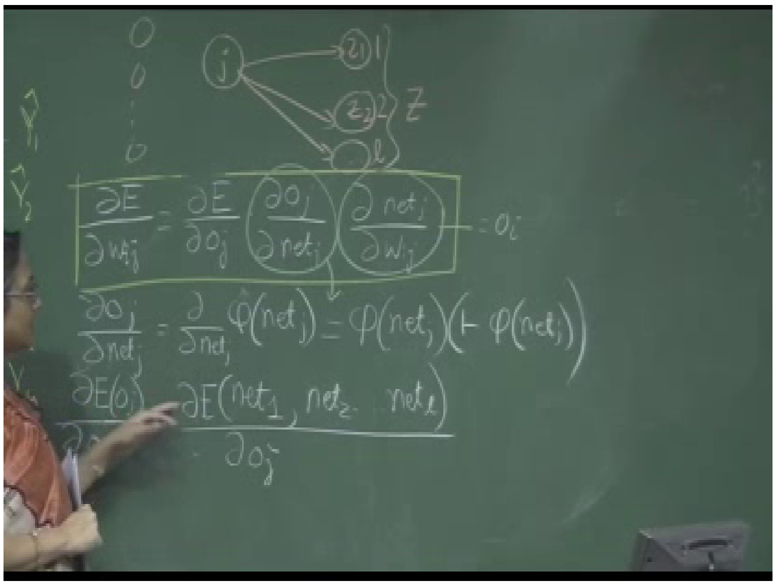
Now, for the output neuron so the error function is E equal to half sigma y minus y hat whole square, and we can take summation over all the training examples if you wish. Now let us think of in the network we have different units, these units are either at the output or at the hidden layer, so let us take any unit chain. For each unit j, so j is a unit and for j the output is o j. First of all we have seen that in the unit there are two components; one is the summation component, the second is the activation function component.

So first, the summation is applied and then the activation function is applied. Let us say that output here we call net j and output after this we call o j, this is a unit j. So, the output of the summation is net j and output of the final output is o j. So for unit j, o j equal to phi of net j and net j equal to summation w i x i, where i ranges over all the inputs to the unit j. This is equal to phi of sigma k equal to 1 to n, k ranges of all the input w k j. So, w k j is the weight from the unit k to this unit j and o k is the output of unit k. So, output of unit k is an input to unit j. Suppose, this is a node which is j and this has these three inputs; 1, 2, 3,, so this is w 1 j, this is w 2 j, this is w 3 j and the output is here o 1, o 2, o 3. So, this is the output computed as unit j.

Now, if we want to find the partial derivative of the error with respect to w i j we get this is equal to del e by del o j, del o j by del net j, del net j by del w i j. So to simplify the computation we have used the chain rule to write del e the partial derivative of the error with respect to w i j as the product of these three partial derivatives, so to make the computation simple for us. This is m j this is my unit j and these are the units which are feeding to this unit, this is the unit i. And j has you know initially the output is net j and then the output is o j. So this is o j and this is o i.

Now, we have decomposed this partial derivative in these three components. Now what we will do is that let us remember this and we will do these three computations separately then we will write them together. So first let us look at del net j by del w i j, del net j by del w i j. So what is net j, net j is sigma o i w i j. And so we have del del w i j sigma of w k j o k equal to 1 to n, so this is equal to o i. Because only here we have w i j and these other link 1 to n they have nothing to do with i. So, del net j by del w i j is simply o i. In corresponding to o i only we have w i j, and we have o 1 w 1 j, o 2 w 2 j, o 3 w 3 j they have nothing to do with w i j. So only here we will have and so this is equal to o i.

(Refer Slide Time: 22:46)



