MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SysEng 5212 /EE 5370
Introduction to Neural Networks and Applications

Week 6 : Backpropagation II and Practical Implementation Considerations

Cihan H Dagli, PhD

Professor of Engineering Management and Systems Engineering
Professor of Electrical and Computer Engineering
Founder and Director of Systems Engineering Graduate Program

dagli@mst..edu

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY
Rolla, Missouri, U.S.A.

SYSTEMS ENGINEERING
Research Center
Department of Defense UARC

Smart Engineering Systems Lab
Engineering Management and Systems Engineering Department

Volunteers Tally

- Murat Aslan
- Tatiana Cardona Sepulveda
- Prince Codjoe
- Xiongming Dai
- Jeffrey Dierker

Volunteers Tally

- Venkata Sai Abhishek Dwivadula
- Brian Guenther
- Anthony Guertin
- Timothy Guertin
- Seth Kitchen

Volunteers Tally

- Gregory Leach
- Yu Li
- John Nganga
- Igor Povarich
- Jack Savage

Volunteers Tally

- William Symolon
- Wayne Viers III
- Tao Wang
- Kari Ward
- Julia White
- Jun Xu

Lecture outline

- Midterm exam is on 3/6/2018
  - Open-book + open notes + open Internet, bring your laptops (Don't forget to credit your sources by giving references)
  - Two and half hours from 4:00-6:30 PM US CST
  - Both on and off campus students will download the exam at the same time from Canvas
  - Turn in the completed exam by uploading it on CANVAS as zip file (on campus students can submit non-electronic version of their answers to Deepak in class)
  - Must sign and attach honor code -exam will not be graded without it

方向传播算法

Today

Backprop Review

Practical Issues in Standard Backprop

I. Weight initialization
II. Generalization
III. Network configuration
IV. Universal approximation
V. Curse of dimensionality
VI. Independent validation
VII. Bias-variance dilemma
VIII. Improving convergence speed

Standard Backpropagation Review

Step 1: Initialize the network synaptic weights to small random Values

Step 2: Select a random input/output training sample, present it to the network and calculate the network response

Step 3: Compute the error in the network response by comparing it to the desired response

MATLAB: y = purelin(v)

Backpropagation Review

Step 4: Use the output error to calculate all local error gradients $delta_j(n)$

- If j is an output neuron,

$$\delta_j(n) = e_j(n)\phi_j'(v_j(n))$$

- If j is a hidden neuron,

$$\delta_j(n) = \phi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n)$$

where,

- $e_j(n)$ is the output error
- $\phi_j'(v_j(n))$ is the derivative of the activation function
- $w_{kj}(n)$ is the weight to be updated

Step 5: Update the network weights using the weight update or delta rule

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

where $y_i(n) = x(n)$ for layer 1.

Step 6: Continue steps 2 through 5, until the network reaches a desired level of performance

Practical Considerations in Implementing Backpropagation Learning

1. Initialization of synaptic weights
   a) Nguyen Widrow initialization algorithm
2. Generalization
   a) Overfitting
   b) Network configuration
   c) Training sample size
3. Universal approximation and network configuration
4. Curse of dimensionality
5. Independent validation
6. Early stopping
7. Bias-variance dilemma
8. Complexity regularization
9. Speed of convergence

Initializing Network Weights

- Avoid very large and very small values of initial weights.
- Weight initialization heuristic: $[\frac{-0.5}{m_1}, \frac{0.5}{m_1}]$ where m_1 is the number of neurons in the first hidden layer
- Alternative weight update method for MLPs with one hidden layer: Nguyen-Widrow's initialization algorithm
  - significantly improves speed of network training

Nguyen Widrow Initialization Algorithm

Define, m_0 = number of input nodes; m_1 = number of neurons in the hidden layer

ϒ= scaling factor

Step 1: Compute the scaling factor as,

$$\Upsilon = 0.7 \sqrt[m_0]{m_1}$$

Step 2: Initialize the weights w_{ij} as uniformly distributed random numbers between -0.5 and 0.5

