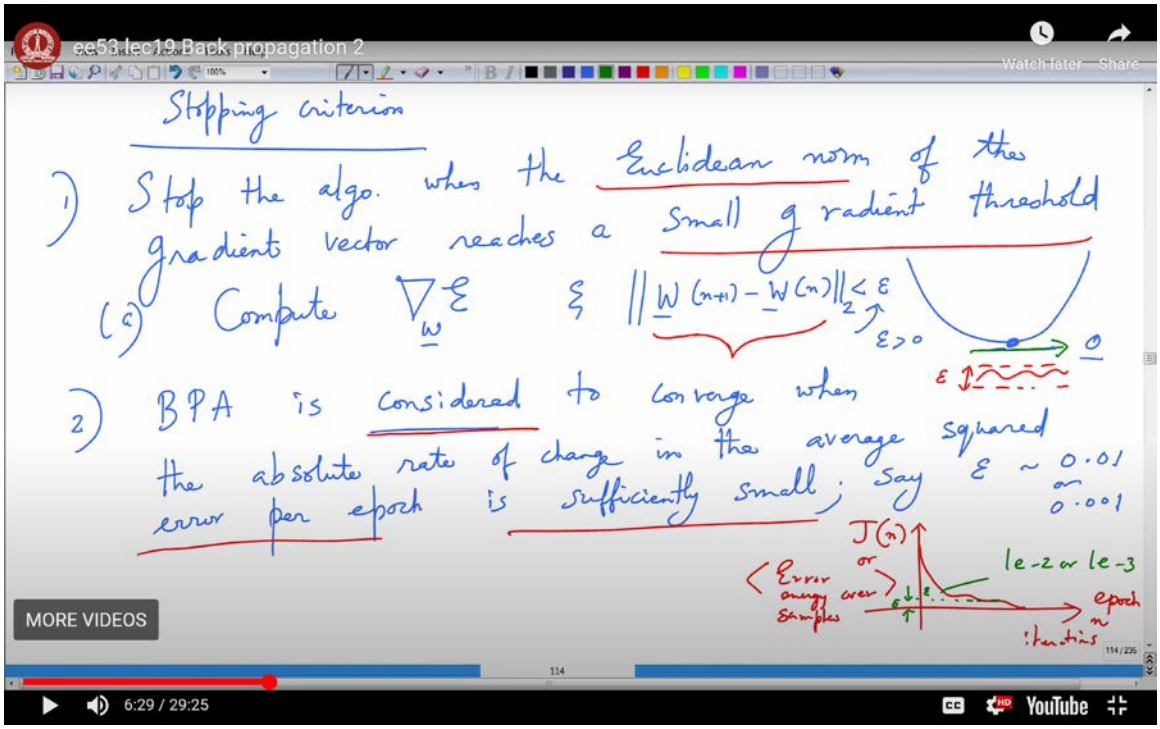
Neural Networks for Signal Processing-I
Prof. Shayan Srinivasa Garani
Department of Electronic System Engineering
Indian Institute of Science, Bengaluru

Lecture – 19
Back Propagation 2

Having explored how the momentum term impacts the learning process and how it can accelerate convergence, it's clear that introducing this term can significantly influence how the synaptic weights adapt in conjunction with the gradient descent updates. We also sketched out how parameters affect the learning rate using a simple signal flow graph.

One crucial aspect of iterative algorithms is the stopping criterion. When dealing with algorithms designed for convergence, it's essential to have a clear criterion for when to stop the process. Here are some general heuristics to consider for determining when to halt the algorithm:

(Refer Slide Time: 06:29)



1. Fixed Number of Iterations: An easy criterion is to set a fixed number of iterations and stop once that number is reached. This approach is particularly useful when dealing with hardware constraints. However, you need to be mindful of the initial conditions. If the initial conditions are not well understood, you might not achieve the desired results with a fixed number of iterations. For example, with hardware constraints, you might decide on 25 or 30 iterations. If the initial conditions are good, you can proceed with this fixed number of iterations confidently. However, if the initial conditions are poor, these iterations might not yield meaningful results.

2. Gradient Norm Threshold: Another stopping criterion involves monitoring the Euclidean norm of the gradient vector. Specifically, you can stop the algorithm when the gradient's Euclidean norm falls below a small predefined threshold. In this case, you calculate the gradient of the error with respect to the weight, then assess the weight vector at time n+1. If the difference between the weight vectors at consecutive iterations, $w_{n+1} - w_n$, remains within an epsilon threshold, you can consider this as a convergence signal and stop the algorithm. Even if there is minor fluctuation in the norm, as long as it's within the epsilon threshold, convergence is deemed to be reached.

