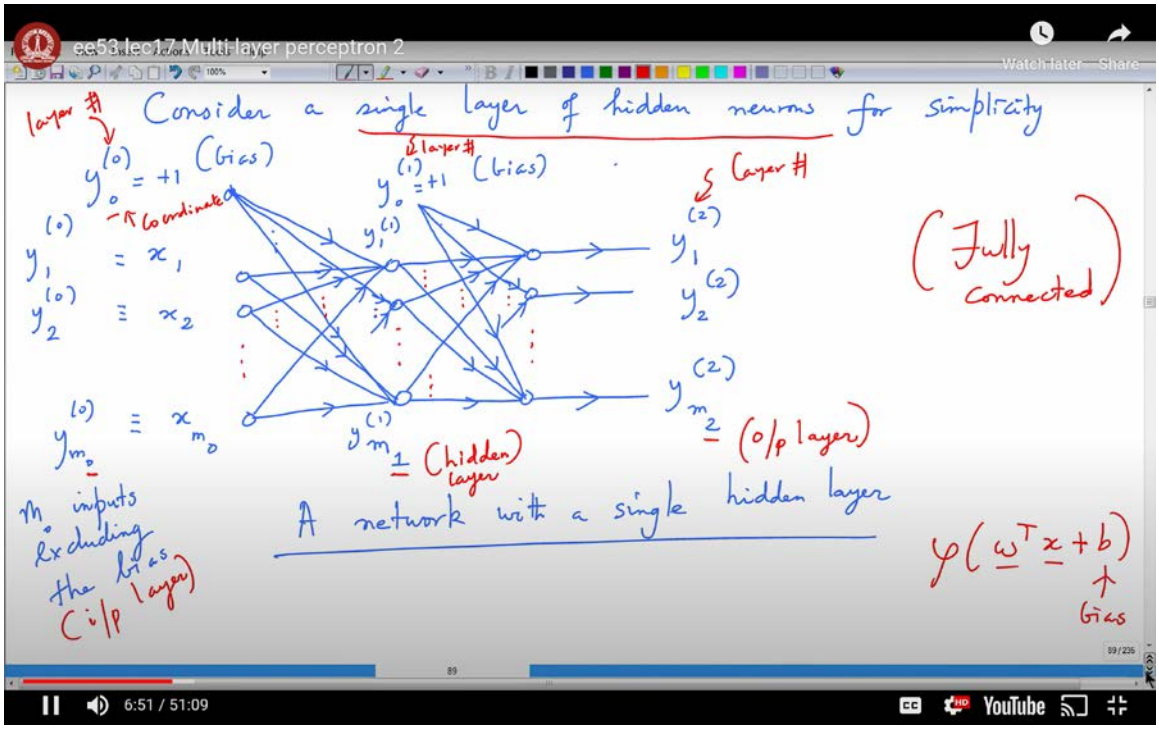
Neural Networks for Signal Processing-I
Prof. Shayan Srinivasa Garani
Department of Electronic System Engineering
Indian Institute of Science, Bengaluru

Lecture – 17
Multi-Layer Perceptron 2

Let's get started with this module. We are going to delve into the multilayer perceptron. So far, we have qualitatively examined the pros and cons of various modes through which the learning algorithm can be explored. Now, we will dive into the mathematical details of the multilayer perceptron, focusing on the derivation of the so-called backpropagation algorithm. This algorithm, initially conceived by Paul Verbos, has found significant applications in modern deep learning techniques and serves as the backbone of neural network architectures based on the feed-forward criterion.

Let's begin by formulating this. We previously discussed the error setup, providing you with a general idea of how to establish the error criterion. Now, we will explore this in greater detail.

(Refer Slide Time: 06:51)



Consider a single layer of hidden neurons. As mentioned earlier, a neural network consists of an input layer, several hidden layers of neurons, and an output layer. For the purposes of derivation, we will focus on a single layer of hidden neurons to simplify the explanation. From this, we can generalize the process for multiple layers.

First, we will discuss the architecture, define all variables, and then derive the algorithm. At the input, we have neurons, which are represented as circles. These circles do not depict actual neurons with full functionality; rather, they simply represent inputs. Excluding the bias, we have M_0 inputs.

We previously discussed the basic model of a neuron, which involves all inputs being processed through a weight matrix. Specifically, we take $W^T X$, add a bias term, and apply a non-linear function φ. Essentially, given some input vectors, we project them with the weights, add the bias, and then apply the non-linear function. This is the core mechanism here: all inputs are connected to the hidden neurons. For our derivation, let us assume that the hidden layer has m_1 neurons.