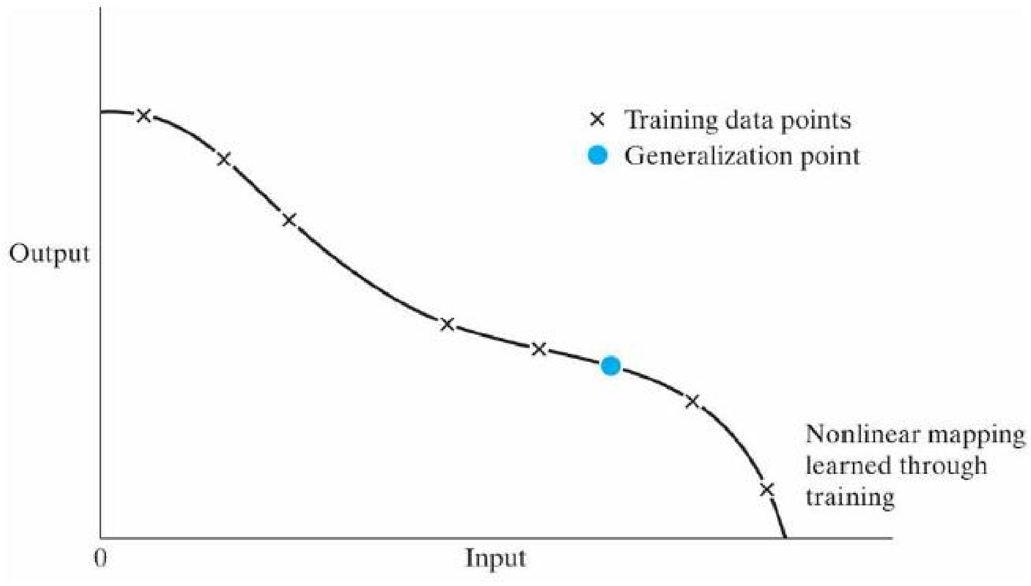
Step 3: Reinitialize weights using,

$$w_{ji} = \gamma \frac{w_{ji}}{\sqrt{\sum_{i=1} m_1 w_{ji}^2}}$$

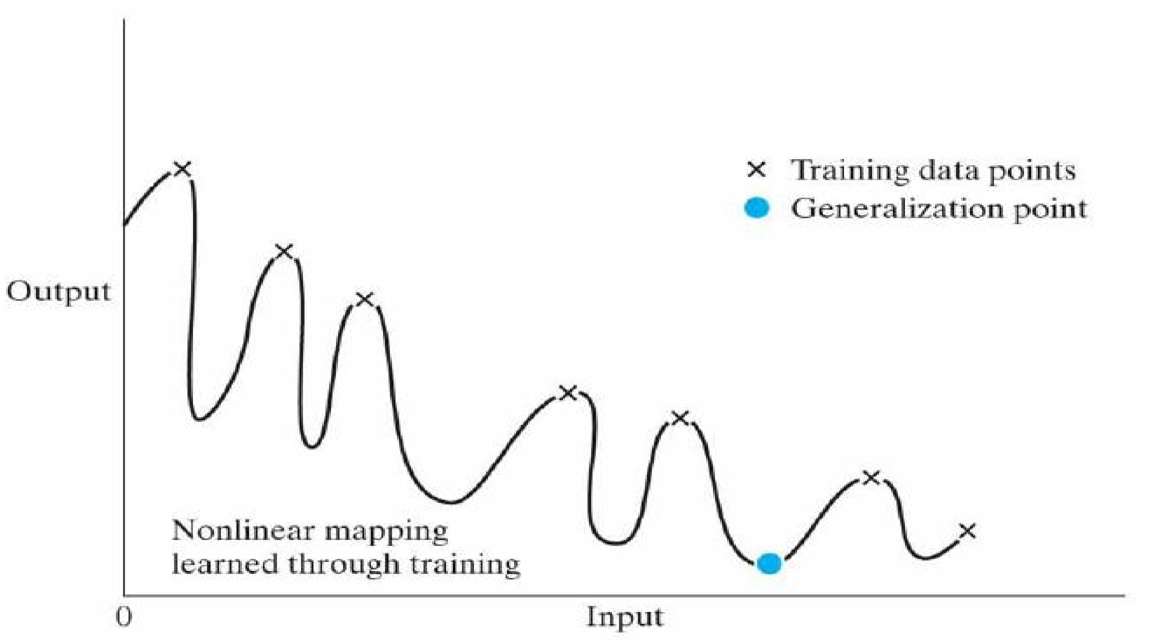
Step 4: For the jth neuron in the hidden layer, set the bias to be a random number between w_{ij} and - w_{ij}

Generalization

- Training: Encoding input output patterns into the nodes in the form of the synaptic weights
- Generalizing: Correctly computing the response to a never-before-seen input



Overfitting



If the data contains noise, the network will identify features that don't exist in the function being mapped. Network memorizes the data and loses ability to generalize!

More on Overfitting

- Overtrained network acts like a “look-up table" 查出
- Input-output mapping computed by the network is not smooth
- What makes a good mapping?
  - Occam's Razor applies
  - The simplest function mapping the inputs to the outputs is often the best
- The smoothest function is usually the simplest function.

Factors Affecting Generalization

Good generalization depends on three factors,

- Size of the training sample
- Architecture of the neural network
- Complexity of the problem

We can look at the problem from two perspectives,

- Fix the architecture, find a training sample set that will achieve good generalization
- Fix the training sample, determine the best architecture for generalizing with the available data

Sufficient Training-Sample Size

In practice, we can achieve good generalization by selecting a training set that satisfies the condition, $N = O(\frac{W}{\varepsilon})$

where, W is the number of free parameters (weights and biases), ε is the fraction of classification errors permitted, and O(.) is the order of the error fraction.

As a rule of thumb, for good generalization, the number of training examples should be greater than the ratio of the total number of free parameters (m0 m1) to the mean-square error estimate.

Impact of Network Configuration on Performance

Network configuration is determined by,
- Number of hidden layers
- Number of neurons in each hidden layer
- Type of activation functions used

An overdesigned architecture will overt data; network cannot generalize.

Choice of activation function has little impact on performance as long as it is nonlinear
- Output layer functions may be nonlinear or linear

More on Network Configuration

- Determining a sufficient number of hidden neurons is frequently done by trial and error

- MLP with one hidden layer acts as a universal approximator
  - Use of more than two hidden layers is rarely justifiable in terms of performance

Multilayer Perceptron as a Universal Approximator

A multilayer perceptron trained with backpropagation performs a nonlinear mapping of input to the output.

If m_0 is the number of input variables and m_L is the number of output neurons,

- m_0 -dimensional input space→ m_L -dimensional output space

What is the minimum number of hidden layers necessary to successfully approximate any given nonlinear input-output mapping?

- The *universal approximation theorem* answers this question.

Universal Approximation Theorem

Let $\phi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit-hypercube $[0,1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then ,given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, there exists an integer m_1 and set of real constants α_j, b_j, and w_{ji} where $i = 1, \ldots, m_0$ and $j = 1, \ldots, m_1$ such that we may define,

$$F(x_1, \ldots, x_{m_0}) = \sum_{j=1} m_1 \alpha_j \phi(\sum_{ij=1} m_0 w_{ji} x_j + bj)$$

as an approximate realization of the function $f(\cdot)$; that is,

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \epsilon$$

for all $x_1, \ldots, x_{m_0}$ that lie in the input space.