3. Rate of Change in Average Squared Error: A third criterion for convergence is based on the rate of change in the average squared error per epoch. The algorithm is considered to have converged when this rate of change becomes sufficiently small.

These stopping criteria help ensure that the algorithm terminates appropriately, either based on fixed iterations, gradient norms, or changes in error rates.

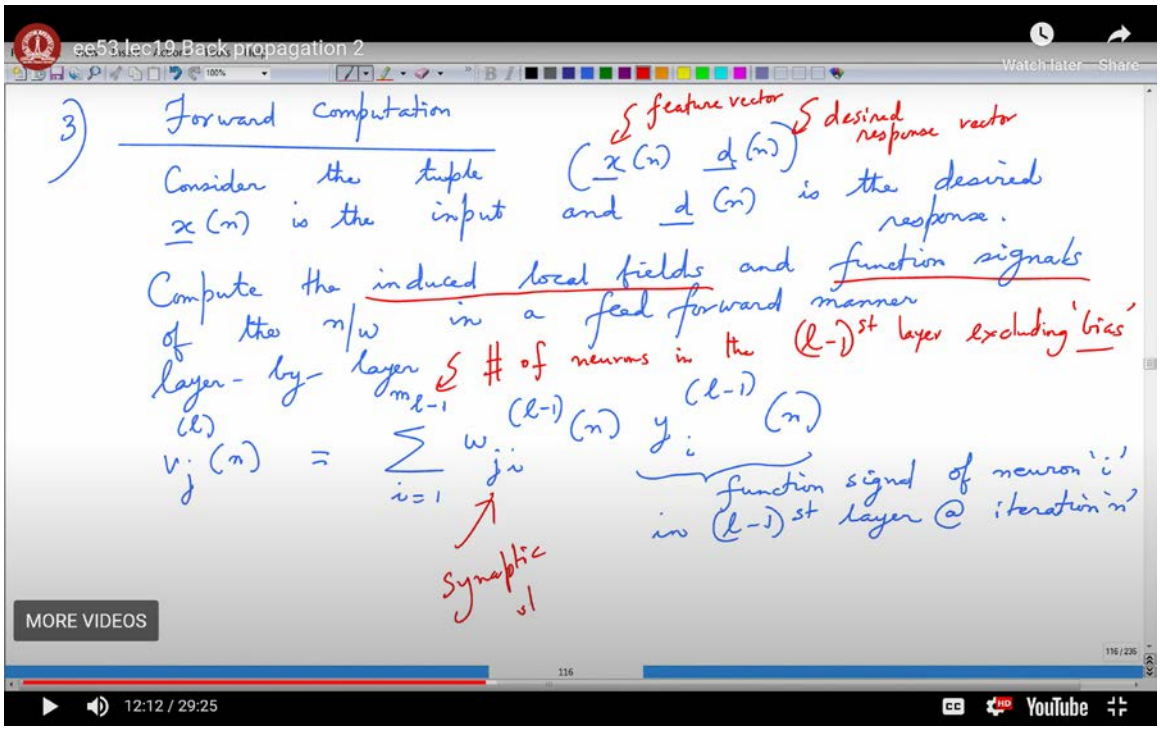
To reiterate, the absolute rate of change in the average squared error refers to plotting the error as a function of the iteration n. You might use notation like $e_j(n)$ or another label to represent the average error energy over samples. Initially, the error might start at a certain value and fluctuate before decreasing. If this error, when within an epsilon tolerance, typically 1×10^{-2} or 1×10^{-3}, meets your stopping criteria, then you can end the process. This is a practical constraint: regardless of the chosen criterion, the error should be small when you stop the algorithm.

If there is minimal change in the norm of the gradient vector, this suggests that you have reached a point where the gradient is effectively zero. In geometric terms, this means you are tangential to the error surface, indicating that you may have reached a saddle point where further significant progress is unlikely. In such cases, the algorithm can be considered to have converged.

For hardware implementations, the first and second criteria may be challenging due to finite resources, such as power consumption and throughput limitations. In such scenarios, it is often necessary to fix the number of iterations. It is crucial to choose initial conditions for the weights wisely. This can be achieved through a combination of hardware and software: using software to set appropriate initial conditions before transferring them to the hardware for faster convergence.

