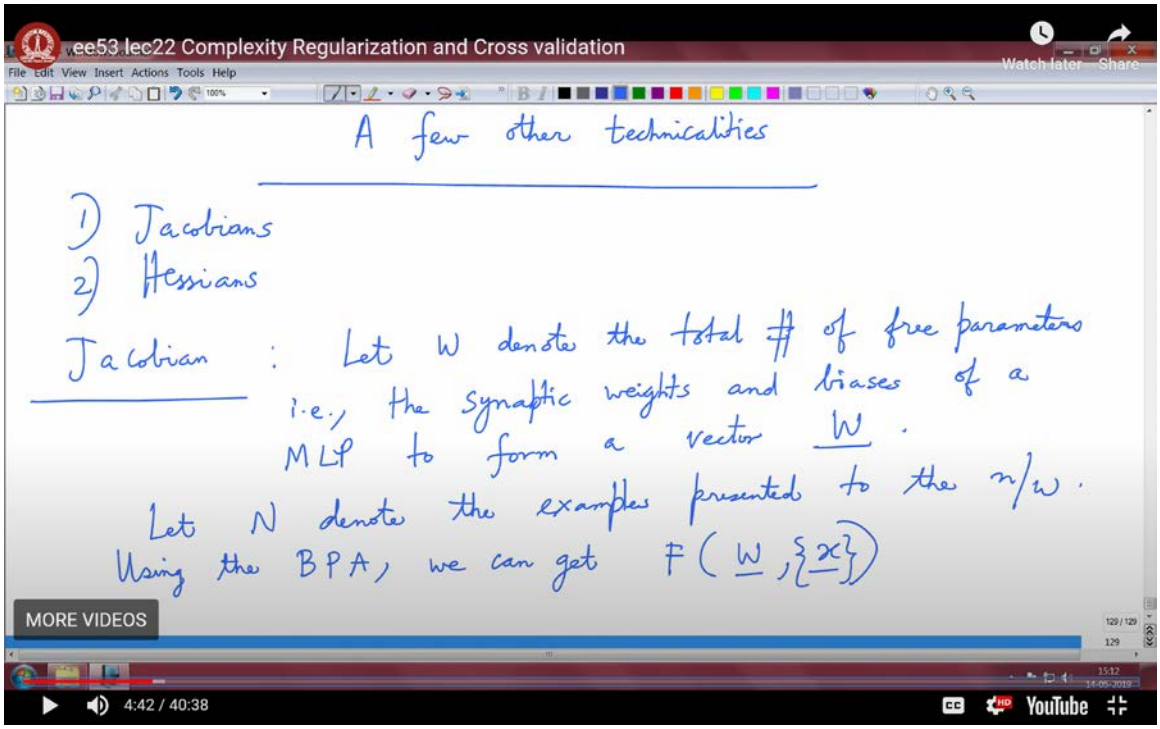
Neural Networks for Signal Processing-I
Prof. Shayan Srinivasa Garani
Department of Electronic System Engineering
Indian Institute of Science, Bengaluru

Lecture – 22
Complexity Regularization and Cross Validation

Let's delve into some additional technical details, starting with Jacobians and Hessians. We are dealing with an approximating function that depends on both the synaptic weights of the multilayer perceptron and the input data samples. Our goal is to compute the partial derivative of this approximating function, which is a function of both the synaptic weights and the input data points, with respect to the synaptic weights. This process yields the Jacobian matrix.