Applying the Universal Approximation Theorem to the MLP

$\Phi(.)$: a nonconstant, bounded, and monotone-increasing continuous function. The sigmoidal nonlinearities as activation functions satisfy this condition

- m_0 -dimensional input space: The network input has m_0 nodes denotes by $x_1, \dots \dots .. x_{m_0}$

integers m_1 and set of real constants α_j , b_i, and w_{ij}: The hidden layer has m_1 neurons; α_j , b_i, and w_{ij} are the output weights, hidden layer bias and weights respectively

MLP with Single Hidden Layer is a Universal Approximator

Stated simply,

The theorem states that a single hidden layer is sufficient for a multilayer perceptron to compute an approximation represented by a given input-output mapping.

The theorem guarantees approximation, not exact representation or optimal implementation!

Practical Considerations In Designing a Multilayer Network

- The universal approximation theorem is not constructive; does not tell us how to determine the most optimal network configuration
- Neurons in a single hidden layer interact with each other
- This makes it difficult to improve the approximation at one point
- without worsening it at some other point

- Research provides some justification for the use of two hidden layers,
  - first hidden layer extracts local features; creates a new feature space
  - Second hidden layer extracts global features partitions new feature space into desired classes
  - Two layers make the approximation much more manageable

Curse of Dimensionality

- Problem complexity increases with increase in number of input dimensions
- Functions with higher-dimensional inputs are much more complex than functions with lower dimensional inputs; Are harder to learn
- The complexity is much more expensive to model
- Modeling outcomes are more likely to be inaccurate
- For accurate outcomes the number of input samples N needs to be,

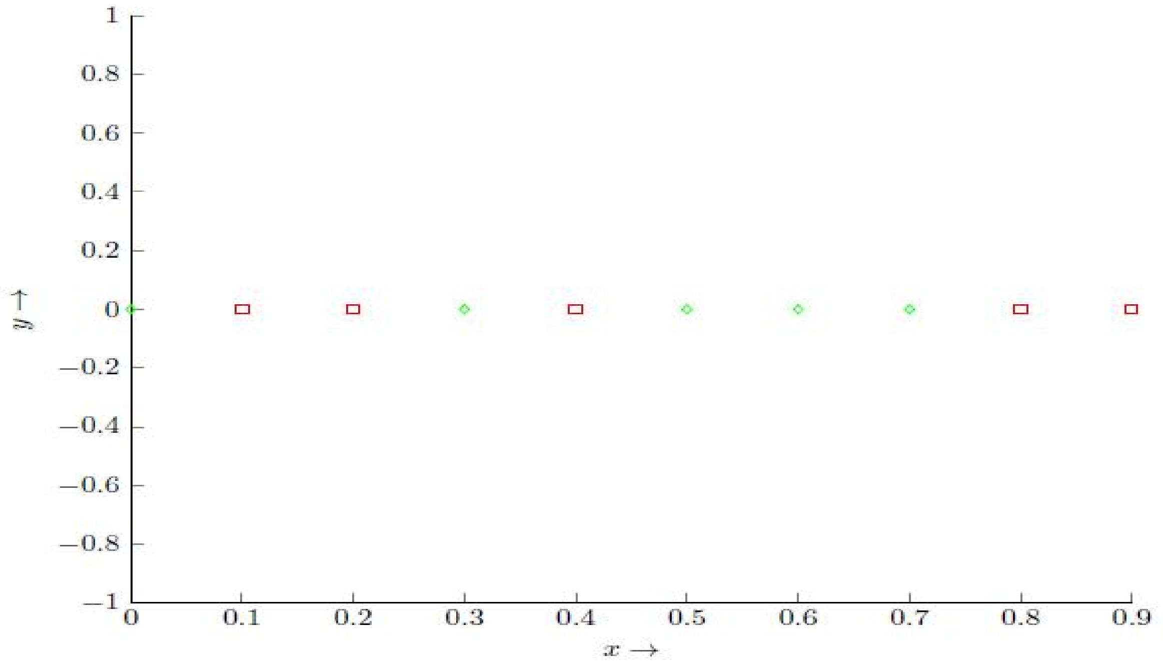
$$N \gg m_{0^2}$$

where m_0 is the dimensionality of the input.

- Otherwise, the model is too complicated for the data which leads to overfitting!

Curse of Dimensionality: Number of input features

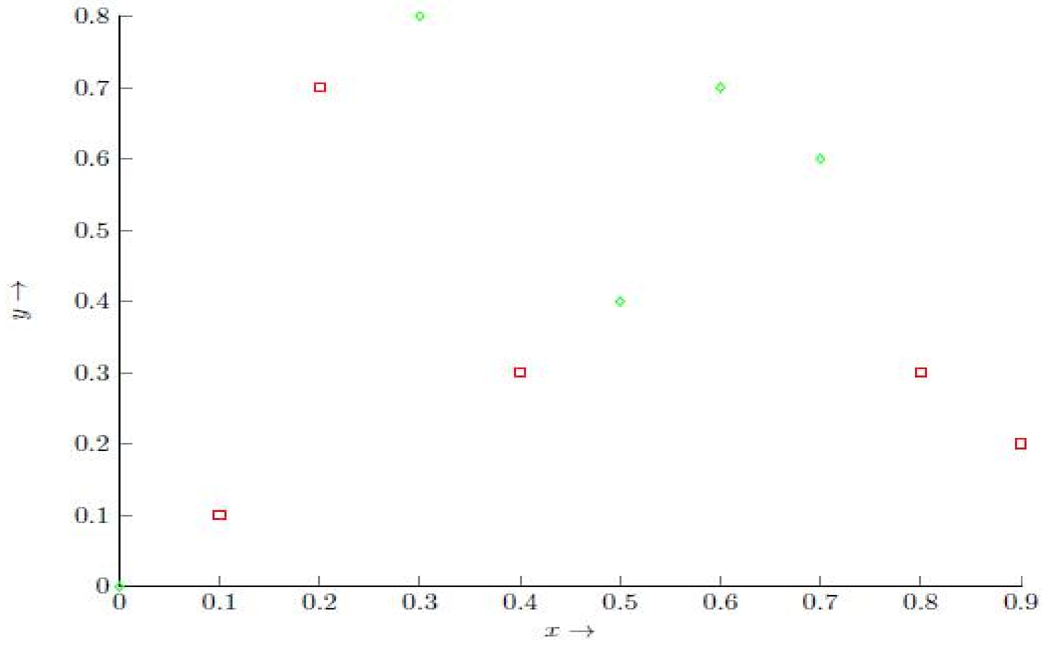
Consider a pattern recognition problem with two classes. To start with train the network with just one input feature



