Artificial Intelligence

Prof. P. Dasgupta

Department of Computer Science & Engineering

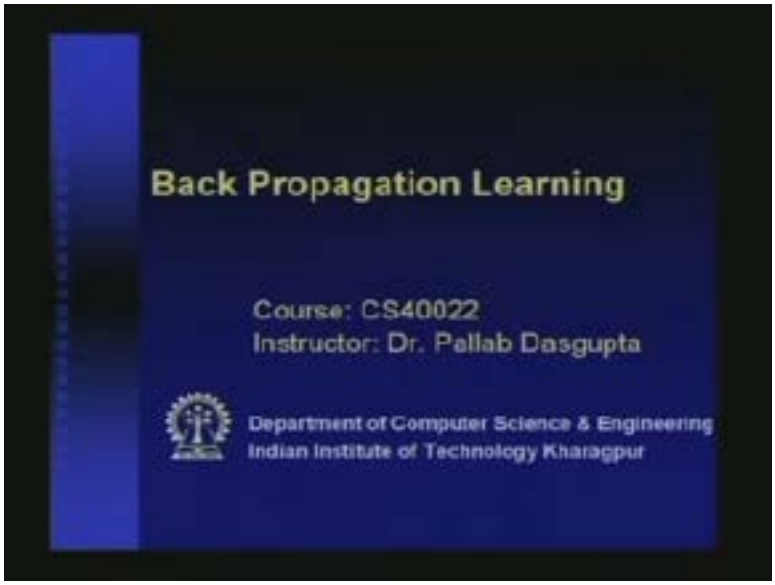
Indian Institute of Technology, Kharagpur

Lecture No - 28

Back Propagation Learning

In the last, we had started off with neural networks and we had seen how, by using very simple processing units called neurons, we attempt to learn different kinds of functions.

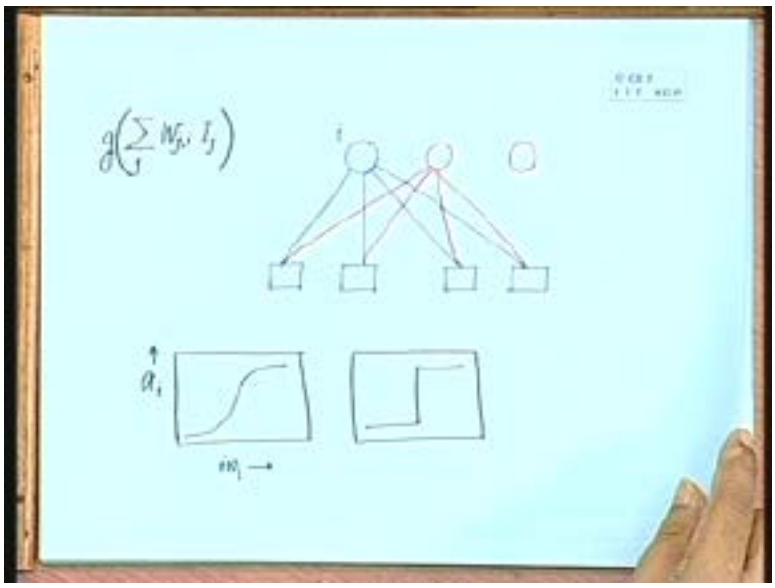
(Refer Slide Time: 01:08)



The model that we looked at in the last lecture was what is called a perceptron and a perceptron is a single layer network, where we have neurons which are simple processing units and we had a set of input units feeding into the neuron. Each neuron was like this and we had actually, a collection of other neurons also and each of these other neurons would also receive inputs from the same set of input lines. This was the simple form of a single layer network that we had looked at in the last class. And then, we wanted to see that how to compute the weight learning function.

Initially, all the weights are randomized and we want to learn the weight learning function, so that after we have learned the weights and we are presented with the inputs, the neuron outputs are the activation values of the neurons, should have the correct output value. For example, 1 way of the these neurons can have different kinds of function that they can compute, of which 1 is where you compute the total input into the unit i as sigma Wji, over all the inputs j times the input that you receive from j. This was defined as the total input.

