Introduction to Artificial Neural Networks

Lecture 5:

Multilayer Perceptrons (Feedforward Neural Net)

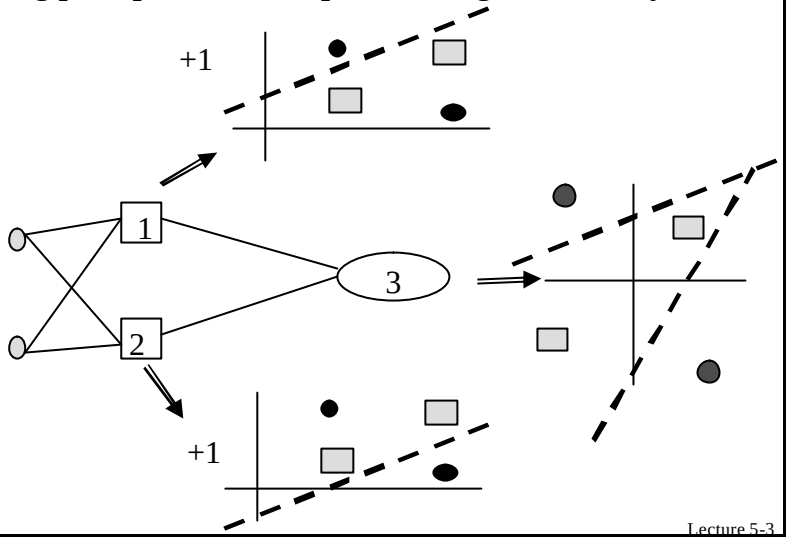
By: Ali Motie Nasrabadi

Outline

- Introduction
- Multilayer Perceptrons
- Back-Propagation Algorithm
- Gradient Descent
- Choosing the Learning Rate
- Mode of Training
- Stopping Criteria
- local and global minimum
- Heuristics for making BP Better
- Momentum term
- Generalization
- Approximation of Functions
- Practical Consideration
- Cross-Validation
- Model selection
- Accelerated Convergence
- MATLAB neural networks toolbox
- Faster Training
- conjugate gradient
- Line Search Routines
- Quasi-Newton Algorithms
- Levenberg-Marquardt
- Under-Fitting and Over-Fitting

Introduction

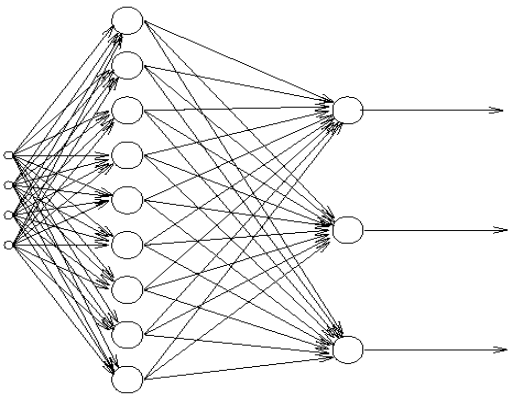
Minsky & Papert (1969) offered solution to XOR problem by combining perceptron unit responses using a second layer of units



Lecture 5-3

Introduction

MLPs: Feed-forward neural network with multiple layers of processing neurons.



Inputs | Processing Units (hidden units) | Outputs

Lecture 5-4

Multilayer Perceptrons (1)

- Hidden layers of computation nodes
- input propagates in a forward direction, layer-by-layer basis
  - also called Multilayer Feedforward Network, MLP
- MSE method can not be applied to problem
- Error back-propagation algorithm
  - supervised learning algorithm
  - error-correction learning algorithm
  - Forward pass
    - input vector is applied to input nodes
    - its effects propagate through the network layer-by-layer
    - with fixed synaptic weights
  - backward pass
    - synaptic weights are adjusted in accordance with error signal
    - error signal propagates backward, layer-by-layer fashion

Multilayer Perceptrons (2)

- Non-linear activation function
  - differentiable
  - sigmoidal function, logistic function
  - nonlinearity prevent reduction to single-layer perceptron

$$y_i = \frac{1}{1+\exp(-v_j)}$$

- One or more layers of hidden neurons
  - progressively extracting more meaningful features from input patterns
- High degree of connectivity
  - Nonlinearity and high degree of connectivity makes theoretical analysis difficult
  - Learning process is hard to visualize
  - BP is a landmark in NN: computationally efficient training

Multilayer Perceptrons (3)

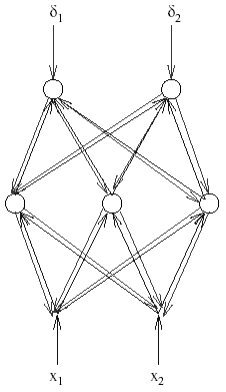
- **Function signal**
  - **input signals comes in at the input end of the network**
  - **propagates forward to output nodes**
- **Error signal**
  - **originates from output neuron**
  - **propagates backward to input nodes**
- **Two computations in Training**
  - **computation of function signal**
  - **computation of an estimate of gradient vector**
    - **gradient of error surface with respect to the weights**

Back-Propagation Algorithm

- **Error signal for neuron j at iteration *n*** $e_j(n) = d_j(n) - y_j(n)$
- **Total error energy** $E(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n)$
  - ***C* is set of the output nodes**
- **Average squared error energy** $E_{av} = \frac{1}{N}\sum_{n=1}^{N} E(n)$
  - **average over all training sample**
  - **cost function as a measure of learning performance**
- **Objective of Learning process**
  - **adjust NN parameters (synaptic weights) to minimize E_{av}**
- **Weights updated pattern-by-pattern basis until one epoch**
  - **complete presentation of the entire training set**

Back-Propagation of Error

- Forward pass: **to compute network output**
- Backward pass: **propagate error back through the network to recursively compute weight changes (via chain rule differentiation).**



Lecture 5-9

BPA

- **Induced local field** $v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$
- **output of neuron j** $y_j(n) = j_j(v_j(n))$
- **Gradient** $\frac{\partial E(n)}{\partial w_{ij}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ij}(n)}$
  - **Sensitivity factor**
  - **determine the direction of search in weight space**
  - **according to chain rule**

