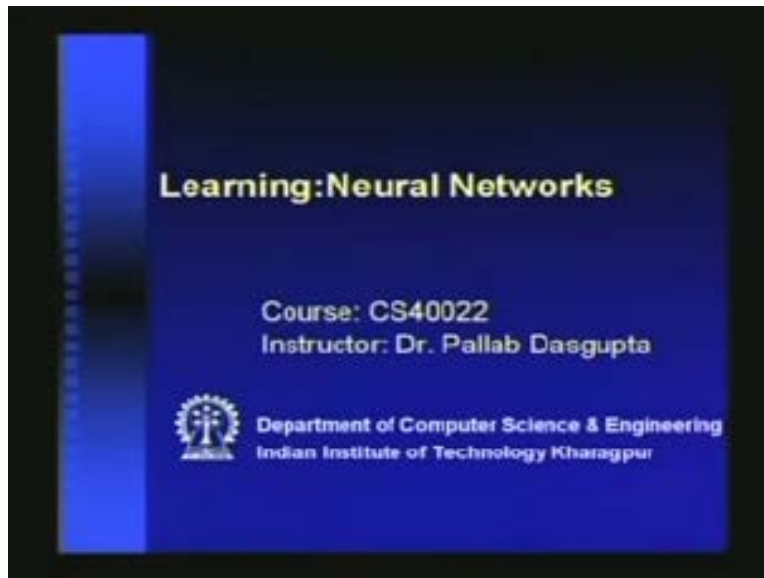


Artificial Intelligence
Prof. P. Dasgupta
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture No - 27
Learning: Neural Networks

In the last lecture, we looked at decision tree learning. Now, we will digress a little and look at kind of learning which comes under statistical learning. So, specifically, we will look at learning using neural networks. I will briefly introduce the structure and the basic definitions of a neural network and we will see how we can use neural networks to learn different kinds of functions.

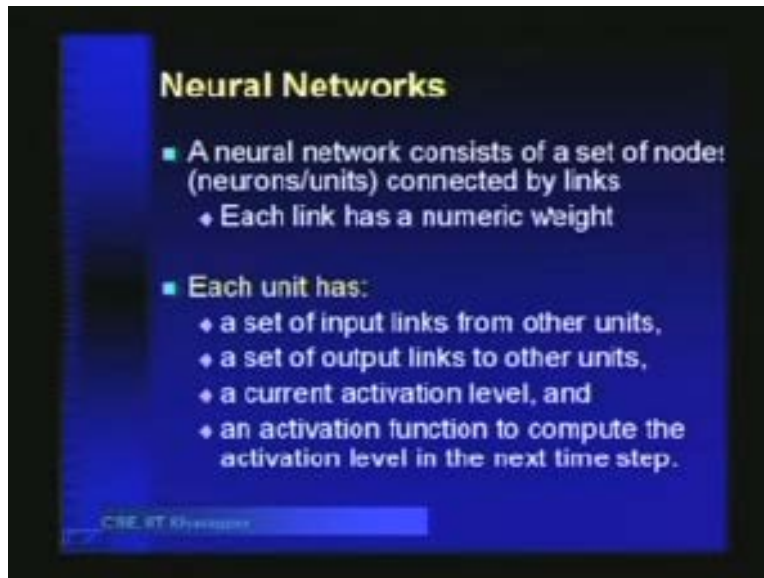
(Refer Slide Time: 01:13)



So, a neural network consist of a set of nodes which are neurons connected by links. Each of these nodes has very simple processing capability and there are lots of them and they are connected links; each link has a numeric weight, associated weight, each unit has a

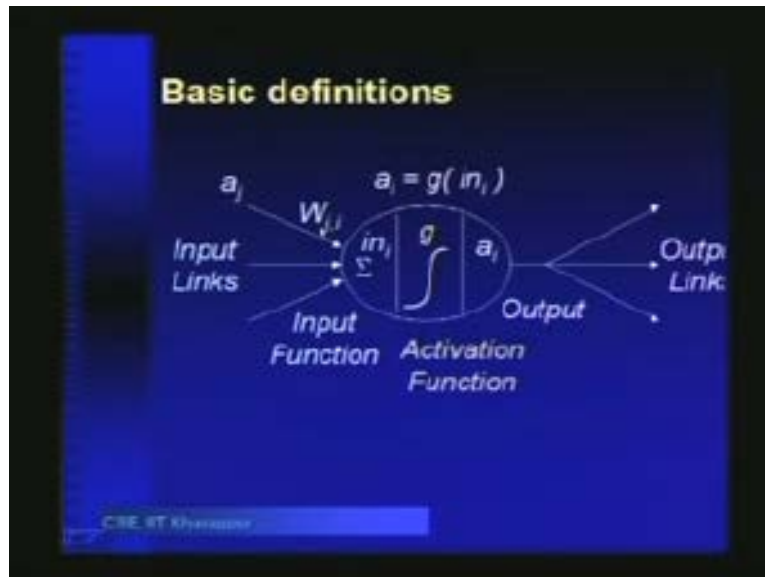
set of input links from other units. It has a set of output links which goes into other units, it has a current activation level which I will define shortly and has an activation function to compute the activation level in the next time step.