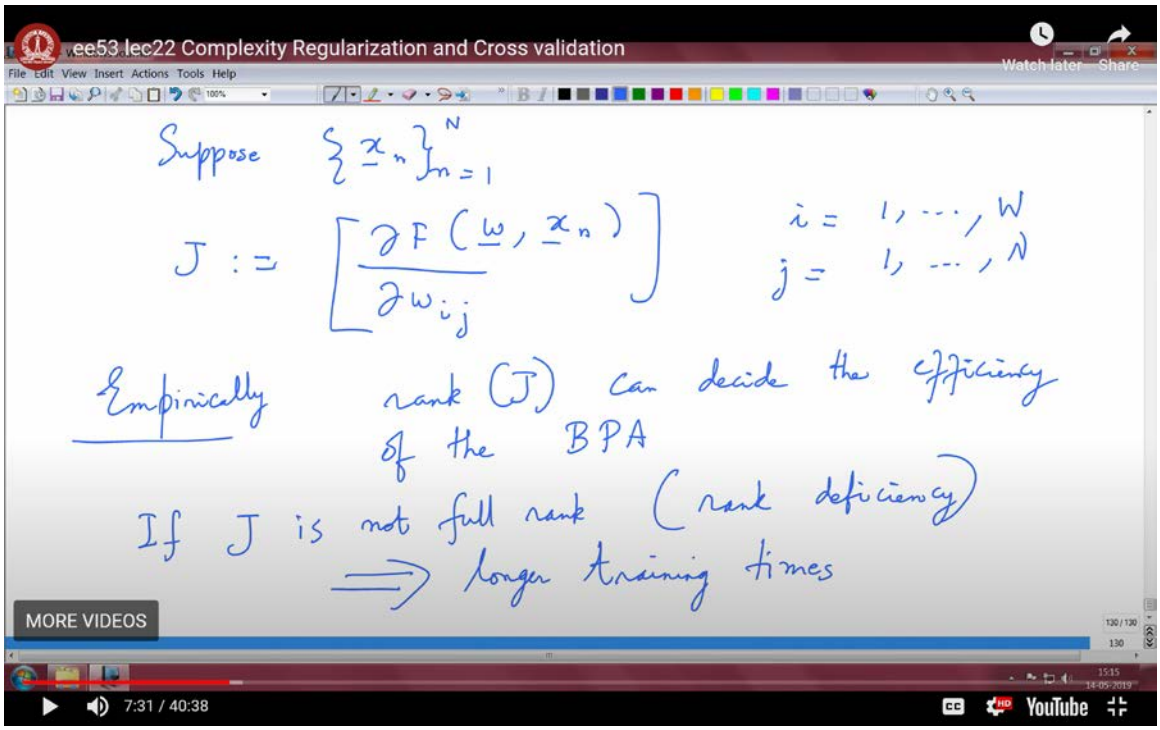
(Refer Slide Time: 04:42)



To formalize this, let W represent the total number of free parameters, including the synaptic weights and biases of the multilayer perceptron, which we denote as $\overline{W}$. It's important to note that $\overline{W}$ should not be confused with the total number of free parameters, and we may choose a different notation for clarity, but for now, we'll use W.

You might wonder why Jacobians and Hessians are important. The Jacobian matrix's rank can significantly impact the performance of the backpropagation algorithm. Any rank deficiency in the Jacobian matrix can cause the backpropagation algorithm to obtain only partial information about potential search directions during gradient descent. This limitation can lead to longer training times. Therefore, understanding the Jacobians is crucial for optimizing algorithm performance.

(Refer Slide Time: 07:31)



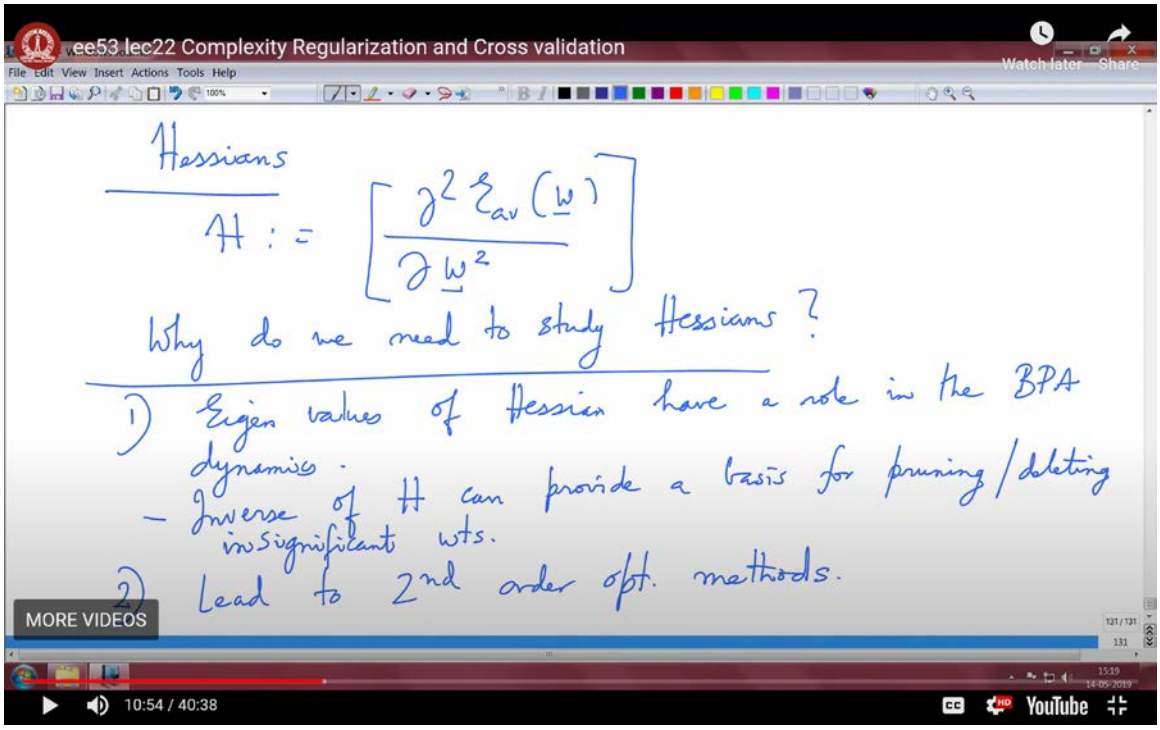
Continuing with our formalism, let N denote the number of examples presented to the network. Using the backpropagation algorithm, we derive the approximating function F, where F represents the approximating function and f denotes the true function. The approximating function F depends on the synaptic weights $\overline{W}$ and the inputs x.