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \qquad \frac{\partial e_j(n)}{\partial y_j(n)} = -1 \qquad \frac{\partial y_j(n)}{\partial v_j(n)} = j'_j(v(n)) \qquad \frac{\partial v_j(n)}{\partial w_{ij}(n)} = y_i(n)$$

Lecture 5-10

Gradient Descent

- **Therefore,** $\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n) j'_j(v_j(n)) y_i(n)$
- **By delta rule** $\Delta w_{ji}(n) = -? \frac{\partial E(n)}{\partial w_{ji}(n)}$

  which is gradient descent in weight space
- **Local gradient**

$$\delta_j(n) = -\frac{\partial E(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$
$$= e(n) j'_j(v_j(n))$$
$$\Delta w_{ij}(n) = ? \delta_j(n) y_i(n)$$

Lecture 5-11

Local Gradient (1)

- **Neuron j is an output node** $e_j(n) = d_j(n) - y_j(n)$
- **Neuron j is a hidden node**
  - **credit assignment problem**
    - **how to determine their share of responsibility**

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} j'_j(v_j(n))$$

- **for output neuron *k***

$$E(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n)$$
$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

Lecture 5-12

Local Gradient (2)

- **Error in neuron k** $e_k(n) = d_k(n) - y_k(n) = d_k(n) - j_k(v_k(n))$
- **Hence** $\frac{\partial e_k(n)}{\partial v_k(n)} = -j'_k(v_k(n))$

  since $v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n)$, $\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$
- **desired partial derivative**

$$\frac{\partial E(n)}{\partial y_j(n)} = -\sum_k e_k(n) j'_k(v_k(n)) w_{kj}(n) = -\sum_k \delta_k(n) w_{kj}(n)$$

- **back-propagation formula for hidden neuron j**

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

Lecture 5-13

BP Summary

$$\begin{pmatrix} \text{Weight} \\ \text{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{learning} \\ \text{parameter} \\ ? \end{pmatrix} \cdot \begin{pmatrix} \text{local} \\ \text{gradient} \\ \delta_j(n) \end{pmatrix} \cdot \begin{pmatrix} \text{input signal} \\ \text{to neuron j} \\ y_i(n) \end{pmatrix}$$

- **forward pass**

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$$

$$y_j(n) = j_j(v_j(n))$$

- **backward pass**
  - **recursively compute local gradient** δ
    - **from output layer toward input layer**
  - **synaptic weight change by delta rule**

Lecture 5-14

Activation Function(logistic function)

$$j_j(v_j(n)) = \frac{1}{1+\exp(-av_j(n))} \quad a > 0 \text{ and } -\infty < v_j(n) < \infty$$

$$j'_j(v_j(n)) = \frac{a\exp(-av_j(n))}{[1+\exp(-av_j(n))]^2} = ay_j(n)[1-y_j(n)]$$

- **local gradient**
  - **for output node**

$$\delta_j(n) = e_j j'_j(v_j(n)) = a[d_j(n) - o_j(n)]o_j(n)[1-o_j(n)]$$

  - **for hidden node**

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

$$= ay_j(n)[1-y_j(n)]\sum_k \delta_k(n)w_{kj}(n)$$

Activation Function(Hyperbolic tangent function)

$$j_j(v_j(n)) = a\tanh(bv_j(n)) \qquad (a,b) > 0$$

$$j'_j(v_j(n)) = ab\,\mathrm{sech}^2(bv_j(n)) = ab(1-\tanh^2(bv_j(n)))$$

$$= \frac{b}{a}[a-y_j(n)][a+y_j(n)]$$

- **local gradient**
  - **for output node**

$$\delta_j(n) = e_j j'_j(v_j(n)) = \frac{b}{a}[d_j(n) - o_j(n)][a-o_j(n)[a+o_j(n)]$$

  - **for hidden node**

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

$$= \frac{b}{a}[a-y_j(n)][a+y_j(n)]\sum_k \delta_k(n)w_{kj}(n)$$

Back-Propagation Summery

- **Initialize weights and thresholds**
  - **Set all weights and thresholds to small random values**
- **Present input and desired output**
  - **Present input x and desires output d. For pattern association x, d represent the pattern to be associated. For classification d is set to zero except for one element set to 1 that corresponds to the class that x is in**
- **Calculate actual output**
  - **Each layer calculates** $v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$

    $y_j(n) = j_j(v_j(n))$
  - **And passes that as input to the next layer. The final layer outputs values**

Back-Propagation Summery

- **Adapt weights**
  - **Start from the output layer, and work backwards:**

$$w_{ij}(n+1) = w_{ij}(n) + hd_j(n) y_j(n)$$

  - $w_{ij}(n)$ **represents the weights from node i to j at time n,** h **is a learning rate, and** d_j **is an error term for a pattern or a set of pattern on node j**
  - **For output units:**

$$\mathrm{d}_j(n) = a[d_j(n) - o_j(n)] o_j(n)[1 - o_j(n)]$$

  - **For hidden units:** $\mathrm{d}_j(n) = a y_j(n)[1 - y_j(n)] \sum_k \mathrm{d}_k(n) w_{kj}(n)$
    - **Where the sum is over the k nodes in the layer above node j**

Choosing the Learning Rate

- **Choosing a good value for the learning rate h is constrained by two opposing facts:**
  - **If h is too small, it will take too long to get anywhere near the minimum of the error function.**
  - **If h is too large, the weight updates will over-shoot the error minimum and the weights will oscillate, or even diverge.**
- **Unfortunately, the optimal value is very problem and network dependant, so one cannot determine reliable general prescriptions. Generally, one should try a range of different values (e.g. 1.0, 0.1, 0.01) and use the results as a guide.**
  - **There is no necessity to keep the learning rate fixed throughout the learning process.**
  - **Common variable learning rates are:**

$$h(n) = \frac{h_0}{n} \qquad h(n) = \frac{h_0}{1 + n/t}$$

Lecture 5-19

Mode of Training

- **Epoch : one complete presentation of training data**
- **randomize the order of presentation for each epoch**