And then, we define the activation function as some g of this input and that is the output that this neuron is going to have. And we we saw that there can be different kinds of functions for g, of which the 2 most common ones are the sigmoid function, which looks like this, where the input changes gradually or it could be a threshold function, which means that the moment you reach the threshold, it will simply switch on. So, it is off- when you reach the threshold, it switches on. Yes. (Student speaking). There- (Student speaking). These ones- they are the input units, they are the inputs to your neuron.
(Refer Slide Time: 05:12)

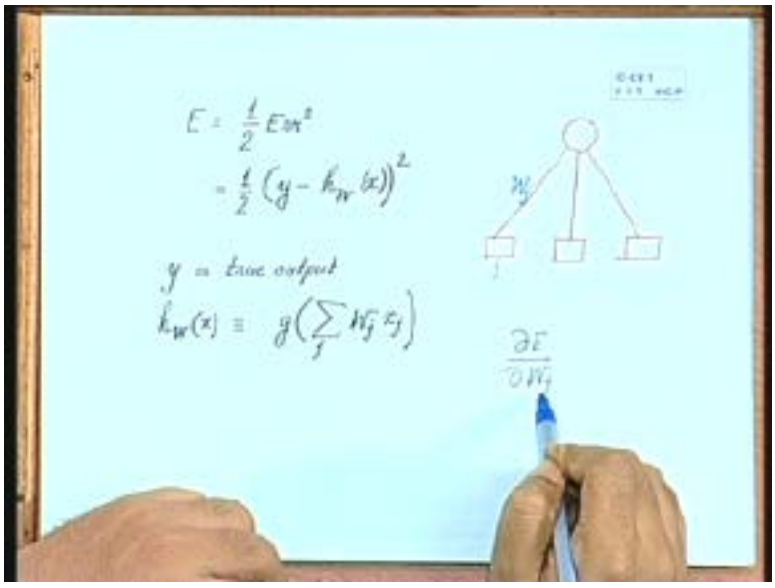


So, a neural network will have a several layers of neurons like this and will have 1 layer at the bottom of input units. In general, multi-layered- I am going to come to multi-layer

network, so, you will have an input layer and then, you can have many layers of neurons. This is a single layer, because just above the input layer, you just have 1 layer of neuron. So, it is called a single layer neural network and these single layer networks are also called- these neurons are called perceptrons. The g function is going to determine that what is the output of the neuron, what is the activation value of the neuron a, given the input that it is received. And input of the unit i is defined as follow sigma over j Wji times ij.

The first thing that we want to know is that how do we adjust these weights, so that given any function, our neural network is going to adjust the weights in such a way that g of the input is going to have the same value as the output, of the correct output of the network? We did a single step of error propagation. We said that okay, let us define the error. The error E is given as the square of the- half of the square of the error, so, this is the RMS error that we want to compute- root mean square error. What is the the error of the output? We are looking at- let me draw the neuron here as well, so, we are looking at the error that we are going to have in the output of this neuron. In general, the total output is defined as the collective output of all these neurons taken together, so it is a vector actually.

(Refer Slide Time: 10:03)



When we talk about half of Err square, it is that vector that we are talking about and what is that error? That error is nothing but y minus- okay, where, what is y? y is the true output, the correct output. Note that this is a vector, this is a vector, this is a correct output vector and its difference with HWx, and what is HWx? And what is x? x is the input and HWx is the total input function that it receives. This is the total g of the total input function that it receives. So, we can write this as g over. Again, see, we are dealing with vectors here. Now, let us look at how we compute the error in terms of the change in the weight values.

What we actually need to see is how can we bring down the error by adjusting the weights. How do we do that? If this is the error, then, what we are actually interested in computing is the change in error with the change in Wj and what is Wj? Wj is the weight that we are feeding into the neuron I. If we take a single neuron, then, Wj means it is the weight of j to the output unit. So, let me recap. What we are doing here? We have the neuron here; we want to compute that how do we adjust these weights, so that the error that we had on the output of this neuron is going to come down. So, if E is the error at the output of this neuron and W1, W2, through Wj, etc., are the weights on these edges, then, we are interested in finding out what is delta E by delta Wj. That is going to give me the change in the error with the change in this weight.

Likewise, we will do it for each of the weights and that is going to give us the weight updating function. So, if we use these formulae, then, let us see how we compute delta E by delta Wj. If we look at delta E by delta Wj, then, this is Err into- okay, let me first write down what was our E. It is delta by delta Wj of half of Err square, where Err was our y minus, so, our Err was y minus g of sigma over j Wj and xj is the input here; it is the value of the input that is coming. So, in Booleans, it will be 0 or 1, but in general for a neural network, it need not be 0 or 1. It can be anything; it can be any value. So, this is Wj times xj or Ij, whatever we might want to call it.

