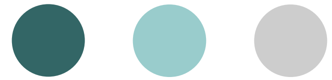


Multilayer feedforward networks

Back Propagation

Introduction

- Simple perceptron has ability to solve a problem depends on the condition that the input patterns of the problem be linearly separable (for threshold units) or linearly independent (for continuous and differentiable units).
- These limitations of simple perceptrons do not apply to feedforward networks with intermediate or "hidden" layers between the input and output layers.
- An example shows why multilayer networks can solve problems that cannot be solved by single-layer networks.

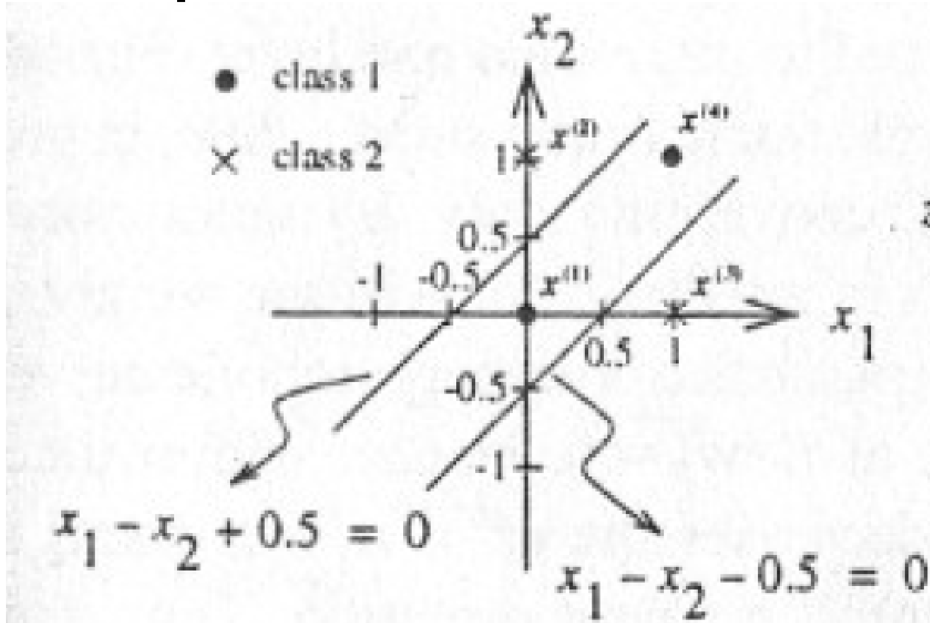
Example

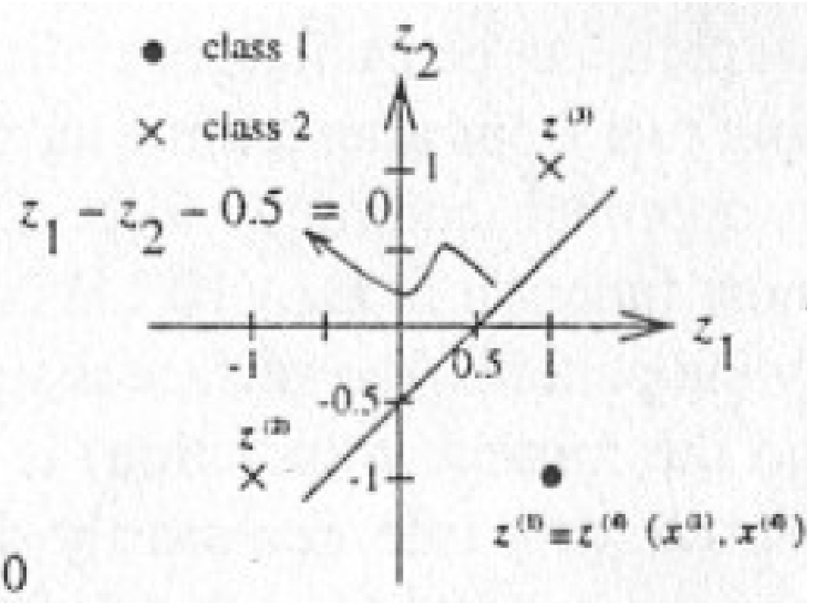
- This example illustrates how a linearly nonseparable problem is transformed to a linearly separable problem by a space transformation and thus can be solved by a multilayer perceptron network with LTUs.
- The problem we consider is the XOR problem.
- Input patterns and the corresponding desired outputs are:

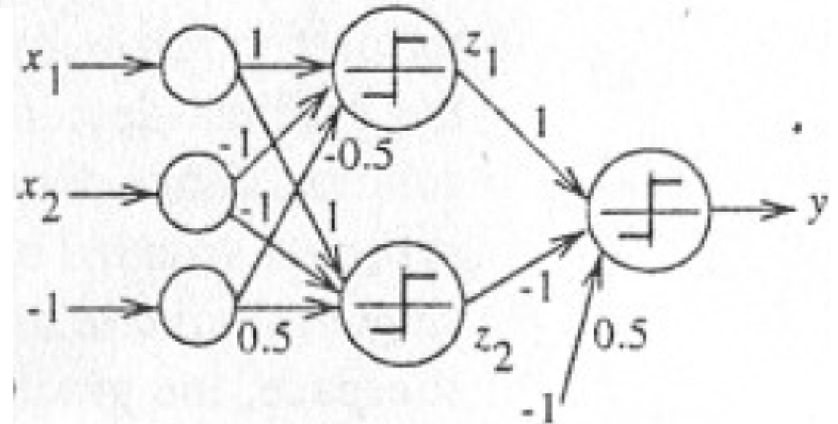
$$\left(\mathbf{x}^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, d^{(1)} = 1\right); \qquad \left(\mathbf{x}^{(2)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, d^{(2)} = -1\right);$$

$$\left(\mathbf{x}^{(3)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, d^{(3)} = -1\right); \qquad \left(\mathbf{x}^{(4)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, d^{(4)} = 1\right).$$

Example







- Multilayer perceptron for the XOR problem
  - (a) Input space (linearly nonseparable)
  - (b) Image space (linearly separable)
  - (c) A multilayer perceptron network for the XOR problem.

Example

- Two selected lines are

$$x_1 - x_2 + 0.5 = 0 \quad \text{and } x_1 - x_2 - 0.5 = 0$$

- Two LTU – outputs

$z_1 = sgn(x_1 - x_2 + 0.5)$ and $z_2 = sgn(x_1 - x_2 - 0.5)$

- z LTUs are defined

$$\left(\mathbf{z}^{(1)} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, d^{(1)} = 1\right); \qquad \left(\mathbf{z}^{(2)} = \begin{pmatrix} -1 \\ -1 \end{pmatrix}, d^{(2)} = -1\right);$$

$$\left(\mathbf{z}^{(3)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, d^{(3)} = -1\right); \qquad \left(\mathbf{z}^{(4)} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, d^{(4)} = 1\right).$$

- Output

$$y = \text{sgn}(z_1 - z_2 - 0.5)$$

Introduction

- Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions
- Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function
- There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods

Back propagation

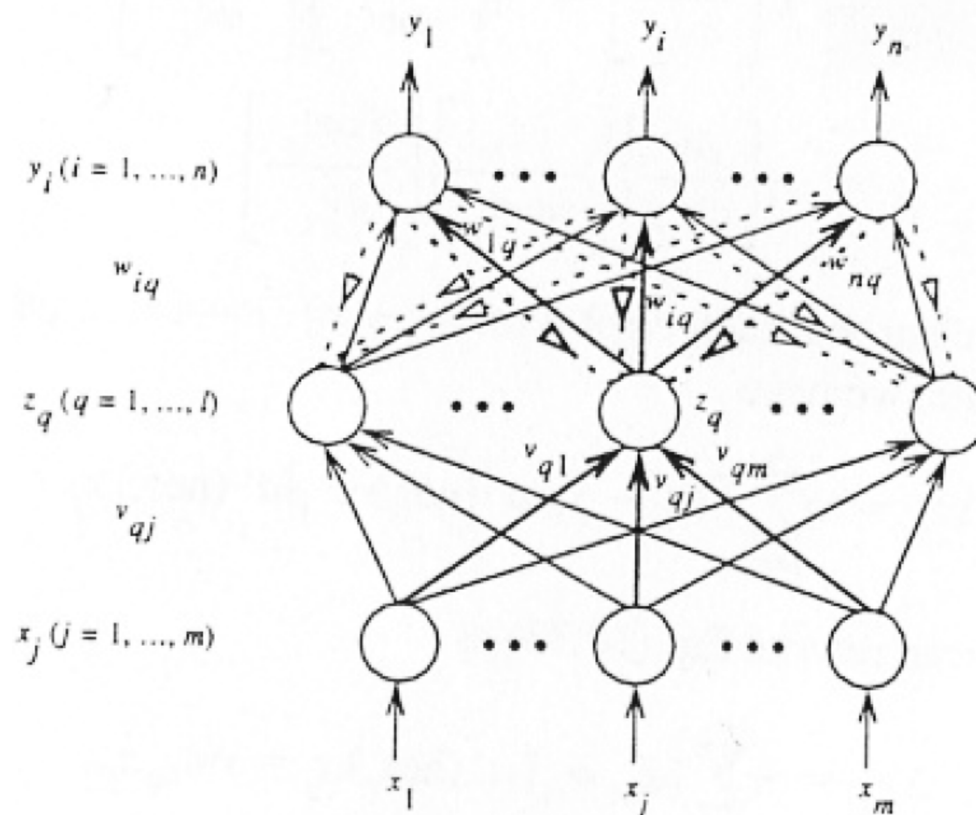
- The back-propagation learning algorithm is one of the most important historical developments in neural networks [Bryson & Ho 1969; Werbos 1974; LeCun 1985; Parker 1985; Rumelhart et al. 1986]
- This learning algorithm is applied to multilayer feedforward networks consisting of processing elements with continuous differentiable activation functions.
- Such networks associated with the back-propagation learning algorithm are also called *back-propagation networks.*

- Given a training set of input-output pairs $\{(x^{(k)}, d^{(k)})\}, k = 1, 2, \ldots p,$ the algorithm provides a procedure for changing the weights in a back-propagation network to classify the given input patterns correctly.
- The basis for this weight update algorithm is simply the gradient-descent method as used for simple perceptrons with differentiable units.