Sample density is 10 samples per unit length.

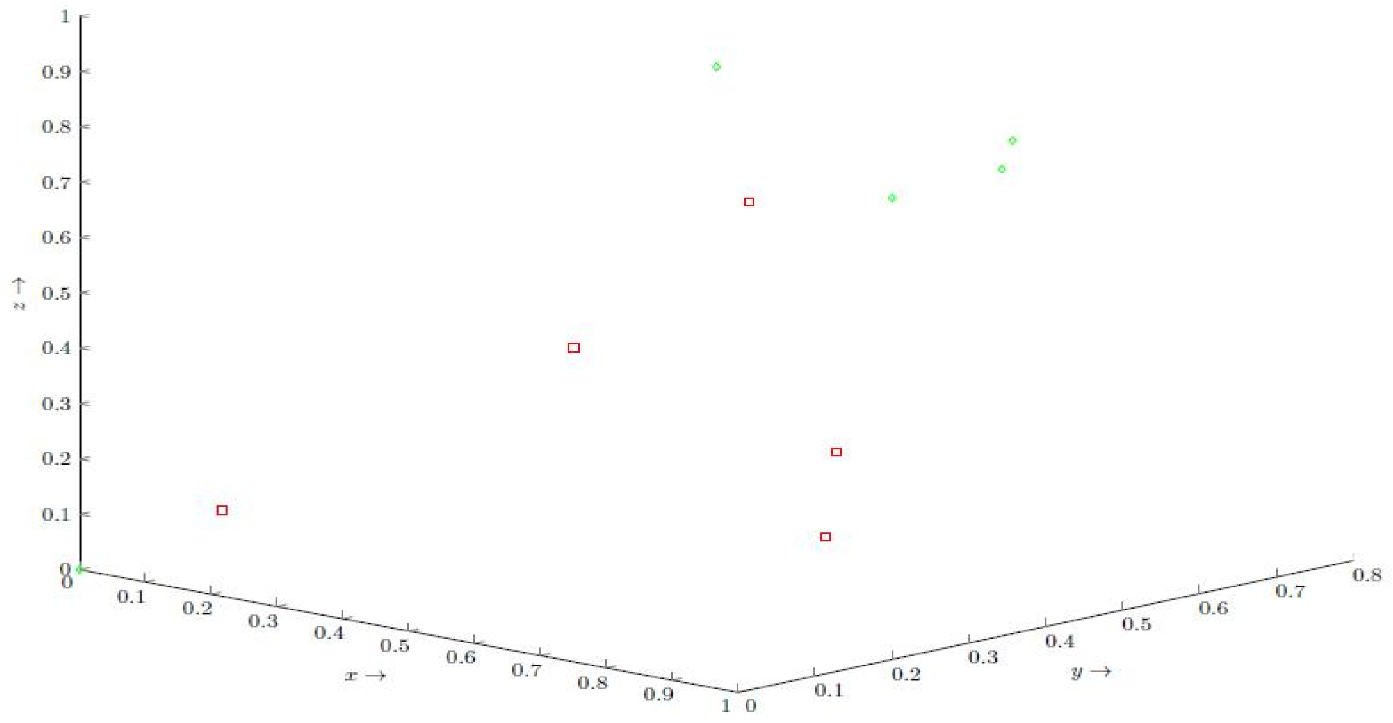
Increase the Number of Features

- Increasing the number of input features improves discrimination but decreases sample density



- To get the same sample density as one feature, 10^2 samples are needed!

With three features, 10^3 samples will be needed!