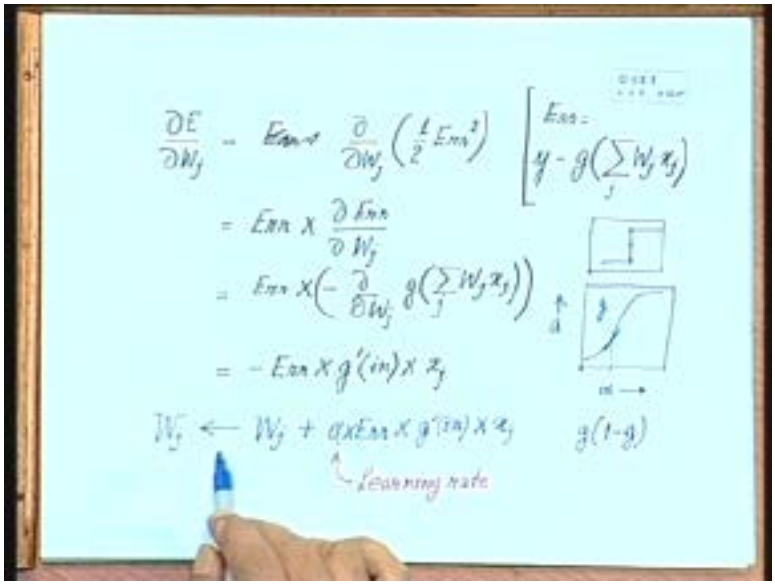
This is Err. So, if we now apply this partial differential on this, so then, we will- yes? (Student speaking). No, now, when we are looking at a single weight updation, so, xj is

just a scalar which is the value of the input j; xj is the scalar equal to the value. (Student speaking). Yes, this is 1 of the inputs for the neuron. If you take a given neuron, if you just take 1 neuron, then, these are the inputs that are feeding into that neuron. xj is the value of the jth input, Wj is the weight of the link connecting the jth input to the output unit. We will do the same for each of the neurons that we have in the upper layer, so, we are doing it only for 1 of them, because you see, when we are talking about the error, this error gets decomposed into the errors of the individual dimensions of the vector.

So, it is the error of this is 1 dimension; the error of the other neuron here will be another dimension. So, now, we are trying to see that how do we reduce the error in 1 dimension and we will do the same for all the dimension. So, this is going to be Err times delta Err by- right? Now, we apply this; if we replace Err by this term, then, because y is a constant, what is y? It is a correct output, so, that is a constant; that is a value that is given to us, so, the differential will eliminate that. And we are going to have delta of this, so, we are going to have Err times- minus of g of- right? And now, when we use the differential on the g, so, we are going to have- this minus comes outside, we have Err times g dash of in times xj; only xj is going to come out, because the rest, when we partially differentiate with respect to Wj- the rest will get eliminated.

So, we will have only the Wjxj term will be significant, and when we differentiate Wjxj with respect to Wj, we get Xj. So, this is the term that we have for the change in the error with respect to the change in the weight and that also gives us a way of learning the neural network. What we are going to do is, we are going to use gradient descent, so, we will just simply apply this rule to update our weights.

(Refer Slide Time: 20:12)



For example, we will use Wj is- can you see this color? So, Wj will be updated as Wj plus some alpha times Err times g dash in times xj. So, this is the change in error with respect to this, so, if we add this term, then, the error is going to decrease. If we add this term to Wj, then, the error will decrease, because we had a negative here. And this alpha here is called the learning rate, so, it is a fraction of the error that we adjust against the weights. See, this term- the change in error with respect to the change in weight was negative. So, if you subtract this from Wj, then, your error will go down. If you subtract the negative term, it is equivalent to adding this thing, right?

So that is going to ta- if you use this updation rule then, the delta E by del, that the delta E by delta j is this. If you add this term, the new error will be lesser, so, if you compute the total output with this Wj, it will come closer to the value y that we intended it to have. Essentially, what it means is that we just compute the error and then, use a fraction of that error to, we adjust the weights of the input to the unit. Now, how do we use this g dash in? For threshold functions, g dash in is the differential of the input output function. Recall that we had different kinds of functions like the sigmoid, where this was in this direction. We have the input and in this direction, we have the activation of the unit. This

is our function g, so, if we compute the derivative of this at any given input point, so, it is the slope at that point, right?

So, for the sigmoid function, I think it is just g times 1 minus g. Just let me check out; I think it is- yes, it is g times 1 minus g, right, of the input. And for the threshold function, recall that it is a step function, so, the derivative does not exist at this point. So, what was done was, in the initial algorithms for updation, this term was simply dropped. It means that you compute the error right times the input, and then, take a fraction of that and adjust the weight accordingly. Is this part clear to you? This learning of single layered neural network? Yes. (Student speaking). Where can you have a 0 slope here, for example? (Student speaking). In the threshold, see, because this is not a continuous function, so, that differential does not exist. It was a design choice to simply drop this term and it work quite well, actually.