- For a given input-output pair $\{(x^{(k)}, d^{(k)})\}$, the back-propagation algorithm performs two phases of data flow.
  - First, the input pattern $x^{(k)}$ is propagated from the input layer to the output layer and, as a result of this forward flow of data, it produces an actual output $y^{(k)}$
  - Then the error signals resulting from the difference between $d^{(k)}$ and $y^{(k)}$ are *back-propagated* from the output layer to the previous layers for them to update their weights.

Example

- Let us consider a three-layer network



- First, let us consider an input-output training pair $(\boldsymbol{x}, \boldsymbol{d})$, where the superscript k is omitted for notation simplification. Given an input pattern x, a PE q in the hidden layer receives a net input of

$$\mathrm{net}_q = \sum_{j=1}^{m} v_{qj} x_j$$

- And produces an output of

$$z_q = a(\mathrm{net}_q) = a\left(\sum_{j=1}^{m} v_{qj} x_j\right)$$

- The net input for a PE i in the output layer is then

$$\text{net}_i = \sum_{q=1}^{l} w_{iq} z_q = \sum_{q=1}^{l} w_{iq} a\left(\sum_{j=1}^{m} v_{qj} x_j\right)$$

- And it produces an output of

$$y_i = a(\text{net}_i) = a\left(\sum_{q=1}^{l} w_{iq} z_q\right) = a\left(\sum_{q=1}^{l} w_{iq} a\left(\sum_{j=1}^{m} v_{qj} x_j\right)\right)$$

Example

- Error signal and back propagation. We first define a cost function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(d_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{n}[d_i - a(\text{net}_i)]^2 = \frac{1}{2}\sum_{i=1}^{n}\left[d_i - a\left(\sum_{q=1}^{l} w_{iq} z_q\right)\right]^2$$

- Then according to the gradient-descent method, the weights in the hidden-to-output connections are updated by

$$\Delta w_{iq} = -\eta \frac{\partial E}{\partial w_{iq}}$$

$$\Delta w_{iq} = -\eta \left[\frac{\partial E}{\partial y_i}\right]\left[\frac{\partial y_i}{\partial \text{net}_i}\right]\left[\frac{\partial \text{net}_i}{\partial w_{iq}}\right] = \eta[d_i - y_i][a'(\text{net}_i)][z_q]$$

$$\triangleq \eta \delta_{oi} z_q$$

Example

- $$\Delta w_{iq} = -\eta \left[\frac{\partial E}{\partial y_i}\right]\left[\frac{\partial y_i}{\partial \text{net}_i}\right]\left[\frac{\partial \text{net}_i}{\partial w_{iq}}\right] = \eta[d_i - y_i][a'(\text{net}_i)][z_q]$$
  $$\triangleq \eta \delta_{oi} z_q$$
- where δ_{oi} the *error signal* and its double subscript indicates the i –th node in the output layer. The error signal is defined by
  $$\delta_{oi} \triangleq -\frac{\partial E}{\partial \text{net}_i} = \left[\frac{\partial E}{\partial y_i}\right]\left[\frac{\partial y_i}{\partial \text{net}_i}\right] = [d_i - y_i][a'(\text{net}_i)]$$
- where net_i is the net input to PE *i* of the output layer and $a'(\text{net}_i) = \partial a(\text{net}_i)/\partial \text{net}_i$. The result thus far is identical to the delta learning rule for a single-layer perceptron whose input is now the output z_q of the hidden layer

Example

- where δ_{hq} the error signal of PE q in the hidden layer and is defined as

$$\delta_{hq} \triangleq -\left[\frac{\partial E}{\partial \mathrm{net}_q}\right] = -\left[\frac{\partial E}{\partial z_q}\right]\left[\frac{\partial z_q}{\partial \mathrm{net}_q}\right] = a'\left(\mathrm{net}_q\right)\sum_{i=1}^{n}\delta_{oi}w_{iq}$$

- where net_q is the net input to the hidden PE q
- The error signal of a PE in a hidden layer is different from the error signal of a PE in the output layer. Because of this difference, the above weight update procedure is called the ***generalized delta learning rule***.

Example

- For the weight update on the input-to-hidden connections, we use the chain rule with the gradient-descent method and obtain the weight update on the link weight connecting PE j in the input layer to PE q in the hidden layer,

$$\Delta v_{qj} = -\eta\left[\frac{\partial E}{\partial v_{qj}}\right] = -\eta\left[\frac{\partial E}{\partial \mathrm{net}_q}\right]\left[\frac{\partial \mathrm{net}_q}{\partial v_{qj}}\right] = -\eta\left[\frac{\partial E}{\partial z_q}\right]\left[\frac{\partial z_q}{\partial \mathrm{net}_q}\right]\left[\frac{\partial \mathrm{net}_q}{\partial v_{qj}}\right]$$

- it is clear that each error term $[d_i - y_i]$, $i = 1,2,\ldots,n$, is a function of z_q

Evaluating the chain rule, we have

$$\Delta v_{qj} = \eta\sum_{i=1}^{n}[(d_i - y_i)a'(\mathrm{net}_i)w_{iq}]\,a'(\mathrm{net}_q)\,x_j$$

$$\Delta v_{qj} = \eta\sum_{i=1}^{n}[\delta_{oi}w_{iq}]\,a'(\mathrm{net}_q)\,x_j = \eta\delta_{hq}x_j$$

Example

- We observe that the error signal δ_{hq} of a hidden PE q can be determined in terms of the error signals δ_{oi} of the PEs, y_i that it feeds. The coefficients are just the weights used for the forward propagation, but here they are propagating error signals (δ_{oi}) backward instead of propagating signals forward.
- This is shown by the dashed lines.
- This also demonstrates one important feature of the back-propagation algorithm-the update rule is local; that is, to compute the weight change for a given connection, we need only quantities available at both ends of that connection.

- The same form of the general weight learning rule, except that the learning signals, $r = \delta$,are different.
- The above derivation can be easily extended to the network with more than one hidden layer by using the chain rule continuously. In general, with an arbitrary number of layers, the backpropagation update rule is in the form

$$\Delta w_{ij} = \eta \delta_i x_j = \eta \delta_{output-i} x_{input-j}$$

- where "$output - i$" and "$input - j$" refer to the two ends of the connection from PE j to PE i, x_j is the proper input-end activation from a hidden PE or an external input, and δ_i is the learning signal (Eq. (10.35)) for the last (or output) layer of connection weights and defined by Eq. (10.35) for all the other layers.

- Consider a network with Q feedforward layers, $q = 1,2,\dots,Q$, and let ${}^{q}\mathrm{net}_i$ and ${}^{q}y$ denote the net input and output of the i-*th* unit in the q-*th* layer, respectively. The network has m input nodes and n output nodes. Let ${}^{q}w_{ij}$ denote the connection weight from ${}^{q-1}y_j$ *to* ${}^{q}y_i$
- Input: A set of training pairs $\{(x^{(k)}, d^{(k)}) | k = 1,\dots,p\}$ where the input vectors are augmented with the last elements as -1, that is, $x_{m+1}^{(k)} = 1$.

- Choose $\eta > 0$ and E_{max}
- Initialize the weights to small random values.
- $E = 0$

Step 1 (Begin training loop)

- Apply k-th input pattern to the input layer $(q = 1)$

$$^{q}y_i = {}^{1}y_i = x_i^{(k)} \quad \text{for all } \; i$$

Step 2 (Forward propagation)

- Propagate the signal forward through the network using

$$^{q}y_i = a({}^{q}\mathrm{net}_i) = a(\textstyle\sum_j {}^{q}w_{ij}^{\,q-1}y_j)$$

- **Step 3 (Output error measure)**
  - Compute error value

$$E = \frac{1}{2}\sum_{i=1}^{n}(d_i^{(k)} - {}^{Q}y_i)^2 + E$$

  - Compute error signal

$${}^{Q}\delta_i = (d_i^{k} - {}^{Q}y_i)\, a'({}^{Q}\mathrm{net}_i)$$

- **Step 4 (Error back propagation)**
  - Propagate the errors backward to update the weights and compute the *error signals* for the preceding layers

$$\Delta^{q}w_{ij} = \eta^{q}\delta_i{}^{q-1}y_j \quad \text{and} \quad {}^{q}w_{ij}^{new} = {}^{q}w_{ij}^{old} + \Delta^{q}w_{ij}$$

$${}^{q-1}\delta_i = a'({}^{q-1}net_i)\sum_j {}^{q}w_{ij}{}^{q}\delta_j \;\; \text{for} \;\; q = Q, Q-1, \ldots,$$

- Check whether the whole set of training data is cycled once
  - YES -> Step 6
  - NO -> Step 1

Step 6 (Total error checking)

- If $E < E_{max}$ -> END
- Else E=0 -> Step 1 (new training epoche)