- **Sequential mode (Incremental, Online)**
  - **for each training sample, synaptic weights are updated**
  - **require less storage**
  - **converge much fast, particularly training data is redundant**
  - **random order makes trapping at local minimum less likely**
- **Batch mode**
  - **at the end of one epoch, synaptic weights are updated**

$$\Delta w_{ji}(n) = -? \frac{\partial E_{avg}(n)}{\partial w_{ji}(n)} = -\frac{?}{\mathrm{N}} \sum_{\mathrm{n}=1}^{\mathrm{N}} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}(n)}$$

  - **may be robust with outliers**

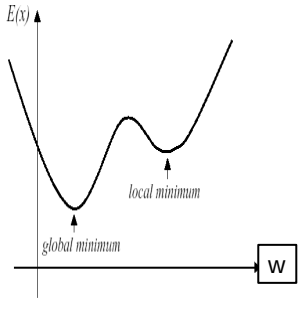
Lecture 5-20

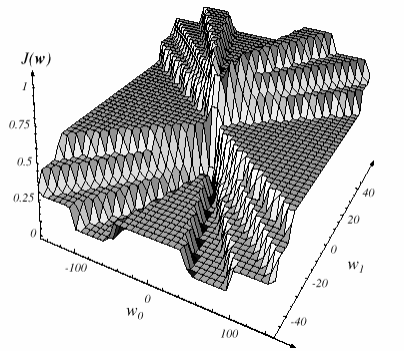
Stopping Criteria

- No well-defined stopping criteria
- Terminate when Gradient vector g(W) = 0
  - located at local or global minimum
- Terminate when error measure is stationary
- Terminate if NN's generalization performance is adequate

local and global minimum

- Error functions can quite easily have more than one minimum (for one and two dimensional error performance)





Heuristics for making BP Better (1)

- **Training with BP is more an art than science**
  - **result of own experience**
- **Sequential vs. Batch update**
- **Maximizing information content**
  - **examples of largest training error**
  - **examples of radically different from previous ones**
- **Randomize the order of presentation**
  - **successive examples rarely belongs to the same class**
- **Activation function**
  - **asymmetric function learns fast** $j(v) = a\tanh(bv)$
- **Target value is within range of sigmoid activation function**
  - **target value should be offset by some e from limiting value**

Heuristics for making BP Better (2)

- **Normalizing the inputs**
  - **preprocessed so that its mean value is closed to zero**
  - **input variables should be uncorrelated**
    - **by principal component analysis**
  - **scaled so that covariance are equal**
- **Weight Initialization**
  - **Large weight value => saturation**
    - **local gradient value is small = slow learning**
  - **small weight value => operate on flat area = slow learning**
  - **somewhere between two extremes.**
  - **For the hyperbolic tangent function, set $\mu = 0$, $\sigma^2 = 1/m$**

Heuristics for making BP Better (3)

- **Learning from hints**
  - **prior information should be included in the learning process**
    - **invariant properties, symmetries, etc.**
- **Learning Rate**
  - **all the neurons learn at the same rate**
    - **last layer has large local gradient (by limiting effect)**
    - **last layer learns fast**
    - **h of last layer should be assigned smaller one**
    - **LeCun's suggestion : learning rate is inversely proportional to square root of the number of synaptic connection ($m^{-1/2}$)**

Momentum term

- **BP approximate the trajectory of steepest descent**
  - **smaller learning-rate parameter makes smoother path**
- **increase rate of learning yet avoiding danger of instability**

$$\Delta w_{ji}(n) = a\,\Delta w_{ji}(n-1) + ?d_j(n)y_j(n)$$

where a is momentum constant

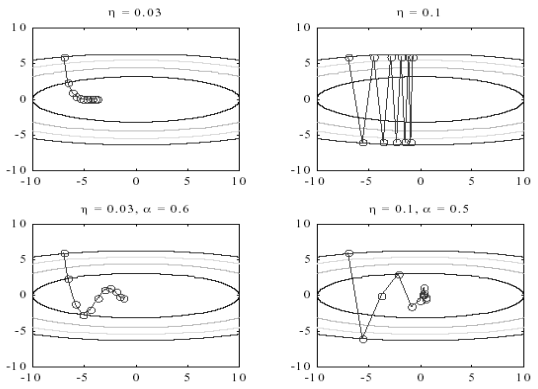
$$\Delta w_{ji}(n) = -? \sum_{t=0}^{n} a^{n-1} \frac{\partial E(n)}{\partial w_{ji}(n)} = ? \sum_{t=0}^{n} a^{n-1} d_j(t) y_i(t)$$

  - **converge if 0 £ | a | £ 1**
  - **the partial derivative has the same sign on consecutive iterations, grows in magnitude - accelerate descent**
  - **opposite sign - shrinks; stabilizing effect**
- **benefit of preventing the learning process from terminating in a shallow local minimum**

Backprop with momentum

No momentur

Momentum



Lecture 5-27

Generalization

- **Input-output mapping is correct for data never seen before**
- **Learning process is curve fitting - non-linear mapping**
- **Overfitting - Overtraining**
  - **memorize training data, not the essence of the training data**
  - **learns idiosyncrasy and noise**
  - **loose the ability of generalize**
- **Occam's Razer**
  - **find the simplest function among those which satisfy given conditions**
  - **smoothest function**

Lecture 5-28

Training Set Size for Generalization

- Genralization is influenced
  - size of training set
  - architecture of Neural Network
- Given architecture, determine the size of training set for good generalization
  - The number of training patterns (N) required to classify test example with an error of e is approximately given by

$$N > \frac{W}{e}$$

    - Where w is number of weights in the network (Haykin).
    - A rule of thumb states that $N \approx 10W$
    - Given set of training samples, determine the best architecture for good generalization