While this course does not delve deeply into hardware systems related to neural networks, it is important to recognize the practical implications as hardware implementations are a significant area of interest.

(Refer Slide Time: 12:12)



To summarize the backpropagation algorithm, having covered its derivation, the role of the activation function, and the adjustments for learning rate and momentum, along with stopping criteria, here's a concise overview:

Start by initializing the synaptic weights, assuming no prior information. Typically, these weights are drawn from a uniform distribution with a mean of zero and a small variance. This setup ensures that the local reception fields are within the linear portion of the sigmoid activation function. This is a heuristic approach to begin the training process effectively.

When training the neural network, it's essential to present the training samples on a per-epoch basis by shuffling the data points. If you feed the data sequentially, you might introduce a bias based on the order of the samples. To mitigate this, randomize the inputs before presenting them to the network. This randomization helps the network learn more effectively by reducing any inherent biases related to the order of data presentation.

To recap the process, we have two main steps: forward computation and backward computation.

Consider a tuple (x_n, d_n) where x_n is our feature vector and d_n is the desired response vector. In real-world scenarios, such as working with speech signals or images, we extract features to form these feature vectors. For instance, the autocorrelation coefficients or the Linear Prediction Coefficients (LPC) from a windowed speech signal can serve as feature vectors. Similarly, for images, you might compute statistics at various orientations. Raw signals are data-intensive, but by extracting and using these features, you can compactly represent the signals, treating each feature vector as a data point for the neural network.

For each feature vector x_n, there is a corresponding desired response vector d_n. These are inputs to the neural network. During the feedforward phase, we compute the induced local fields and functional signals layer by layer.

Specifically, for a layer l and a neuron j within this layer, the value v_j^l at time instant n is calculated as:

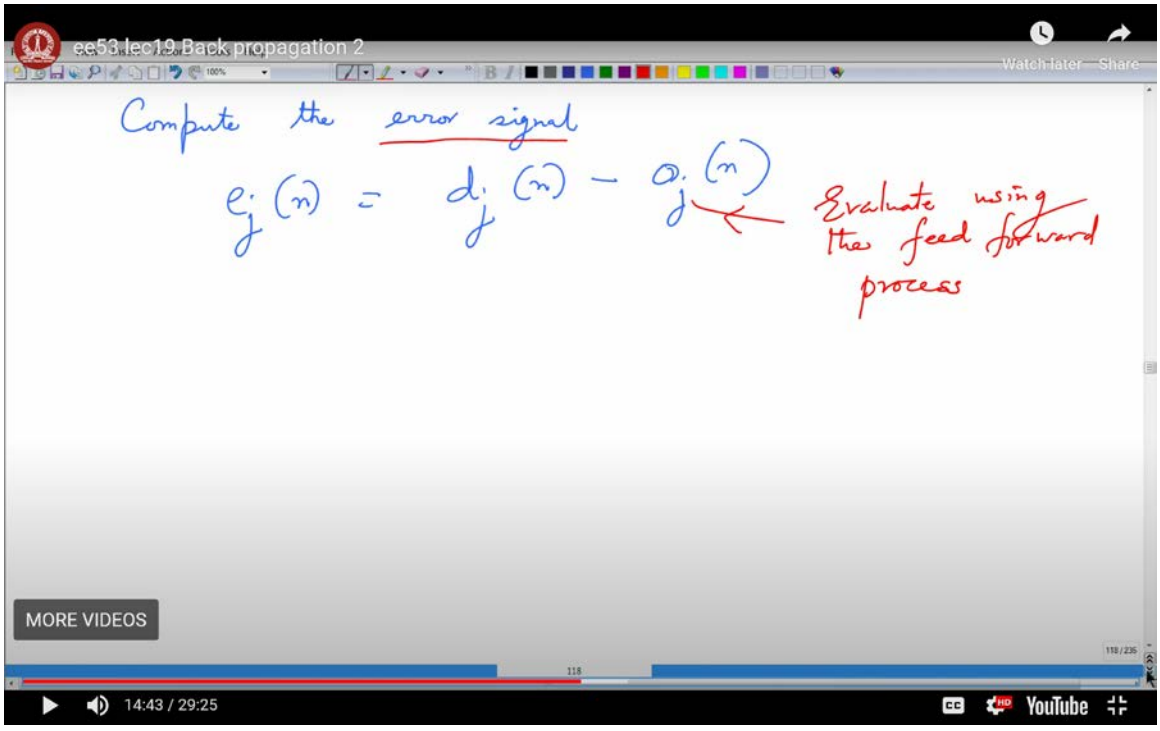
$$v_j^l(n) = \sum_{i=1}^{m_{l-1}} w_{ji}^l \cdot x_i^{l-1}(n)$$

where m_{l-1} represents the number of neurons in the previous layer (excluding the bias). This means that all functional signals from neurons i in layer l-1 are scaled by their respective synaptic weights.