(Refer Slide Time: 01:52)



Now, when we talk about such nodes and etc., for the time being, we will assume that these are all done. This is just a model for computation. It is not that we actually have a processor which is doing all this kind of stuff, it is just that this is a model of computation that we are looking at, where we have set of nodes having very limited computational capability and we have these interconnections. So, a typical picture of a node is going to be like this- you see it clearly, right?

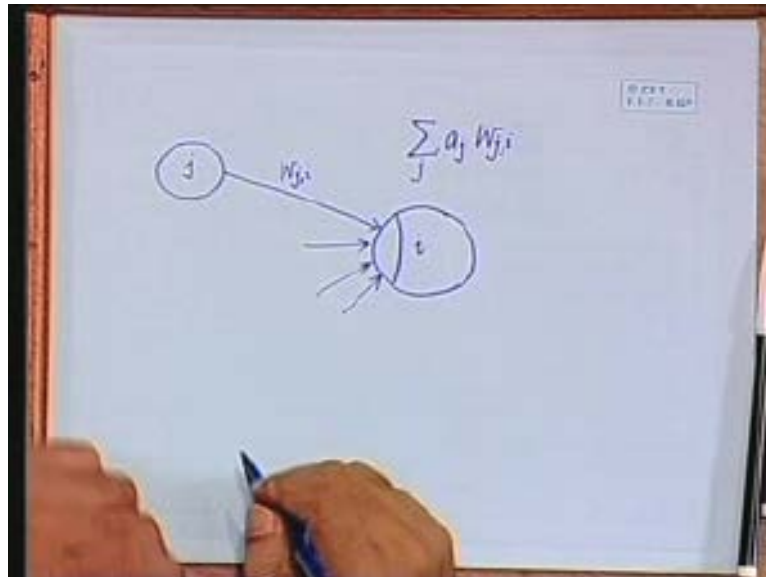
(Refer Slide Time: 03:25)



These are the set of inputs that we have, so, it is like, A_j is an input to this particular node or neuron; there is a weight associated with every link. So, the weight from a neuron j to neuron i is given as W_{ji} , so, this is directed link from j to i and it has weight W_{ji} . Then, we have a sigma function here, which computes the total input that it receives from the other neurons. I will define what is the total input and then, there is this activation function g , which is a function of the total input that the neuron i receives. The sum function of this and that defines the activation A_i of the neuron i . And then, this activation value is propagated through the output links to other neurons.

The total weight- the total weighted input- is the sum of the input activations times their respective weights. What does that mean? It means that if I have a neuron i - suppose this is our neuron i - and I have neuron j here feeding into this, this has a weight of W_{ji} . Then, the input that i receives from j is the activation of j times W_{ji} . This is the input that it receives from this. And if I want to compute the total input that i receives from all of its neighbors- preceding neighbors- then, I sum this over j , for all j which feed into this network, this node.

(Refer Slide Time: 05:41)



And then, in each step, we compute the activation a_i which is a function of the total input. So, it is a function g of $\sum_j W_{ji} A_j$. Is this clear? Now, this function can be of different types; it can be a threshold function, which says that A_i will be 1, if the total input exceeds some value. if the total input is more than 0.7, then, A_i will become 1; if the total input is less than 0.7, then, A_i will become 0. So, we could have something like that and so on.

(Refer Slide Time: 05:55)

Basic definitions

- The total weighted input is the sum of the input activations times their respective weights:

$$in_i = \sum_j W_{j,i} a_j$$

- In each step, we compute:


$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$

CSE, IIT Kharagpur

Now, let us see that how do we use a neural network like this for learning. So, this is a single layer network. I have 1 layer of input units here. These are the input units of the network and I have 1 layer of output units of the network. These are nodes of the network.