Approximation of Functions

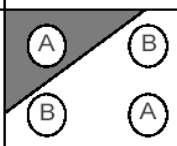
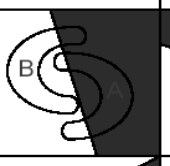
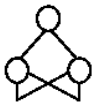
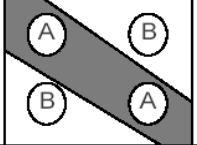
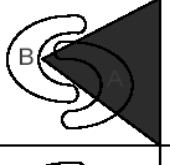
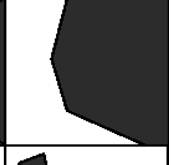
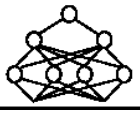
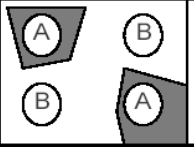
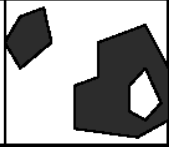
- Non-linear input-output mapping
  - M_0 input space to M_L output space
- What is the minimum number of hidden layers in a MLP that provide approximate any continuous mapping ?
- Universal Approximation Theorem
  - existence of approximation of arbitrary continuous function
  - single hidden layer is sufficient for MLP to compute a uniform e approximation to a given training set
  - not saying single layer is optimum in the sense of training time, easy of implementation, or generalization
- Bound of Approximation Errors of single hidden node NN
  - larger the number of hidden nodes, more accurate the approximation
  - smaller the number of hidden nodes, more accurate the empirical fit

Practical Consideration

- **Single hidden layer vs. double(multiple) hidden layer**
  - **single HL NN is good for any approximation of continuous function**
  - **double HL NN may be good some times**
- **double(multiple) hidden layer**
  - **first hidden layer - local feature detection**
  - **second hidden layer - global feature detection**

Lecture 5-31

Different Non-Linearly Separable Problems

Structure	*Types of Decision Regions*	*Exclusive-OR Problem*	*Classes with Meshed regions*	*Most General Region Shape*
Single-Layer 	*Half Plane Bounded By Hyperplane*			
Two-Layer 	*Convex Open Or Closed Regions*			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

Lecture 5-32

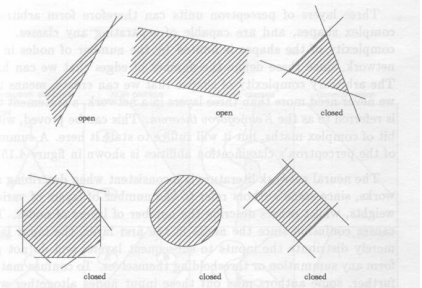


Figure 4.13 Examples of closed and open convex hulls.

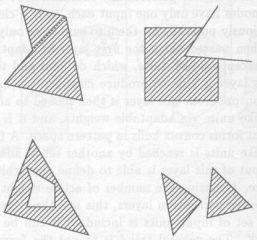
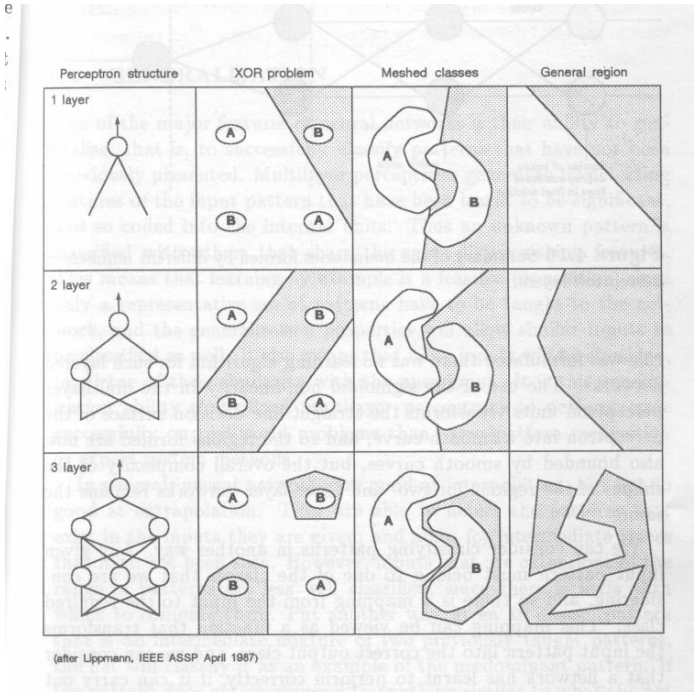


Figure 4.14 Examples of arbitrary regions formed by the combination of various convex regions.

Lecture 5-33



Lecture 5-34

Cross-Validation

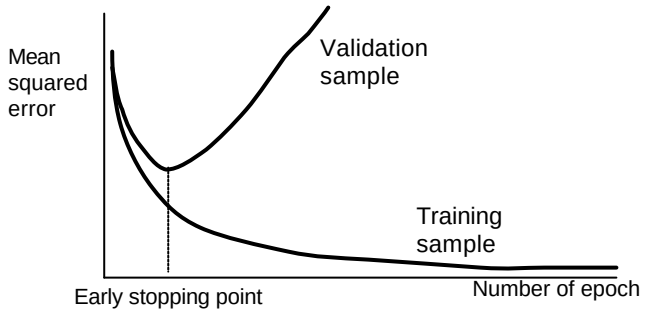
- Validate learned model on different set to assess the generalization performance
  - guarding against overfitting
- Partition Training set into
  - Estimation subset
  - validation subset
- cross-validation for
  - best model selection
  - determine when to stop training

Model selection

- Choosing MLP with the best number of free parameters with given N training samples
- Issue is to choose r
  - that determines split of training set between estimation set and validation set
  - to minimize classification error of model trained by the estimation set when it tested with the validation set
- Kearns(1996) : Qualitative properties of optimum r
  - Analysis with VC Dim
  - for small complexity problem (desired response is small compared to N), performance of cross-validation is insensitive to r
  - single fixed r nearly optimal for wide range of target function
- suggest $r = 0.2$
  - 80% of training set is estimation set

Stopping method of training

- **Right time to stop training**
  - **to avoid overfitting**
- **Early stopping method**
  - **after some training, with fixed synaptic weights computed validation error**
  - **resume training after computing validation error**



Lecture 5-37

Stopping method

- **Amari(1996)**
  - **for N<W**
    - **early stopping improves generalization**
  - **for N<30W**
    - **overfitting occurs** $r_{opt} = 1 - \frac{\sqrt{2W-1}-1}{2(W-1)}$

      $r_{opt} \approx 1 - \frac{1}{\sqrt{2W}}$ for large W
    - **example: w=100, r=0.07**
      - **93% for estimation, 7% for validation**
  - **for N>30W**
    - **early stopping improvement is small**
- **Leave-one-out method**