For the output layer l = L, the output y_j^L is denoted as O_j, where O_j represents the output of the neuron j in layer L. We use this notation because it aligns with the activation functions we discussed, making it easier to compute local gradients.

In the input layer l = 0, the input y_j^0 is simply x_j, which are the raw inputs excluding the bias. Biases are treated separately and are initialized to 1. For all layers from 0 through L-1, the 0th coordinate is hardwired to 1 to represent the bias term.

(Refer Slide Time: 14:43)



The bias weight w_{j0}^{L} in layer L at time instant n corresponds to the bias b_j. This weight connects the constant input of 1 through the bias weight w_{j0}^{L}. In formula terms, we denote this as b_j^L, where b_j^L is the bias value for neuron j in layer L.

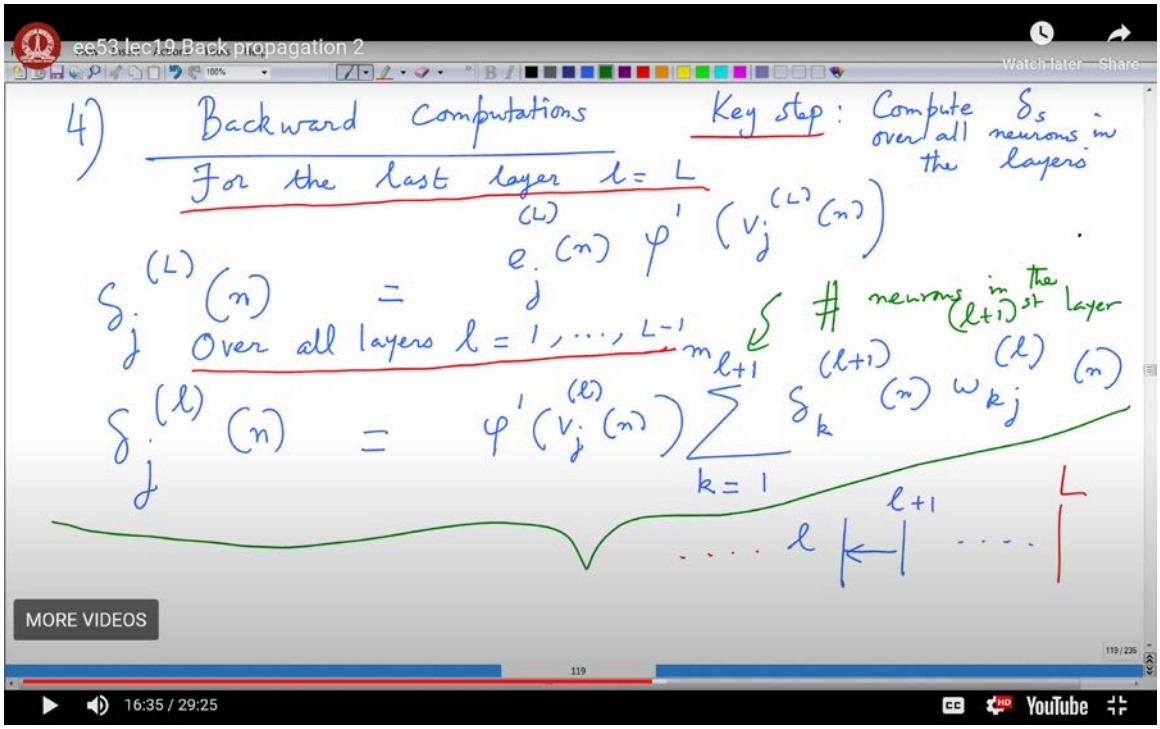
Finally, the error signal $E_j(n)$ is computed as:

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

where d_j is the desired response vector, and O_j is the actual output of the network.

In the feedforward computation, you can compute the outputs O_j by progressing through each layer until you reach the final layer. The errors are then calculated based on the outputs obtained from this forward pass.