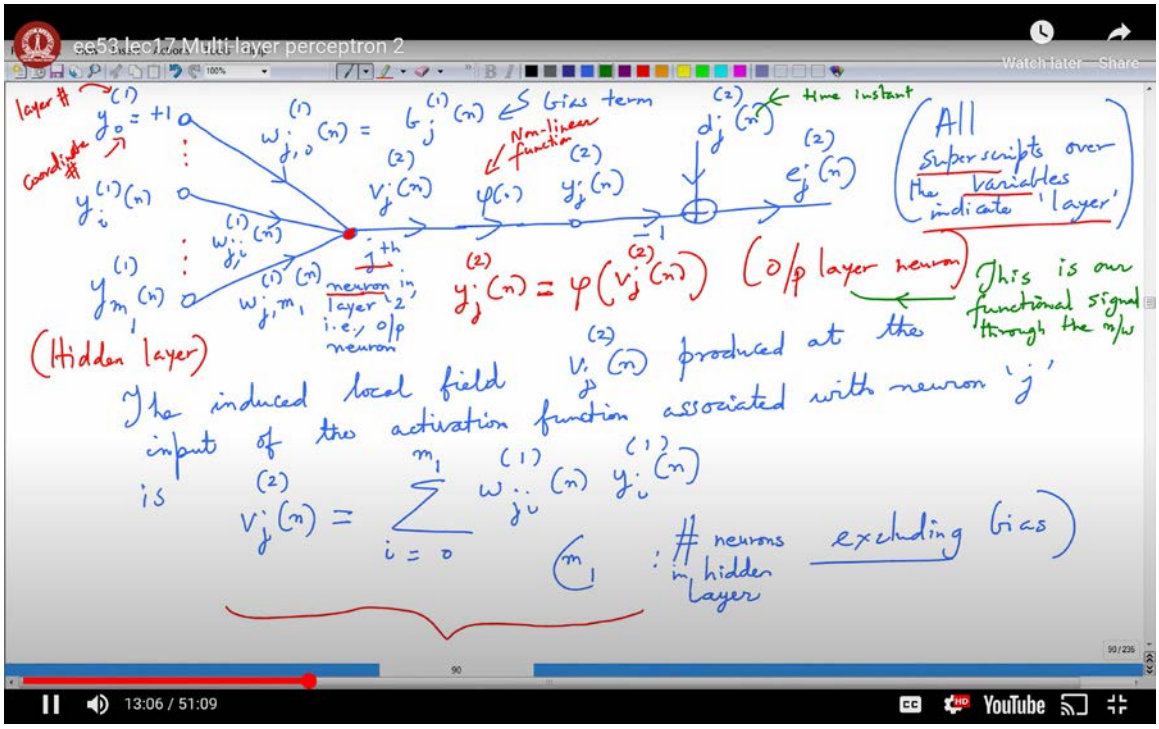
Again, our convention is very important here, which I will highlight using red. The zeroth layer, denoted as layer 0, is our input layer. The first layer, denoted as layer 1, is our hidden layer. The second layer, denoted as layer 2, is our output layer. Now, we have weights that connect the inputs to the nodes of the network in the hidden layer. Additionally, there are synaptic weights connecting the hidden layer to the output layer, maintaining a fully connected network. This means every processing element is connected to every other processing element.

We introduce bias to incorporate the function $\varphi(W^T x + b)$. To include this bias, we introduce it in both the input and hidden layers. We denote the bias in the input layer as $y_0^{(0)}$ and in the hidden layer as $y_0^{(1)}$, where the number in the parentheses indicates the layer. This notation is crucial for our convention, and the subscript '0' simply refers to the coordinate.

These biases are connected to the processing elements, and this structure continues throughout the network. Now, let us delve into the details of the algorithm. You might

wonder why there is no bias at the outputs. The reason is that the output layer merely produces functional signals, so bias is not needed. Therefore, the output layer contains m_2 neurons. In the hidden layer, we have m_1 neurons, excluding the bias, so we refer to it as $m_1 + 1$ nodes. Similarly, the input layer has m_0 inputs, excluding the bias.

(Refer Slide Time: 13:09)



This may seem like a minor detail, but it is crucial for correctly formulating the equations. Let's approach this systematically. The slide might appear cluttered at first glance, but I have noted down all the variables, which will help us go through each step methodically. Whether you find the slide messy or visually appealing depends on your perspective. If you enjoy equations, you might find it intriguing; if not, it might seem overwhelming. However, from a conceptual standpoint, it's not overly complex.

We can easily and quickly understand this process. In the hidden layer, the inputs are essentially the functional signals, denoted as $y_0^{(1)}$, which includes a bias term of +1. Remember, our superscripts indicate the layer number, while the subscripts denote the coordinate number.

These functional signals are connected through synaptic weights. Let's consider a neuron in the output layer, specifically the j-th neuron. This j-th neuron is part of layer 2, the output layer, where we compute W^T Y. You might want to use n to denote the time step for these signals, but note that the bias remains fixed at +1 and does not change over time.

To make this clear, I'll shade the j-th neuron in red. This neuron is connected to the 0th node in the hidden layer via the weight W_{j0}. Similarly, the connection between the j-th neuron and the i-th functional signal is given by W_{ji}. Recall that the superscript indicates the layer number, and the time instant is within the braces. The subscript j, i shows that the j-th neuron in the output layer is connected to the i-th neuron in the previous layer. The j-th neuron in layer 2 is also connected to m_1 neurons in the hidden layer, with the weight W_{j,m_1}.

We formulate the signal $v_j(n)$, which is the summation of y_0 times the corresponding synaptic weight. This involves multiplying $y_0^{(1)}$ by $W_{j,0}^{(1)}$, $y_1^{(1)}$ by W_{j1}, and so on, until $y_{m_1}^{(1)}$ is multiplied by W_{j,m_1}. This results in a linear summation, as represented by the equation.

Here, W_j represents the local receptive field. This summation is passed through a non-linear function φ, producing $y_j(n)$ for layer 2. This $y_j(n)$ is the functional signal, or the response of the network, indicated in green.

However, a careful reader or student will realize that this explanation covers just a single hidden layer. To generalize, you can imagine cascading these functional signals through multiple layers. For simplicity and better understanding, we are focusing on one hidden layer.