Lecture 5-38

Network Pruning

- **Minimizing network improves generalization**
  - **less likely to learn idiosyncrasies or noise**
- **Network growing**
- **Network pruning**
  - **weakening or eliminate synaptic weights**
- **Complexity-regularization**
  - **tradeoff between reliability of training data and goodness of the model**
  - **supervised learning by minimizing the risk function**

$$R(w) = E_s(W) + ?\ E_c(W)$$

where

$E_s(W)$ **: standard performance measure**

$E_c(W)$ **: complexity penalty**

Complexity-regularization

- **Weight Decays** $E_c(W) = \| W \|^2 = \sum w_i^2$
  - **some weights are forced to take value zero**
  - **weights in network are grouped into two categories**
    - **those of large influence**
    - **those of little or no influence : excess weights**
- **Weight Elimination**

$$E_c(W) = \sum \frac{(w_i / w_o)^2}{1 + (w_i / w_o)^2}$$

  - **when $w_i << w_0$, eliminated**
- **Approximate Smoother**

Hessian-based Network Pruning

- **Identify parameters whose deletion will cause the least increase in E_{av}**
- **by Tayer series**

$$E_{av}(\mathbf{w}+\Delta\mathbf{w}) = E_{av}(\mathbf{w}) + \mathbf{g}^t(\mathbf{w})\Delta\mathbf{w} + \frac{1}{2}\Delta w^t\mathbf{H}\Delta\mathbf{w} + O(\|\Delta\mathbf{w}\|^3)$$

  - **Parameters are deleted after training process has converged**
  - **quadratic approximation**

$$\Delta E_{av} = E(\mathbf{w}+\Delta\mathbf{w}) - E(\mathbf{w}) \approx \frac{1}{2}\Delta w^t\mathbf{H}\Delta\mathbf{w}$$

- **eliminate the weights of** $\Delta w_i + w_i = 0$
- **Solve the constrained optimization problem :** $\Delta\mathbf{w} = -\frac{w_i}{[\mathbf{H}^{-1}]_{i,i}}\mathbf{H}^{-1}{}_{1i}$
  - **if** $[\mathbf{H}^{-1}]_{i,i}$ **is small, even small weight is important**

Optimal Brain Surgeon

- **Saliency of w_i** $S_i = \frac{w_i^2}{2[\mathbf{H}^{-1}]_{i,i}}$
  - **represent the increase in the mean-squared error from delete of w_i**
- **OBS procedure**
  - **weight of small saliency will be deleted**
- **Optimal Brain Damage**
  - **with assumption of the Hessian matrix is diagonal**
- **computation of the inverse of Hessian**

Accelerated Convergence

- **Heuristics**

1. Adjustable weights should have own learning rate parameter

2. Learning rate parameters should be allowed to vary on iteration

3.If sign of the derivative is same for several iteration, learning rate parameter should be increased

- **Apply the Momentum idea even on learning rate parameters**

4. If sign of the derivative is alternating for several iteration, learning rate parameter should be decreased

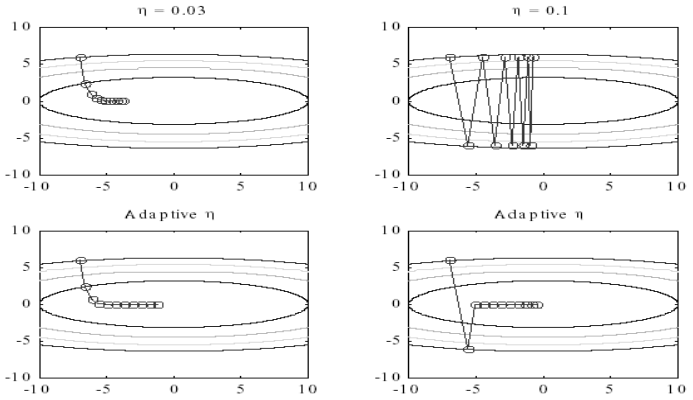
Adaptive η ("Bold driver")

$$\Delta \mathbf{w}(t) = -\eta(t)\nabla_w E(t)$$

$$\eta(t) = \begin{cases} 0.5\eta(t-1) & \text{if } E(t) \geq E(t-1) \\ 1.2\eta(\text{t - 1}) & \text{if } E(t) < E(t-1) \end{cases}$$

If things are going well $\Rightarrow$ increase speed
If things are going bad $\Rightarrow$ decrease speed

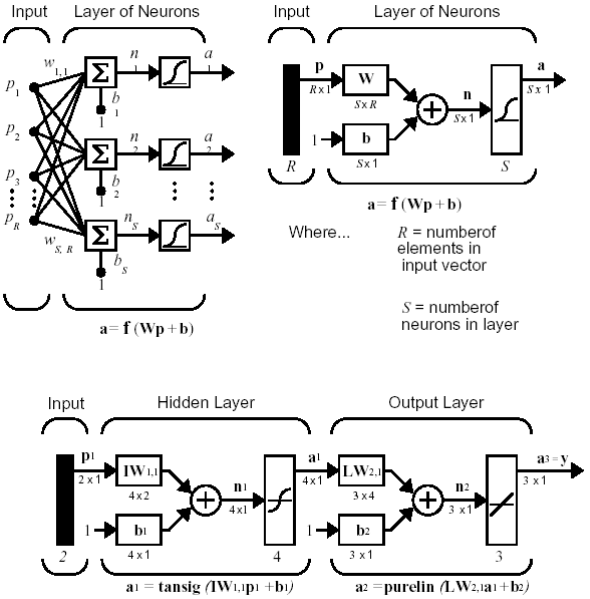
Adaptive η
"Bold driver"