(Refer Slide Time: 07:48)

Learning in Single Layer Network



- If the output for a output unit is O , and the correct output should be T , then the error is given by:
$$Err = T - O$$
- The weight adjustment rule is:
$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

where α is a constant called the learning rate

CSE, IIT Kharagpur

There is a W_{ji} which is the weight of the link from i_j to the output i O_i . If the output for an output unit is O and the correct output should be T , then, what do we mean by the correct output? So, what we are trying to do is, we are trying to read; we are trying to learn a function from the inputs to the outputs. Again, this structure of the neural network for a function which has 4 inputs and 3 outputs is like this. Now, what will happen is that we will be given a set of training data set, just like we had in the decision tree scenario. We will be given training sets; those training sets will be valid input output pairs. So, it will be a set of a cases where we are given i_1, i_2, i_3, i_4 and for a given value of i_1, i_2, i_3, i_4 , what are the values of O_1, O_2 and O_3 ?

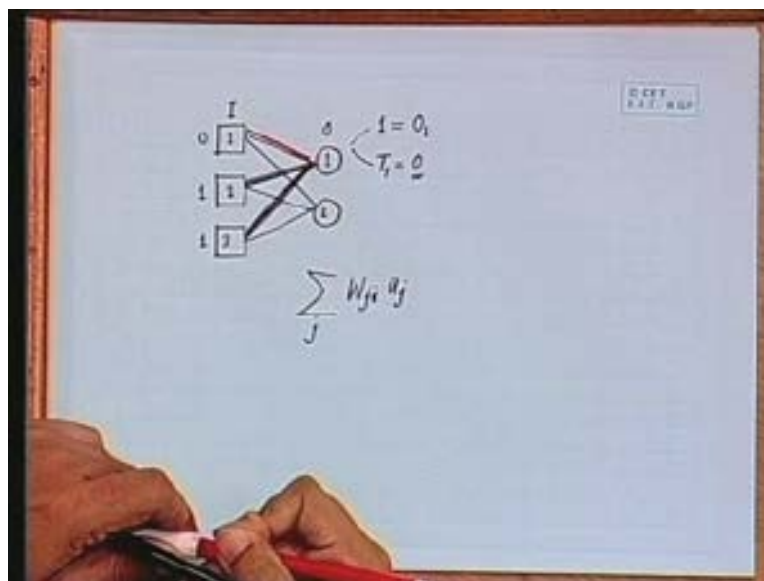
We will be given several such cases and the objective is to make this network learn that function. That at the end of the training, if I repeat any example from that, if I give the inputs corresponding to any of the sample data sets, then, the correct output should be displayed in the output. Also, if I give some inputs which was not there in the training set, then also, correct output should be displayed. Now, again, just like the previous case also, we can never be 100 percent sure whether it is giving the correct output for the others, but the objective is to make the neural network to learn the function, so that it is able to extrapolate also correct values for the input scenarios, which were not given in the training set.

Obviously, we have to define an error term and the objective will be to learn, so that this error is minimized. So, if the output for an output units is O and remember that the output is actually a real valued stuff, because our activation values that we have are real value; it can be real value, it can be 0 and 1 also. If we use a threshold function, then, the activation will be 0 or one. If you use some other kind of function which gives real values as output, you can give that also, and then, in case that case O will be a real (words unclear) and if the correct output is T , then, the error is given by T minus O . Now, the weight adjustment rule is W_j tends W_j plus α into i_j into Err .

Now, let us see what- I will explain this in a moment, so, what we are trying to do is, we are going to train the neural network in the following weight. So, we will present it with some input value. Initially, the weights are randomly assigned; they are all randomly assigned weights. I will give some input from the training set and then, I will see what output it produces. Based on the output that it produces, I will compute the error, because in the training set, the correct outputs are given, which is T. So, I will compute the error and depending on the error, I will readjust the weights on the inputs. Now, let us see in very crude terms, that what would that mean?

Suppose I had- I will start by given a giving an example where the inputs are all Booleans. So, let us say that I have these 3 inputs and these 2 outputs and I have a complete connection, so, it represents a complete bi-partite graph. Now, let us say that this is one, 2, 3; this is one, 2, so- this is i_1, i_2, i_3 and this is O_1, O_2 . Now, I have given some value, let us say, 0, 1, and let us say that at this point of time, this produces a value of 1, whereas this is our O and the value of O_1 and actual value, which means the correct value, T_1 , should have been 0. That means that what we need to do is- and let us say that the function that we are using here is a threshold function.