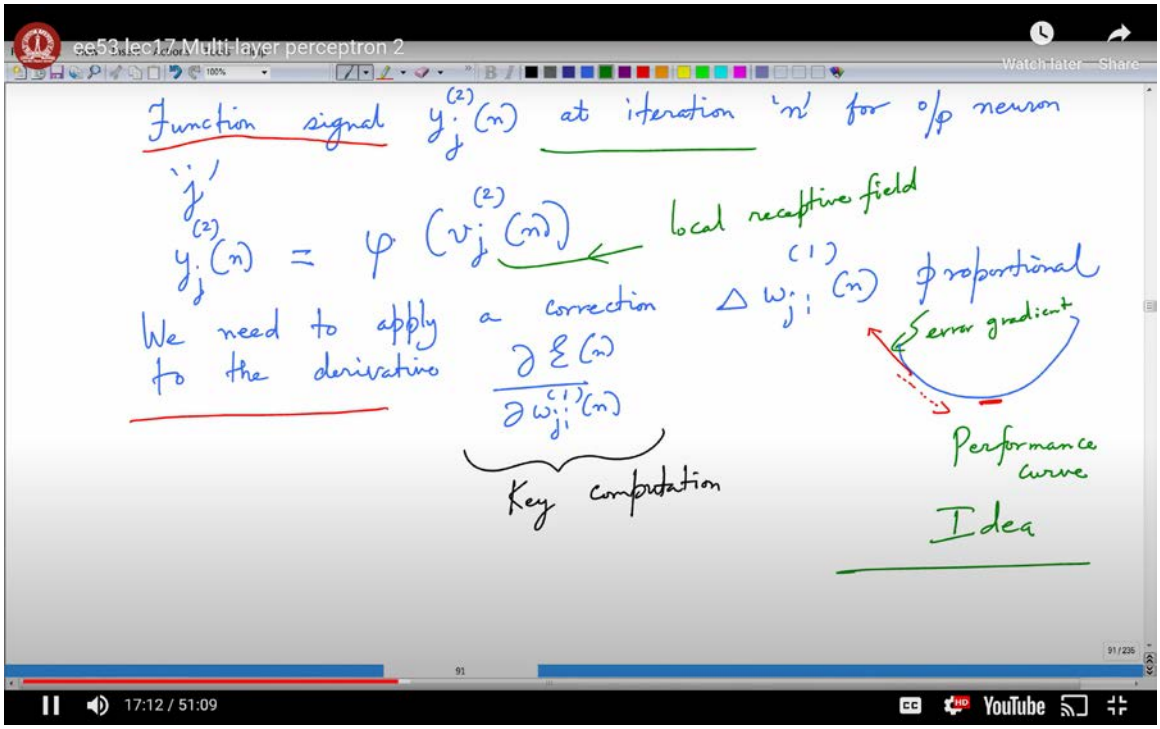
After computing the functional signal y_j for node j, we compare it with the corresponding desired signal d_j. Here, n denotes the time instant. The difference between these two signals gives us the instantaneous error signal $e_j(n)$. This provides a complete picture of how to compute some of the latent variables given the network's parameters.

In mathematical terms, the functional signal at neuron j in the second layer (the output layer) at iteration n is given by:

$$y_j^{(2)}(n) = \varphi\left(v_j(n)\right)$$

This $v_j(n)$ is also referred to as the local receptive field.

(Refer Slide Time: 17:12)



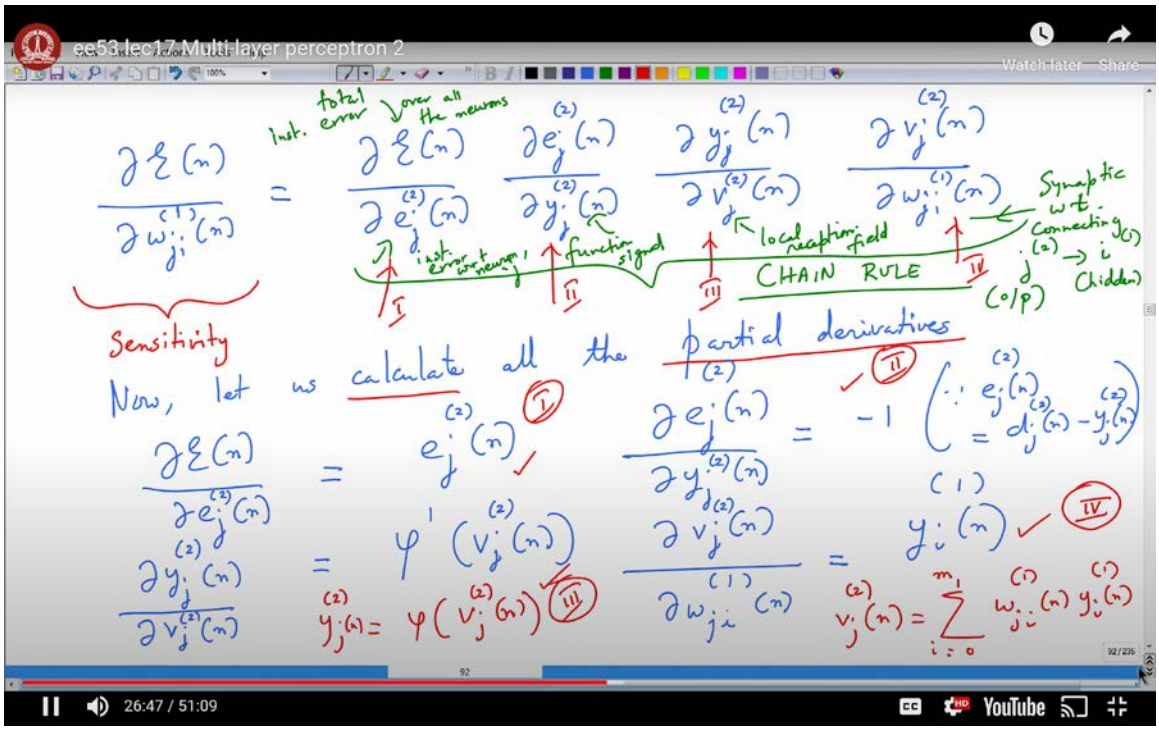
Now, once we compute the functional signals, we can determine the error. As we saw earlier, the error is essentially the difference between the functional signal and the desired signal, regardless of the order, since we square the error and the sign doesn't matter. You know the desired signal for neuron j in the output layer, and you have analytically computed the functional signal for this node. Therefore, you can compute the error. With this parameter in hand, the next step is to apply a correction to the weight connecting neuron j in the output layer to neuron i in the previous layer at time instant n. This correction is proportional to the gradient of the error.

Let's talk about gradient descent techniques. The key idea here is to move in the direction opposite to the error gradient. Imagine you have a performance curve representing the error surface. If the error gradient points in a particular direction, indicated by the bold red arrow,

you need to move in the opposite direction to reach the bottom of this curve, which represents the minimum error.

Instantaneously, we may not know the exact shape of the error surface. However, we understand that moving in the direction opposite to the error gradient reduces the error. As you move away from the bottom of the curve, the error increases, but as you move towards it, the error decreases. Therefore, the correction term must be proportional to the negative derivative of the instantaneous error with respect to the synaptic weights at time instant n. Although the negative sign is crucial, we will discuss this in detail as we progress through the algorithm.