MATLAB Neural Networks Toolbox

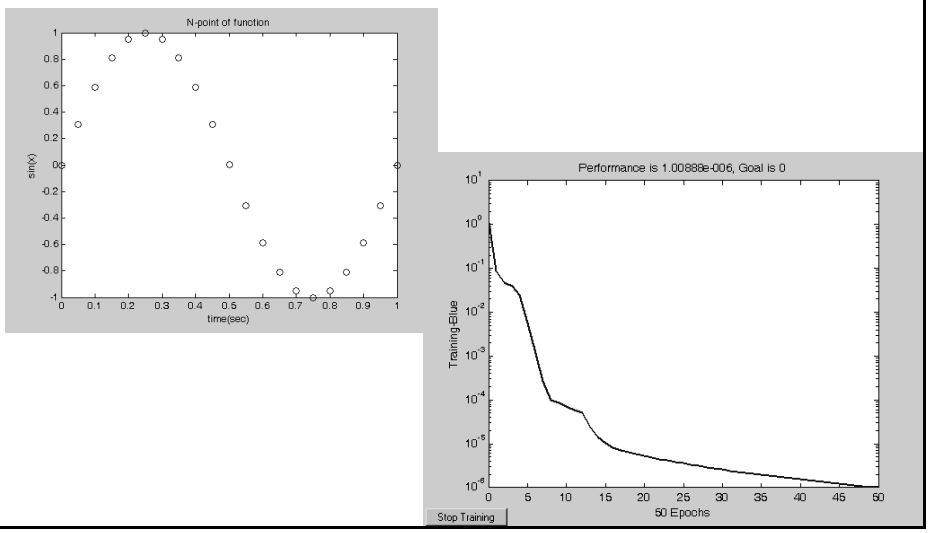
- **NEWFF Create a feed-forward backpropagation network.**
- **net = newff(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF)**
- **NEWFF(PR,[S1 S2...SNl],{TF1 TF2...TFNl},BTF,BLF,PF) takes,**
- **PR - Rx2 matrix of min and max values for R input elements.**
- **Si - Size of ith layer, for Nl layers.**
- **TFi - Transfer function of ith layer, default = 'tansig'.**
- **BTF - Backprop network training function, default = 'trainlm'.**
- **BLF - Backprop weight/bias learning function, default = 'learngdm'.**
- **PF - Performance function, default = 'mse'.**
- **and returns an N layer feed-forward backprop network.**