Assuming we have a set of data points x_n for n = 1 to N, if we are performing stochastic or online learning, we present a data vector x at each time step n. This sequence forms our set of data points.

Let's discuss the Jacobian and Hessian matrices in more detail.

The Jacobian matrix is computed as follows: it represents the partial derivatives of the approximating function with respect to the synaptic weights W_{ij}, where i ranges from 1 to W, the total number of free parameters, and j ranges from 1 to N, the number of examples. The rank of the Jacobian matrix is crucial for the efficiency of the backpropagation algorithm. If the Jacobian matrix is not full rank—meaning it is rank deficient—this can lead to longer learning times. This inefficiency occurs because the algorithm may search in suboptimal directions, which can prevent convergence to a solution. Hence, understanding the Jacobian matrix is a critical technical consideration.

(Refer Slide Time: 10:54)

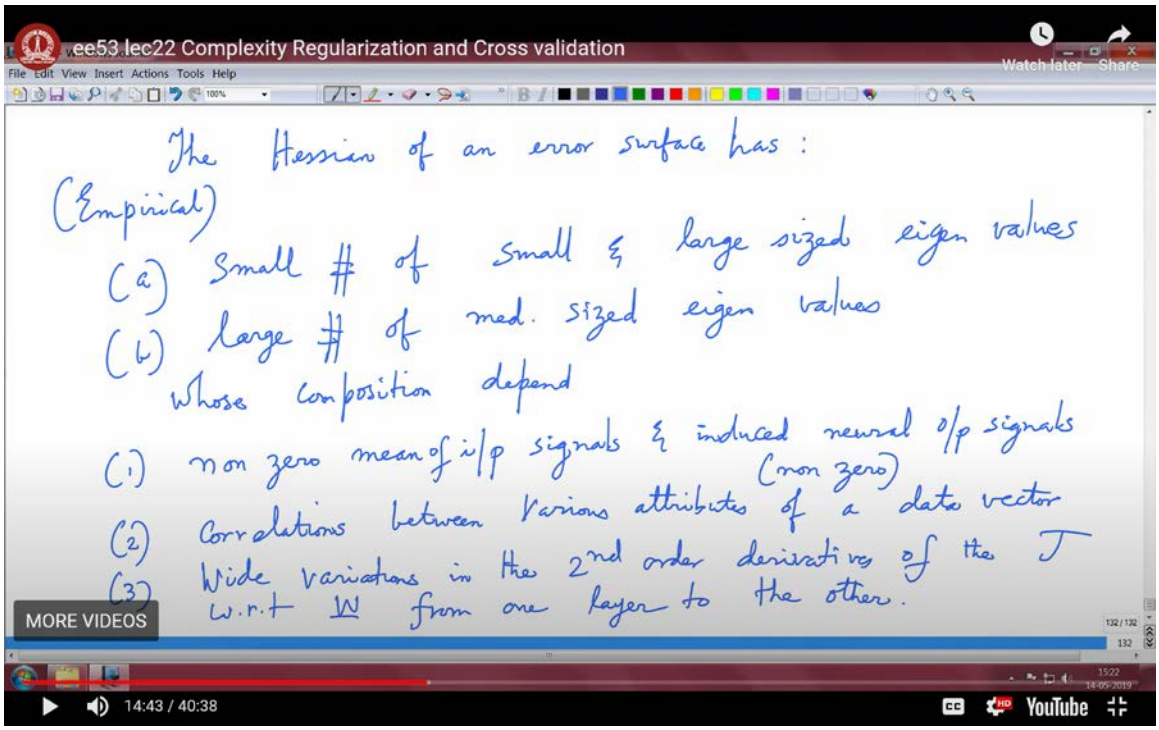


Similarly, we need to consider the Hessian matrix, which is the second-order derivative of the cost function with respect to the weights. The study of Hessians is important because they provide insights into second-order effects. The eigenvalues of the Hessian matrix play a significant role in the dynamics of backpropagation. Specifically, the inverse of the Hessian can be used for pruning or deleting insignificant weights. By performing an eigen decomposition on the inverse of the Hessian, we can identify and remove weights that do

not significantly contribute to the network, which in turn can simplify the network and reduce both computational and spatial complexity.

This approach can lead to second-order optimization methods, which go beyond the first-order gradient-based adaptations used in standard backpropagation. While traditional backpropagation involves adjusting weights based solely on first-order gradients, incorporating Hessians allows us to account for second-order effects, potentially leading to more optimized network training. Thus, understanding and leveraging Jacobians and Hessians can significantly enhance the efficiency and effectiveness of neural network training.

(Refer Slide Time: 14:43)



There are additional technical aspects to consider when dealing with Hessians. The Hessian matrix of an error surface generally contains a range of eigenvalues. Empirically, it tends to exhibit a few very small and very large eigenvalues, while the majority of eigenvalues are of medium size. These eigenvalues are influenced by various factors, including the distribution of the data, the non-zero means of input signals, and the biases in the neural

output signals. Moreover, there could be correlations between different attributes of a data vector, which also impact the eigenvalue distribution.

Additionally, the second-order derivatives of the cost function with respect to the weights can vary significantly from one layer to another. For instance, the output layer might learn faster than the hidden layers, leading to dynamic learning behaviors as errors propagate through different layers. Understanding the distribution of eigenvalues is crucial because, in linear dynamical systems, the system’s behavior is largely determined by its largest eigenvalue. For example, if you have two eigenvalues, 2 and 0.5, the system's response will be dominated by the largest eigenvalue, 2.

Thus, studying the eigenvalue distribution of the Hessian matrix is essential. By knowing how these eigenvalues are influenced by factors such as non-zero biases in input and output signals, and correlations between data elements, we can employ second-order optimization techniques to accelerate convergence. While traditional techniques rely on gradients for optimization, incorporating Hessians can enhance the efficiency of convergence, providing a more refined approach to optimizing the network.

Next, we will discuss complexity regularization. When designing learning networks, it's important to recognize that they are fundamentally statistical in nature. A neural network operates as a statistical engine, heavily reliant on data points and governed by the dynamics of the optimization function used in the learning process.