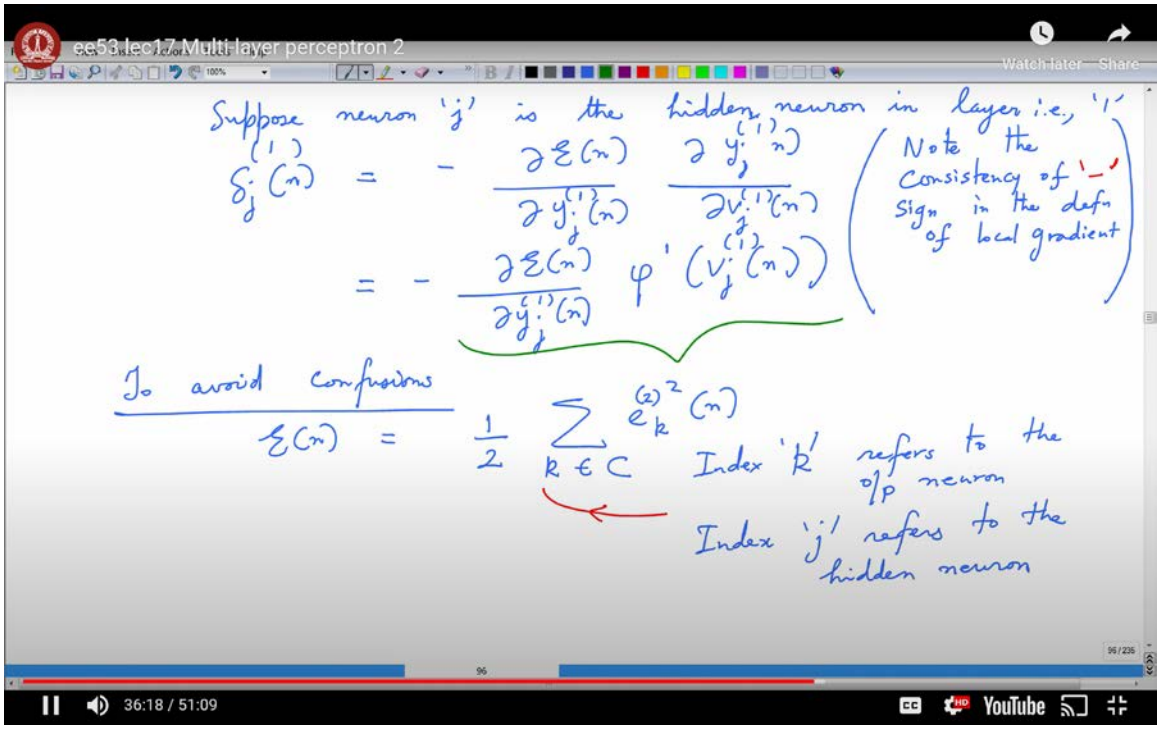
(Refer Slide Time: 26:47)



The idea should be clear: we need to follow the gradient descent. Now, how do we decompose this error? Recall that the total error is the summation of all instantaneous error energies over all neurons. When taking the derivative of the error with respect to the synaptic weights, we use the chain rule.

Here's how we proceed: we need the partial derivative of the error energy at time instant n with respect to the synaptic weight w_{ji} at time instant n in layer 1. In our notation, synaptic weights connecting the zeroth layer to the first layer are indicated by $w_{ji}^{(0)}$, and those from the hidden layer to the output layer are $w_{ji}^{(1)}$.

(Refer Slide Time: 36:18)



We apply the chain rule to take the partial derivative of the total error (instantaneous error summed over all neurons) with respect to the instantaneous error of neuron j. Then, we take the partial derivative of the instantaneous error of neuron j with respect to the functional signal in the output layer for this neuron. The next step involves taking the partial derivative of the functional signal in the output layer at node j with respect to the local receptive field v_j. Finally, we take the partial derivative of the local receptive field with respect to the synaptic weight connecting neuron j in the second layer (output layer) to neuron i in the first layer (hidden layer).

So, our chain rule steps are as follows:

1. $\frac{\partial E}{\partial e_j}$: The partial derivative of the total error with respect to the instantaneous error of neuron j.

2. $\frac{\partial e_j}{\partial y_j}$: The partial derivative of the instantaneous error of neuron j with respect to the functional signal y_j in the output layer.

3. $\frac{\partial y_j}{\partial v_j}$: The partial derivative of the functional signal y_j with respect to the local receptive field v_j.

4. $\frac{\partial v_j}{\partial w_{ji}}$: The partial derivative of the local receptive field v_j with respect to the synaptic weight w_{ji}.

By following these steps, we systematically decompose the error and apply the necessary corrections to the weights. This method, grounded in gradient descent, ensures that we minimize the error effectively.

To compute the necessary partial derivatives using the chain rule, we must recognize that there are four main derivatives to consider. First, we need to compute the partial derivative of the total instantaneous energy with respect to the synaptic weight w_{ji} in layer 1, where j is the neuron in the output layer and i is the neuron in the hidden layer. This can be broken down into four distinct parts:

1. The partial derivative of the total instantaneous energy with respect to the instantaneous energy at neuron j.

2. The partial derivative of the instantaneous energy at neuron j with respect to the functional signal at neuron j.

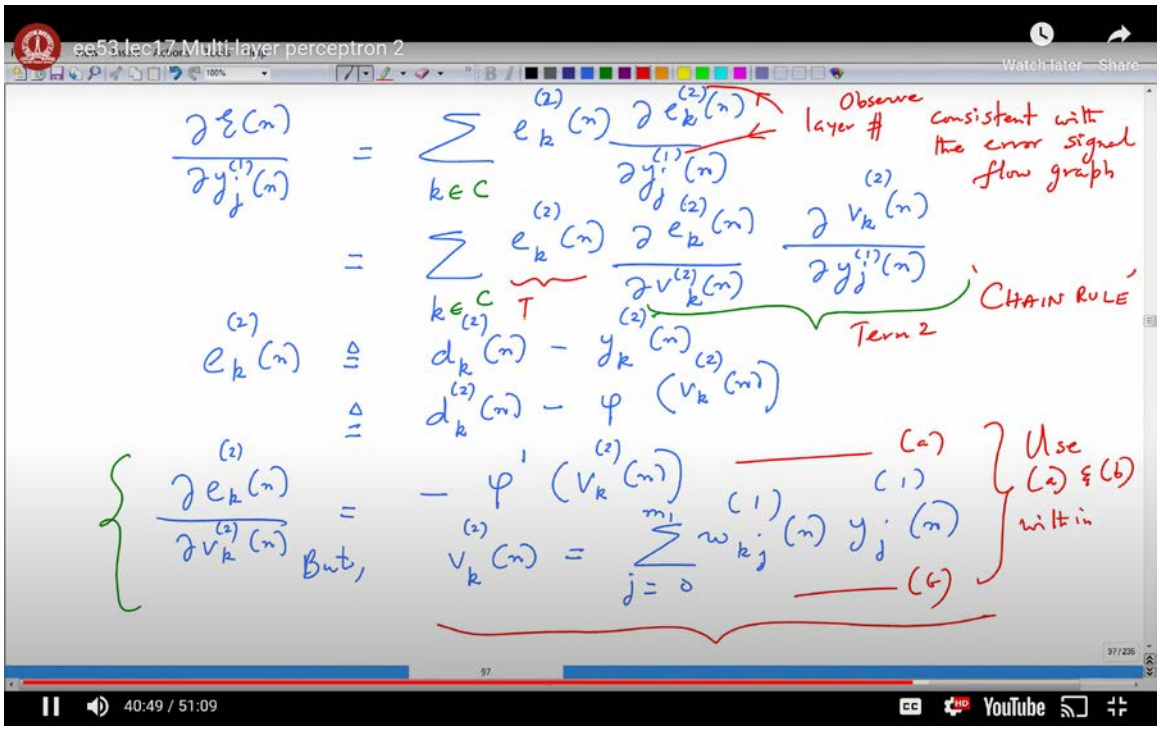
3. The partial derivative of the functional signal at neuron j with respect to the local receptive field at neuron j.

4. The partial derivative of the local receptive field at neuron j with respect to the synaptic weight connecting neuron j in the output layer to neuron i in the hidden layer.

This process involves a change in the superscript value to indicate the connection. Once we understand this concept, we can straightforwardly calculate all the partial derivatives.

First, consider the partial derivative of the total instantaneous error over all neurons in the output layer with respect to the instantaneous error in neuron j in the output layer, which is essentially $E_j(n)$. This is straightforward because the summation of $\frac{1}{2}E_j^2(n)$ over all j in the set of output neurons leaves us with $E_j(n)$.

(Refer Slide Time: 40:49)



Next, we address the second term. The instantaneous error at neuron j is defined as the desired signal minus y_j. When we take the derivative, we obtain a value of -1.

For the third term, we take the derivative of the functional signal y_j with respect to the local receptive field v_j. The functional signal y_j is given by $\varphi(v_j)$, where φ is a non-linear activation function. Therefore, the derivative is $\varphi'(v_j(n))$.

Finally, we consider the partial derivative of v_j with respect to w_{ji}. Recall the equation $v_j(n) = \sum_{i=0}^{m_1} w_{ji}(n)y_i(n)$. Taking the partial derivative with respect to w_{ji}, we get $y_i(n)$.

Summarizing, we have the four terms:

$$1.\frac{\partial E}{\partial e_j} = E_j(n)$$

$$2.\frac{\partial e_j}{\partial y_j} = -1$$

$$3.\frac{\partial y_j}{\partial v_j} = \varphi'\left(v_j(n)\right)$$

$$4.\frac{\partial v_j}{\partial w_{ji}} = y_i(n)$$

Combining these, the partial derivative of the total instantaneous energy with respect to the synaptic weight w_{ji} is given by:

$$\frac{\partial E}{\partial w_{ji}} = E_j(n) \cdot (-1) \cdot \varphi'\left(v_j(n)\right) \cdot y_i(n)$$

This result simplifies to:

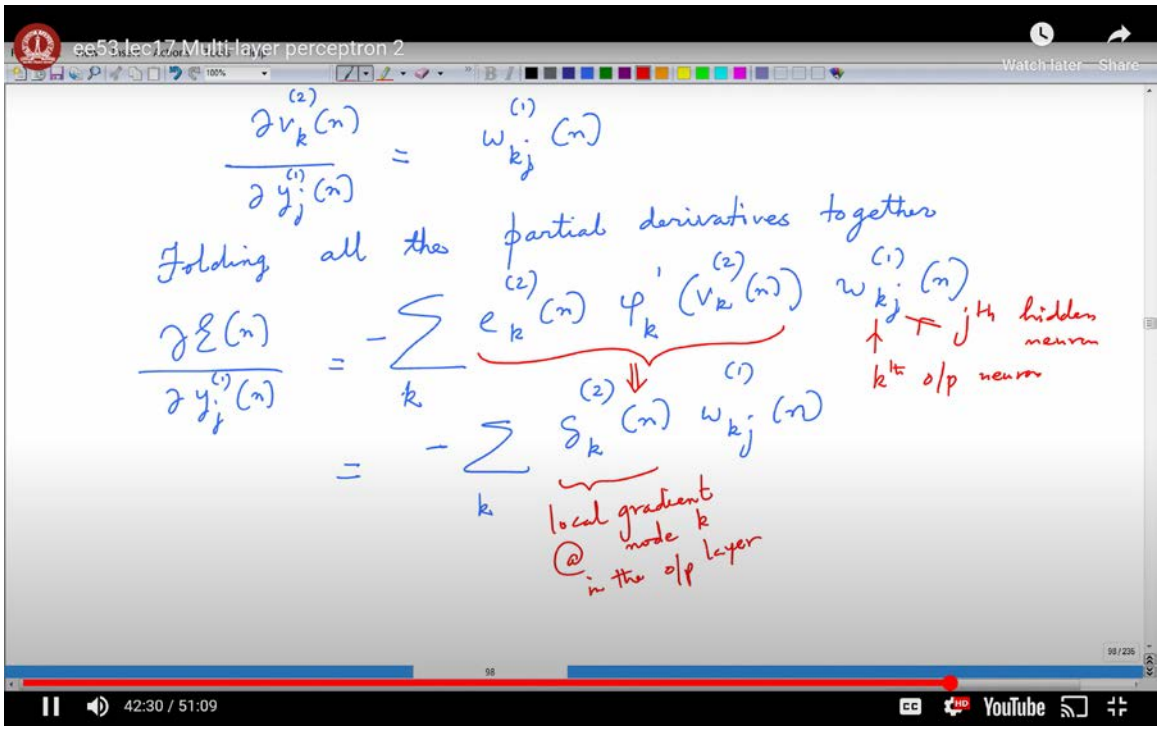
$$\frac{\partial E}{\partial w_{ji}} = -E_j(n)\varphi'\left(v_j(n)\right) y_i(n)$$

Understanding how to compute these variables within the network using the chain rule allows us to determine the partial derivative of the total instantaneous energy with respect to the synaptic weight w_{ji} in the hidden layer, ultimately leading us to the desired correction for the weights.