So, then, people started examining the performance of this single layer networks vice-versa other models of learning, like we had studied previously, the learning of decision trees. And it was found that in some cases, perceptrons would perform much better. For example, if you look at the mAjority function- what does the mAjority function give you? If you have k inputs; if you have unit and you have k input lines and if a mAjority of them are 1, then, this output will be 1. Otherwise, this output will be 0. That is the mAjority function. Now, imagine trying to learn a mAjority function using a decision tree. In a decision tree, you will have these variables x1, x2 through some xk. Now, if you just first take, examine the value of x1, suppose it is 1. Can you decide which is the mAjority? You cannot; then, you take x2; you find it is 0, you still do not know what is the mAjority.

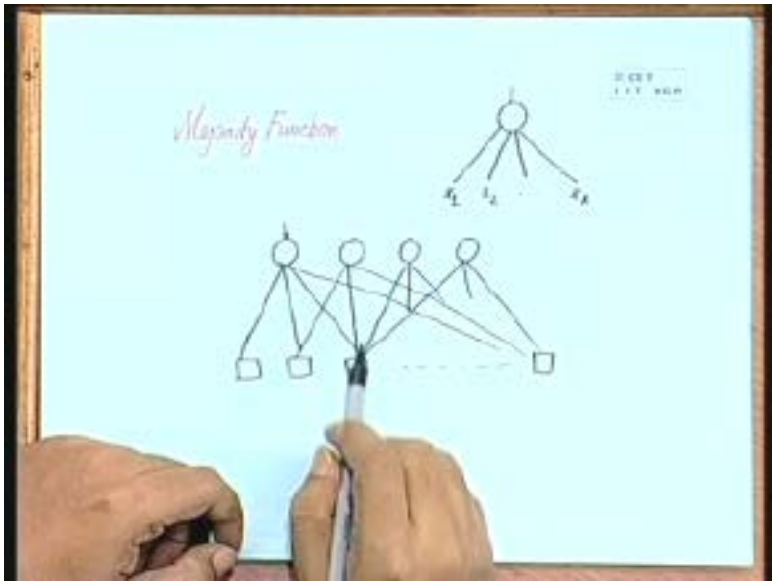
So, in a decision tree, if you just think that if in a decision tree, you have to learn the mAjority function, then, you may have in many cases, to go right up to the maximum depth. In any case, without examining half of the variables, he will never be able to decide, but even if you examine all the half of half of the variables and all of them are 1 or all of them are 0; only then, you can reach a decision. So, it is only in 2 cases that you

will be able to reach a decision, be examining half of the variable, right? But in order to reach a very decision for all cases, you will definitely have to look at all the k variable. So, if you learn try to learn the mAjority function using decision trees, it does not worked too well.

On the other hand, perceptrons can learn the mAjority function pretty well, because remember that they are good at learning a linearly separable functions. Why? Linearly separable functions are better learned by perceptrons, because we are individually trying to tailor each of the inputs independently, each of the outputs independently. So, suppose we have 4 neurons and several input units and obviously, all of these are feeding into all of those. What we were doing so far is trying to tailor these errors of these individuals, so that the error is minimized. Now, if it so happens that you tailor this error, then, for some other example, the error is going to become larger. Then, we have a problem, right?

But in case of linearly separable functions where you have all the yes cases on 1 side of a hyper plane and all the no cases on the other side of a hyper plane; if you can separate out the yes cases and the no cases from this hyper plane, then, individually, if you move the dimensions towards these plane, then, your error is going to progressively go down, gradient descent will succeed and we will have to learn that function and the mAjority function is of these cases.

(Refer Slide Time: 26:53)



Because if you look at the mAjority values, they are all going to be on 1 side and the minorities are going to be on the other side. So, you can always fit a hyper plane between the yes cases and the no cases, so, that is why it was found that for such kinds of functions which are linearly separable and for which decision trees are difficult to learn, perceptrons were very useful. But the moment when we came up with cases where the variable starts in the- the outputs are not independent in that sense and start influencing each other, then, perceptrons will fare pretty badly. For example, if you look at that restaurant example that we had seen previously; for that, what was found was that decision trees can learn that pretty easily and perceptrons fare really bad.