(Refer Slide Time: 16:35)



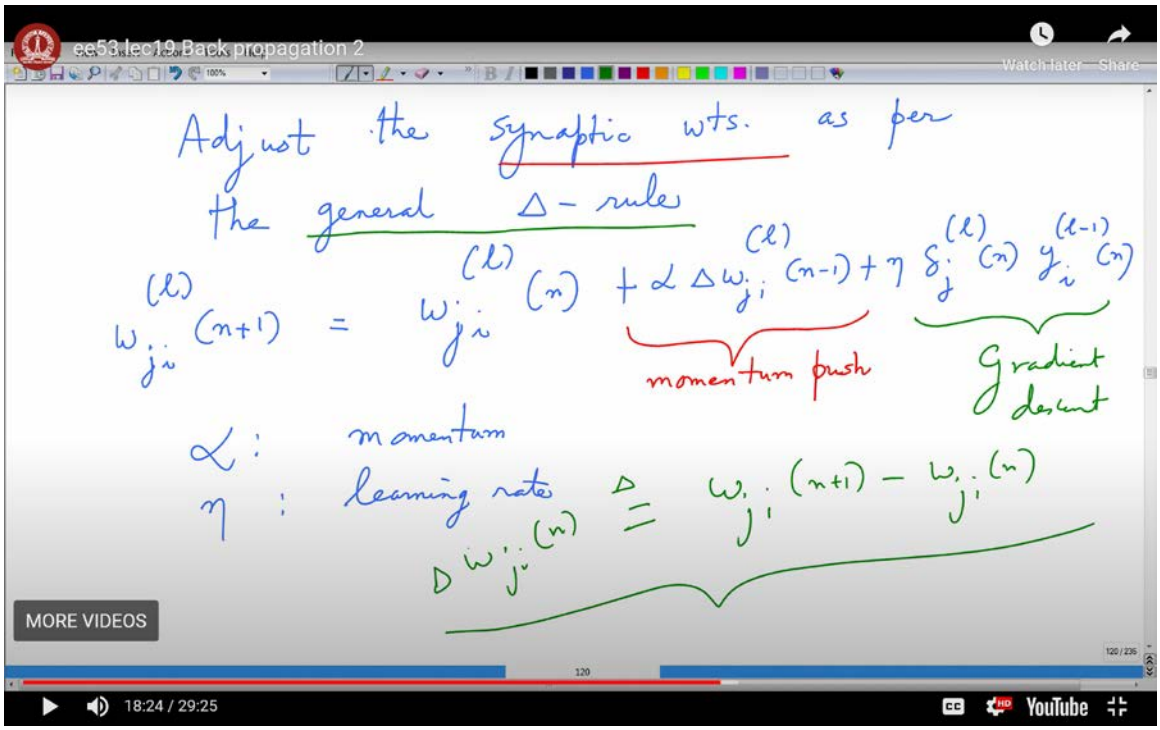
When we move to the backward computation, the process begins at the final layer, L. Specifically, we start with layer l = L and compute the local gradients. These gradients are determined by multiplying the error observed at node j in layer L by the derivative of the

activation function, evaluated at the local reception field for node j in layer L. This is a straightforward calculation.

For layers from 1 to L-1, the local gradients are computed by scaling the gradients from layer L+1 with the synaptic weights connecting nodes in layer L+1 to nodes in layer L. You sum these contributions from all nodes k in layer L+1 to obtain the gradient for each node j in layer L. The formula involves summing from k = 1 to m_{L+1}, where m_{L+1} is the number of neurons in the (L+1)-th layer. This gradient is then scaled by the derivative of the activation function evaluated over the local reception field at node j.

For a single-layer network, if L was 1 and L+1 was 2, this process would be simpler. However, for any number of layers, you follow these steps precisely. Start at the final layer L and move backwards through the layers until you reach the first layer.

(Refer Slide Time: 18:24)



Once you have computed the deltas (gradients), you can adjust the synaptic weights. In gradient descent, this adjustment involves the local gradient times the functional signal (input). When using momentum, you incorporate the momentum parameter α and adjust

based on the synaptic weights from the previous time step. Specifically, the update rule for the weight w_{ji} is given by:

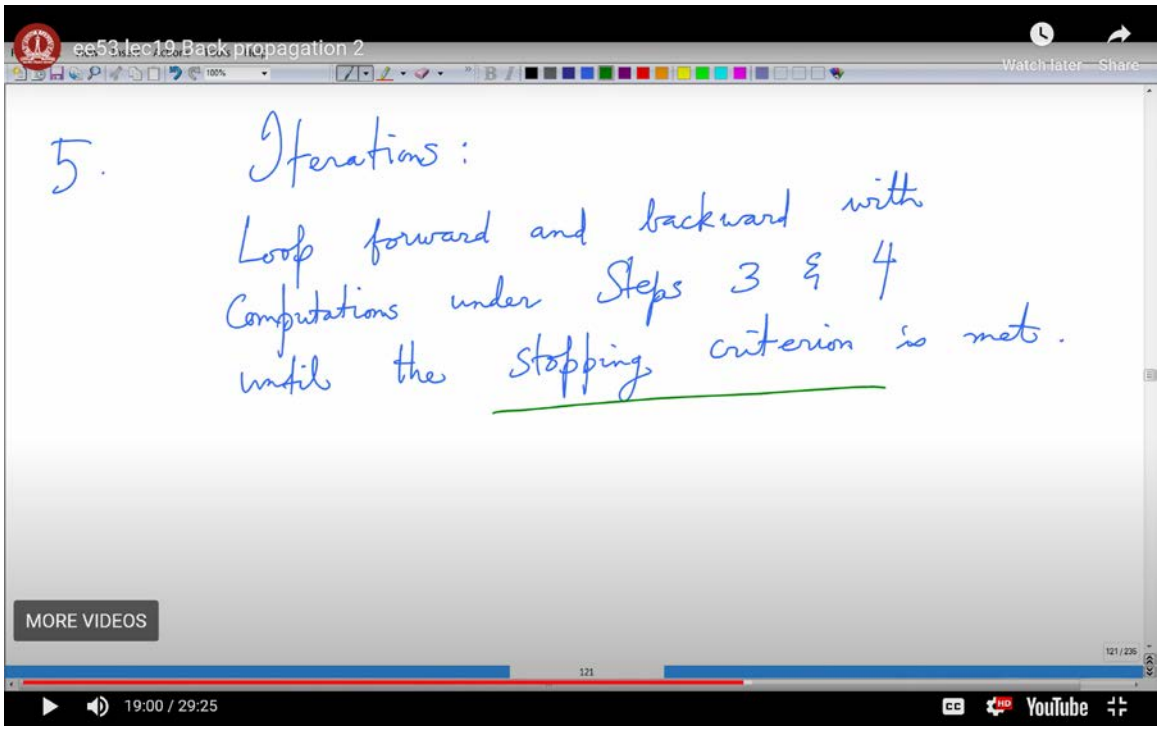
$$\Delta w_{ji}(n+1) = \alpha \Delta w_{ji}(n) + \eta \cdot \text{gradient}$$

Here, $\Delta w_{ji}(n)$ is the change in weight from the previous iteration, and η is the learning rate.

You continue iterating through the forward and backward computations until a stopping criterion is met. The stopping criteria could include a fixed number of iterations, reaching an epsilon tolerance in the error, or ensuring that the norm of the weight differences across iterations is within an epsilon threshold, where epsilon is a positive quantity.

Choose the stopping criterion that best suits your needs, but be mindful of the details of the variables involved. With the backpropagation algorithm derived and the computational steps understood, we can now explore heuristics to enhance the performance of the backpropagation algorithm.

(Refer Slide Time: 19:00)