To compute the weight adjustments, δw_{ji}, we need them to be proportional to the error gradient with respect to the synaptic weight. By introducing a learning rate η, which we assume to be constant throughout the algorithm, we obtain:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

(Refer Slide Time: 42:30)



This expression highlights the necessity of calculating the partial derivative of the instantaneous error energy with respect to the synaptic weight connecting the j-th neuron to the i-th neuron in the hidden layer.

To achieve this, we must carry out several calculations involving four distinct partial derivatives. Let's define the local gradient $\delta_j(n)$ in layer 2 as:

$$\delta_j(n) = -\frac{\partial E}{\partial v_j(n)}$$

Here, $v_j(n)$ is the local receptive field at node j, and the superscript indicates the layer number. The term "local gradient" is used because, in the output layer, we can directly compute the error by comparing the functional signal with the desired response. This gives us the instantaneous error at the output.

Recall that the error signal must propagate backward from the output layer to the weights connecting the inputs to the first hidden layer. To compute the changes in synaptic weights,

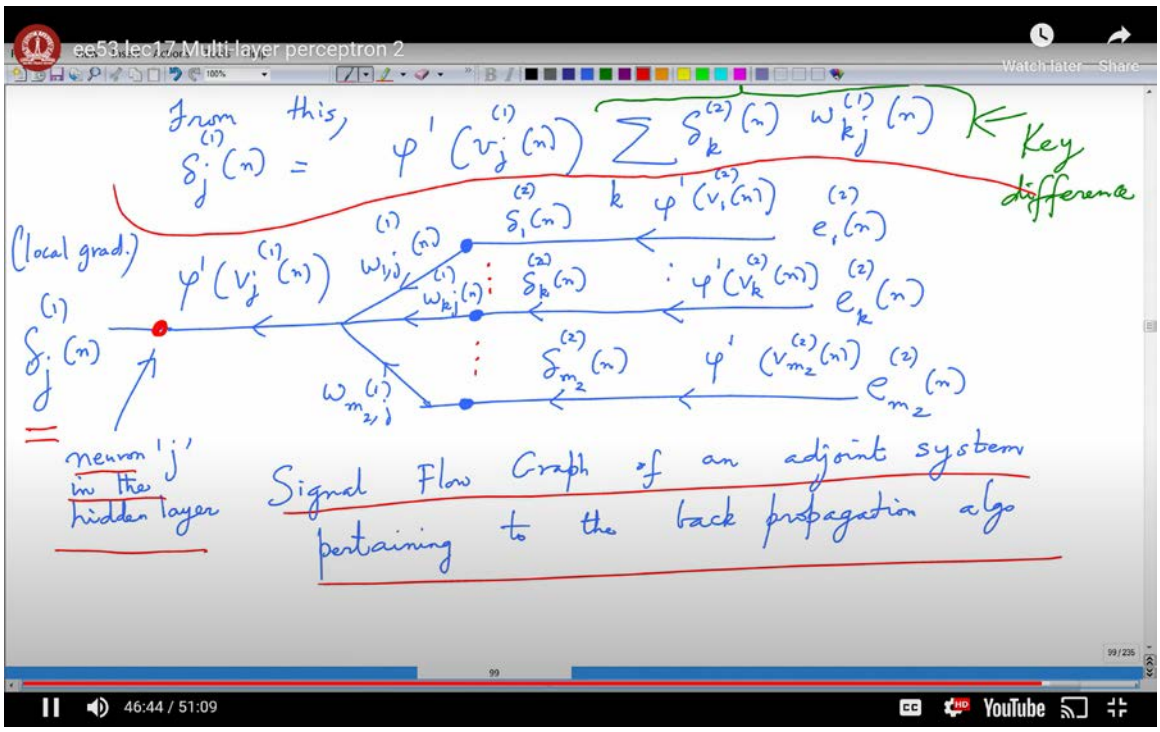
we need to examine the error gradients or local gradients in each layer. This is a critical concept for understanding the algorithm.

In the output layer, the local gradient $\delta_j(n)$ is defined as:

$$\delta_j(n) = E_j(n)\varphi'\left(v_j(n)\right)$$

where $E_j(n)$ is the instantaneous error, and φ' is the derivative of the non-linear activation function acting on the local receptive field in the second layer. Maintaining consistent layer numbering is crucial here.

(Refer Slide Time: 46:44)



The adjustment to the weight, δw_{ji}, is given by:

$$\delta w_{ji} = \eta\delta_j(n)y_i(n)$$

This aligns with our previous discussions. Even if the concept of the local gradient seems complex, recalling the gradient descent rule and the method for taking partial derivatives will suffice for proceeding forward. Defining the local gradient as an interim variable is

useful because it simplifies the process of updating the weights. Thus, we can easily update the weights using this definition.

Let's proceed further. There are two scenarios to consider, as neurons can exist in the output layer as well as in the hidden layers. Since we are discussing a single layer of hidden neurons, we have two cases: neuron j as an output node and neuron j as a hidden node.

When neuron j is located in the output layer, it is supplied with a desired response. This is crucial because the architecture provides it with the desired response. We can compute the instantaneous error $E_j(n)$ associated with this neuron, making it straightforward to compute $\delta_j(n)$ for that layer. Calculating the local gradient at neuron j in the output layer is easy because we know the error, the activation function, and the local reception field. We assume the same activation function for all neurons to avoid complications, although biological neurons might differ.

Now, computing the local gradient in the output layer is simple because we can directly handle the error at the output node. However, for a hidden node, it is not straightforward since there is no specified desired response. Here, we need a clear conceptual understanding of the algorithm. The error signals flow backward from the output layer to the hidden layer. We compute the sum of all the local gradients scaled by the synaptic weights connecting all neurons in the output layer to a particular neuron in the hidden layer.

