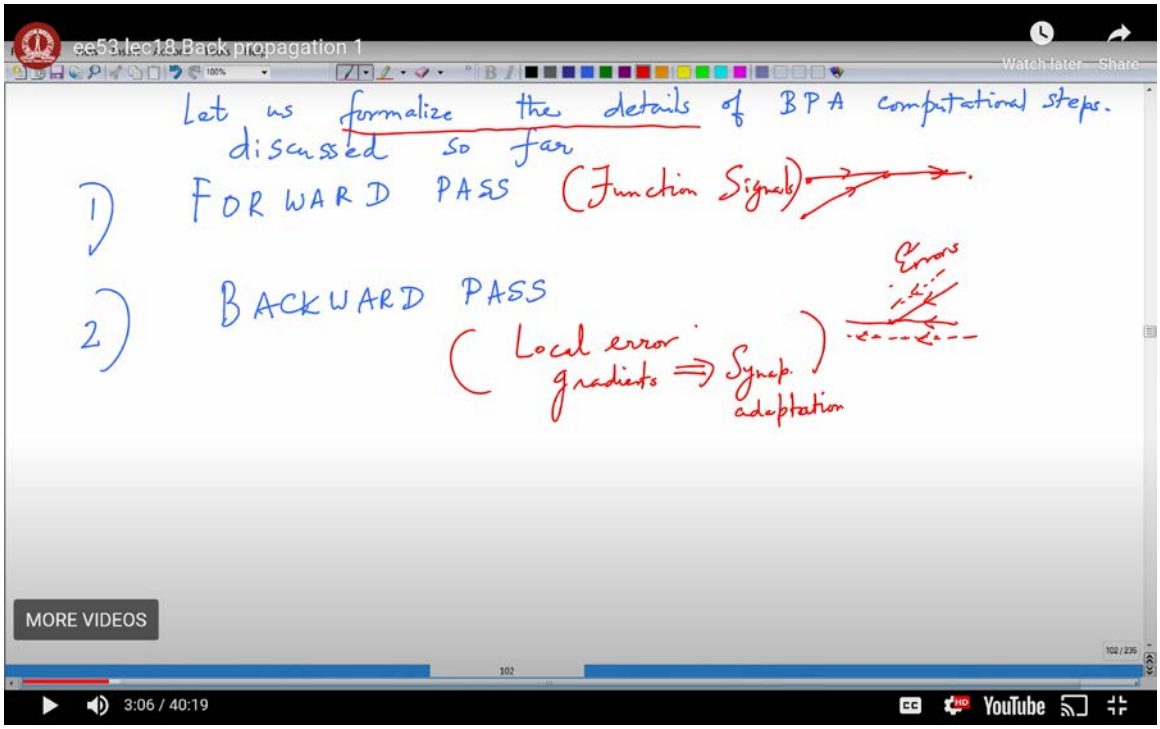
Neural Networks for Signal Processing-I
Prof. Shayan Srinivasa Garani
Department of Electronic System Engineering
Indian Institute of Science, Bengaluru

Lecture – 18
Back Propagation 1

Having derived the backpropagation algorithm from first principles for a single layer of hidden neurons, let us now summarize the computational steps and outline the algorithm. The backpropagation algorithm involves two main passes: the forward pass and the backward pass.

In the forward pass, we compute the functional signals from one node to the next, scaling them by the synaptic weights. This is how the computation progresses. In contrast, during the backward pass, we propagate the error signals (or local gradients of the errors) from the output neurons, scaled by the synaptic weights and the derivative of the activation function. This propagation continues until we reach the synaptic adaptation phase.

(Refer Slide Time: 03:06)



Understanding this distinction is crucial. In the forward pass, we deal with all the function

signals, while in the backward pass, we focus on the local error gradients that ultimately lead to synaptic adaptation. Let us summarize the steps of the forward pass:

- The synaptic weights remain unaltered throughout the network because we are not performing synaptic adaptation in this phase. The weights are fixed based on the synaptic adaptation done during the previous backward pass.
- We aim to compute the function signals at each node within a layer. For a given layer L, the function signal y_j at node j and time step n is computed as the non-linear activation function of the local reception field that node j sees.
- The local reception field at neuron j at time n in layer L is given by:

$$\text{Local Reception Field} = \sum_{i=0}^{L-1} w_{ji}^{(L-1)} \cdot y_i^{(L-1)}$$