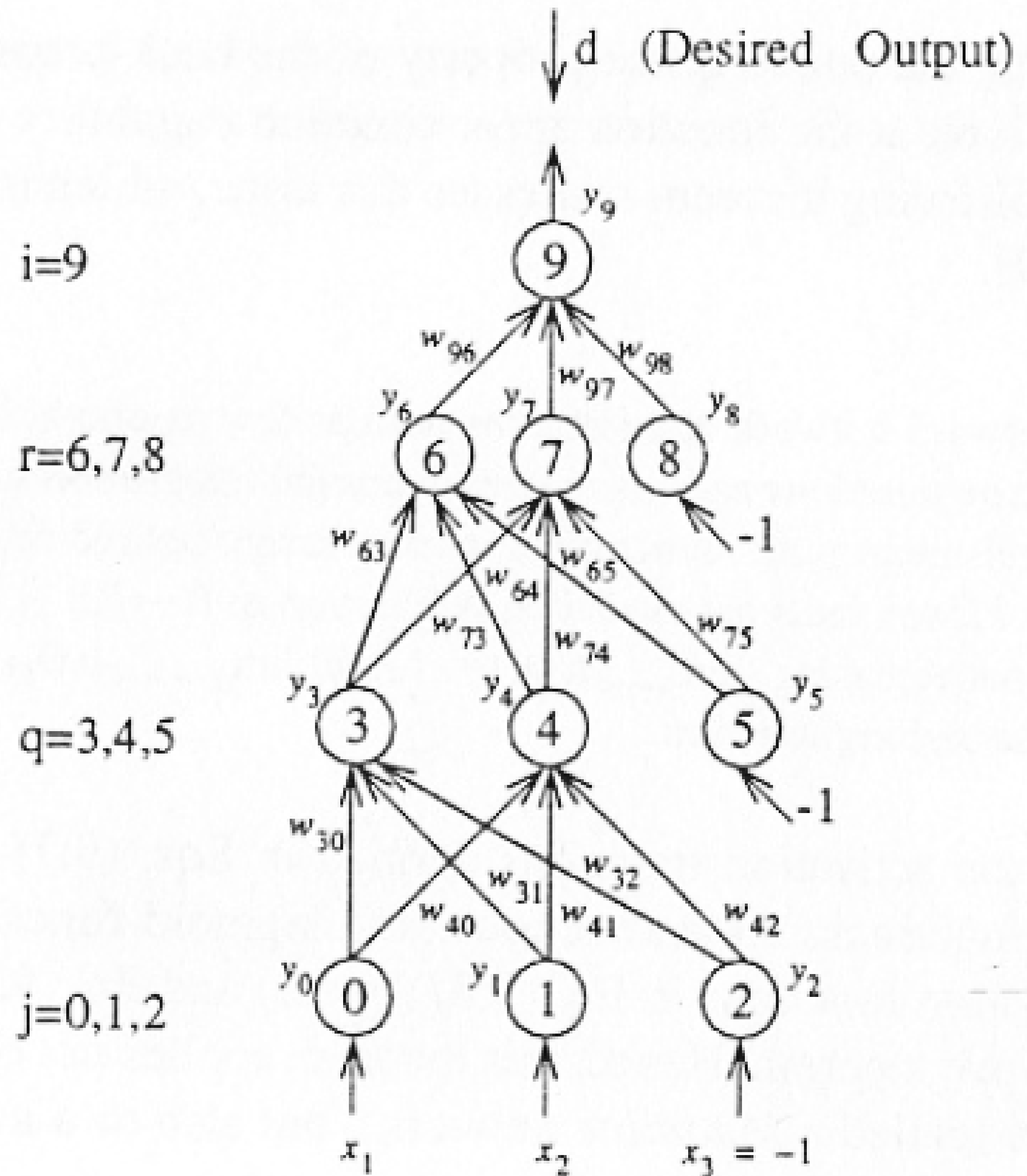
- Simple NN with unipolar sigmoid activation function

$$y = a(\text{net}) = \frac{1}{1 + e^{-\lambda \text{net}}}$$

- $\lambda = 1,$

$$a'(net) = y(1 - y)$$

Example



- Step 3

$$\delta_9 = a_9'(\text{net}_9)(d - y_9) = y_9(1 - y_9)(d - y_9)$$

- Step 4
  - Error signals

$$\delta_6 = a_6'(\text{net}_6)\sum_{i=9}^{9} w_{i6}\delta_i = y_6(1 - y_6)w_{96}\delta_9,$$

$$\delta_7 = a_7'(\text{net}_7)\sum_{i=9}^{9} w_{i7}\delta_i = y_7(1 - y_7)w_{97}\delta_9,$$

$$\delta_3 = a_3'(\text{net}_3)\sum_{r=6}^{7} w_{r3}\delta_r = y_3(1 - y_3)(w_{63}\delta_6 + w_{73}\delta_7),$$

$$\delta_4 = a_4'(\text{net}_4)\sum_{r=6}^{7} w_{r4}\delta_r = y_4(1 - y_4)(w_{64}\delta_6 + w_{74}\delta_7).$$

- Step 4
  - Weight update

$$\Delta w_{96} = \eta\delta_9 y_6, \qquad \Delta w_{97} = \eta\delta_9 y_7, \qquad \Delta w_{98} = \eta\delta_9 y_8 = -\eta\delta_9,$$

$$\Delta w_{63} = \eta\delta_6 y_3, \qquad \Delta w_{64} = \eta\delta_6 y_4, \qquad \Delta w_{65} = \eta\delta_6 y_5 = -\eta\delta_6,$$

$$\Delta w_{73} = \eta\delta_7 y_3, \qquad \Delta w_{74} = \eta\delta_7 y_4, \qquad \Delta w_{75} = \eta\delta_7 y_5 = -\eta\delta_7,$$

$$\Delta w_{30} = \eta\delta_3 y_0, \qquad \Delta w_{31} = \eta\delta_3 y_1, \qquad \Delta w_{32} = \eta\delta_3 y_2 = -\eta\delta_3,$$

$$\Delta w_{40} = \eta\delta_4 y_0, \qquad \Delta w_{41} = \eta\delta_4 y_1, \qquad \Delta w_{42} = \eta\delta_4 y_2 = -\eta\delta_4.$$

Algorithm

- There are two different ways in which this gradient descent algorithm can be implemented:
  - **incremental mode** (gradient is computed and the weights are updated after each input is applied to the network)
  - **batch mode** (all the inputs are applied to the network before the weights are updated)

Learning factors of Back Propagation

- Learning factors
  - initial weights
  - learning constant
  - cost function
  - update rule
  - size and nature of the training set
  - architecture (number of layers and number of nodes per layer)

Initial weights

- The initial weights of a multilayer feedforward network strongly affect the ultimate solution.
- Typically initialized at *small random* values.
- Equal initial weight values cannot train the network properly if the solution requires unequal weights to be developed
- The initial weights cannot be large, otherwise the sigmoids will saturate from the beginning and the system will become stuck at a local minimum
- One proper way is to choose the weight w_{ij}, in the range of $\left[-\frac{3}{\sqrt{k_i}}, +\frac{3}{\sqrt{k_i}},\right]$, where k_i is the number of PE j that feedforward to PE i (the number of input links of PE i)

Learning constant

- Another important factor that affects the effectiveness and convergence of the back-propagation learning algorithm significantly is the learning constant η
- There is no single learning constant value suitable for different training cases and η is usually chosen experimentally for each problem.
- A larger value of η could speed up the convergence but might result in overshooting, while a smaller value of η has a complementary effect. Values of η ranging from 10^{-3} to 10 have been used successfully for many computational back-propagation experiments

Learning constant

- Another problem is that the best values of the learning constant at the beginning of training may not be as good in later training.
- A more efficient approach is to use an adaptive learning constant.
- The intuitive method is to check whether a particular weight update has decreased the cost function.
  - If it has not, then the process has overshot and η should be reduced.
  - If several steps in a row have decreased the cost function, then we may be too conservative and should try increasing η.
- The learning constant should be updated according to the following rule:

$$\Delta\eta = \begin{cases} +a & if\ \Delta E < 0\ consistently \\ -b\eta & if\ \Delta E > 0 \\ 0 & \end{cases}$$

- where ΔE is the change in the cost function and a and b are positive constants

Learning constant

- The learning constant should be updated according to the following rule:

$$\Delta\eta(t+1) = \begin{cases} +a & \text{if } \bar{\lambda}(t-1)\lambda(t) > 0 \\ -b\eta & \text{if } \bar{\lambda}(t-1)\lambda(t) < 0 \\ 0 & \text{otherwise} \end{cases}$$

- where

$$\lambda(t) = \frac{\partial E}{\partial w_{ij}} \quad \text{and} \quad \bar{\lambda}(t) = (1-c)\lambda(t) + c\bar{\lambda}(t-1)$$

- where $c \in [0,1]$ is a constant
- Even without an adaptive rule, it may be appropriate to have different η for each pattern or each connection weight according to the fan-in of the corresponding node

Cost functions

- The quadratic cost function is not the only possible choice. The squared error term $(d_i - y_i)^2$ can be replaced by any other differentiable function $F(d_i, y_i)$
- Based on this new cost function, we can derive a corresponding update rule. It can be easily seen that only the error signal δ_{oi} for the output layer changes for different cost functions, while all the other equations of the back-propagation algorithm remain unchanged
- The cost functions usually used are those based on L_p norm $(1 \leq p \leq \infty)$ because of the advantage of easier mathematical formulation. Such cost functions are in the form of

$$E = \frac{1}{p}\sum_i (d_i - y_i)^p \quad where\ 1 \leq p < \infty$$

Cost functions

- The least squares criteria (L_2 norm) used in the quadratic cost function is widely employed because of its simplicity.
- L_∞ norm, which is also called the *Chebyshev norm*

$$E^\infty = \sup_i |d_i - y_i|$$

- where $\sup|\cdot|$ denotes a function selecting the largest component in the vector. The above definition implies that the overall error measure E^∞ equals the largest component of the error vector, while all other error components are negligible.
- We can derive the error signal δ_{oi} of the output layer as

$$\delta_{oi} = -\frac{\partial E}{\partial \mathrm{net}_i} = \begin{cases} 0 & if\ i \neq i* \\ a'(\mathrm{net}_i *)\mathrm{sgn}(d_i * - y_i *) & if\ i = i* \end{cases}$$

- where $i*$ is the index of the largest component of the output error vector. This error signal indicates that the only error with the back propagation is the largest one among the errors of all output nodes and that it is propagated from the $i*$th output node, where it occurs, back' to the preceding layers.