This process is opposite to the forward pass, where a neuron in the output layer sums all the inputs or functional signals from the hidden layer, scaled by the weights. For a hidden neuron, the error signals, scaled by the synaptic weights, are summed. This recursive determination of the error signal for a hidden neuron is crucial and involves working backward from the output neurons to which the hidden neuron is directly connected. If confused, refer to the layer numbers and neuron connections to derive it yourself.

Now, with this conceptual clarity, we proceed. If j is a neuron in the hidden layer, the local gradient δ_j is:

$$\delta_j = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$$

By performing straightforward mathematical computations and considering the consistency of the negative sign throughout the process, we arrive at this equation. Following this principle, we compute the partial derivative of the error with respect to the functional signal in the hidden layer.

The error energy is essentially the sum of the squared errors of the output neurons:

$$E = \frac{1}{2}\sum_{k} E_k^2$$

Taking the partial derivative, we sum over all k belonging to the set of output neurons:

$$\frac{\partial E}{\partial y_j} = \sum_{k} E_k \frac{\partial E_k}{\partial y_j}$$

Next, we apply the chain rule to split this into two terms:

$$\frac{\partial E_k}{\partial y_j} = \frac{\partial E_k}{\partial v_k}\frac{\partial v_k}{\partial y_j}$$

By understanding these steps, we can derive the necessary partial derivatives and proceed with the algorithm accurately.

This involves using the chain rule, and it's crucial to be very attentive to the layer numbers. When you carefully examine the diagram or the signal flow graph, it becomes clear. Here, we are considering the error at the kth node in layer 2 (the output layer) and evaluating it with respect to the partial derivative at node j in layer 1. Observing the layer number is essential as it aligns with the error signal flow graph, which moves backward through the network.

Once you grasp this concept, the next steps are straightforward. You need to compute the partial derivative of each error in the output layer with respect to the functional signal. This involves summing over k:

$$\sum_k \frac{\partial E_k(n)}{\partial y_j}$$

Here, $E_k(n)$ is computed as:

$$E_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi\big(v_k(n)\big)$$

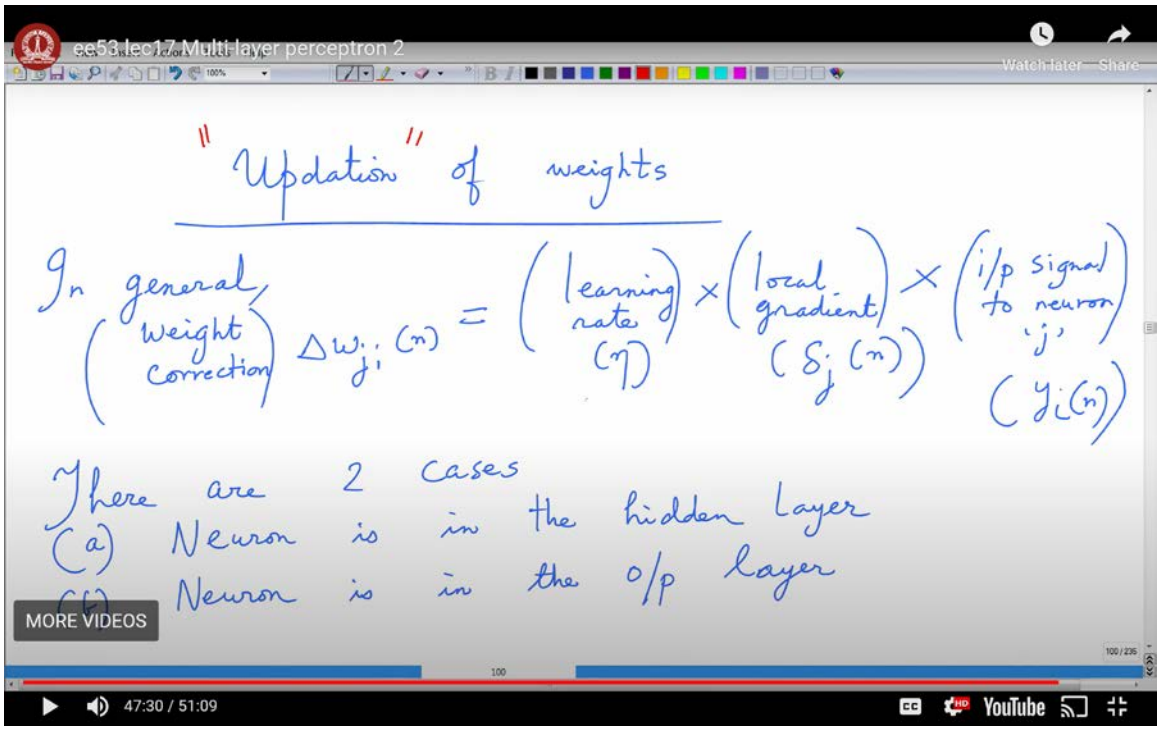
Calculating the partial derivative of $E_k(n)$ with respect to $v_k(n)$ is straightforward:

$$\frac{\partial E_k(n)}{\partial v_k(n)} = -\varphi'\big(v_k(n)\big)$$

This represents the local reception field. Next, taking the partial derivative of $v_k(n)$ with respect to y_j results in:

$$\frac{\partial v_k(n)}{\partial y_j} = W_{kj}(n)$$

(Refer Slide Time: 47:30)



These two equations can be referred to as terms A and B. Using these, we can simplify our equation for the partial derivative of the total instantaneous energy with respect to the functional signal at node j in layer 1:

$$-\sum_{k} E_k(n) \cdot \varphi'\big(v_k(n)\big) \cdot W_{kj}$$

Here, W_{kj} is the synaptic weight between the kth output neuron and the jth hidden neuron. If we treat $E_k(n) \cdot \varphi'\big(v_k(n)\big)$ as the local gradient at node k in the output layer, we link this with the synaptic weights between the hidden and the output layers.