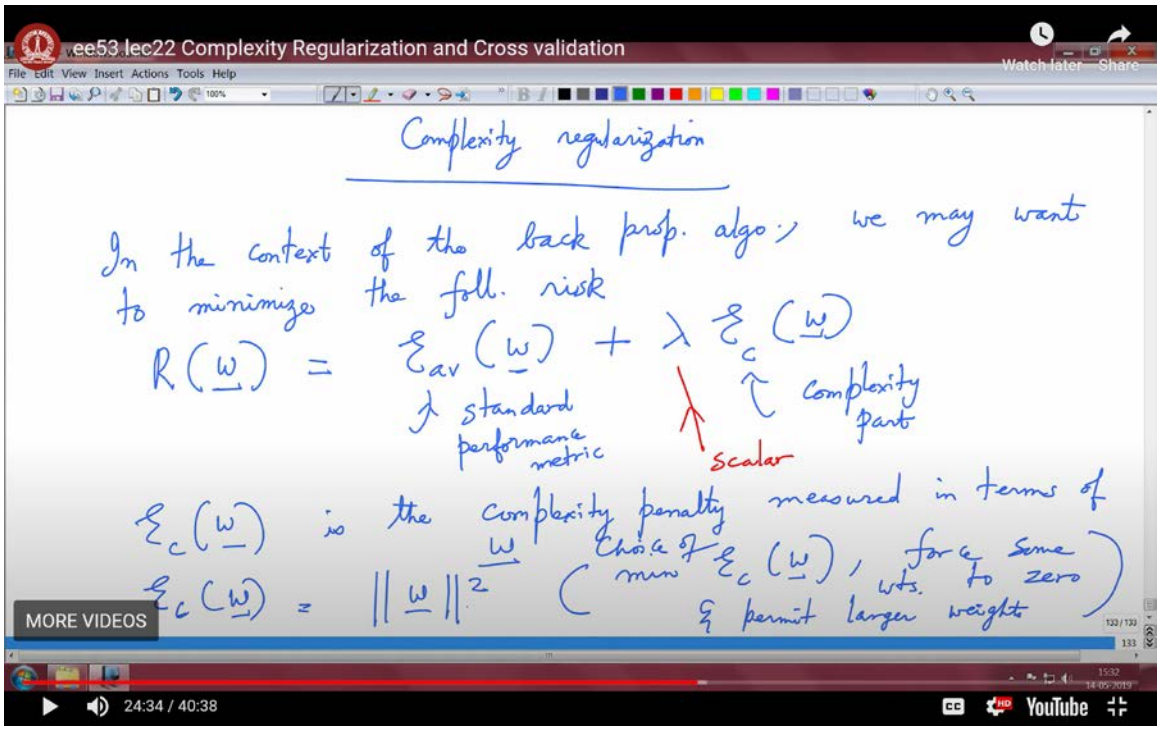
We've already discussed gradient-based methods and explored some technical aspects related to Jacobians and Hessians, including how second-order optimization techniques can accelerate convergence. One crucial aspect of neural network design, particularly in our context, is determining the optimal network complexity. This is essential because, when considering data, two key factors come to mind: the complexity of the network and the learning efficiency of the network. We aim for a balance between rapid learning times and a minimal number of parameters. A more complex network increases both space and time complexities, which can pose implementation challenges in hardware systems or neuromorphic systems.

Let’s delve into the concept of complexity regularization and its fundamental principles. In the context of the backpropagation algorithm, we want to minimize a risk function, R, which depends on the network parameters. This risk function can be decomposed into two components:

1. The Average Error: This component measures the standard performance of the network, reflecting how well the network is performing in terms of accuracy, such as squared error or cross-entropy.

2. The Complexity Term: Represented as E_C in terms of the weights, this component addresses the complexity of the network. We introduce a scalar parameter, λ, to balance the trade-off between optimizing the average error and minimizing the complexity.

(Refer Slide Time: 24:34)



When λ is set to 0, we focus solely on the standard performance metric, such as the squared error, which evaluates the difference between the predicted and actual outputs at the output layer. This is our primary performance metric.

On the other hand, if λ is set to 1, the focus shifts entirely to the complexity term, which is not practical since an excessively low complexity might reduce the network to a trivial form, like a single neuron.

Instead, we aim to balance both metrics: the standard performance metric and the complexity metric. For instance, $E_C(W)$ represents the complexity penalty in terms of the network weights W. One common approach for $E_C(W)$ is the squared norm of the weights. By setting an appropriate threshold and minimizing this complexity function over all weights, we can enforce some weights to be zero while allowing larger weights to remain, thereby controlling network complexity effectively.

In this approach, we are essentially pruning weights that do not contribute significantly to the network's performance, effectively weeding out "dead neurons." You might wonder how to choose the complexity functional. In any optimization problem, the choice of what to optimize is guided by the specifics of the problem at hand. The square of the norm of the weights is one meaningful choice for this functional, but other metrics can also be considered depending on the application. The key point is that the regularization parameter λ plays a crucial role; it balances the trade-off between the standard performance metric and the complexity component. By selecting an appropriate complexity functional, you can fine-tune the network's performance.

Now, let's explore alternative strategies for learning based on second-order updates. Up to this point, we've discussed the qualitative aspects—why Hessians and Jacobians are important. However, we haven't yet expanded the performance function, specifically our squared error function, to include second-order and higher-order terms. To understand how learning can be influenced by these terms, let's start by expanding the performance function $E_{\text{avg}}(W + \delta W)$ through a Taylor series.