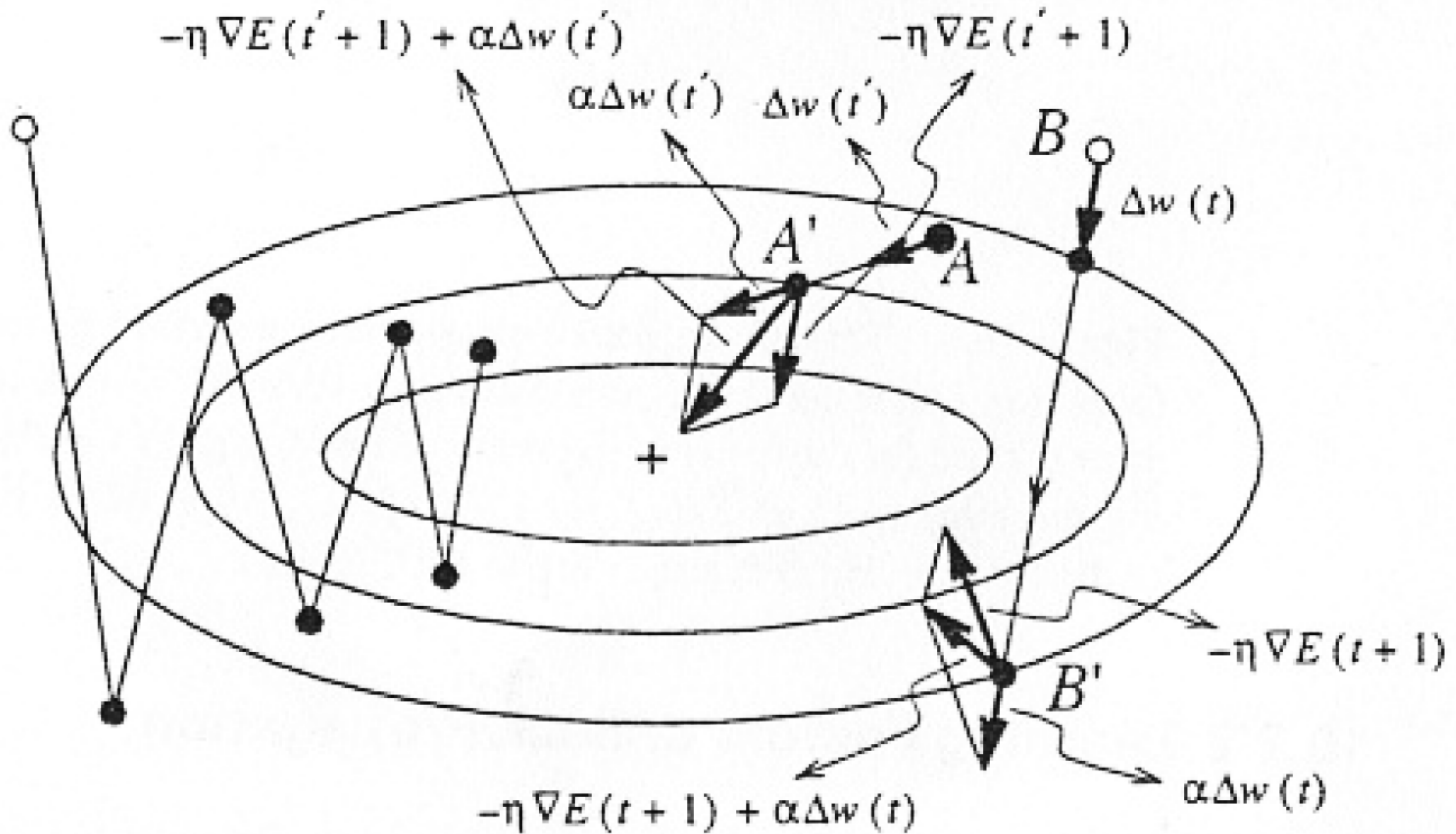
Momentum

- The gradient descent can be very slow if the learning constant η is small and can oscillate widely if η is too large.
- This problem essentially results from error surface valleys with steep sides but a shallow slope along the valley floor. One efficient and commonly used method **that allows a larger learning constant without divergent oscillations** occurring is the **addition of a momentum term** to the normal gradient-descent method.
- The idea is **to give each weight some inertia or momentum** so that it tends to change in the direction of the average downhill force that it feels. This scheme is implemented by giving a contribution from the previous time step to each weight change:

$$\Delta w(t) = -\eta \nabla E(t) + \alpha \Delta w(t-1)$$

- where $\alpha \in [0,1]$ is a momentum parameter and a value of 0.9 is often used

Momentum



- Trajectory without momentum (the left curve) has larger oscillations than the one with momentum (the right curves).
- We further observe from the right curves that the momentum can enhance progress toward the target point if the weight update is in the right direction (point A to A'). On the other hand, it can redirect movement in a better direction toward the target point in the case of overshooting (point B to B').
- Momentum term typically helps to speed up the convergence and to achieve an efficient and more reliable learning profile.
- A momentum term is useful with either pattern-by-pattern or batch-mode updating. In the batch mode, it has the effect of complete averaging over the patterns. Although the averaging is only partial in the pattern-by- pattern case, it can leave some beneficial fluctuations in the trajectory

- Although the gradient-descent (or steepest-descent) method is one of the simplest optimization techniques, it is not a very effective one.
- Further numerical optimization theory [Luenberger, 1976] can be applied to make convergence of the back propagation algorithm significantly faster.
- Numerical optimization theory provides a rich and robust set of techniques which can be applied to neural networks to improve learning rates.

Update rules

- The gradient-descent method considers only the first-order derivative of an error function. It is helpful to take into account higher-order derivatives. Using Taylor's series expansion on $E(w)$ around the current point w_0 we have

$$E(\boldsymbol{w}) = E(\boldsymbol{w_0}) + (\boldsymbol{w} - \boldsymbol{w}_0)^T \nabla E(\boldsymbol{w}_0) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_0)^T \boldsymbol{H}(\boldsymbol{w})(\boldsymbol{w} - \boldsymbol{w}_0) + \cdots$$

- where $\boldsymbol{H}(\boldsymbol{w})$ is called a Hessian matrix and is the the second derivate evaluated at $\boldsymbol{w}_0$

$$\boldsymbol{H}(\boldsymbol{w}) \triangleq \nabla^2 E(\boldsymbol{w}) \quad \text{or} \quad H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$$

- To find the minimum of $E(\boldsymbol{w})$, set gradient to zero

$$\nabla E(\boldsymbol{w}) = \nabla E(\boldsymbol{w}_0) + \boldsymbol{H}(\boldsymbol{w})(\boldsymbol{w} - \boldsymbol{w}_0) + \cdots = 0$$

- If we ignore the third- and higher-order terms, we obtain

$$\boldsymbol{w} = \boldsymbol{w}_0 - \boldsymbol{H}^{-1}(\boldsymbol{w}) \nabla E(\boldsymbol{w}_0)$$

- or use k to indicate the k –th step of learning, we obtain

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \boldsymbol{H}^{-1}\left(\boldsymbol{w}^{(k)}\right) \nabla E\left(\boldsymbol{w}^{(k)}\right)$$

Update rules

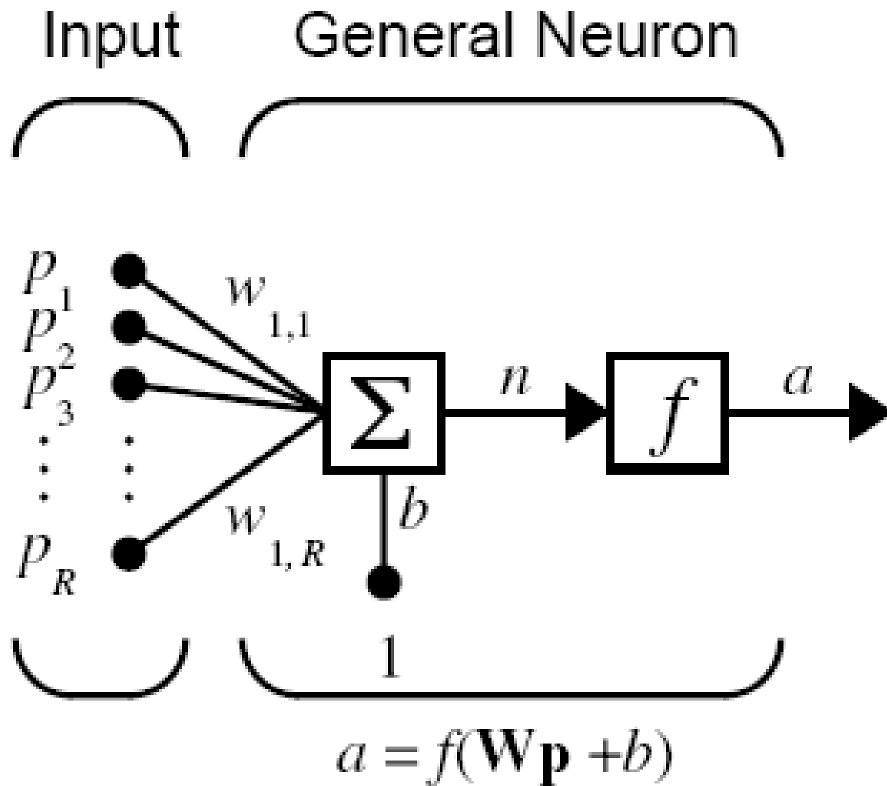
- This is called *Newton's method* of weight updating. Newton's method uses the second derivative in addition to the gradient to determine the next step direction and_step size.
- It can converge quadratically when close to the solution of a convex function.
- However, there are several drawbacks in Newton's method.
  - In order to converge, it requires a good initial estimate of the solution.
  - For a convex function, it can converge quickly; however, for a nonconvex function, it may easily converge to a local minimum or a saddle point.
  - The key drawback is that each iteration requires computation of the Hessian matrix and also its inversion, and so the method is expensive in terms of both storage and computation requirements.
- Hence, it is not a practical technique, and alternative or revised methods have been proposed. These include the conjugate-direction method and the quasi-Newton method [Luenberger, 1976].