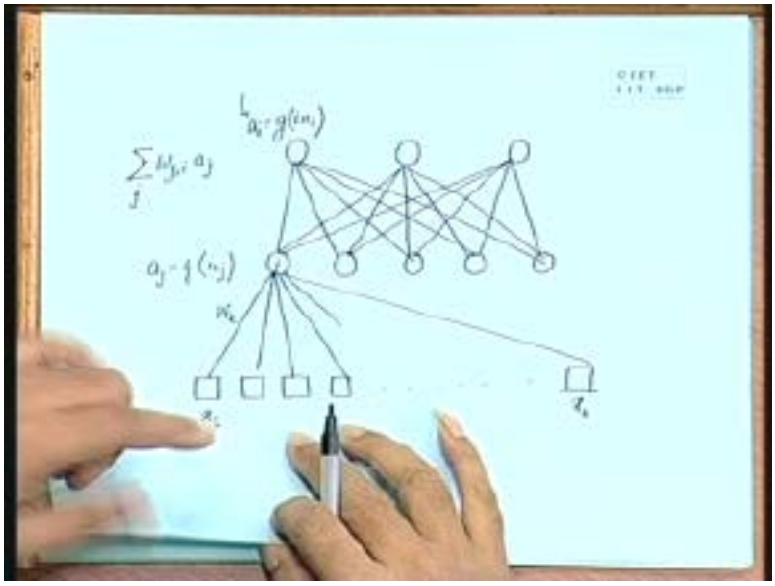
People started arguing that what do we lack in the perceptron. What we lack here is as follows: that if we look at a layer here, so, if these outputs are not independent of each other, then, there is no way of having a common variable shared between these 2 which we could tune to bring the 2 into c, right? So, what people thought of is, let us have a multi-layer network. So, we will have, say, something like this. So, we have some layers; an output layer, then, we have some second layer and finally we have the input layer. So, this is a 2 layer network and we can generalize this to many layers between 2 layers; we will have a complete bi-partite connection, which means that this is going to be

connected to all of these. Likewise, this is going to be connected to all of these, and similarly for this, and we will have a similar connection between this layer and these, so each of these units connected to- right?

Now, see, if you examine a weight here, suppose, let us call this Wk. If you examine this weight by tuning this weight, I am able to change the activations of all of these output units, something which was not happening previously. See, we cannot tune the inputs here; it was only the inputs which was feeding into everything, but inputs are given, there is nothing to tune. And these weighs are direct connections between inputs and the output layers. So, if we want to learn something which is in a sense higher order kind of relationship, then, we have here a mechanism of tuning these weights, which is going to affect the activations of all the output units. So, the question was, that can we have a learning algorithm which will able not only to tune these weights but also these weights, so that we are now able to learn functions which are not linearly separable.

And then, people say that okay, let us start from the top and try to see whether we can propagate the error backwards into this. See, what do we lack here? In order to update the weights that are feeding into this neuron, we need to know what is the error here, but error is given only at the output, because we know the correct output; we know the correct output and we can see what output we have got through our network, but we do not know what should be the correct output of this to get the correct output in the output layer. The training set that is given to us only gives us the final output; it does not tell us what should be the correct activation values of this hidden layer, so, this layer is called the hidden layer. We do not- what is the correct output of this?

(Refer Slide Time: 34:49)



So then, the idea was that let us do the following thing: let us take the errors of these units and make a proportionate distribution of that error back into the hidden layer. That is why we have back propagation learning. We will compute the errors in the output layer and back propagate that error into the hidden layers, so that we have some heuristic for the errors that are there in the hidden layer. We do not know actual errors on the hidden layer; we are guessing that if this fellow is feeding into all of those, then, whatever errors that these fellows have, some of that blame goes to this 1. We take a fraction of the errors from these propagated backwards into the hidden layer and then, once you have the errors or the guesses of the errors that these hidden layers, then, you have the updation rules for these exactly as before.

Now, let us go though this analysis quickly, so, at the output layer- this is a just a 2 layer network that we have talked of- we can have a multiple layer network where we started the output layer, and then, progressively, start propagating the error backwards, layer after layer, until we reach the input layer, so, that is the multi-layer learning algorithm that we are going to formalize now. So, if we look at Wji, which is the weight from the- okay, let us follow this nomenclature that this is the output i, output unit i, and this is- let

us say, the hidden unit j and this is some input k. So, Wji is this weight and Wkj will be this weight, so, this is 1 segment of this whole network that we have taken; 1 path, and obviously, we will sum over the js and sum over the ks when we do the analysis.

So, Wji is simply given as Wji plus alpha, which is the learning rate plus a times Aj, where Aj is the activation of this unit. How do we compute the activation of this unit by feed forward, when we present this network with the training sample. We are feeding it with inputs x1 through xk, so, then, we compute the activation of this as g of the total input that this jth unit receives, so, that is the activation Aj. Similarly, when we have the activations of all of these, then, we can compute the activation of Ai as g of in i, where in i is sigma over Wji A of j and summation over j. That is how we compute the activations of all of these, given 1 training sample. After doing that, we will back propagate the errors, because the training sample will also have the correct output values, which we will compare with the Ai.