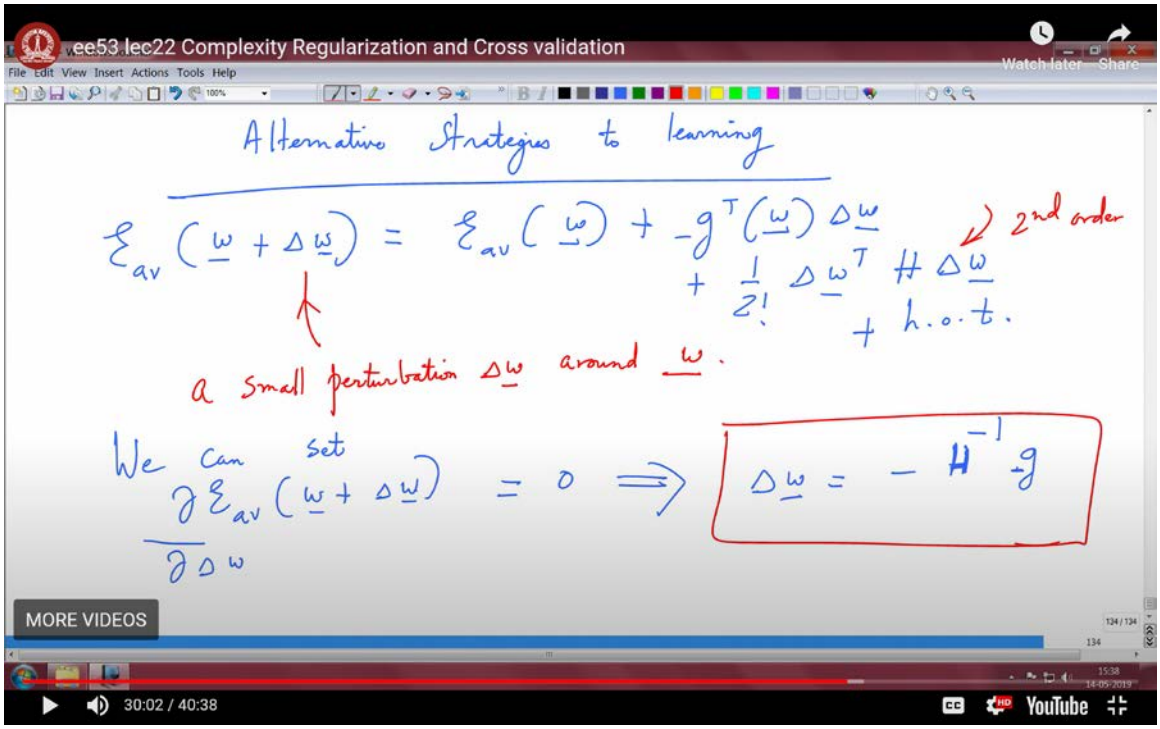
The expansion can be written as follows:

$$E_{\text{avg}}(W + \delta W) = E_{\text{avg}}(W) + \nabla E_{\text{avg}}(W) \cdot \delta W + \frac{1}{2}(\delta W)^T H(\delta W) + \text{higher-order terms}$$

Here, $E_{\text{avg}}(W)$ represents the performance function at the weight vector W, $\nabla E_{\text{avg}}(W)$ is the gradient of the performance function with respect to the weights, and H denotes the Hessian matrix. This expansion considers a small perturbation δ W around the weight vector W. To find the optimal perturbation δ W, we set the partial derivative of this expanded error function with respect to δ W to zero. Solving this using matrix calculus yields:

$$\delta W = -H^{-1}\nabla E_{\text{avg}}(W)$$

(Refer Slide Time: 30:02)