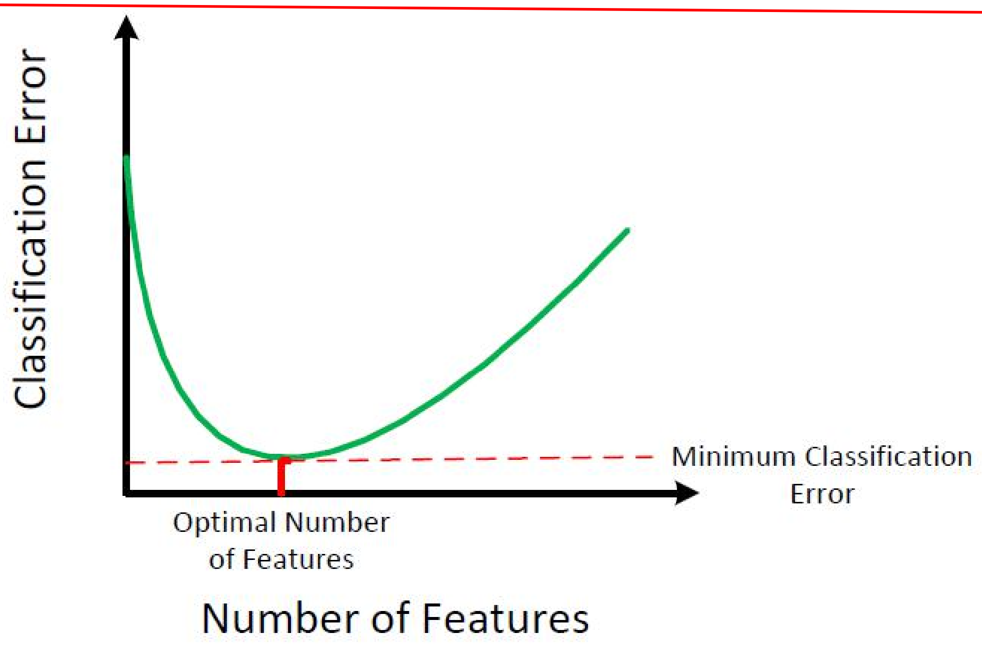


Number of Features and Sample Size

- If we have n samples in one dimension, we need n^m samples in m dimensions.
- n^m increases exponentially with increase in number of input features.
- Adding more input features to improve performance works only if we can increase the number of input samples proportionately
- Adding more features without increasing sample size will worsen performance
- Practically, finding more data is very very hard!
- Dimensionality reduction is a more feasible approach: Data preprocessing, Principal component analysis

How to Determine the Optimal Number of Features?

For a given sample size, there is an optimal number of features to use,



Empirical approach: Keep adding features until the error reaches a minimum before it begins to increase.

Independent Validation

- Testing performance with training data is unreliable
- It may lead to overfitting
- This may be avoided by using a standard statistical procedure called *independent validation or cross-validation*
  - Randomize the dataset
  - Partition data into training and testing set
  - Further section the training data into 2 sets: training (or estimation) set and validation set
  - Use the training set to update the weights; Use the validation set to assess network performance
  - Use the test data to test how well the trained network generalizes
- This method of cross-validation is called the holdout method

Using Cross-Validation for Model Selection

- Model selection is basically the problem of finding the optimal number of nodes (weights and biases) in the network
- We train the network and validate its performance for various network configurations; choose the one that minimizes the generalization error
- The optimal ratio of the sizes of the training set and validation set plays an important role in minimizing the generalization error.
- For N data samples, (1 - r)N samples are allocated to the training set and rN are allocated to the validation set.

How do we specify r?

- low complexity problem → number of samples much larger than number of input features: performance is almost insensitive to r
- High complexity problem: choice of r has a more pronounced effect on performance
- A single fixed value of r has been shown to work nearly optimally for a wide range of problems
- Kearns (1996) reported that r = 0.2 is generally a sensible choice.

  20% Validation samples, 80% Training samples

Using Cross-Validation as an Early-Stopping Criterion

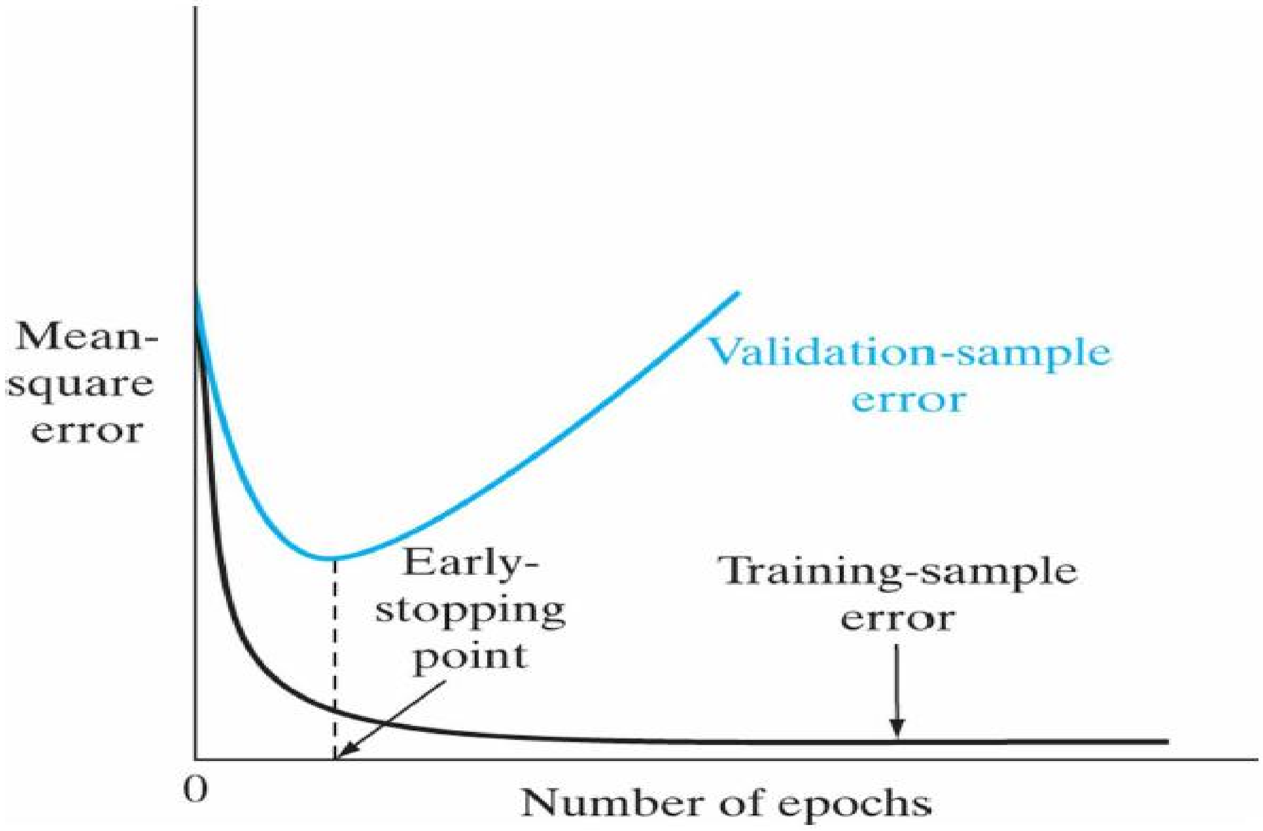
- Typically, as training progresses the mean square error decreases, and the network moves towards the minimum
- For good generalization, it is necessary to stop training at the `right point' on the error curve.
- Stopping too early leads to poor generalization, stopping too late leads to overfitting.
- Cross-validation can be used to identify the onset of overfitting.