Objective

- The primary objective is to explain how to use the backpropagation training functions in the toolbox to train feedforward neural networks to solve specific problems. There are generally four steps in the training process:
  1. Assemble the training data.
  2. Create the network object.
  3. Train the network.
  4. Simulate the network response to new inputs.

Arcitecture

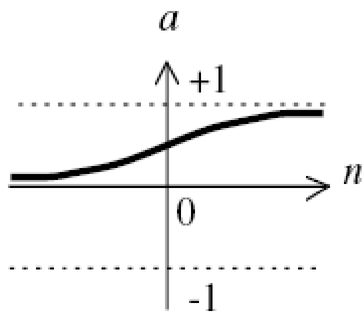
Neuron Model (logsig, tansig, purelin)

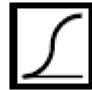


Where

R = number of elements in input vector

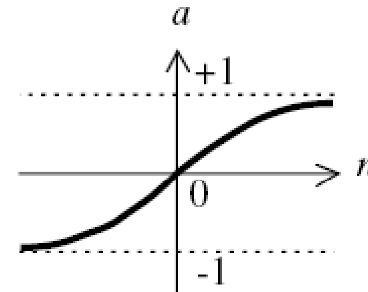
Neuron Model (logsig, tansig, purelin)

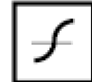




$a = logsig(n)$

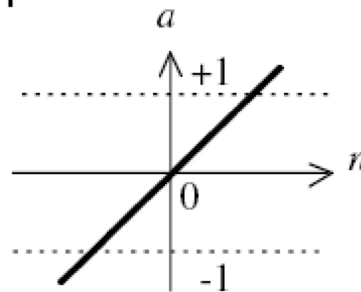
Log-Sigmoid Transfer Function

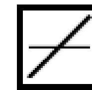




$a = tansig(n)$

Tan-Sigmoid Transfer Function



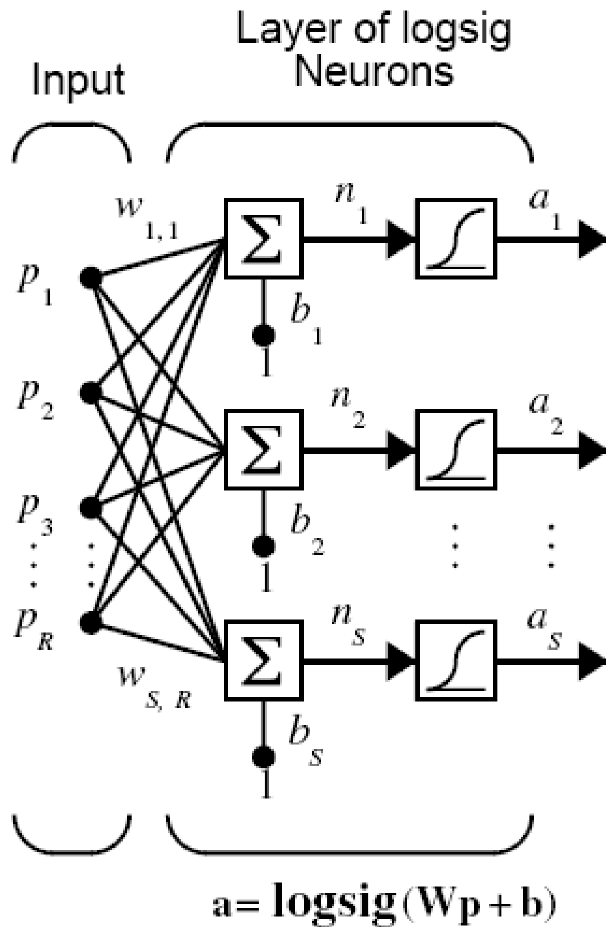


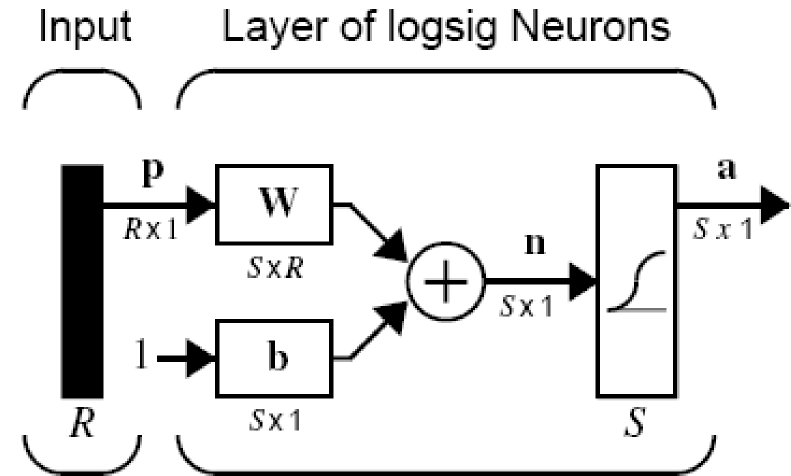
$a = purelin(n)$

Linear Transfer Function

Feedforward network

One layer





$$\mathbf{a} = \mathbf{logsig}(\mathbf{Wp} + \mathbf{b})$$

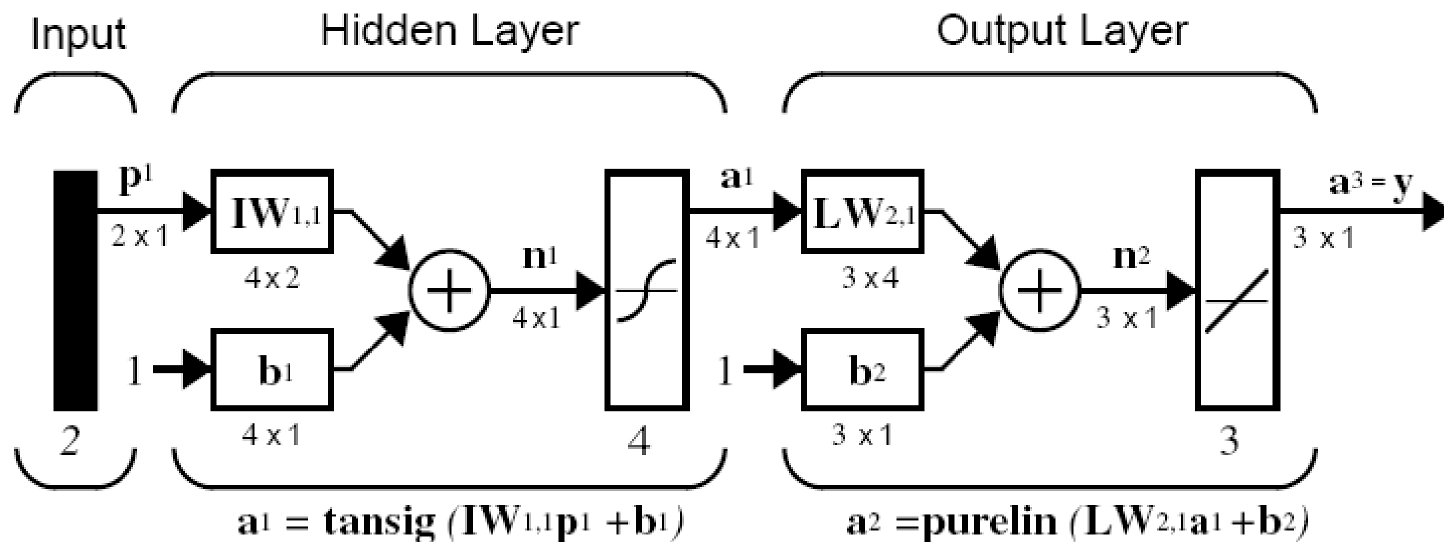
Where...