Then, back propagate the errors, adjust the weights and then, again, present it with the second training sample and continue doing this, until we have run all the training samples several number of times and we hope that the neural network has learned the function. At that point of time, we seen the weights and thereafter, evaluate the performance of the network on the training samples and on new samples of the data. That is what the whole learning is going to do. The updation rule for these weights between the output layer and the hidden layer is Wji, is Wji plus alpha times Aj times delta i, where delta i is that function that we had seen; it is basically Err i times g dash ini. Now, recall that the overall this thing was the learning rule between the, from the output layer was this, which we had already derived, so this term is the delta.

(Refer Slide Time: 36:16)

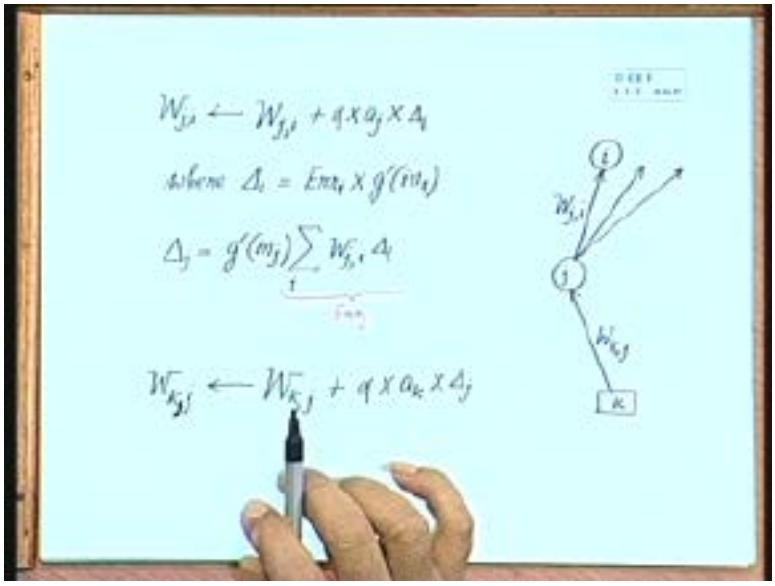
Just using this term for the delta, it is just a short form, so, this is that Erri times g dash ini. So, this is the rule that we will use for updating this 1; this part is very simple, it is exactly as before. Now comes the question of how do we compute the errors in these

units, so, that is why a back propagation will come. We will define delta j as g dash inj times sigma over i Wji delta i; now, I have not yet derived this, I am just stating this. We will derive this later, but intuitively, let us see that what is this error term for delta j that we are having here. This is g dash of inj, just like we have g dash of ini, and then, instead of Erri, this is the error that we are proportioning back into the hidden layer. And what is this term? This is sigma over all i Wji times delta i. What is sigma over this? It means that this j unit is feeding into other neurons as well in the output layer.

You take the errors of all of those, right, take proportionate with the weights of the links connecting to those, so, it is sigma over i, mind you. It is the sigma over all the output layer units and that fraction is what we are going to a proportion back, as the error of this. So, that is going to be- this term is going to be our Err j term, that we obtain by back propagation learning. Now, this term- how we arrive at? Exactly. This term I will formally derive later, but as of now, you can appreciate that this is something which is the errors of all the output layer units biased by the weights of the connections from this hidden layer unit to those.

So, it is fair enough that it is proportionate to- this is what the error coming back should be, and we are- the proportionate function is this g dash inj; we will derive this later. But now, if we have this as the error here, then, the remaining part of the updation rule- which means, how do we update Wkj? Sorry, k comma j; this is going to be exactly as before, so, we are going to be have Wkj plus that alpha into Ak into delta j, where Ak is the input. If you have multi-layered networks, then, Ak is the activation of the next layer of units; otherwise, in case of this, Ak is the same as xk. It is the input, right? As you can see, these 2 are absolutely similar, so, the whole trick is in computing this 1 in a proportioning the error at the output by back propagating it backwards into the hidden layer.

(Refer Slide Time: 40:10)



We will now derive the formal basis for this; the other part of the updation is as before and it has the same derivation as for just the pair of layers. Is that clear? Now, let us get into the derivation of this 1. Let us see what we want to actually derive; we want to actually derive the change in error with the change in Wkj, so, what we are interested in is how does changing this weight affect the change in the error at the output layer? Delta E is the error in the output, change in error in the output layer, and we are interested in finding out how does these links- weights of these links- affect those.