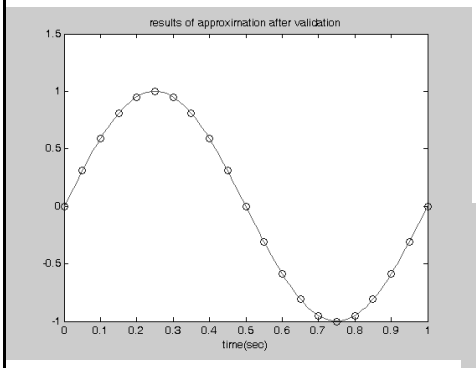
- **Feedforward neural net**

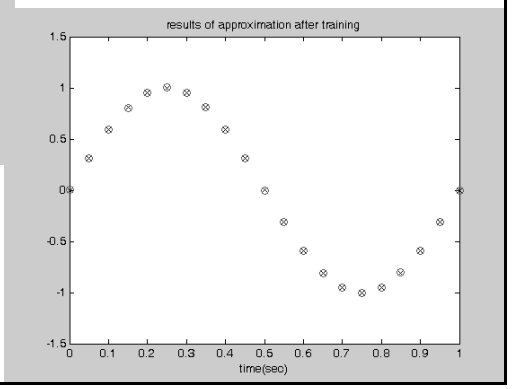


Function approximation

- **Approximation of sin(2*pi*x) in x=[0 1] with five neurons**







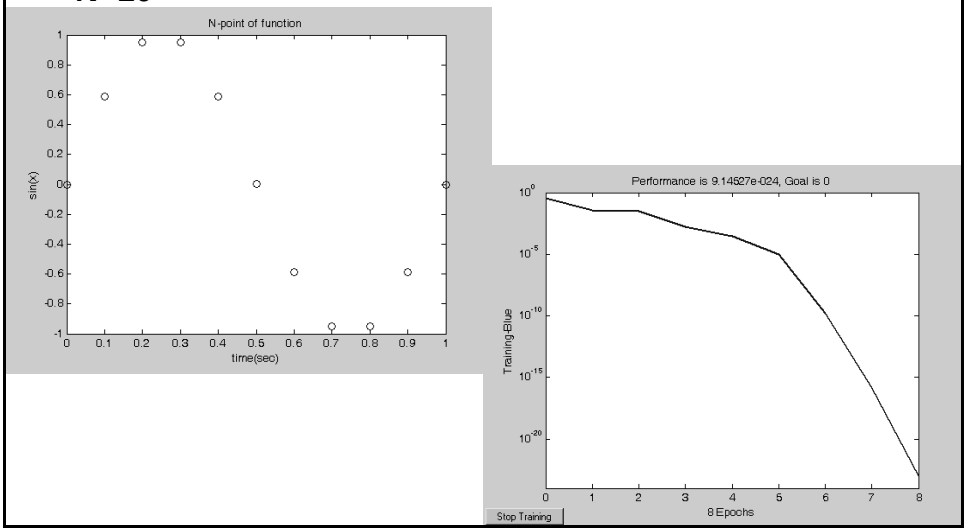
M_file of function approximation

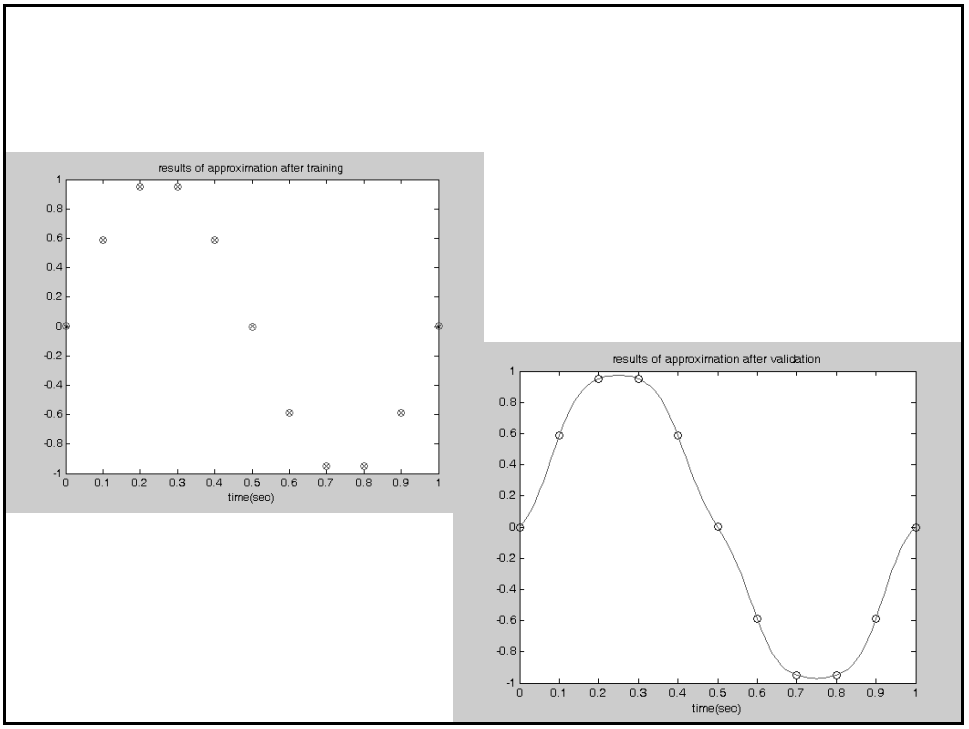
E:\Matlab6.5\work\mlpsin.m*

File Edit View Text Debug Breakpoints Web Window Help

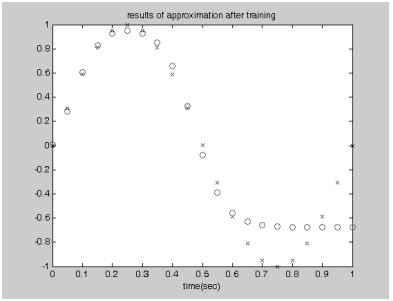
```
%function approximation :sin(2*pi*x) in [0 1]
clear
close all
x=0:0.05:1;
y=sin(2*pi*x);
plot(x,y,'o');xlabel('time(sec)');ylabel('sin(x)');
title('N-point of function')
P=x;
T=y;
%learnig phase
net = newff(minmax(P),[5 1],{'tansig' 'purelin'});
net.trainParam.epochs = 50;
net = train(net,P,T);
 Y = sim(net,P);
figure;plot(P,T,'x',P,Y,'o');xlabel('time(sec)');
title('results of approximation after training')
%validation phase
t=0:.01:1;
Y = sim(net,t);
figure;plot(P,T,'o',t,Y)
xlabel('time(sec)');
title('results of approximation after validation')
```

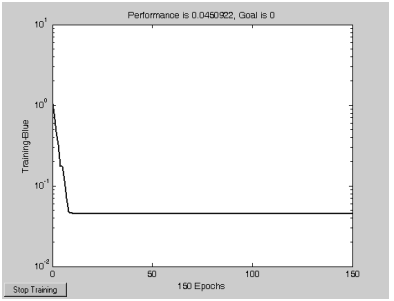
■ **N=10**

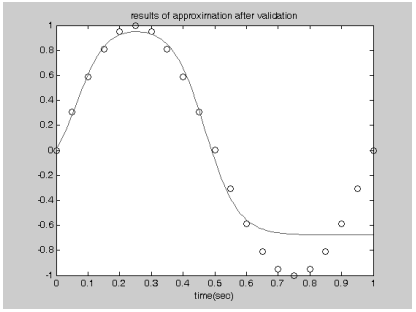

N-point of function
sin(x)
time(sec)
Performance is 9.14627e-024, Goal is 0
Training-Blue
8 Epochs
Stop Training


results of approximation after training
time(sec)
results of approximation after validation
time(sec)