R = number of elements in input vector

S = number of neurons in layer

Feedforward network

Two layers (Hidden and output)



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities arbitrarily well, given sufficient neurons in the hidden layer

Creating a Network (newff)

```
net = newff(P,T,S,TF,BTF,BLF,PF,IPF,OPF,DDF)
```

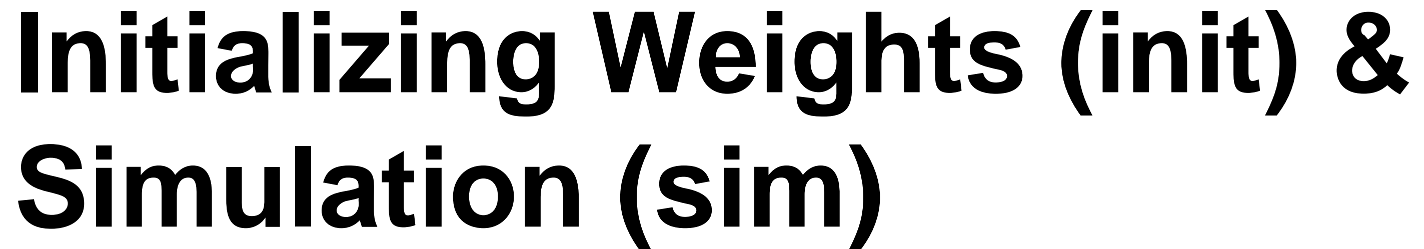
NEWFF(P,T,S,TF,BTF,BLF,PF,IPF,OPF,DDF) takes,

- P - RxQ1 matrix of Q1 representative R-element input vectors.
- T - SNxQ2 matrix of Q2 representative SN-element target vectors.
- Si - Sizes of N-1 hidden layers, S1 to S(N-1), default = [].
  (Output layer size SN is determined from T.)
- TFi - Transfer function of ith layer. Default is 'tansig' for
  hidden layers, and 'purelin' for output layer.
- BTF - Backprop network training function, default = 'trainlm'.
- BLF - Backprop weight/bias learning function, default = 'learngdm'.
- PF - Performance function, default = 'mse'.
- IPF - Row cell array of input processing functions.
  Default is {'fixunknowns','remconstantrows','mapminmax'}.
- OPF - Row cell array of output processing functions.
  Default is {'remconstantrows','mapminmax'}.
- DDF - Data division function, default = 'dividerand';
  and returns an N layer feed-forward backprop network

Example

- `P = [0 1 2 3 4 5 6 7 8 9 10];`
- `T = [0 1 2 3 4 3 2 1 2 3 4];`

- Here a network is created with one hidden layer of 5 neurons, (1 input, 1 output)
  - `net = newff(P,T,5);`
  - `Y = sim(net,P);`
  - `plot(P,T,P,Y,'o')`
- Training
  - `net.trainParam.epochs = 50;`
  - `net = train(net,P,T);`
  - `Y = sim(net,P);`
  - `plot(P,T,P,Y,'o')`



Initializing Weights (init) & Simulation (sim)

- Init network:

```
net = init(net);
```

- Simulation for a single input vector:

```
p = [1;2];
a = sim(net,p)
```

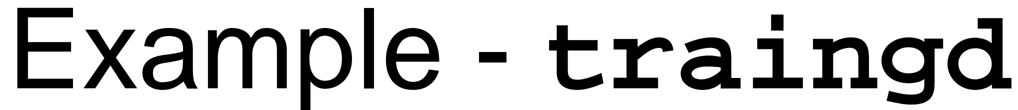
- Simulation for a concurrent set of three inputvectors:

```
p = [1 3 2;2 4 1];
a=sim(net,p)
```

Training

- The network can be trained for
  - function approximation (nonlinear regression),
  - pattern association,
  - pattern classification
- The training process requires a set of examples of proper network behavior - network inputs p and target outputs t
- During training the weights and biases of the network are iteratively adjusted to minimize the network performance function `net.performFcn`
- The default performance function for feedforward networks is mean square error mse - the average squared error between the network outputs a and the target outputs t

- There are seven training parameters associated with traingd:
  - **`epochs`**
  - **`show`**
  - **`goal`**
  - **`time`**
  - **`min_grad`**
  - **`max_fail`**
  - **`lr`**

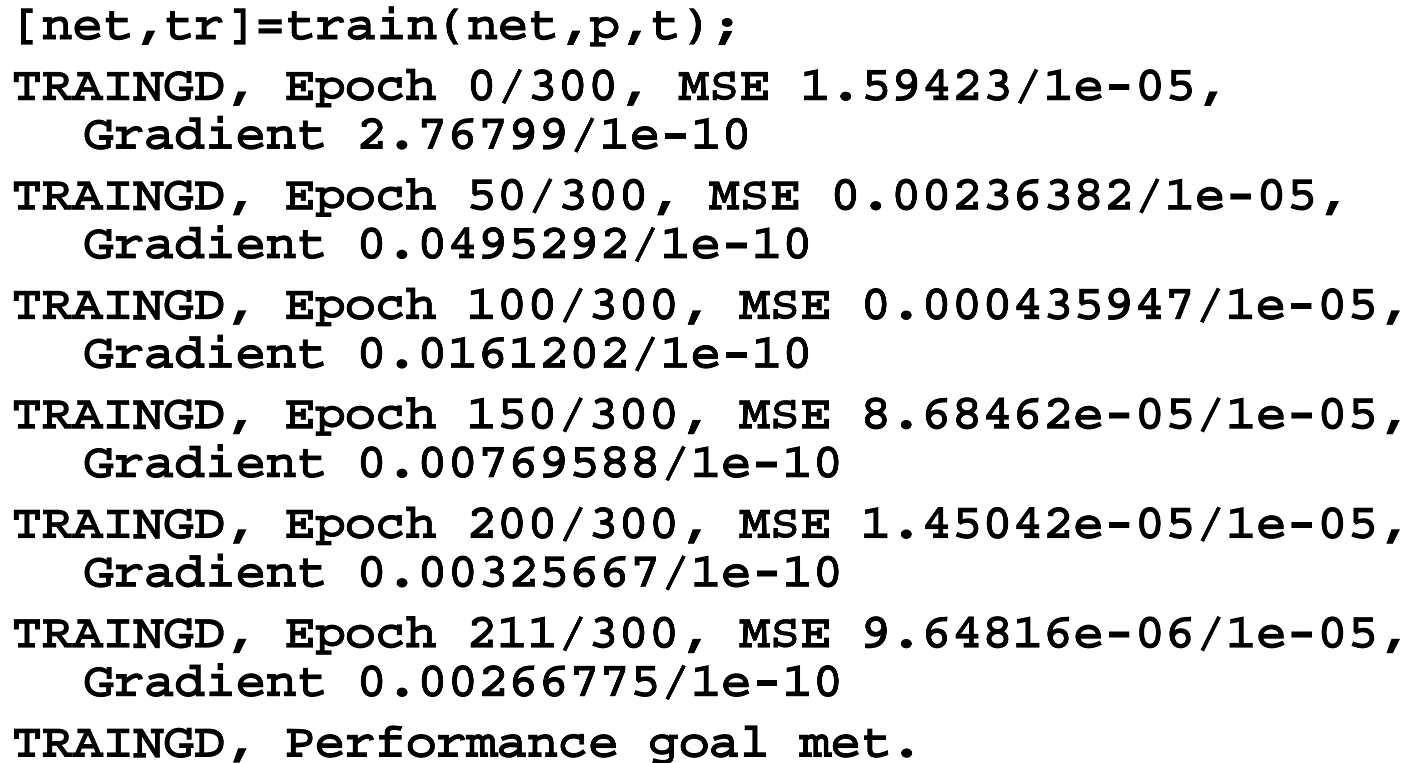


Example - `traingd`

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(p,t,3,{'tansig','purelin'},'traingd');
```

- modify some of the default training parameters

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```



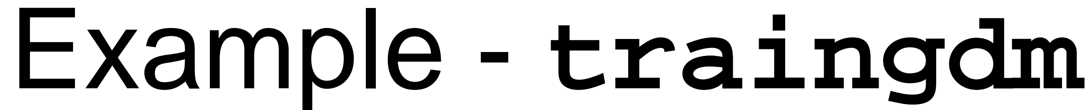
```
[net,tr]=train(net,p,t);
TRAINGD, Epoch 0/300, MSE 1.59423/1e-05,
  Gradient 2.76799/1e-10
TRAINGD, Epoch 50/300, MSE 0.00236382/1e-05,
  Gradient 0.0495292/1e-10
TRAINGD, Epoch 100/300, MSE 0.000435947/1e-05,
  Gradient 0.0161202/1e-10
TRAINGD, Epoch 150/300, MSE 8.68462e-05/1e-05,
  Gradient 0.00769588/1e-10
TRAINGD, Epoch 200/300, MSE 1.45042e-05/1e-05,
  Gradient 0.00325667/1e-10
TRAINGD, Epoch 211/300, MSE 9.64816e-06/1e-05,
  Gradient 0.00266775/1e-10
TRAINGD, Performance goal met.
```

- Simulate

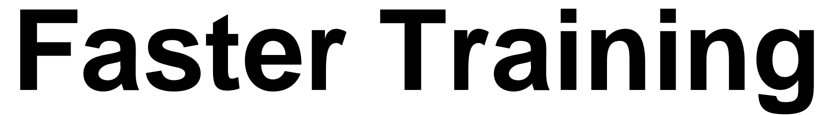
```
a = sim(net,p)
a =
-1.0010 -0.9989 1.0018 0.9985
```

Batch Gradient Descent with Momentum (`traingdm`)

- Another batch algorithm for feedforward networks that often provides faster convergence: traingdm, steepest descent with momentum
- Acting like a lowpass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network can get stuck in a shallow local minimum. With momentum a network can slide through such a minimum
- You can add momentum to backpropagation learning **by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the backpropagation rule**. The magnitude of the effect that the last weight change is allowed to have is mediated by a **momentum constant**, mc, which can be any number between 0 and 1
  - When the momentum constant is 0, a weight change is based solely on the gradient
  - When the momentum constant is 1, the new weight change is set to equal the last weight change and the gradient is simply ignored