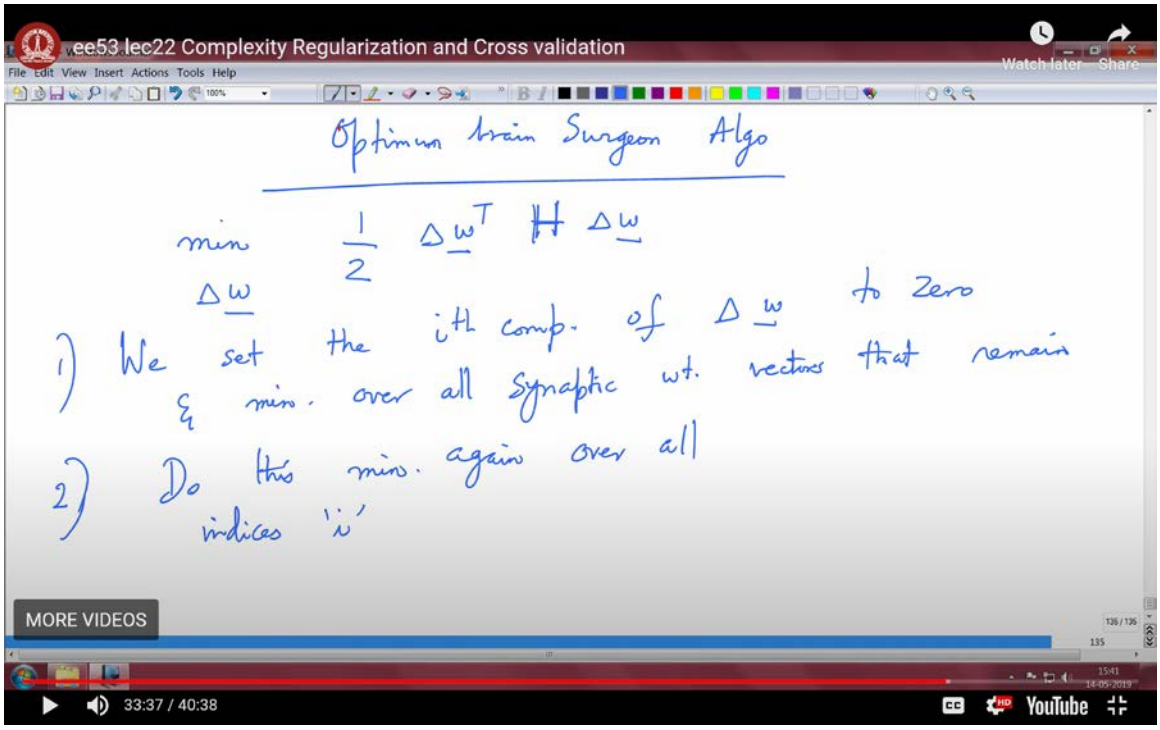
In this equation, H is the Hessian matrix (written in bold to denote a matrix), and $\nabla E_{\text{avg}}(W)$ is the gradient vector. This provides a way to update weights based on second-order statistics. Higher-order terms can also be considered for better approximations, but in practice, we often stick to second-order statistics due to their manageable complexity and effectiveness.

Additionally, there's an algorithm known as the Optimum Brain Surgeon algorithm, which has an intriguing name. The idea is quite straightforward: to optimize the function involving δ W, we aim to minimize:

$$\frac{1}{2}(\delta W)^T H(\delta W)$$