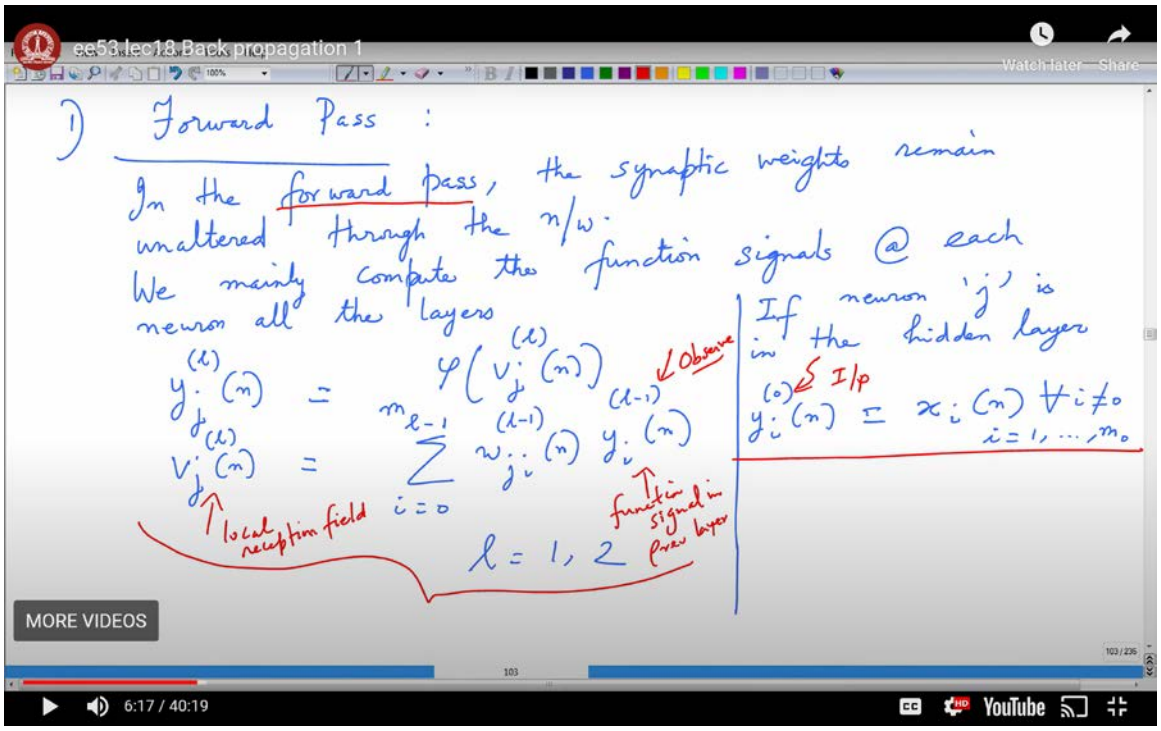
Here, $w_{ji}^{(L-1)}$ represents the synaptic weight connecting neuron j with neuron i in the previous layer, and $y_i^{(L-1)}$ is the function signal in the previous layer. Pay close attention to the superscripts to ensure clarity.

- If neuron j is in the hidden layer, the function signal $y_i(n)$ in layer 0 is essentially the input x_i for all i = 1 to m_0, except for the bias term. When i = 0, it corresponds to the bias term, which is the 0th coordinate.

With these steps in mind, we can accurately summarize and implement the backpropagation algorithm. This structured approach will help in understanding the intricate details and computations involved in both the forward and backward passes, ultimately leading to effective synaptic adaptation and learning.

The forward pass is relatively straightforward, but when implementing it in code, it's crucial to pay close attention to the weights. The weights used in the forward pass are the result of the synaptic adaptation that occurred during the backward pass of the previous computational step.

(Refer Slide Time: 06:17)

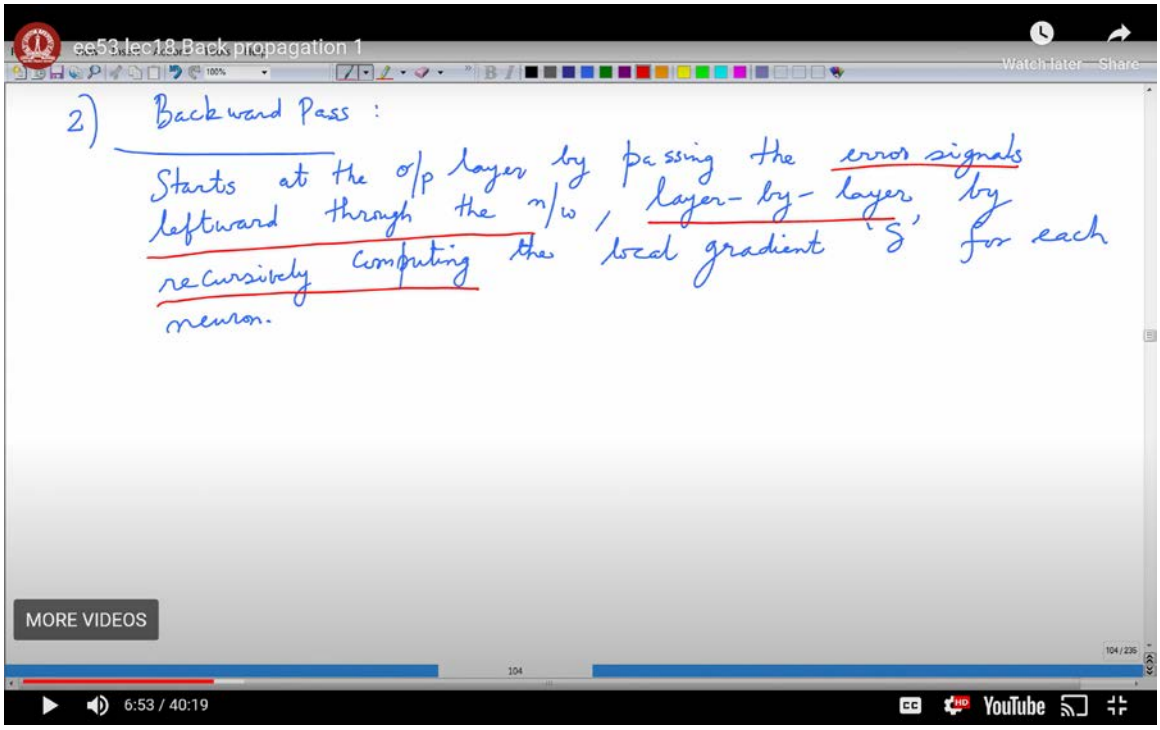


The backward pass, on the other hand, is also straightforward. It begins at the output layer and propagates the error signals backward through the network, layer by layer. This involves computing the local gradients recursively, which is a critical aspect of the process. If you have a good understanding of how to compute local gradients for both output layer neurons and hidden layer neurons from the signal flow graph, you're well-prepared to summarize the steps involved in the backward pass.

Now, let's discuss the handling of different activation functions, which is an important consideration. Activation functions are generally non-linear functions with inherent non-linearities. However, to compute derivatives, these functions must be differentiable, meaning they must be continuous. Some rules may need to be applied to handle derivatives for activation functions that are not smooth. For instance, a smooth function is one for which you can take multiple derivatives—first-order, second-order, and so on. In our case, since gradient descent relies on the gradient, which is the first derivative, it's sufficient if the activation function is differentiable for the first derivative.