```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
net=newff(p,t,3,{'tansig','purelin'},'traingdm');
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.mc = 0.9;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
[net,tr]=train(net,p,t);
TRAINGDM, Epoch 0/300, MSE 3.6913/1e-05, Gradient
4.54729/1e-10
TRAINGDM, Epoch 50/300, MSE 0.00532188/1e-05, Gradient
0.213222/1e-10
TRAINGDM, Epoch 100/300, MSE 6.34868e-05/1e-05, Gradient
0.0409749/1e-10
TRAINGDM, Epoch 114/300, MSE 9.06235e-06/1e-05, Gradient
0.00908756/1e-10
TRAINGDM, Performance goal met.
a = sim(net,p)
a =
-1.0026 -1.0044 0.9969 0.9992
```



Faster Training

- heuristic techniques
  - variable learning rate backpropagation (`traingda, traingdx`)
  - resilient backpropagation (`trainrp`)
- standard numerical optimization techniques
  - Conjugate Gradient Algorithms (`traincgf,`
  - `traincgp, traincgb, trainscg`)
  - Quasi-Newton Algorithms (`trainbfg, trainoss`)
  - Levenberg-Marquardt (`trainlm`)

- We always require that training data be *sufficient* and *proper*. However, there is no procedure or rule suitable for all cases in choosing training data.
- One rule of thumb is that training data should cover the entire expected input space and then during the training process select training-vector pairs randomly from the set.
- More precisely, assume that the input space is linearly separable into M disjoint regions with boundaries being part of the hyperplanes. Let P be the lower bound on the number of training patterns. Then choosing P such that $\frac{P}{M} \gg 1$ hopefully allows the network to discriminate pattern classes using fine piecewise hyperplane partitioning.
- In some situations, scaling or normalization is necessary to help the learning.
- For example, if the output function is sigmoidal, then the output values need to be scaled properly.

Training data and generalization

- The back-propagation network is good at generalization.
- The network is said to generalize well when it sensibly interpolates input patterns that are new to the network.
- Networks with too many trainable parameters for the given amount of training data **learn well but do not generalize well**. This phenomenon is usually called **overfitting**.
- With too few trainable parameters, the network fails to learn the training data and performs very poorly on the test data.
- In order to improve the ability of a network to generalize from a training data set to a test data set, it is **desirable that small changes in the input space of a pattern do not change the output components**. This can be done by including variations in the input space of training patterns as part of the training set but this is computationally very expensive.

Training data and generalization

- One way is to form a cost function that is the sum of the normal cost term E_f found in the back-propagation algorithm and an additional term that is a function of the Jacobian:

$$E_b = \frac{1}{2}\left(\frac{\partial E_f}{\partial x_1}\right)^2 + \frac{1}{2}\left(\frac{\partial E_f}{\partial x_2}\right)^2 + \cdots + \frac{1}{2}\left(\frac{\partial E_f}{\partial x_m}\right)^2$$

- where x_j refers to the j –th input. The rationale for this approach is that if the **input changes slightly, the cost function E_f should not change**. To minimize the new cost function, a double back-propagation network was constructed such that one back propagation was for $\frac{\partial E_f}{\partial w_j}$ and the other for $\frac{\partial E_b}{\partial w_j}$.
- This technique has been shown to improve the generalization capability of a trained network by forcing the output to be insensitive to incremental changes in the input

Pruning neural networks

- One possible method of obtaining a neural network of appropriate size for a particular problem is **to start with a larger network and then prune it to the desired size**.
- network-pruning techniques for general feedforward or recurrent networks
  - Weight Decay
  - Connection and Node Pruning

Weight Decay

- One approach to having the network itself remove nonuseful connections during training is called weight decay. This is to give each connection w_{ij} a tendency to decay to zero so that connections disappear unless reinforced

$$w_{ij}(k+1) = -\eta \frac{\partial E}{\partial w_{ij}}(k) + \beta w_{ij}(k)$$

- for some positive $\beta < 1$. The same weight decay term was introduced when we discussed the basic unsupervised learning rules. With the weight decay term, weights that do not have much influence on decreasing the error while learning (i.e., have $\frac{\partial E}{\partial w_{ij}} \approx 0$) experience an exponential time decay:

$$w_{ij}(k) \approx \beta^n w_{ij}(0)$$

- This is equivalent to adding a penalty term w_{ij}^2 to the original cost function E, changing it to

$$E' = E + \gamma \sum_{ij} w_{ij}^2$$

- and performing gradient descent $\Delta w_{ij} = -\eta \partial E' / \partial w_{ij}$ on the resulting total E'. The β parameter is then just $\beta = 1 - 2\gamma\eta$.

- Although previous equations penalizes the use of more weights than necessary, it discourages the use of only large weights. That is, one large weight costs much more than many small ones. To solve this problem, the following different penalty term can be used:

$$E' = E + \gamma \sum_{ij} \frac{w_{ij}^2}{1 + w_{ij}^2}$$

- which is equivalent to making β dependent on w_{ij}

$$\beta_{ij} = 1 - \frac{2\gamma\eta}{\left(1 + w_{ij}^2\right)^2}$$