(Refer Slide Time: 15:10)



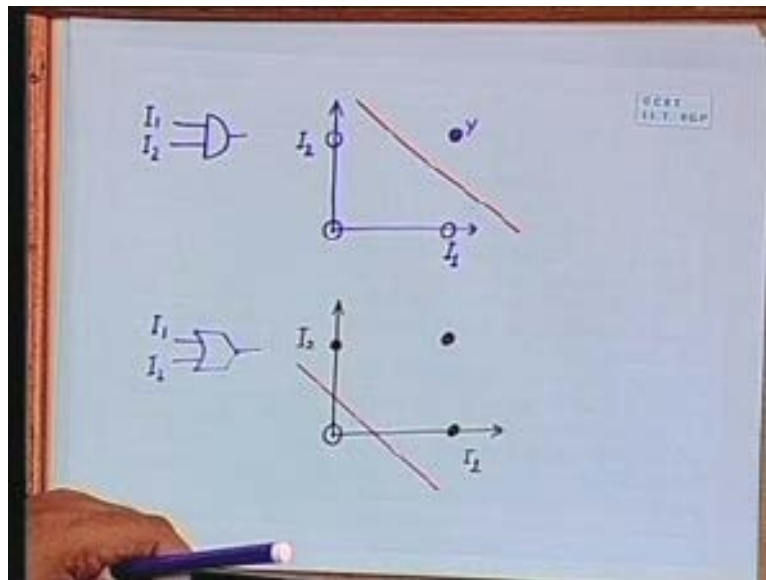
So, it is a threshold function, which says that if the total input is greater than 0.5, then, this is going to be high, otherwise, it is going to be low. So, that means that V_1 - that the total input should be - for this case, should be below 0.5, so that the unit remains at 0, the correct value. Now, how can we do that? What we are going to do is, we are going to reduce the weights on the edges, which connect to the 1 values. So, we will pick up these 2 edges and reduce their weight. What effect is that going to have? The input value will go down, because our input was $\sum_j W_{ji} A_j$. So, A_j is 1 for these 2 and if we decrease W_{ji} , then, the total input is going to go down. But at the same time, if we unilaterally decrease the weights here, then, the total weight balance is going to change.

So, what we are going to do is, we are going to reduce the weights on these and at the same time, increase the weight on this, so that the total weight constitution remains fixed. It is just transferring weights from the ones which we want to reduce to the ones where we want to increase it. If we do that, then, for this case, we will move a little bit closer to the goal and we repeat this case over all the training sets, with the hope that at the end of the training, we will be able to correctly classify the samples and be able to produce the right kinds of outputs. Now, as it turns out, I am going to come into the formal analysis in a moment, but as it turns out, that this kind of a single layer of neural network is able to learn only functions which are linearly separable.

Now, let us understand what is linearly separable; linearly separable functions are ones where you can have a plane in the Euclidean space which separates the positive cases from the negative cases; the yes answers from the no answers. For example, if you look at say, an AND gate; suppose we want to learn the AND function. This has 2 inputs; i_1 and i_2 . If you look at the 2 dimensional plane with this being i_1 and this being i_2 , then, if both are 0, then, we have 0. So, this is a no answer. If 1 of them is 0; if i_1 is 0, i_2 is 1, then also, it is a no answer. If both - if i_2 is 0 and i_1 is 1, then also, it is a no answer. If both are one - that is the only case where we have a yes answer.

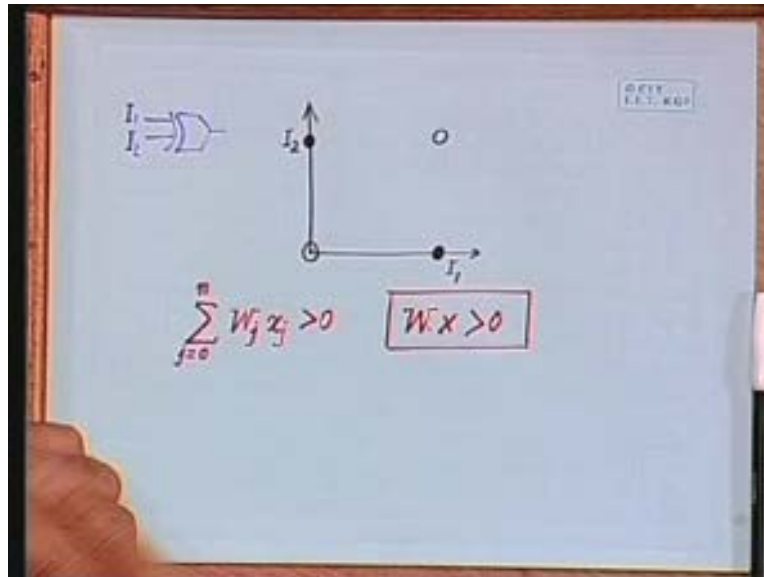
Now, this is linearly separable, because I can have a plane which drives through this. Now, you might be wondering that what does this have to do with the learning here. I will come to that in a moment. Let us look at the OR function. If we have the OR function, then, what we will have is, this will be 0. So, I have I_1 , I_2 here. Again, this will be 0 and these 3 will be 1.