Now, having computed the partial derivatives, we need to calculate the local gradient in the hidden layer. For the jth neuron in the hidden layer, this is given by the derivative of the activation function evaluated at the local reception field times the summation of the product of the synaptic weight connecting the output neuron k with the hidden neuron j and the local gradient of the output neuron k:

$$\delta_j = \varphi'\big(v_j\big) \sum_{k} W_{kj} \cdot \delta_k$$

This process emphasizes the key difference between computing the local gradient in the output layer and the hidden layer. For the output layer, it is straightforward as we compute the error times the partial derivative of the error with respect to the functional signal. However, for a hidden neuron, we sum all the local gradients from the output nodes, scaled by the synaptic weights.

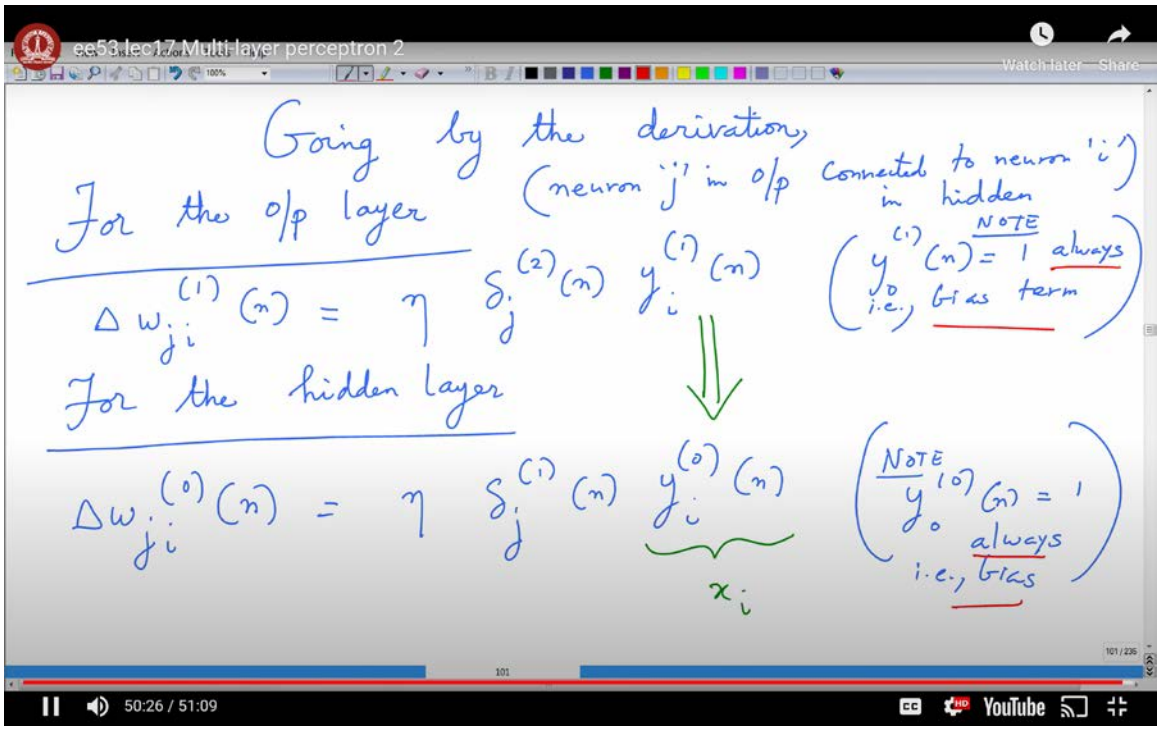
Pay close attention to the layer numbers indicated in the superscripts. This detail is crucial, especially if you plan to code this algorithm yourself. While many packages are available, understanding how the algorithm works is essential for developing your intuition and creating your own algorithms if needed.

Therefore, pay particular attention to the layer numbers and all the superscripts and subscripts. Though they might seem messy at first, if you closely follow the signal flow

graphs, you will find them extremely useful. From this equation, we can finally sketch the signal flow graph of an adjoint system pertaining to the backpropagation algorithm.

In the output layer, we have errors $e_1(n), e_2(n), \ldots, e_{m_2}(n)$. We take the derivative of the activation function with respect to the local reception field, $V_1, V_2, \ldots, V_{m_2}$. These are scaled by the local gradients $\delta_1, \delta_2, \ldots, \delta_{m_2}$, and then scaled by the synaptic weights that connect the jth neuron in the hidden layer to all these scaled local gradients through the synaptic weights. This is how we compute the local gradient at neuron j in the hidden layer.

(Refer Slide Time: 50:26)



With these tools in place, we can update the synaptic weights and realize our learning update. The general rule for weight update is that the weight correction is proportional to the local gradient times the input signal to that particular neuron. In mathematical terms, this proportionality is represented as:

$$\Delta w_{ji}(n) = \eta \cdot \delta_j(n) \cdot y_i(n)$$

Here, η is the learning rate, $\delta_j(n)$ is the local gradient at neuron j, and $y_i(n)$ is the input signal at neuron i. Although I have left it in a broad manner without indicating the layer numbers, if you are savvy, you will realize that there are two types of neurons: neurons in the hidden layer and neurons in the output layer. It is crucial to distinguish between them because a neuron in the hidden layer does not see the error directly.

For the output layer, where neuron j is connected to neuron i in the hidden layer, the weight update is given by:

$$\Delta w_{ji}^{1}(n) = \eta \cdot \delta_j(n) \cdot y_i(n)$$

This represents the input from the ith hidden node to the jth neuron in the output layer. Similarly, for the weights connecting the input to the hidden layer, the update is given by:

$$\Delta w_{ji}^{0}(n) = \eta \cdot \delta_j(n) \cdot y_i(n)$$

Here, $y_i^0(n)$ is essentially x_i, which are our input signals.

One small detail to remember, particularly when coding the algorithm, is that the 0th coordinate, y_0, in both the input layer and the hidden layer is always 1 because they are bias terms. Do not leave this floating or it will complicate things. This is a DC bias with a fixed value of +1, but it will be scaled by the synaptic weight, which is a variable in the equation.

We have now derived the backpropagation algorithm, detailing all the necessary steps for updating the synaptic weights, which is essentially how learning should proceed. We will proceed further from this step towards summarizing the algorithm. Let's stop here for now.