- so that the **small w_{ij} decay more rapidly** than the large ones

Weight Decay

- The above weight decay rules aim at removing unnecessary weights (i.e., *connections)*. However, we often want to remove whole nodes so that we can start with an excess of hidden nodes and later discard those not needed.
- This can be achieved by making weight decay rates larger for nodes that have small outputs or that already have small incoming weights [Chauvin, 1989].

$$\beta_i = 1 - \frac{2\gamma\eta}{\left(1 + \sum_j w_{ij}^2\right)^2}$$

- and the same β_i is used for all connections feeding node i

Connection and Node Pruning

- Instead of waiting for weight decay in the learning process: we can trim the trained network by removing unimportant connections and/or nodes. With this approach, it is necessary to *retrain* the network after the "brain damage," but this retraining is usually rather fast.
- When a network has learned, then an arbitrary setting of a weight w_{ij} to zero (which is equivalent to eliminating the connection from node j to node i) typically results in an increase in the error E. Hence, **efficient pruning means finding the subset of weights that, when set to zero, lead to the smallest increase in E.**
- The same concept is applied to node elimination. Sietsma and Dow [1988] proposed that all nodes of a network are examined when the entire set of training data is presented and that each node (along with its synaptic connections) that does not change state or replicate another node is removed. Thus, they found a subset of the network that had the same

Connection and Node Pruning

- Another approach to connection and/or node pruning is based on **estimation of the *sensitivity* of the global cost function** to the elimination of each connection.
- The idea is to keep track of the incremental changes in the connection weights during the learning process. The connections are then **ordered by decreasing sensitivity values**, and so the network can be efficiently pruned by **discarding the last items on the sorted list**. In terms of possible connection elimination, the sensitivity with respect to w_{ij} denoted *by* S_{ij} is defined as

$$S_{ij} = E\left(w_{ij} = 0\right) - E\left(w_{ij} = w_{ij}^{f}\right)$$

- where w_{ij}^{f} is the final value of the connection on completion of the training phase and all other weights are fixed at their final states. To calculate the sensitivity value S_{ij} after training, we need to present all the training patterns to the network for each special weight setting $w_{ij} = 0$. It is better if we can

Connection and Node Pruning

$$S_{ij} = \frac{E\left(w_{ij} = w_{ij}^{f}\right) - E(w_{ij} = 0)}{w_{ij}^{f} - 0} w_{ij}$$

- Since a typical learning process does not start with $w_{ij} = 0$ but rather with some small, randomly chosen initial value w_{ij}^{i},

$$S_{ij} = \frac{E\left(w_{ij} = w_{ij}^{f}\right) - E(w_{ij} = {w_{ij}}^{i})}{w_{ij}^{f} - w_{ij}^{i}} w_{ij}$$

- Furthennore, the numerator can be approximated by

$$E\left(w_{ij} = w_{ij}^{f}\right) - E\left(w_{ij} = w_{ij}^{i}\right) = \int_{I \to F} \frac{\partial E}{\partial w_{ij}} dw_{ij}$$

- where the integration is along the line from an initial point *I* in the weight space to the final weight state *F*. This expression can be further approximated by replacing the integral by a summation over the discrete steps that the network takes while learning

Connection and Node Pruning

- Thus, the estimated sensitivity to the removal of connection w_{ij} is evaluated as

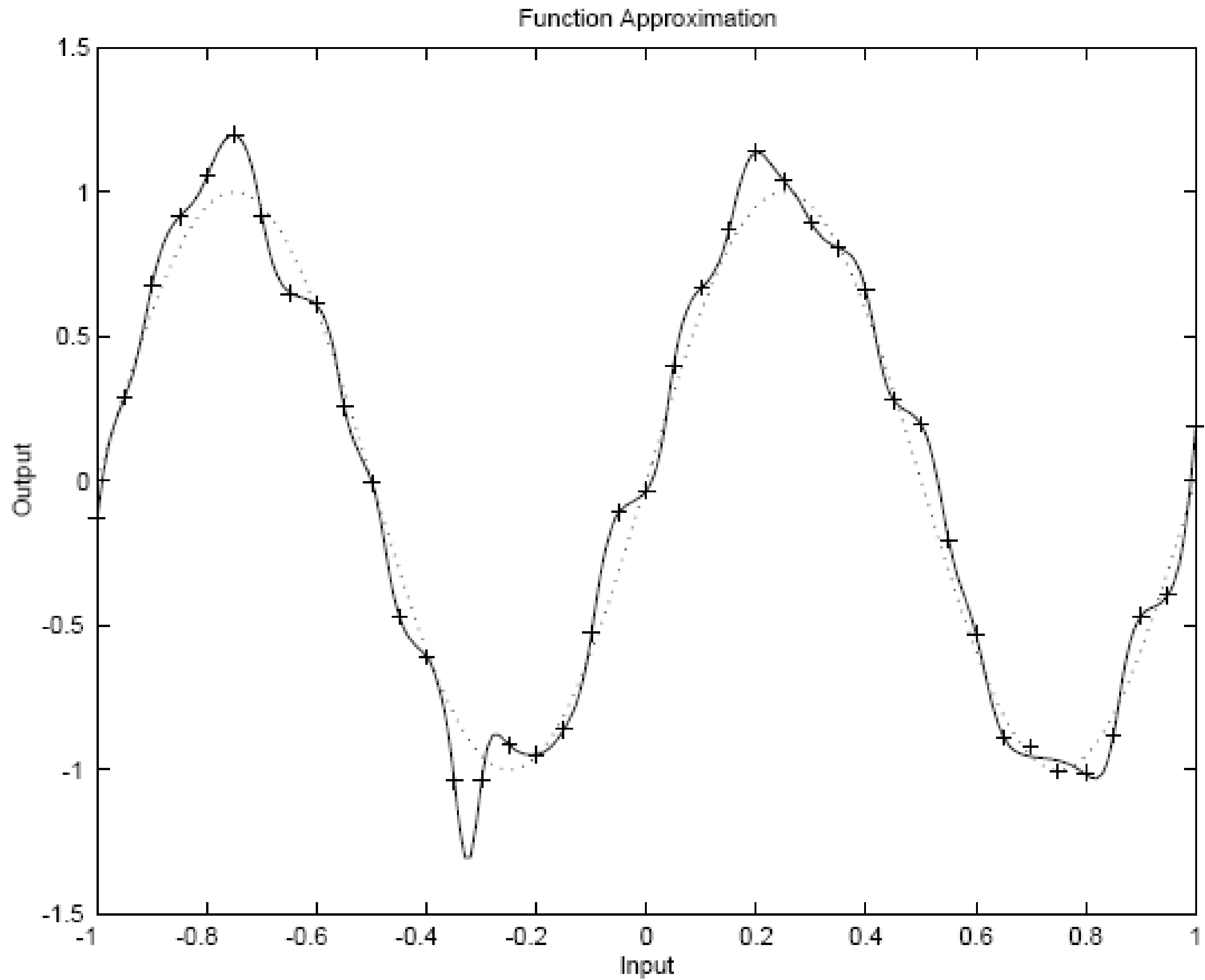
$$S_{ij} = -\sum_{k=0}^{N-1} \frac{\partial E}{\partial w_{ij}}(k) \Delta w_{ij}(k) \frac{w_{ij}^{f}}{w_{ij}^{f} - w_{ij}^{i}}$$

- where N is the number of training epochs. The terms used are readily available during the nonnal course of training using gradient-descent methods.
- For the special case of back propagation, weights are updated according $\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$, then

$$S_{ij} = -\sum_{k=0}^{N-1} \left(\Delta w_{ij}(k)\right)^{2} \frac{w_{ij}^{f}}{\eta(w_{ij}^{f} - w_{ij}^{i})}$$

- The size of a hidden layer is a fundamental question often raised in the application of multilayer feedforward networks to real-world problems.
- The exact analysis of this issue is rather difficult because of the complexity of the network mapping and the nondeterministic nature of many successfully completed training procedures.
- **The size of a hidden layer is usually determined experimentally**.
- One empirical guideline is as follows. For a network of reasonable size (e.g., hundreds or thousands of inputs), the size of hidden nodes needs to be only a relatively small fraction of the input layer.
  - If the network fails to converge to a solution, it may be that more hidden nodes are required.
  - If it does converge, you may try fewer hidden nodes and then settle on a size based on overall system performance

Improving Generalization

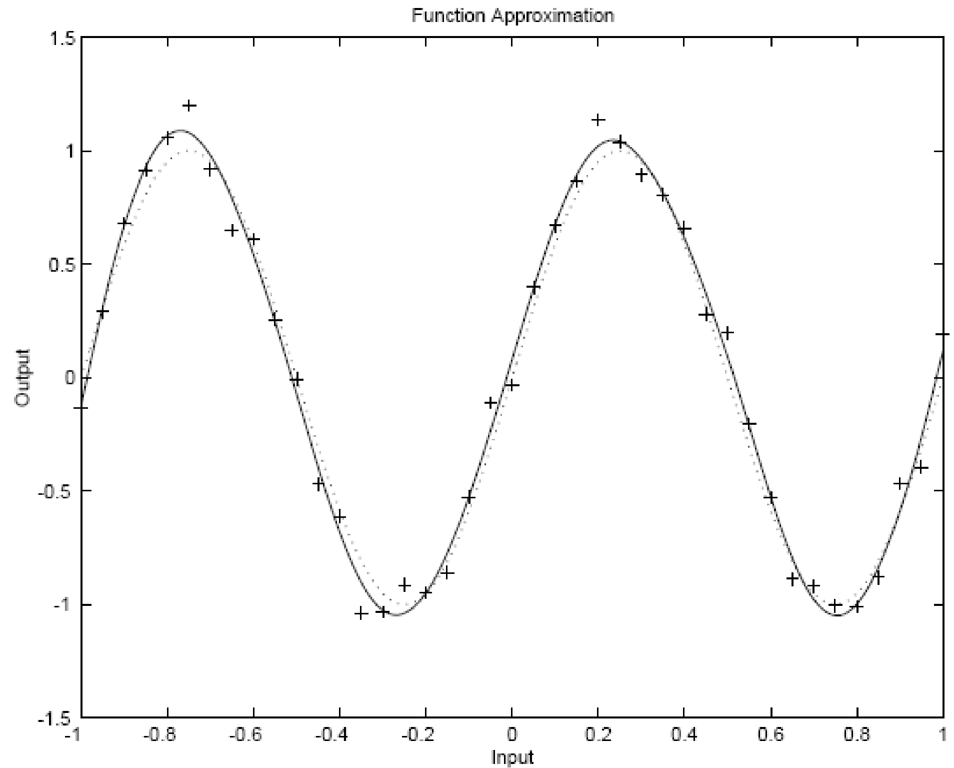


Regularization

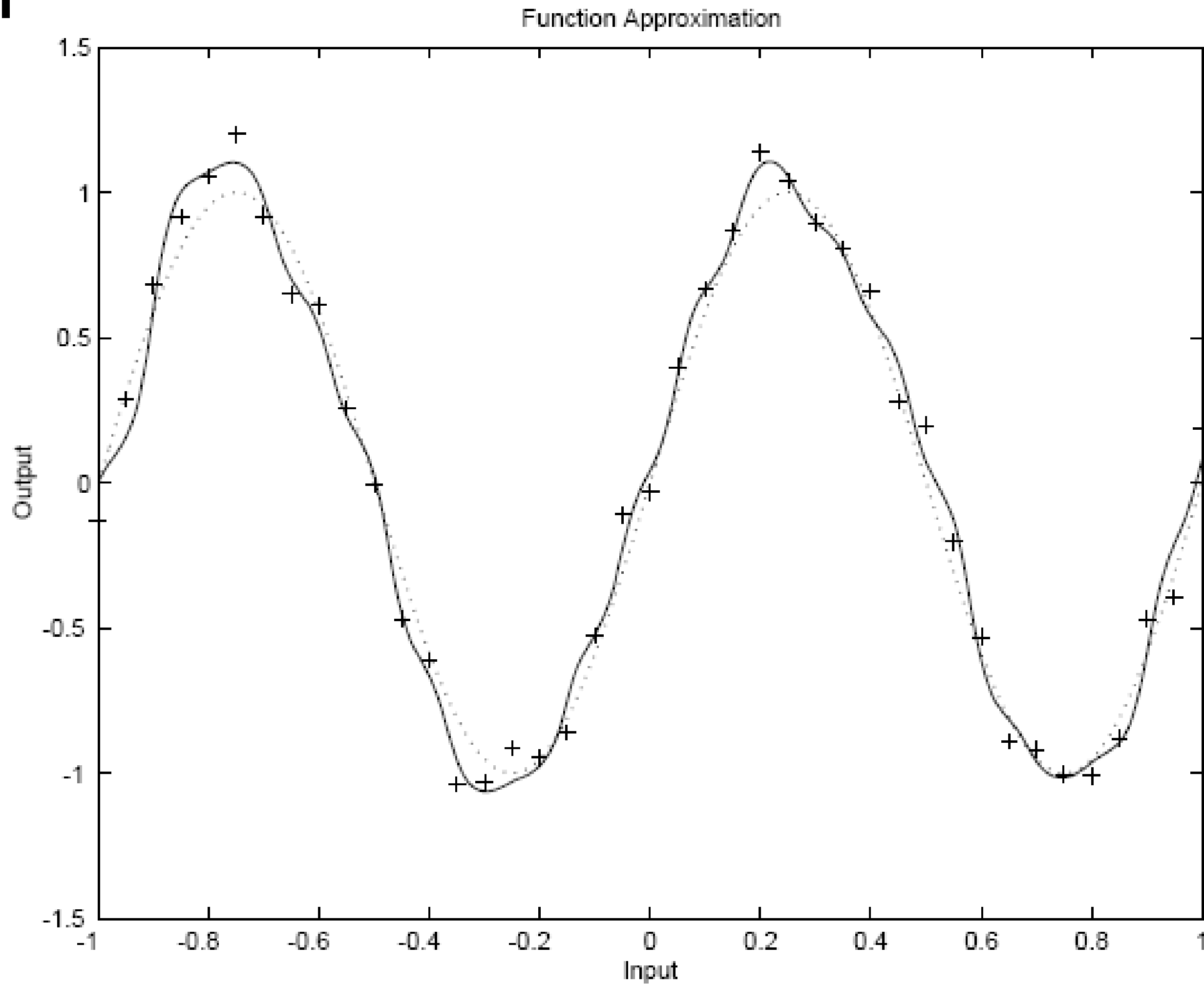
$$F = mse = \frac{1}{N}\sum_{i=1}^{N}(e_i)^2 = \frac{1}{N}\sum_{i=1}^{N}(t_i - a_i)^2$$

$$msereg = \gamma mse + (1-\gamma) msw$$

$$msw = \frac{1}{n}\sum_{j=1}^{n} w_j^2$$



Early Stopping



- **60% are used for training**
- **20% are used to validate that the network is generalizing and to stop training before overfitting**
- **The last 20% are used as a completely independent test of network generalization**

Train, validation and test