Train then Validate Periodically

- Train the network for a period
- Periodically stop training, every 5 epochs for instance, present the validation input to the network and find the validation error
- Continue training and periodic training until the validation error reaches a minimum

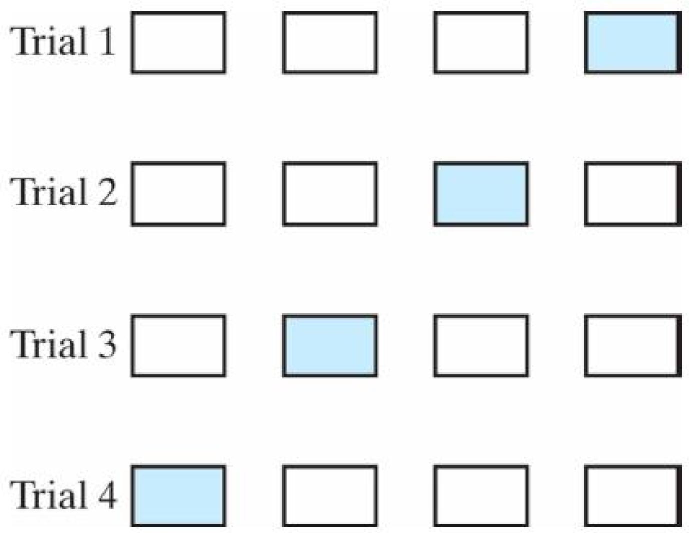
This procedure is called the early-stopping method of training.

Illustration of Early-Stopping



Multifold Cross-Validation

- When there is a scarcity of data samples, we can use *multifold cross-validation.*
- Divide the dataset of N samples into K subsets
- Train the network with K-1 subsets and validate with the Kth subset.
- Repeat until all subsets have been used for training and validation.
- Network performance is the average of the error over all trials.



This process is computationally expensive. The model has to be trained K times.

With a severely limited dataset, we may use *the leave-one-out* method. Train the network on N- 1 samples, test with the final sample. Network must be trained N times.

Bias and Variance in a Neural Network

- Overfitting is one the most serious problems that arises in supervised learning networks.
- We use statistical techniques like independent validation to prevent overfitting.
- These methods are employed to balance the statistical bias and statistical variance when doing neural network learning to achieve smallest average generalization error.
- Statistical bias accounts for the degree of fitting the training data.
- The statistical variance accounts for the generalization ability of the network.
- Ideally we would like to minimize both statistical bias and statistical variance to improve performance.

Bias-Variance Dilemma

- There is a tradeoff between bias and variance; cannot minimize one without increasing the other.
- A network that fits the data closely has low bias but a high variance. If we reduce variance this will lead to a decrease in the degree to which the data is fitted.
- To reduce both the statistical bias and variance we can add more data samples; not always possible.
- Statistical bias can be reduced by increasing the network size.
- Statistical variance can be decreased by pruning the neural network.

Pruning refers to the removal of synaptic connections

Complexity Regularization

- The tradeoff between the reliability of data and goodness of the model can be represented as,
$$R(w) = \varepsilon_{av}(w) + \lambda \varepsilon_c(w)$$
- The $\varepsilon_{av}(w)$ term is the standard average error of the backpropagation algorithm, which depends on the both data and the network configuration.
- The $\varepsilon_c(w)$ term is the complexity penalty measured in terms of the network weights.
- λ is the regularization parameter representing the importance assigned to the complexity penalty.
- A small λ indicates unconstrained learning; A large λ indicates a constrained learning process.

More on regularization theory in later lectures.

Weight-Decay Procedure

- A type of complexity regularization process
- The complexity penalty term is defined as,

$$\varepsilon_c(w) = \|w\|^2 = \sum_{i \in C_{total}}$$

where C_{total} refers to all synaptic weights in the network

- This procedure drives some network weights to zero; effectively disconnecting the weight.
- Weights that have a significant impact on the performance have large values.
- Weights that have little to no influence on performance are disconnected; also called excess weights.
- Removal of network weights is called pruning; improves generalization.