■ Two neurons

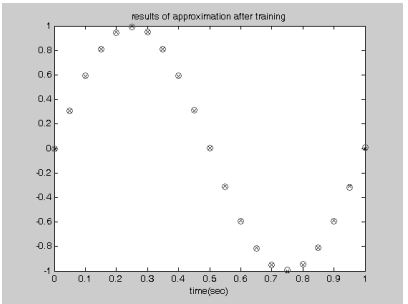


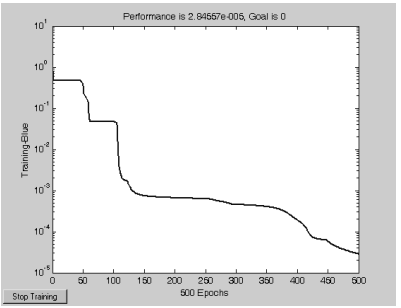


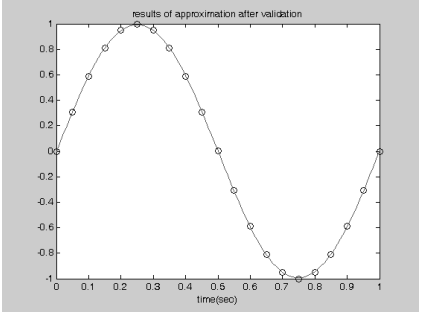


Lecture 5-53

■ Two hidden layers , each layer has two neurons

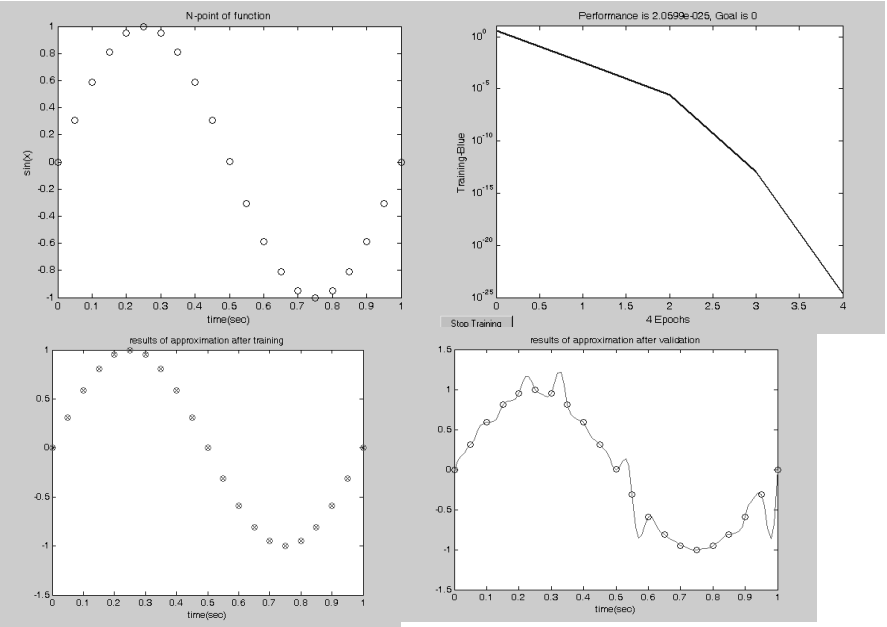






Lecture 5-54

- **Thirty hidden neurons**



Lecture 5-55

Faster Training

- **The previous slides presented three backpropagation training algorithms:**
  - **gradient descent**
  - **gradient descent with momentum (fast training)**
  - **gradient descent with** variable learning rate **(fast training)**
- **The second category of fast algorithms uses standard numerical optimization techniques**
  - Conjugate Gradient Algorithms
  - Line Search Routines
  - Quasi-Newton Algorithms
  - Levenberg-Marquardt (trainlm)

Lecture 5-56

Conjugate Gradient

- **All of the conjugate gradient algorithms start out by searching in the steepest descent direction (negative of the gradient, g,) on the first iteration:** $p(0) = -g(0)$
- **A line search is then performed to determine the optimal distance to move along the current search direction:** $W(n+1) = W(n) + a(n)p(n)$
- **The general procedure for determining the new search direction is to combine the new steepest descent direction with the previous search direction:**

$$p(n) = -g(n) + b(n)p(n-1)$$

- **For the Fletcher-Reeves update the procedure is**

$$b(n) = \frac{g^T(n)g(n)}{g^T(n-1)g(n-1)}$$

Lecture 5-57

Line Search Routines

- There are several method
- Golden Section Search
  - **The golden section search is a linear search that does not require the calculation of the slope. This routine begins by locating an interval in which the minimum of the performance occurs. This is accomplished by evaluating the performance at a sequence of points, starting at a distance of delta and doubling in distance each step, along the search direction. When the performance increases between two successive iterations, a minimum has been bracketed. The next step is to reduce the size of the interval containing the minimum. Two new points are located within the initial interval. The values of the performance at these two points determines a section of the interval that can be discarded, and a new interior point is placed within the new interval.**

Lecture 5-58

Quasi-Newton Algorithms

- **Newton's method is an alternative to the conjugate gradient methods for fast optimization. The basic step of Newton's method is**

$$W(n+1) = W(n) - H^{-1}(n)g(n)$$

- **where H is the Hessian matrix (second derivatives) of the performance index at the current values of the weights and biases. Newton's method often converges faster than conjugate gradient methods. Unfortunately, it is complex and expensive to compute the Hessian matrix for feedforward neural networks.**
- **For Quasi-Newton the Hessian matrix can be approximated as** $H(n) = J^T(n)J(n)$ $g(n) = J^T(n)e(n)$

Where J is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases, and e is a vector of network errors.

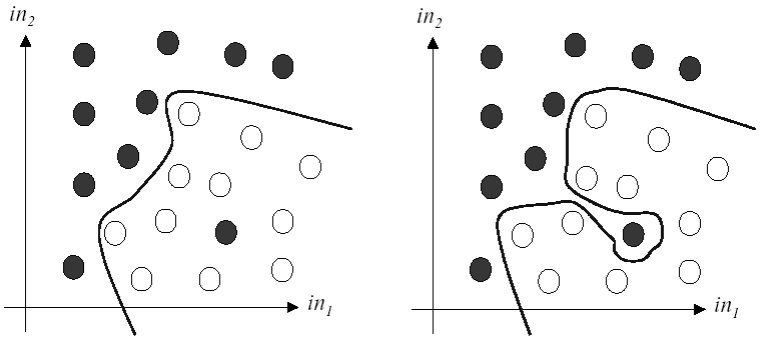
Levenberg-Marquardt

- **Like the quasi-Newton methods, the Levenberg-Marquardt algorithm was designed to approach second-order training speed without having to compute the Hessian matrix.**
- **The Levenberg-Marquardt algorithm uses this approximation to the Hessian matrix in the following Newton-like update:**

$$W(n+1) = W(n) - [J^T(n)J(n) + mI]^{-1} J^T(n)e(n)$$

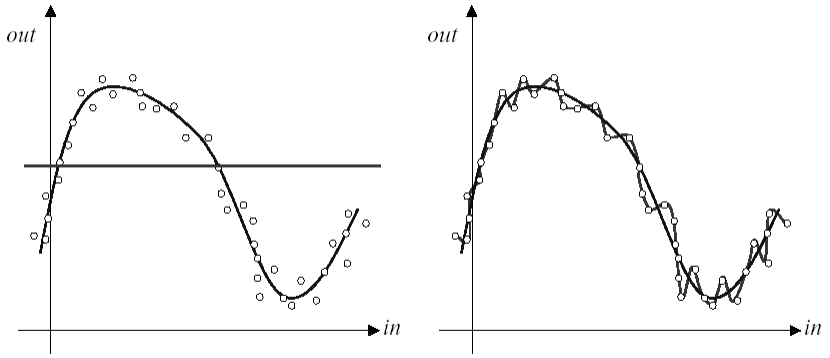
Under-Fitting and Over-Fitting

- **Recall the idea of getting a neural network to learn a classification decision boundary:**



- **Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately as that is likely to reduce the generalization ability.**

- **Similarly if our network is required to recover an underlying function from noisy data:**



Ignore the data ⇒

Get every data point ⇒

Preventing Under-fitting and Over-fitting

- **To** *prevent under-fitting* **we need to make sure that:**
  - **The network has enough hidden units to represent to required mappings.**
  - **We train the network for long enough so that the sum squared error cost function is sufficiently minimized.**
- **To** *prevent over-fitting* **we can:**
  - **Stop the training early – though if we stop too early we will get under-fitting.**
  - **Restrict the number of adjustable parameters the network has – e.g. by reducing the number of hidden units, or by forcing connections to share the same weight values.**
  - **Add some form of *regularization* term to the error function to encourage smoother network mappings.**
  - **Add noise to the training patterns to smear out the data points.**