This approach involves a two-level optimization strategy. First, minimize the function over the synaptic weight vectors while keeping the ith weight vector fixed at zero. This method provides a refined approach to optimizing the network by focusing on significant weight adjustments and effectively managing complexity.

(Refer Slide Time: 33:37)



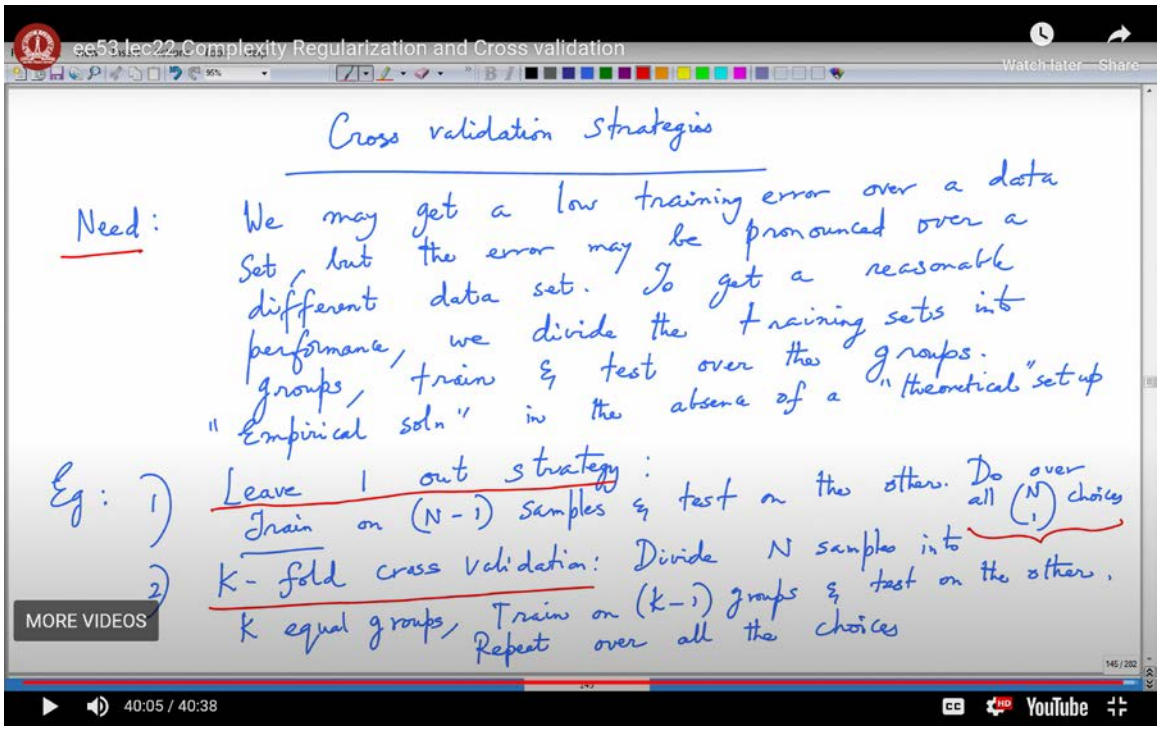
In this approach, we first set the i-th component of δ W to zero. We then minimize the error over the remaining weight vectors with this component fixed at zero. This process involves optimizing over all the other weight vectors after setting the i-th component to zero. We repeat this process for each index i, allowing us to determine the optimal choice of δ W. Essentially, this two-step optimization process involves setting one component to zero,