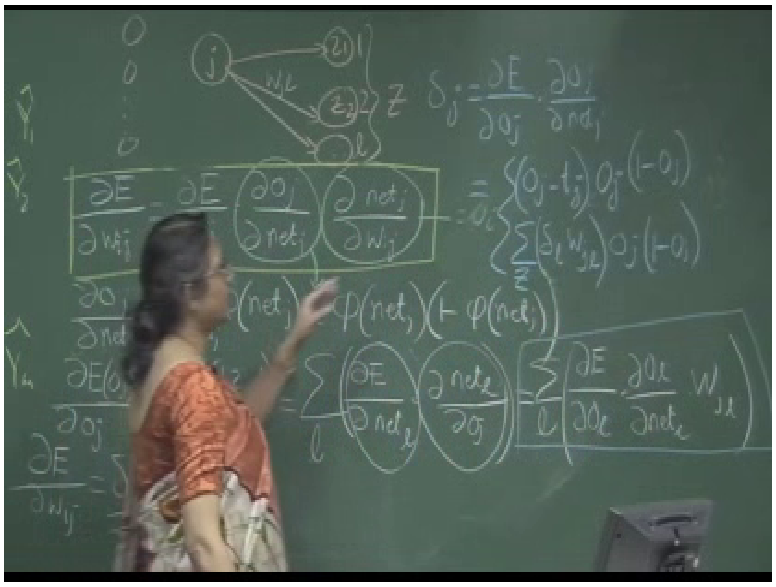
So, del net j by del w i j this we have this part we have computed and this a is equal to o i. Let me clean this so that we can work out the other two components of this expression. Next let us take this second term del o j by del net j. What is o j? O j equal to phi of net j, so del del net j we have phi of net j. We have already seen that this particular derivative will depend on the form of the activation function if you assume sigmoid activation function, for sigmoid activation function we will have this is equal to phi of net j 1 minus phi of net j. So, this is the second term here which we have computed given like this.

Now, let us look at the third term del e by del o j. Now we can take the derivative with respect to o j and we can get recursive expression of the derivative as follows; this is equal to summation over i. Now first of all let us say that the inputs to this network are 1, 2, 3, 4 up to n, so, let us say z equal to 1, 2, n these are the inputs to the unit o j. We can write this as follows; summation i. So, del e o j, before that let me write this.

So, this is e of the output o j is equal to this error is dependent on or rather write let me just write it in a proper way. It is other way round. So, this error at the unit o j has come from the units which are upstream of o j. If you allow let me rub this part, so that we can make the drawing here.

So, this is my the unit j and j has certain input which we have already seen, but j outputs to the next layer, and let the output of j go to the input of these units and let us call this set of nodes z. And let set say that z comprises of z 1, z 2, etcetera, or just for simplicity let me denote it for this step as 1 2 l. These are the nodes to which the output of j is feeding. During back propagation the error of these nodes is due to the error here. So, the error of o j is based on error of net z 1 net z 2 net z 3, or for simplicity let me write net 1, net 2, etcetera net l by del o j.

(Refer Slide Time: 27:56)



So, these components due to net 1, net 2, net l we can do a summation. We can write this as let me again compact it error of 1, 2, l, and this we can write as summation over l, this index letters call it i d or let us use l as the index and l we have del e by del net l del net l by del o j. So, this error is coming from these units and i am taking the summation over that so summation of del e by del o j del e where it is coming from this unit. So, we were writing it in chain form as del e by del net l del net l by del o j. So, for this node we have net 1 and then o 1, net 2 then o 2, net 3 then o 3, and the output of this contributes to the net 1 contributes to net 2 contributes to net 3.

Now what we can write is that, if we look at this so we can write this as summation of over l del e by. So, let us look at the output here, output here is o l. We can write it as del e by del o l, del o l by del net l. This part i have expanded to del e by del o l times del o l by del net l and we have del net l by del net j. So, del net l by del o j, so net l will depend on sigma w j l o j, so from this we get w j l. This component is equal to w j l. So, we have this expression.

We can compute this derivative of error with respect to o j if all the derivatives with respect to the output of the next layer are already computed. Now, we can write this as. Suppose we have del e by del w i j equal to delta j o i. We have the output of the current unit and the delta or the errors which are available here these errors are the errors which are available here, which is being proper contributing to the error here. So, this delta j corresponds to the error which was computed at this layer which is brought here. And the component of this delta j are, so delta j is one of this units and it is component is del e by del o j times del o j by del net j.

And this will be treated in different ways depending on whether the unit is the output unit or it is an intermediate unit. If it is an output unit we can use the formula which we have already seen so it will be equal to o j minus t j, t j is the target output at that node which we wrote us here, so o j minus t j times o j times 1 minus o j. This will be if j is an output neuron. And if it is an intermediate neuron this will be equal to it will get the outputs from the next layer, get the errors or the delta values from the next layer as we have written here so it will be sigma over the set z delta l w j l times o j times 1 minus o j, if j is an in a neuron.

So, this delta j is if we are looking at for the output neuron we have already seen this delta j equal to this, but for an intermediate neuron it comes from this formula where we will get delta j as sigma over z. Where, z is the set of those units to which this unit feeds input, so sigma over z the delta of those nodes the errors computed at those nodes times w j l times o j into 1 minus o j. This particular part, this comes due to the sigmoid activation function, if you change the activation function this will be a little different.

So, we stop the class today and in the next class we will look at how this is incorporated into the back propagation algorithm.

Thank you very much.