(Refer Slide Time: 19:01)



Again, this is linearly separable, because we can have a plane like this in contrast. Let us look at the XOR function. So, for that, this is going to be, 0 this is 1; this is 1 and this is 0. Now, there is no way that we can drive a plane between yes cases and no cases. So, this is a case which our single layer network will not be able to learn. It will not be able to learn the XOR function.

(Refer Slide Time: 21:59)



Now, what has this got to do with our weight adjustment, etc.? Why can it not learn this why? Can it learn the other ones? So, let us simply look at the single layer network, where, if the total input is positive, we will switch on the unit; if the total input is negative, we will switch off the unit. So, activation will be 1 if the total input is positive, activation will be 0 if the total input is negative. So then, we have this sigma of j equal to 0 to n, where n is the set of inputs $W_j x_j$, where x_j is the input. If this is greater than 0, then, the input switches on; otherwise, it switches off.

Actually, the equation $W \cdot x > 0$, where W is the weight vector and x is the input vector. So when this happens, then only we switch the unit on. Now, if you look at this function, this actually defines the hyper plane; this defines a hyper plane- see this x . For 2 dimensions, this is going to be just a single line; if you have multiple dimensions, it will become a hyper plane, because each of these x can be k dimensional vector. So, this hyper plane is separating out- is acting as a threshold. If that is greater than 0 is on the other side, anything which is less than 0 is on this side.

And so, because that is the decision for switching the neuron on or off, so, that is the plane which separates the positive cases from the negative cases. Our objective is to learn the values of the weights, so that the the weight vector that we construct along with the input vector will actually come to this plane; the weight vector will coincide with this weight vector which separates these 2. That is intuitively the objective that we are trying to do. Let me quickly derive this particular equation for updating the rules, then, we will see some example cases of this learning and its applications also.

First, we define the error. For the error, we are going to use the the root mean square error, RMS, which is the standard error that people wish to minimize between functions. You have studied RMS error? What we are going to do is, we are going to keep this as the error term. So, y minus this whole square where perceptron is, the network is the simple network node that we talked about. It is the simple neural network node that we talked about; it is popularly called perceptron. Now, our objective is to update the weights in such a way that in each step, this error will reduce. What we are going to do is, we are going to do gradient descent.

You remember gradient decent? What does gradient decent do? It has some objective function and we take steps, so that that objective function gradually decreases. Of course, we have the problem of getting stuck in local minima; we have the same problem here also, but let us see how can we do gradient descent to minimize this error function. Every step is going to reduce the error and we want to do this monotonically; that is why gradient descent- we want to monotonically reduce the weights. Remember that we are going to put different training sets and each time, we are going to bring down the error for the (words unclear).

To reduce this error, let us first see what we have as ΔE over ΔW_j . I want to see that what is the change in error with respect to a change in the weight that I receive from j . So, this is given by Err . What is Err ? It is half Err^2 . So, if I do this, then, it is Err times- see this- 2 and half will get cancelled out, because of the differential. This 2 and half will get cancelled out and I will have Err times ΔErr by Δ . I will have Err

times- now, I substitute this out here. Delta by delta W_j of g of y minus i ; think I missed a g here, it should have been this- should be a g here; this is g of g is the activation function of y minus- this is the total input- j equal to 0 to n $W_j x_j$. No, the difference, the error, is in terms of the output that we get.

So, this comes to minus of E_{nn} into g dash in, where in is this total input times x_j and g dash is this derivative. See the other terms; see this, besides x_j , it has other terms also- x_i , the ones which are non j . Now, those terms are going to get eliminated, because this is a partial differential. So, the only term that we will have out here is the 1 which corresponds to X_j and we will also have the derivative of the whole thing. Now, could I make myself clear? No? This minus is getting propagated outside this because this is a constant, it gets eliminated, so, I have this term.