optimizing the remaining weights, and then iterating this procedure for all components to identify the best weight configuration.

This approach, based on the Hessian, helps us update the differential of the weight vector, which can be useful for refining our backpropagation algorithm with the delta rule, especially when using second-order methods like the SCM.

To summarize, we derived the backpropagation algorithm from first principles in our earlier discussions, focusing initially on gradient-based methods. We then explored the roles of Jacobians and Hessians, discovering that the inverse of the Hessian matrix is particularly valuable due to its influence on the distribution of eigenvalues. This insight helps us retain useful eigenvalues while pruning those that do not impact system performance, contributing to complexity regularization.

(Refer Slide Time: 40:05)



We also discussed expanding the performance function using higher-order statistics. Restricting ourselves to second-order statistics involves expanding the performance function up to the second order using a Taylor series and solving for the optimal δW using

matrix calculus. The Optimum Brain Surgeon algorithm offers a two-step optimization procedure, which can be beneficial for refining weight updates.

While these methods are useful, it is essential to remember that unlike linear signal processing techniques, which often involve straightforward matrix computations, non-linear signal processing techniques like those used in neural networks require additional considerations. Solutions based on gradient or Hessian calculations alone are not sufficient due to the complexity of network parameters. Heuristics and update rules, such as the momentum learning rule, are necessary for effective optimization, even though they cannot always be derived from first principles.

As you work with these techniques, it's crucial to keep these considerations in mind. The tools and strategies discussed should aid in your research and practical applications. Finally, when dealing with a dataset, it's important to partition it into training and testing samples using strategies like cross-validation. This approach helps address potential overfitting by ensuring that the model generalizes well to unseen data.

When applying this approach to different datasets, the results may not always be optimal due to inherent anomalies caused by varying feature vector observations across datasets. Different datasets can lead to different feature vectors, which may affect the neural network's ability to generalize. Thus, a model trained on one dataset may not perform well on another. To address this issue, we employ a strategy known as cross-validation.

There are two common cross-validation methods. The first is the leave-one-out strategy. In this method, if you have n samples, you train the model on n - 1 samples and test it on the remaining one. This process is repeated for each sample, so you perform n iterations. This approach is feasible because n is the number of possible choices. However, if you opt to choose k samples out of n for training and test on the remaining n - k samples, the number of possible combinations, $\binom{n}{k}$, can be quite large, especially when both n and k are large. This combinatorial complexity can make this method challenging to handle.

An alternative is k-fold cross-validation. In this approach, you divide the n samples into k equal groups. You then train the model on k - 1 of these groups and test it on the remaining

one. This process is repeated for each possible grouping, ensuring that every group is used for testing at least once. K-fold cross-validation is a practical method used in the absence of a more rigorous theoretical framework for data partitioning. It helps in empirically determining how well the model generalizes across different subsets of the data.

In summary, cross-validation is a crucial step to ensure that your algorithms generalize well in practice. It helps in assessing the model's performance on various data subsets and is essential for robust model evaluation.