Speed of Convergence

- Backprop is a generalization of LMS algorithm
- Learning parameter affects the speed of convergence
- This issue had led to the development of accelerated learning algorithms
- Heuristic improvement to standard backpropagation
  - Backprop with momentum (MATLAB: *traingdm*)
  - Variable learning rate gradient descent algorithms
- Use of numerical optimization techniques with backpropagation

Variable Learning Rate Algorithms

Adaptive learning rate backpropagation

- At each weight update step if the error increases by a given threshold, the new weights are discarded and a new (lower) learning rate is calculated.
- The new learning rate is decreased by a given fraction
- If the error decreases by the threshold amount, the learning rate is increased by a given fraction.
- MATLAB function: *traingda*

Adaptive gradient descent with momentum

– MATLAB function: *traingdx*

Accelerated Learning Algorithms

- Vector matrix form of backpropagation
- Conjugate gradient backpropagation
- Recursive least squares backpropagation
- Levenberg-Marquardt backpropagation

Vector Matrix Form of Backpropagation

Step 1: Initialize the network synaptic weights to small random values

Step 2: Select a random input/output training sample, present it to the network and calculate the response of each layer simultaneously

Step 3: Use the output error to calculate the weight correction at each hidden layer,

- For the output layer j,

$$\mathbf{D}_j = \mathbf{G}(\mathbf{v}_j)(\mathbf{d} - \mathbf{y}_j)$$

- For a hidden layer,

$$\mathbf{D}_{j-1} = \mathbf{G}(v_{j-1})\mathbf{D}_j\mathbf{W}_j$$

where,

- $\mathbf{G}(\mathbf{v}_j))$ is the gradient matrix at layer j
- $\mathbf{W}_j$ is the weight at layer j

- Step 4: Update the network weights according to,

$$\mathbf{W}_j(n+1) = \mathbf{W}_j(n) + \eta D_j(n) y_j(n)$$

where $y_i(n) = x(n)$ for layer 1.

- Step 5: Continue steps 2 through 4, until the network reaches a desired level of performance

Conjugate Gradient Backpropagation

- Conjugate gradient method is a numerical techniques for solving optimization problems
- Presents a compromise between simplicity of steepest descent and fast quadratic convergence of Newton's method
- The error function is approximated as a quadratic function

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{Q}\mathbf{w} - \mathbf{b}^T\mathbf{w}$$

where **w** is the vector of network weights and **Q** is the Hessian matrix

$$\begin{bmatrix} \frac{\partial^2 \mathcal{E}}{\partial w_1^2} & \frac{\partial^2 \mathcal{E}}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 \mathcal{E}}{\partial w_1 \partial w_m} \\ \frac{\partial^2 \mathcal{E}}{\partial w_2 \partial w_1} & \frac{\partial^2 \mathcal{E}}{\partial w_2^2} & \cdots & \frac{\partial^2 \mathcal{E}}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 \mathcal{E}}{\partial w_m \partial w_1} & \frac{\partial^2 \mathcal{E}}{\partial w_m \partial w_2} & \cdots & \frac{\partial^2 \mathcal{E}}{\partial w_m^2} \end{bmatrix}$$

The dimension of the Hessian is equal to the total number of weights in the network.

- Computing the Hessian is impractical for even moderately large networks. Most conjugate gradient methods use alternative approaches for calculating the Hessian matrix.