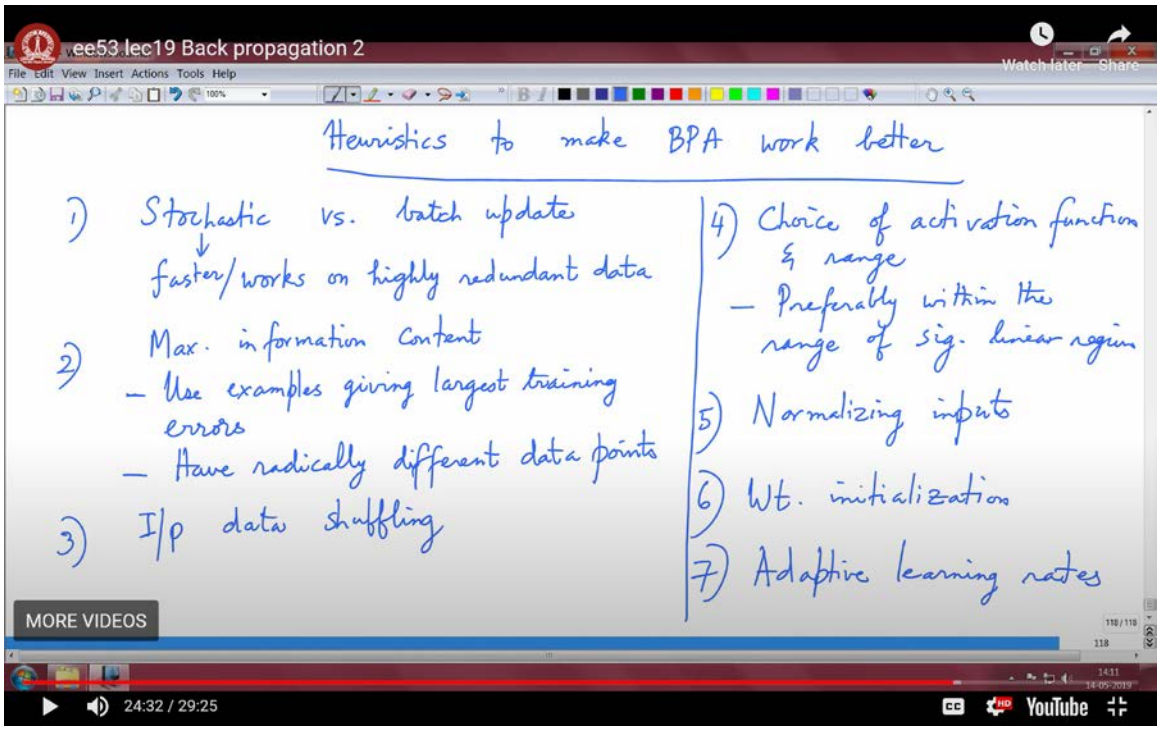


Here are some general guidelines and heuristics to consider:

1. Stochastic vs. Batch Updates: The choice between stochastic and batch updates depends on the redundancy of your data. Stochastic algorithms are advantageous when dealing with large datasets and can handle redundancy effectively. On the other hand, batch updates, which incorporate all errors from the training examples, are more refined and may provide better accuracy. The decision ultimately hinges on the specific conditions and goals of your algorithm.

2. Maximizing Information Content: When selecting examples for training, it is beneficial to include those that exhibit the largest errors. These are often the outliers, and capturing their information helps ensure that the network learns from the most challenging and diverse data points. This approach can significantly improve the robustness of the network.

(Refer Slide Time: 24:32)



3. Input Data Shuffling: Shuffling data before presenting it to the network helps prevent any unintended bias that may arise from the order of the training samples. By randomizing

the order of data samples, you introduce more entropy into the learning process, which can enhance the network's ability to generalize and adapt effectively.

4. Choice of Activation Function and Range: It is generally preferable to operate within the sigmoidal or linear regions of the activation function to avoid issues with saturation. Activation functions like sigmoid or hyperbolic tangent should be chosen based on their performance with your specific system. In convolutional neural networks (CNNs), for example, the Rectified Linear Unit (ReLU) is often preferred over sigmoid or tanh due to its efficiency and effectiveness in deep networks. We will discuss these choices in more detail when we cover CNNs.

5. Normalizing Inputs: Normalization of input data is crucial, especially when data vectors vary significantly in magnitude. For instance, if one vector ranges from 0 to 1000 while another spans from 10,000 to 100,000, normalization ensures that all inputs are scaled similarly. This prevents large discrepancies in updates and helps maintain consistent learning dynamics by preserving the direction of the data.

6. Weight Initialization: Weights should be initialized from a distribution with a mean of 0 and a variance that is compatible with the activation function’s range. While uniform distribution is a common choice, other distributions like Gaussian might also be used, depending on what works best for your system. Initializing weights randomly, rather than setting them all to zero, is important to avoid symmetry and ensure effective learning.

7. Learning Rates: The learning rate is a critical parameter in training neural networks. It is typically set to a small constant value between 0 and 1. While a constant learning rate is common, experimenting with different rates can help you find the optimal setting for your specific application.

These heuristics provide a foundation for optimizing neural network training, but it is essential to explore and adapt these guidelines based on the specific needs and constraints of your system.

When considering the learning rate for neural networks, a common choice is a constant value, typically between 0.01 and 0.05, depending on the specific conditions and dataset.

However, to achieve better convergence, you might want to explore adaptive learning rates. These allow each neuron to adjust its learning rate based on the data it processes and the conditions it encounters. Adaptive learning rates enable individual neurons to align more quickly with the optimal solution, enhancing overall performance.

In practice, applying these heuristics can significantly improve your algorithm's efficiency. As you code and experiment with your neural network, you should try various heuristics to determine which ones work best for your particular system. This summary provides an overview of the backpropagation algorithm and the computational steps involved. Following these steps will allow you to implement and test the algorithm effectively.