(Refer Slide Time: 31:57)

The whiteboard contains the following text and equations:

$$E = \frac{1}{2} (y - h_w(x))^2 = \frac{1}{2} E_{nn}^2$$

where $h_w(x)$ is the output of the perceptron and y is the true value.

$$\begin{aligned} \frac{\partial E}{\partial W_j} &= E_{nn} \times \frac{\partial E_{nn}}{\partial W_j} \\ &= E_{nn} \times \frac{\partial}{\partial W_j} g\left(y - \sum_{j=0}^n W_j x_j\right) \\ &= -E_{nn} \times g'(in) \times x_j \end{aligned}$$

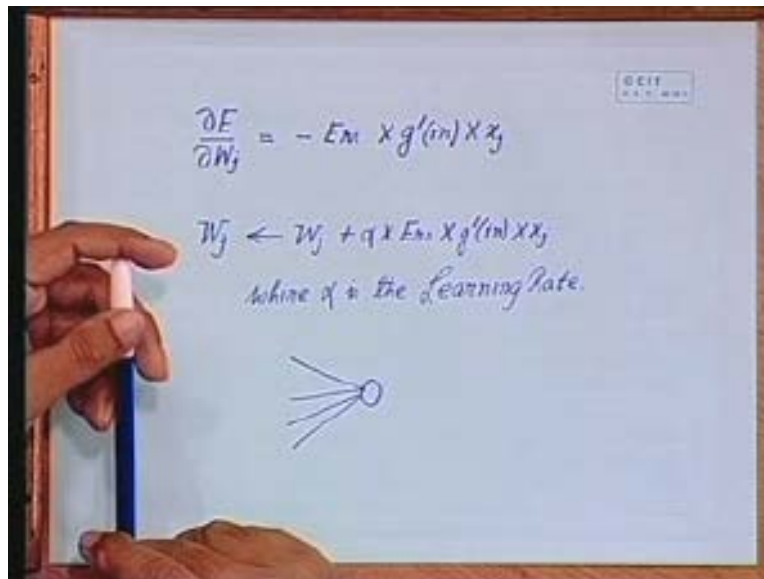
Now, is this clear? How we arrive at this? See, this is a constant, so, it gets eliminated and then this g - because this is a function, it becomes g dash of this whole thing- and then, the partial differential moves inside and when it moves inside, then, everything which is non j gets eliminated and I am just left with x_j . Are you with me? This is the

formula that tells us the weight updation rule. From this, what we will get is- okay, so, let me write down what we have obtained so far. We have obtained ΔE . Yes- (Student speaking). See, this is the- I think this is- wait, yes- (Student speaking). Now, what is the confusion? The output is g of the total input and this is the incorrect input that we are getting, this is the correct input that we should get and this is the incorrect input that we have got, because our weights are not yet tuned.

Yes, I think what we are trying to do here is to minimize the error in the input, because if you are able to bring the total input to the correct value, then, the output will obviously be the correct one. Let me reflect on this a little more and I will clarify it. Maybe in the next lecture, because we are going to revisit good part of this when we look at back propagation learning. So, for the time being, let us say that we have obtained that ΔE by ΔW_j is given by minus of E times g' of in , where in is the total input into the perceptron times x_j .

From this, we set W_j to be W_j plus α into E into g' of in into x_j , where α is the is called the learning rate. See, rather than adding this whole error term into W_j , we are adding only a fraction of it. So, we are not just jumping into the same this thing, because that would amount- is something like you know, quenching, but we do not want to do that; we just have to incrementally tune the weights, so that over all the samples- we arrive at a set of steady state values of the weights.

(Refer Slide Time: 37:08)



Therefore, this alpha is called the learning rate and we just add a fraction of this error into W_j . Is it clear? And we do this for each of the W_j . This was just computing delta E by delta W_j for 1 j and we do it for each of the j s, so that we have updated the weights into of all the links that are feeding into this. Now, I will digress a little bit from this; we will come back to this analysis again when we look at 2 layered networks, where we will study method called back propagation learning, where instead of learning in just 2 layer networks, we will learn in multiple layer networks and the interesting thing will be- what are the internal nodes? What do will they do?