Now, in case of just 2 layers, this error would get split up into the individual output, because the link is only connecting 1 input with 1 output, but in this case, this link is actually affecting the weights of all of these links in an indirect way. So, each of these links are dependent on this; so, we cannot separate out the is as we were doing in the case of 2 links. So, that means this is going to look like minus of sigma. What is our error? Let me bring back our error term; just a minute. Yes, this was our error.

(Refer Slide Time: 42:15)

Let me rewrite- it is half of y minus Ai square, because this HWx is actually the Ai in our case. Now, this is the activation of the output layer unit. Now, if we differentiate this with respect to Wkj, if we differentiate this term with respect to Wkj, this 1, then, we get minus over sigma over I yi minus Ai, because see, this is for just 1 unit. If you take it over all yis, which means all the output layer units, then, you get yi minus Ai times delta of Ai by delta of Wkj. It simply differentiating that error term with respect to Wkj, but because Wkj affects all the activations of the output layers, so, we cannot eliminate any of these; they are all there with the sigma. So then, I can simplify this as sigma of this thing yi minus ai delta over g is ai is g over ini g of ini times delta Wkj.

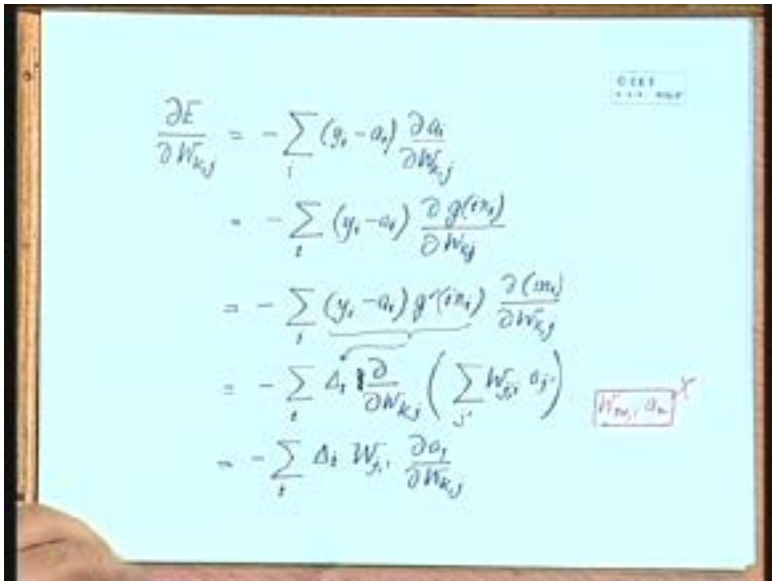
Now, what do we have here? This will become g dash of ini this 1, and delta over ini. Now, what is ini? What is i, first of all? I is the output layer unit. So, ini is sigma over Wji sigma over j Wji times Aj. There is the total input, so, we get- okay, we write this as minus over sigma over I. This term is our delta I, because this is our Err i, Err i times g dash ini, and remember that we had defined delta i as Err times g dash of in, so, this is delta i. And then, when we differentiate this, we will have- wait, we will have- let me not jump steps; this is going to be sigma over j Wji Aj. This is clear? This is just rewriting ini as W sigma over this and then, taking this term and calling it delta i, that is all.

When we propagate this in, what are we going to have? We will simply have the Wji term coming out, because we are differentiating with respect to Wkj. I am not yet done.
So, it is Wji times delta over Aj by delta over Wkj. Now, Wji is there because you have Wji times Aj; Wji is a constant, Wji is a constant because you are differentiating with respect to Wkj. If you differentiate Wji with respect to Wkj, you will just have Wji, because that is a constant. See, this is this times this, so, it is the coefficient; the ones which do not have j will there sum over this.

So, those terms will get- they are just individual constant terms, so, they will get eliminated by the differential. Only those terms which has Aj involved will remain and

the coefficients of those will come out. The only term which has Aj here is Wji Aj. Say, Wmi, WAmi- if you look at a term in this summation; in this summation, if you have a term Wmi Am- see, this is totally constant with respect to Wkj. So, this whole thing is going to get eliminated because of the differential, whereas when you have this 1, well, Aj is not independent of Wkj. Therefore, this coefficient will come out and you will have the differential propagating into Aj. This is clear? (Student speaking). Yes. (Student speaking). Wji Aj- this is the whole ini- (Student speaking). In which term? (Student speaking). Okay, if- just do not treat this j as the j here, okay. It is my mistake; just, instead of this j, call it something else- call it j dash.