There are, however, cases where activation functions have discontinuities, such as the Heaviside function. Despite its discontinuity at the origin, we can still define derivatives at specific points where discontinuities occur and proceed with our calculations.

(Refer Slide Time: 06:53)



Let's review a few basic activation functions before delving into more complex cases like the sine function and other pathological functions. For now, consider the logistic function, which is commonly used in neural networks. The logistic function is defined as:

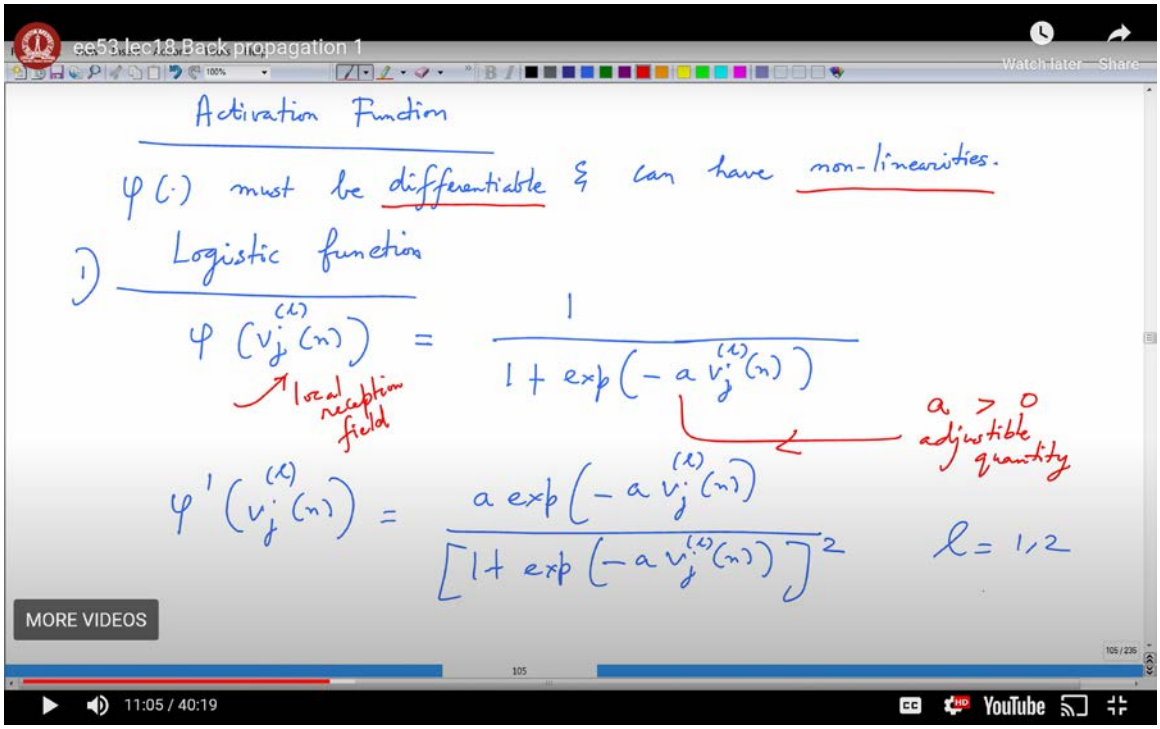
$$\phi\left(v_j(n)\right) = \frac{1}{1 + \exp\left(-a \cdot v_j(n)\right)}$$

Here, $v_j(n)$ represents the local reception field at neuron j in layer l at time step n. The term "local reception field" is qualitative, but $v_j(n)$ provides a quantitative measure, reflecting the value at neuron j in layer l at time n.

To find the derivative of the logistic function, apply basic calculus. Given:

$$\phi\left(v_j(n)\right) = \frac{1}{1 + \exp\left(-a \cdot v_j(n)\right)}$$

(Refer Slide Time: 11:05)



The derivative is:

$$\phi'\left(v_j(n)\right) = \frac{-a \cdot \exp\left(-a \cdot v_j(n)\right)}{\left(1 + \exp\left(-a \cdot v_j(n)\right)\right)^2}$$

This can be simplified to:

$$\phi'\left(v_j(n)\right) = a \cdot \phi\left(v_j(n)\right) \cdot \left(1 - \phi\left(v_j(n)\right)\right)$$

where $\varphi(v_j(n))$ is the logistic function itself. This derivative calculation is straightforward and critical for implementing the backpropagation algorithm.

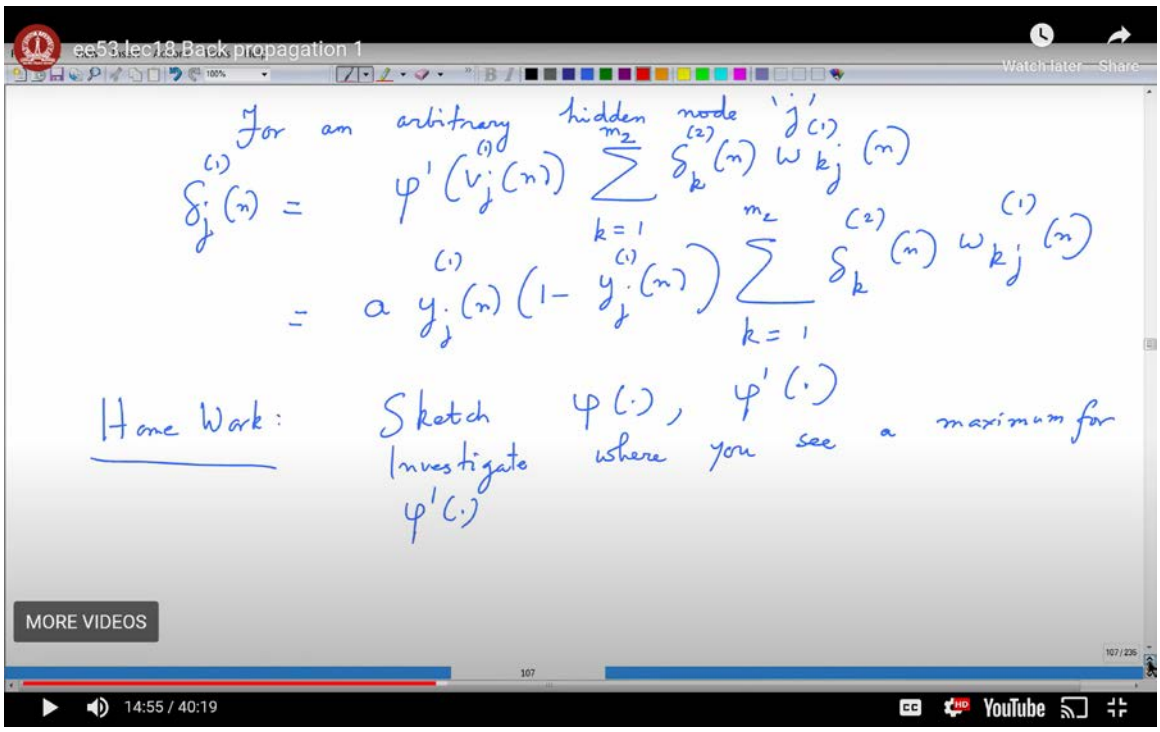
The function:

$$\phi\left(v_j(n)\right) = \frac{1}{1 + \exp\left(-a \cdot v_j(n)\right)}$$

where $v_j(n)$ is the local reception field at neuron j in layer l, is applicable for any layer, not just a single layer. In this case, $y_j(n)$ represents $\varphi(v_j(n))$, and by performing straightforward algebra, we find that the derivative of the local reception field is given by:

$$\phi'\left(v_j(n)\right) = a \cdot y_j(n) \cdot \left(1 - y_j(n)\right)$$

(Refer Slide Time: 14:55)



Thus, if you know the function signals at node j for layer l, you can compute the derivative of the local reception field in terms of the function signals $y_j(n)$, scaled by the constant a. Specifically, this derivative is:

$$a \cdot y_j(n) \cdot \left(1 - y_j(n)\right)$$

This holds true for layers l = 1 and l = 2. However, this approach is not restricted to just these two layers; it can be extended to any layer.

For a neuron j in the output layer, y_j is denoted as o_j, where o stands for output. If you prefer not to use o, you can continue to use y_j as long as the layer number is clearly indicated. The local gradient δ_j for the output layer, which is layer 2, is given by:

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

where $E_j(n)$ is the error at neuron j at time step n, and $\varphi'(v_j(n))$ is the derivative of the activation function evaluated at the local reception field for neuron j in the output layer. All calculations are performed for the specific time step n since there is no cell feedback or recursion involving different time steps.

The error $E_j(n)$ is defined as:

$$E_j(n) = d_j(n) - o_j(n)$$

I have omitted the superscript 2 here because it is implied that this measurement pertains to the output layer.

Using this information, the derivative of the activation function, $\varphi'(v_j(n))$, can be expressed as:

$$\phi'\left(v_j(n)\right) = a \cdot o_j(n) \cdot \left(1 - o_j(n)\right)$$

where $o_j(n)$ is used in place of $y_j(n)$ since it is defined as such. This computation is straightforward.

For an arbitrary hidden node j, the local gradient δ_j can be computed using:

$$\delta_j = \phi'\left(v_j(n)\right) \cdot \sum_k \left(W_{kj} \cdot \delta_k\right)$$

where $\varphi'(v_j(n))$ is the derivative of the activation function evaluated at the local reception field at neuron j in layer l. The summation is over all scaled local gradients from the output layer, scaled by the synaptic weights W_{kj} connecting output neurons k with hidden node j.

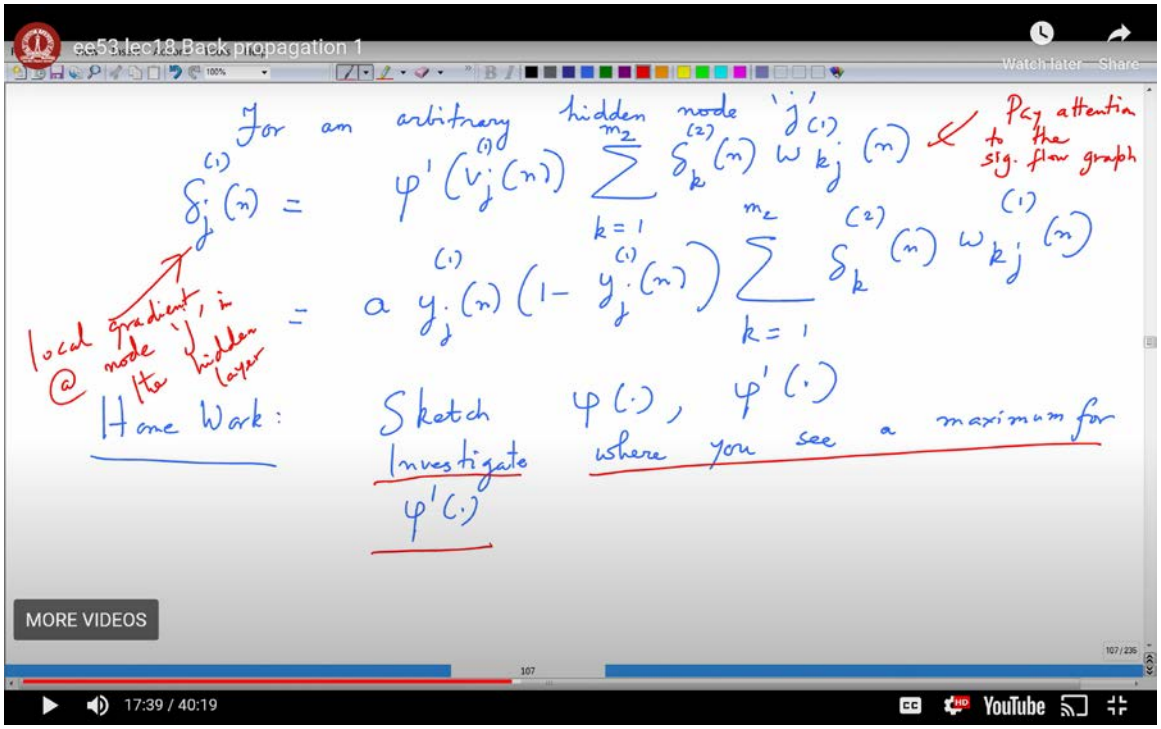
When performing these computations, pay close attention to the signal flow graph to ensure that layer numbers and calculations are accurately managed. At this stage, the primary focus is on the derivative of the activation function.