The most effective way to master any algorithm is through hands-on experience. I recommend experimenting with data, writing your own code, and using tools available in MATLAB, Python, and other environments. By coding the algorithm yourself, you gain a deeper understanding of the variables and dynamics, which is crucial for thorough analysis and simulation.

Let’s delve into the XOR problem, a classic example used to illustrate neural network concepts. An XOR gate is a fundamental Boolean gate with two inputs. Here’s the truth table for a two-input XOR gate:

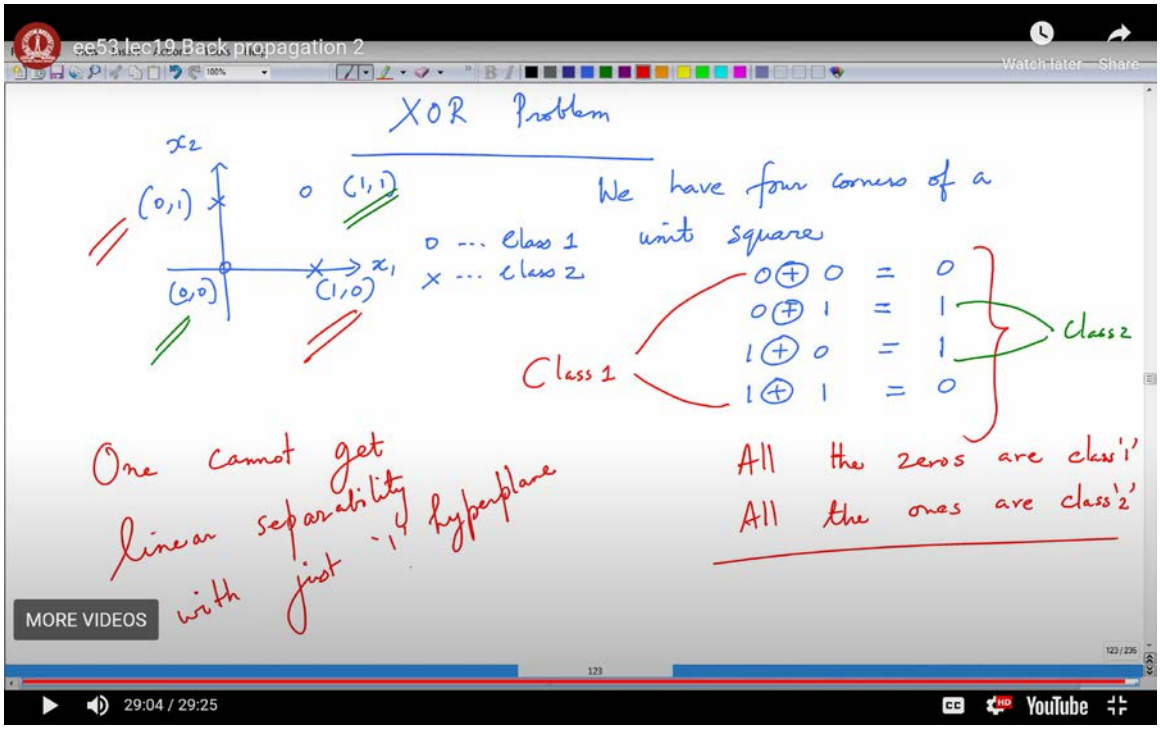
- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0

We categorize the outputs as follows: outputs of 0 form one class (Class 1), and outputs of 1 form another class (Class 2). To visualize this, we plot the points on a two-dimensional plane with x_1 and x_2 as the axes:

- (0, 0) and (1, 1) yield 0 (Class 1)
- (0, 1) and (1, 0) yield 1 (Class 2)

Class 1 is represented by the points (0, 0) and (1, 1), while Class 2 is represented by (0, 1) and (1, 0). On the plot, Class 1 points can be marked with one color, and Class 2 points with another.

(Refer Slide Time: 29:04)



The challenge here is that a single linear hyperplane cannot separate these classes. In other words, you cannot draw a straight line that perfectly divides Class 1 from Class 2. This limitation shows that a single-layer perceptron cannot solve the XOR problem due to its inability to handle non-linear separability.

To address this, we employ a multilayer perceptron (MLP) with hidden layers. By incorporating hidden neurons, the MLP can learn to classify the XOR problem effectively. We will explore the detailed solution to the XOR problem using the multilayer perceptron and then generalize this approach to problems with three variables.