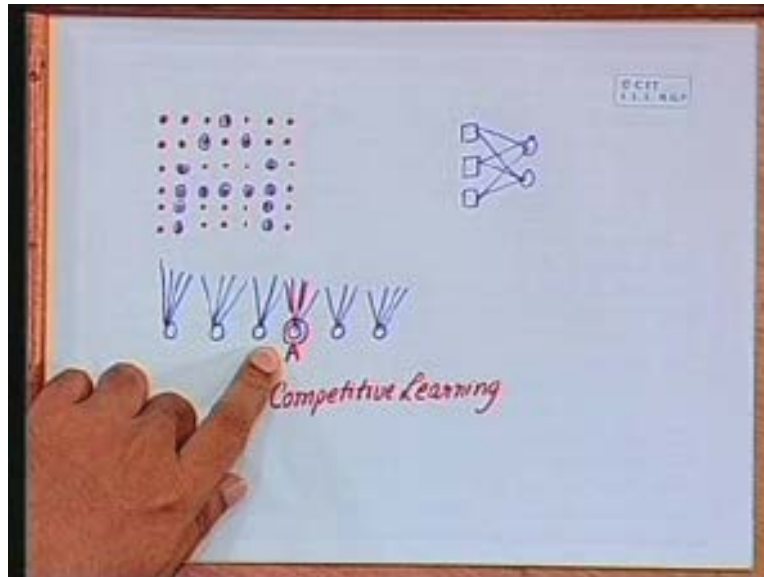
Here, we have just the output layer and the input layer and we are tuning the weights between the output and input layer, so that the output layers come closer to the desired values- the weights become closer to the desired values, so that it is able to give the proper output for all the training examples and others. Now, let us look at slightly different problem- we will look at the problem of recognizing text. Let us say that we are given a matrix of dots. This is the matrix of dots that is given to us and on this matrix, we can have different letters by setting these to 1 and 0. For example, if we set this to 1, this t 1, then, that gives us A. Similarly, you can have B, C, D, whatever.

Now, what we want to do is that if somebody writes A slightly differently- maybe instead of writing it this way, it writes it that instead of lighting this dot, it lights this dot; slightly different, this 1 is off and this 1 is on. We should be able to classify all those cases. I will train it with a set of different A, B, C, D, etc., and then, it should be able to make out a slightly different perturbed A, it should be able to make out the slightly perturbed B and so on, and be able to say that yes, this is still an A, this is still a B and so on. Now, how do we model this into a neural network framework? 1 option is that we create a neural network where these are the inputs; each of these dots is an input which can take a value 0 or one.

And I have a set of output nodes and each of these inputs will be feeding into those outputs. I will be (word unclear) receiving these output nodes, receive inputs from each of these elements; so, I have again that 2 layer kind of complete network that we have. So, it is that complete bi-partite graph that we have here as well. What we are going to do is, we are going to- this is going to have at least twenty 6, could have more also; at least twenty 6. Then, we are going to train this network, so that whenever we have A, 1 of these glows; wherever we have B, some other 1 of these glows; whenever we have C, some third 1 glows, and the others remain off.

Now, 1 way of doing this which was suggested, was doing what is called competitive learning. Competitive learning sets up a competition between these nodes and whichever is the winner is the 1 which will be declared as the value. Which means that whenever we give yes, there should be 1 particular node which should become the winner for all As and 1 particular node should become the winner for all Bs. Which 1 of these will classify the As and which 1 of these will correspond to Bs? We still do not know. So, initially, all weights are random. How the learning will progress initially- all weights are random, so, when I present it with the first A, 1 of these will win. Let us say that this 1 wins for A. Now, what we want is that in future, whenever we present A, even with slight perturbations, this is the 1 which should win.

(Refer Slide Time: 46:55)



So, what we will do is, we will strengthen this so that its activation value will further increase when we present an A. How do we do that? When you have an A, then, there are some when you presented it with an A, then, some of these units were one, which means that there are some of these links which correspond to the ones and some of the links which correspond to the 0s of the input. We will do that weight transferring, so, we will take a fraction of the weights from the 0 ones and transfer that weight and distribute it equally to the ones that we have here. What we are going to have here is that the total weight will always be one, so, for every unit, the total weight of the edges incident on that sum of the weight will be one.

Whenever we redistribute the weights, the total weight will still remain one. But now, we have moved weight away from the 0 inputs to the links corresponding to the 1 input. So, next time, when we give A that activation of this, the total input to this will be more. So, it will stand a larger chance of winning. On the other hand, what do we do with the ones which had loosed the competition? For the ones which loosed the competition, we will do the same, but a much lesser fraction for the winner; the fraction of weight that we transfer will be larger than compared to the losers. For the losers, we will take out weight from

the 1 inputs and transfer it to the 0 inputs, but the fraction of weights that we transfer will be much lesser. Having done this weight adjustment, we then again present it with another sample and repeat the procedure. And the idea is that our expectation is that eventually, these nodes will start classifying some particular letter.