For the logistic function, the derivative of the activation function is given by:

$$\phi'\left(v_j(n)\right) = a \cdot y_j(n) \cdot \left(1 - y_j(n)\right)$$

where a is a constant greater than 0 and $y_j(n)$ is the function signal at neuron j at time step n in layer 1. Plug this derivative into your calculations to compute the local gradient in the hidden layer.

(Refer Slide Time: 17:39)



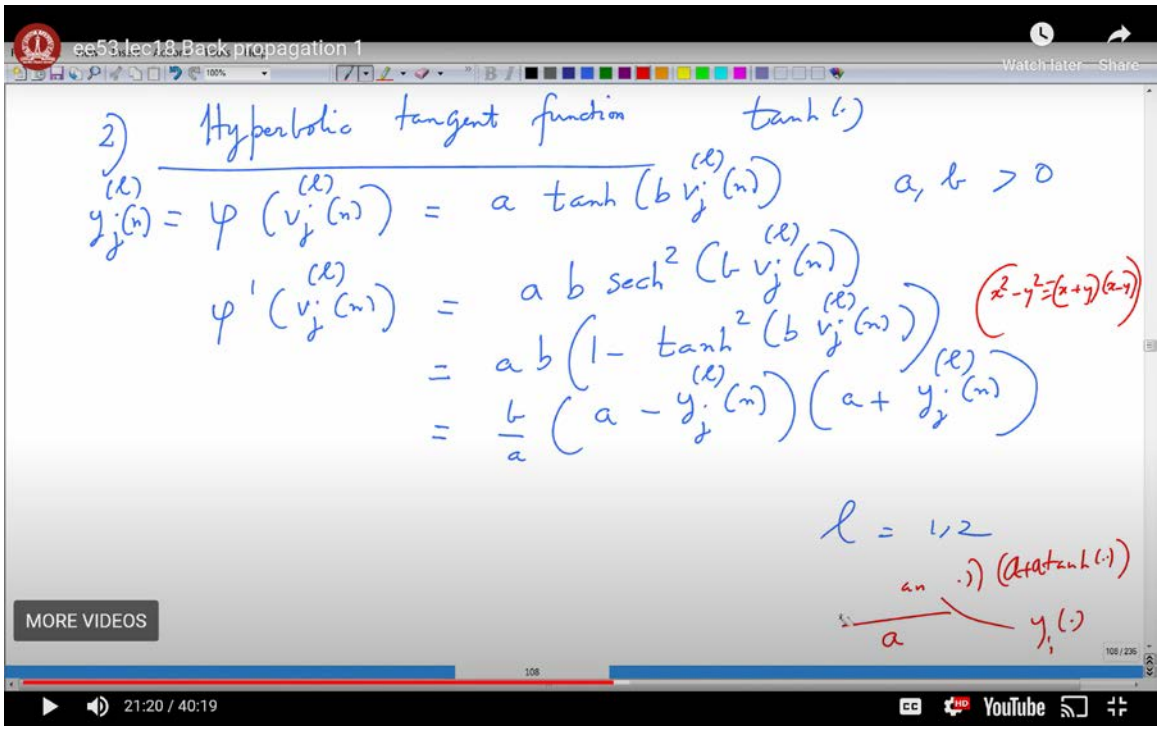
As a homework exercise, I encourage you to sketch this logistic activation function, given by the equation provided earlier. Plot this function for various values of a (greater than 0) and take its derivative. Investigate if you can identify a maximum for the derivative. This exercise will help you understand the function's behavior in more depth.

Another commonly used activation function is the hyperbolic tangent function, often encountered in calculus exercises. The hyperbolic tangent function, denoted as tanh, is used in neural networks and is defined as:

$$\tanh\left(b \cdot v_j(n)\right)$$

where tanh is $\frac{sinh}{cosh}$ you can express sinh and cosh in terms of exponential functions. This is a straightforward process.

(Refer Slide Time: 21:20)



The derivative of the hyperbolic tangent activation function is given by:

$$\phi'\left(v_j(n)\right) = a \cdot b \cdot \mathrm{sech}^2\left(b \cdot v_j(n)\right)$$

where b is a constant greater than 0 and a is also greater than 0. The sech (hyperbolic secant) function is defined as:

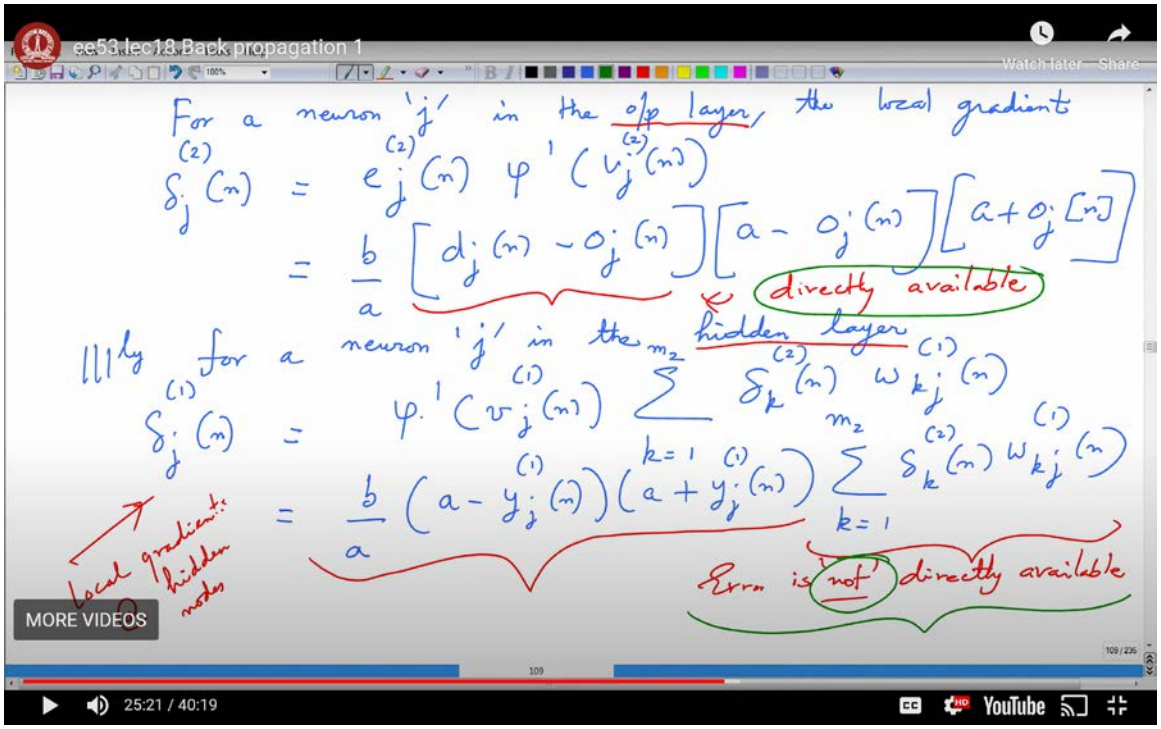
$$\mathrm{sech}(x) = \frac{1}{\cosh(x)}$$

Thus, the derivative can be simplified to:

$$\phi'\left(v_j(n)\right) = a \cdot b \cdot \left(1 - \tanh^2\left(b \cdot v_j(n)\right)\right)$$

Here, $\text{sech}^2\left(b \cdot v_j(n)\right)$ simplifies to $1 - \tanh^2\left(b \cdot v_j(n)\right)$. To simplify further, you can use the identity $\text{sech}^2(x) = 1 - \tanh^2(x)$.

(Refer Slide Time: 25:21)



This result applies to layers 1 and 2. For a neuron j in the output layer, the local gradient δ_j is computed as:

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

where E_j is the error at neuron j and $\varphi'(v_j(n))$ is the derivative of the activation function evaluated at the local reception field.

Substitute the formula for $\varphi'(v_j(n))$ into this equation:

$$\delta_j = \frac{b}{a} \cdot (d_j - o_j) \cdot (a - o_j) \cdot (a + o_j)$$

where o_j is the output response of neuron j in the output layer.

For neurons in the hidden layer, compute the local gradient similarly by applying the derivative of the hyperbolic tangent function at the local reception field using:

$$\delta_j = \frac{b}{a} \cdot (a - y_j) \cdot (a + y_j)$$

where y_j is the function signal in the hidden layer.

This approach is straightforward, and by carefully following these steps, you can effectively compute the local gradients for both output and hidden layer neurons using the hyperbolic tangent activation function.

The remaining calculations can be performed recursively. Once you've computed δ_j for the output layer, you can use this result to evaluate the local gradients for neurons in the hidden layer. This dependency across layers is crucial; if you cannot determine the local gradient for a neuron in the output layer, you cannot compute the local gradients for neurons in the hidden layers.

Thus, you must proceed sequentially, starting from the output layer and moving back through each layer until you reach the first hidden layer. It's also important to note that while the error for neurons in the output layer is directly observable and available, the error for neurons in the hidden layers is not. Therefore, you must compute the local gradients for hidden layer nodes using the gradients obtained from the output layer nodes. Be meticulous with superscripts during these calculations to avoid errors in your code.

Having discussed the computation of the derivative of the activation function and how to evaluate these derivatives, let us now turn our attention to the influence of the learning rate on the dynamics of the backpropagation algorithm.

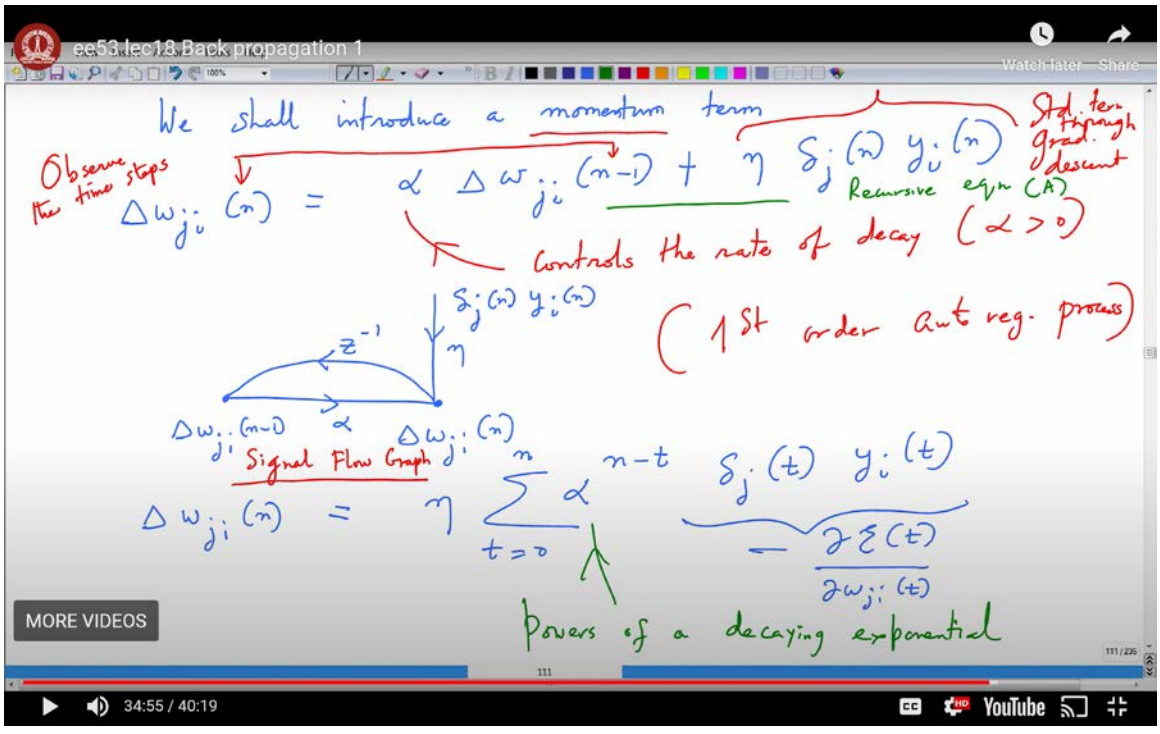
The learning rate, denoted by η, determines how the synaptic weights are adjusted in proportion to the derivative of the total instantaneous error energy observed at the output.