(Refer Slide Time: 49:33)



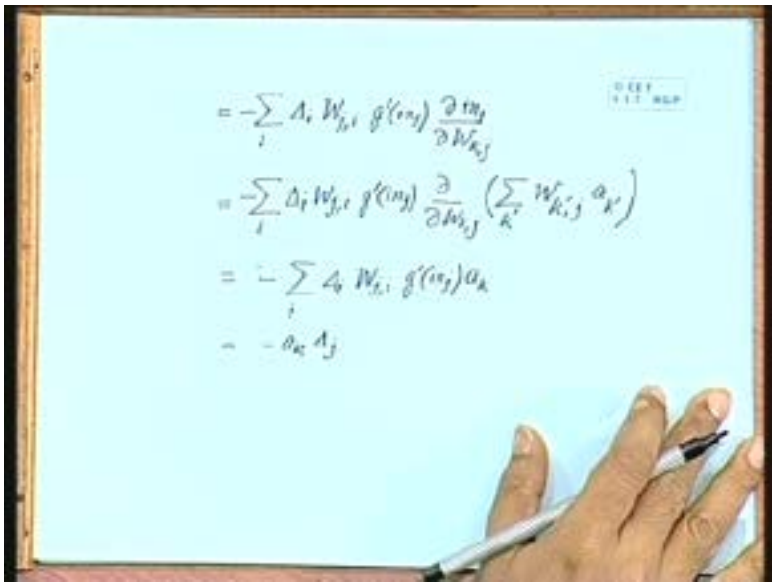
Now, this j that we are talking about here- that is also there among this set; that is the fellow, which, coming up- clear? I use the same variable here, so, that is probably confusing. Now, is it clear? So, this we have derived. Then, so, let me have it; here, I have this. Then, this is equal to sigma over i, delta i Wji g dash inj times delta inj by delta of Wkj. How do we get this? Because our Aj is nothing but sigma; sorry, it is just g of inj, right? Aj is g of inj. So, if we apply the differential on that, we will have g dash inj coming out and we will have delta of inj by delta. Then, again expand out this term. What

is this inj? Okay, let me write it down. Sigma over i delta i Wji g dash inj delta over delta by Wkj of sigma over.

Okay, this time, let me write k dash- Wk dash j Ak dash, again by the same- this thing out of these, only the terms which have k in it; they are going to come out, so, we will have this as minus. What did I do with the minus? There was a minus here, I forgot. So, we will have minus over sigma over i delta i Wji g dash inj times Ak. That is the only term which going to come out, because Wkj Ak will be there in this set and because we are differentiating with respect to Wkj, we will just have Ak.

And we are referring to as Ak times delta j. Now, is this what we had hope to get? So, this is exactly what we are doing here. See, we are adjusting Wkj as Wkj plus alpha times Ak time delta j. This explains the updation rule for the other layers, except the updation rule for the weights feeding into their output layer. Quickly- to conclude, what is the algorithm? The algorithm is, we start with the neural network- say, single layer, multi-layer, whatever. We start by presenting it with a training sample; we present the training sample, compute the updation the activation values of all the units until we compute the activation values of the output layer units.

(Refer Slide Time: 52:48)



Then, compare the outputs that we have received with the training sample outputs to compute the error. Having computed the error, our objective is to re-adjust the weights so that the error decreases, so that, with the change (unclear words) the new activation values will be closer to the real values, to the correct values. Having adjusted the weights for 1 training sample, we will again resume with another training sample and again readjust the weights every time. We use a proportionality factor called alpha which is the learning rate, to decide on how much we are going to adjust. So, you can learn fast; you can learn slow. Slow learning is better learning in general; because you are doing gradient descent, more likely to come down into the global minimum.

And then, you continue with this learning for the different training samples until ideally you reach a state where the weights are not changing too much. So, if the weights are not changing too much, which means that you have converged on the correct set of weights and then, you freeze the weights and you say that I have learned their function. And in order to do this error computation, we have studied the back propagation learning which computes the error at the output layers, and then, a proportions that error back into the hidden layers, and then, once we have the errors for the hidden layers, we use the 2 layer learning algorithm to readjust the weights, as we have seen by the weight adjustment formula.

It has been found that more than multi-layer neural networks perform better for cases where perceptrons were not working well. For example, with a 2 layer network, the restaurant example that we saw was learnt to a much better extent; the decision tree was still better, but only marginal. It is still very not well intuitively understood, that what happens if you have more than 2 layers. I mean the claim is that you will learn better and better functions, but there is no formal way to establish that as of now, but still, we are looking at different kinds of architectures for neural network and see how you can learn. So, in the best interest of time, we will close this chapter here, but there are actually lots of published papers on learning different kinds of functions using neural network and other kinds of models.

So, if you are interested, you can look into different kinds of- and there is a specialized course on machine learning, also offered by our department.