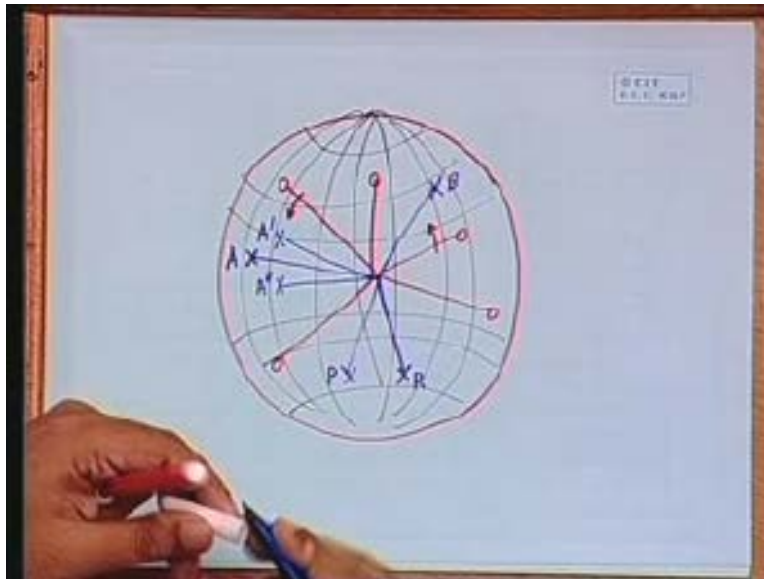
There will be 1 which will always come up for A one, which will always come up for B one, will always come up for C and so on. But just like- this is also gradient descent. Why? Because if you take any particular node, it is being dragged on to something. What is it being dragged on? So, 1 view of looking at this is that each of these is a vector; this is a vector, this is a Boolean vector and it has so many different dimensions. If there are some twenty dots here, then, is a twenty dimensional vector. Let us think of the twenty dimensional hyper plane. If you look at the twenty dimensional hyper plane, then, in that plane, each of these samples is a point, because every vector is a point in that twenty dimensional hyper plane.

I have the A as 1 of the points in this hyper plane, so, just remember that this is just not a circle; it is actually a hyper plane and this is 1 point which corresponds to the A. Then, similarly, we may have another point on the hyper plane that corresponds to B and another plane which corresponds to P. The 1 which corresponds to R is going to be close to P and so on. And where are our vectors here? Each of these vectors- the weight vectors that we have- they also correspond to points in this plane. So, I will have some vector here. Initially, say 1 is here, 1 is here, 1 is here, 1 is here. So, what is happening is that when I present an A, let us say that this fellow wins. So, the weight adjustment rule is moving it towards A and for B- if this 1 wins, this is going to move towards B. And also, the weight adjustment rule is going to take to a much smaller extent the ones which are here to slightly away.

As we were saying, that the ones which are- (Student speaking). Yes, now, why do we move them away? We move them away because of certain scenario. This part is clear? That the weight adjustment is actually taking it closer to this, to the vectors that correspond to the actual thing and when you move it closer, if you give a slight

perturbation of A; if you give a slight perturbation of A, let us say A dash, which is here or even if A double dash, which is here or even if A double dash, which is here, then, it is more likely that this fellow will win once it moves closer to it.

(Refer Slide Time: 50:53)



This vector will not only have learned the A that you presented but also learn As which are close to them. That is a good thing about this. There can be some problematic situations for which we have the other kinds of roots. Suppose we have A here and I have say, B here, and incidentally, it turns out that both of these are pretty close to this one. Now, what is happening is that every time this 1 wins- okay let me not; not this case, forget about this case. I have xB here, sorry, B here and then, I have A here and other ones here. And now, see, the problem is that every time you present A, it is this 1 which is going to win and it is going to move slightly towards this direction.

And every time you present B, it is this only which is winning, because this is absolutely the opposite way. And if this 1 again wins, then, that means that this is going to again be dragged on to this side. So, it is going to oscillate between these 2, whereas there is another vector which is not being used at all. So, whenever you have a losing one,

whenever you present A: this 1 wins, this 1 loses. So, you push this away slightly, then, that is going to have the effect in the long term of moving this slowly around, so that at some point of time, it is going to come pretty close to B. Yes, B is also going to push it away. So, if you have a scenario where you are presenting A and B alternately, then, again, you would have a bit of a problem.

(Refer Slide Time: 53:52)



You have to mix up your training in such a way that it eventually starts classifying. The intuitive idea of moving it away is this: to move away those vectors which were not being used at all, which are not winning on any cases. If we can move them away slightly, then, maybe somewhere down the line, they will move close enough to some 1 else and actually start participating in the classification. Having vectors- having outputs- which are not winning in any case is not useful, right? As you can still imagine, that there will be cases where we will get stuck in local minima and you will have 1 vector which is moving around between 2 of them.

But in many cases, we will be able to do this and if we put in more vectors, more outputs, then, it is more likely that we will- another option is that if you have a particular output

which is not playing a role, you randomize the weights to that, so that it now moves into a entirely different place and maybe starts participating. So, this is 1 paradigm of learning that we have learned today. In the next class, we will be talking about learning algorithm called back propagation learning.