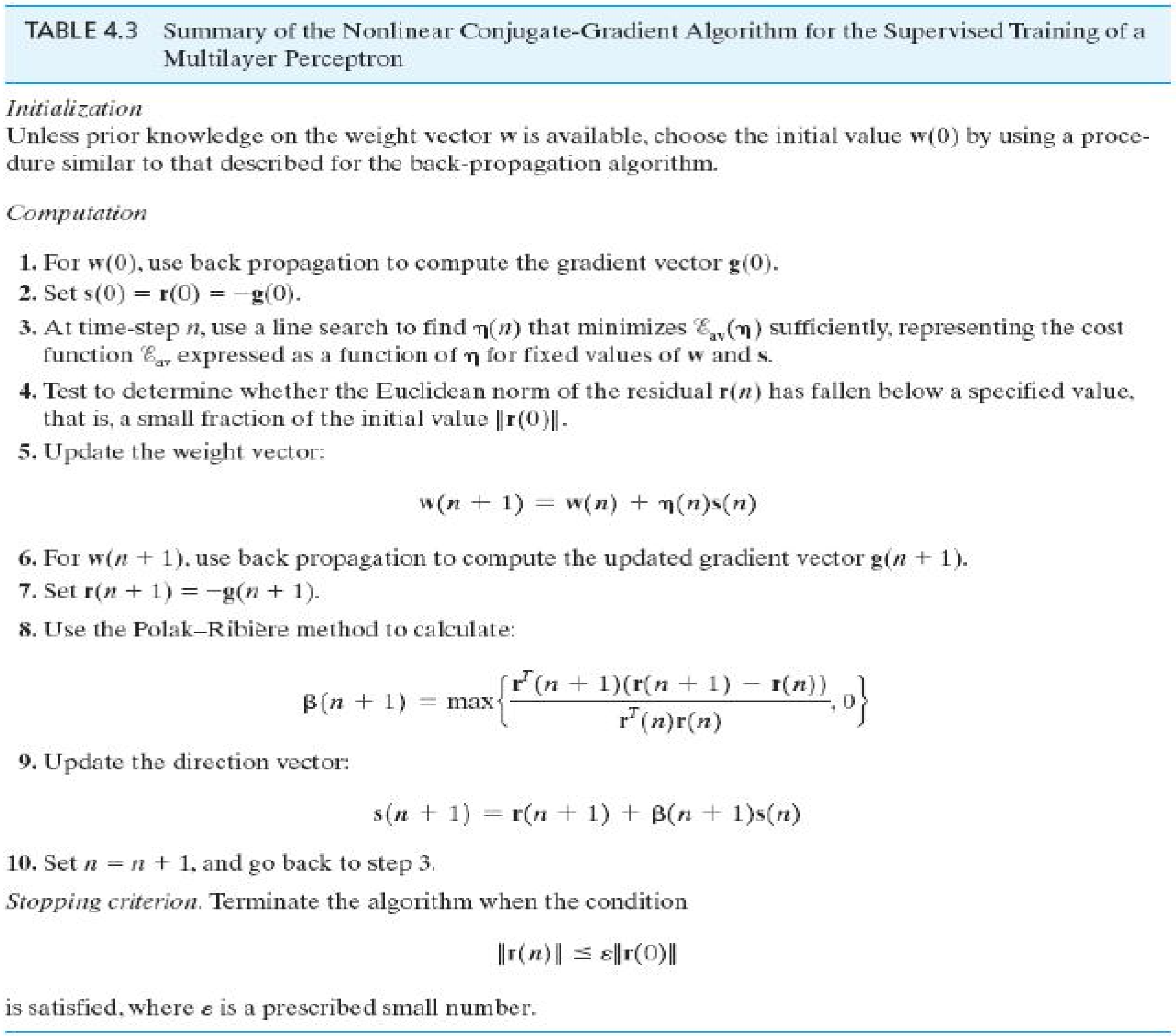


TABLE 4.3 Summary of the Nonlinear Conjugate-Gradient Algorithm for the Supervised Training of a Multilayer Perceptron

Initialization
Unless prior knowledge on the weight vector $\mathbf{w}$ is available, choose the initial value $\mathbf{w}(0)$ by using a procedure similar to that described for the back-propagation algorithm.

Computation

1. For $\mathbf{w}(0)$, use back propagation to compute the gradient vector $\mathbf{g}(0)$.
2. Set $\mathbf{s}(0) = \mathbf{r}(0) = -\mathbf{g}(0)$.
3. At time-step n, use a line search to find $\eta(n)$ that minimizes $\mathscr{E}_{\text{av}}(\eta)$ sufficiently, representing the cost function $\mathscr{E}_{\text{av}}$ expressed as a function of η for fixed values of $\mathbf{w}$ and $\mathbf{s}$.
4. Test to determine whether the Euclidean norm of the residual $\mathbf{r}(n)$ has fallen below a specified value, that is, a small fraction of the initial value $\|\mathbf{r}(0)\|$.
5. Update the weight vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n)$$

6. For $\mathbf{w}(n+1)$, use back propagation to compute the updated gradient vector $\mathbf{g}(n+1)$.
7. Set $\mathbf{r}(n+1) = -\mathbf{g}(n+1)$.
8. Use the Polak–Ribière method to calculate:

$$\beta(n+1) = \max\left\{\frac{\mathbf{r}^T(n+1)(\mathbf{r}(n+1) - \mathbf{r}(n))}{\mathbf{r}^T(n)\mathbf{r}(n)}, 0\right\}$$

9. Update the direction vector:

$$\mathbf{s}(n+1) = \mathbf{r}(n+1) + \beta(n+1)\mathbf{s}(n)$$

10. Set $n = n + 1$, and go back to step 3.

Stopping criterion. Terminate the algorithm when the condition

$$\|\mathbf{r}(n)\| \leq \varepsilon\|\mathbf{r}(0)\|$$

is satisfied, where ε is a prescribed small number.

Comments on Conjugate-Gradient Algorithms

- Four variations of the Conjugate-Gradient method based on how the constant β is calculated.
  - Polak-Ribieri update (MATLAB: traincgp)
  - Fletcher-Reeves update (MATLAB: traincgf)
  - Powell-Beale restarts (MATLAB: traincgb)
  - Scaled conjugate gradient(MATLAB: traincgf)

- The learning rate is calculated at each iteration in the algorithm

Levenberg-Marquardt Backpropagation

- The Levenberg-Marquardt algorithm is a compromise between the Newton methods that converges rapidly near a minimum, and the gradient descent which is guaranteed to converge, but slowly.
- The basic update step of the Newton method is,

$$w(n+1) = w(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n)$$

where **H** is the Hessian matrix and **g** is the error gradient.

- The LM algorithm was designed to improve the training speed by avoiding the calculation of the Hessian.

Computation of Levenberg-Marquardt Weight Update Equation

Consider the approximation of the Hessian,

$$\mathbf{H} = \mathbf{J}^T\mathbf{J}$$

the error gradient is given by,

$$\mathbf{g} = \mathbf{J}^T\mathbf{e}$$

and $\mathbf{J}$ is the Jacobian matrix that contains the first derivatives of the error function.

$$\begin{bmatrix} \frac{\partial e_1}{\partial w_1} & \frac{\partial e_1}{\partial w_2} & \cdots & \frac{\partial e_1}{\partial w_m} \\ \frac{\partial e_2}{\partial w_1} & \frac{\partial e_2}{\partial w_2} & \cdots & \frac{\partial e_2}{\partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e_N}{\partial w_1} & \frac{\partial e_N}{\partial w_2} & \cdots & \frac{\partial e_N}{\partial w_m} \end{bmatrix}$$

where m is the number of input variables and N is number of input samples

Levenberg-Marquardt Weight Update

Levenberg-Marquardt weight update is given by

$$w(n+1) = w(n) - [\boldsymbol{J}^T\boldsymbol{J} + \mu\,\boldsymbol{I}]^{-1}e$$

- Computing the Jacobian is much simpler than calculating the Hessian matrix, reducing computational complexity.

When μ is 0, the weight update reverts to basic Newton's method.

When μ is large the weight update becomes gradient descent with a small learning rate.

MATLAB function: *trainlm*

Virtues and Limitations of Backpropagation

- Computational efficiency
- Connectionism
- Replicator mapping
- Function approximation
- Sensitivity analysis
- Robustness
- Convergence
- Local minima
- Scaling

[1] Ham and Kostanic, Principles of Neurocomputing for Science and Engineering, 2001.

[2] Kearns et. al., An Experimental and Theoretical Comparison of Model Selection Methods, Proc. Eigth ACM Conference on Computational Learning Theory.

[3] Simon Haykin, Neural Networks and Learning Machines, 3rd Ed., 2008.

All figures and content adapted from reference [3], unless specified otherwise.