The backpropagation algorithm approximates the trajectory in the weight space using gradient descent. Visualize this process as a bowl representing the performance surface.

If η is small, the steps you take toward minimizing the error are also small. For example, with small steps (indicated in green), you move smoothly towards the bottom of the bowl. This results in a slower learning rate, which means progress is more gradual but steady.

Conversely, if η is large, the steps become larger, potentially causing oscillations. Imagine a marble sliding through a bowl: if you push the marble gently, it gradually settles at the bottom. However, if you push it forcefully, it might bounce off the sides and oscillate before eventually settling. This analogy helps visualize how a large learning rate can lead to oscillations as the algorithm converges.

(Refer Slide Time: 34:55)



To avoid excessive oscillations and ensure efficient convergence, it's essential to adjust the learning rate carefully. An optimal learning rate allows you to balance the speed of convergence with stability. Additionally, advanced techniques like Hessian-based

optimization methods, which we will cover in more advanced classes, can further enhance the optimization process.

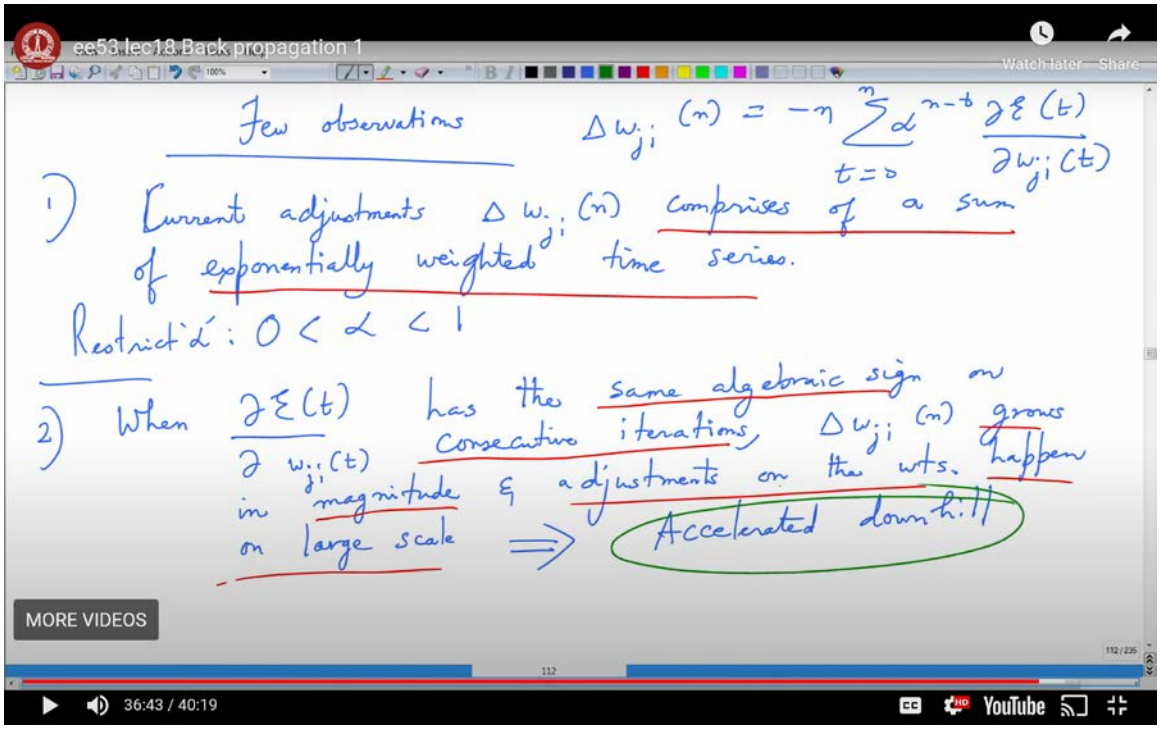
The learning process is not solely about adjusting the learning rate. Introducing a momentum term can also improve learning. Momentum helps accelerate convergence by considering previous adjustments to the synaptic weights, providing an additional push based on past changes.

The synaptic adaptation at time n is defined by the equation:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

Here, α represents the weight of the previous time step's adaptation, while η is the learning rate that controls the step size of the update. The term $\delta_j(n)$ is the local gradient at time n, and $y_i(n)$ is the input at time n. This update rule includes the standard gradient descent term, where $\eta\ \delta_j(n)\ y_i(n)$ represents the error times the input.

(Refer Slide Time: 36:43)



The parameter α controls the rate at which past adaptations influence the current update.

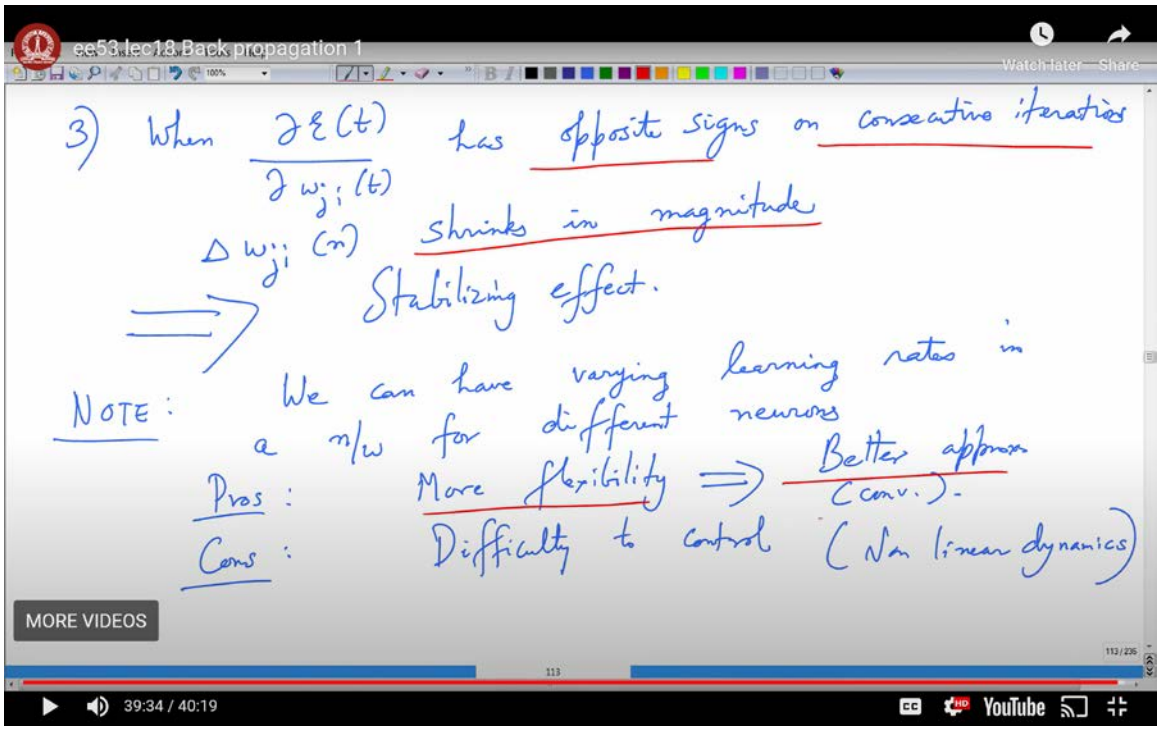
By adjusting α, you can control how quickly or slowly the adaptation decays. If α is close to 1, the influence of past adaptations is significant, leading to faster acceleration. Conversely, if α is very small, the influence of previous time steps is minimal.

To visualize the momentum update process, consider the diagram of a signal flow graph. The adaptation at time step n involves applying $\Delta w_{ji}(n)$, which is influenced by $\alpha \Delta w_{ji}(n - 1)$ and the local gradient and input at the current time step. This process resembles a first-order autoregressive model with time-varying weights, reflecting how the weights adapt based on past examples.

Expanding the recursive equation, we get:

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

(Refer Slide Time: 39:34)



Here, $\delta_j(t)\, y_i(t)$ represents the negative gradient of the error with respect to the synaptic weight, and α^{n-t} indicates how the influence decays over time. When n - t is small, α^{n-t} is

close to 1, and as n - t increases, α^{n-t} approaches 0. This decay term controls the influence of past gradients on the current weight adjustment.

Observations on the weight updates:

- If the partial derivative of the error with respect to w_{ji} has the same algebraic sign across consecutive iterations, w_{ji} grows in magnitude. This means the weights adjust on a larger scale, leading to accelerated descent in the weight space.
- If the partial derivative of the error with respect to w_{ji} alternates in sign over iterations, w_{ji} may shrink in magnitude, leading to a stabilizing effect. This creates a situation where adjustments may oscillate, eventually stabilizing the weight updates.

Thus, controlling the learning rate η and the decay parameter α is crucial for balancing convergence speed and stability in weight adjustments.

The dynamics of the algorithm hinge on the instantaneous gradient that is computed. Since the sign of this gradient is unpredictable from one iteration to the next, you might encounter different behaviors. If the gradient's sign remains consistent, you accumulate adjustments, which can lead to faster progress. On the other hand, if the gradient alternates in sign, this introduces a stabilizing effect, preventing excessive growth and maintaining balance.

Controlling the learning rate, η, is crucial for managing the algorithm's dynamics. At this stage, you might wonder why η should remain fixed. Given that this is a nonlinear dynamical system, adjusting η can influence how the algorithm behaves. Varying learning rates complicate the algorithm's dynamics, making it challenging to analyze and stabilize the system. This complexity would require local stability analysis and the examination of eigenmodes, which can be quite difficult.

However, if you allow η to be adjustable, you gain better control over the dynamics, potentially leading to accelerated learning. This flexibility allows for a more nuanced approach, where heuristics or analytical methods can help determine the optimal learning rates. While this introduces the benefit of improved approximation, it also comes with the challenge of managing the nonlinear dynamics, making it harder to control.

In summary, understanding the role of the learning rate and its impact on the algorithm's dynamics is essential. In the following lectures on backpropagation, we will explore the XOR problem and delve into pattern classification challenges, particularly focusing on scenarios that are not linearly separable. We'll examine how to address these problems using hidden networks with appropriately chosen activation functions. For now, we will conclude our discussion here.