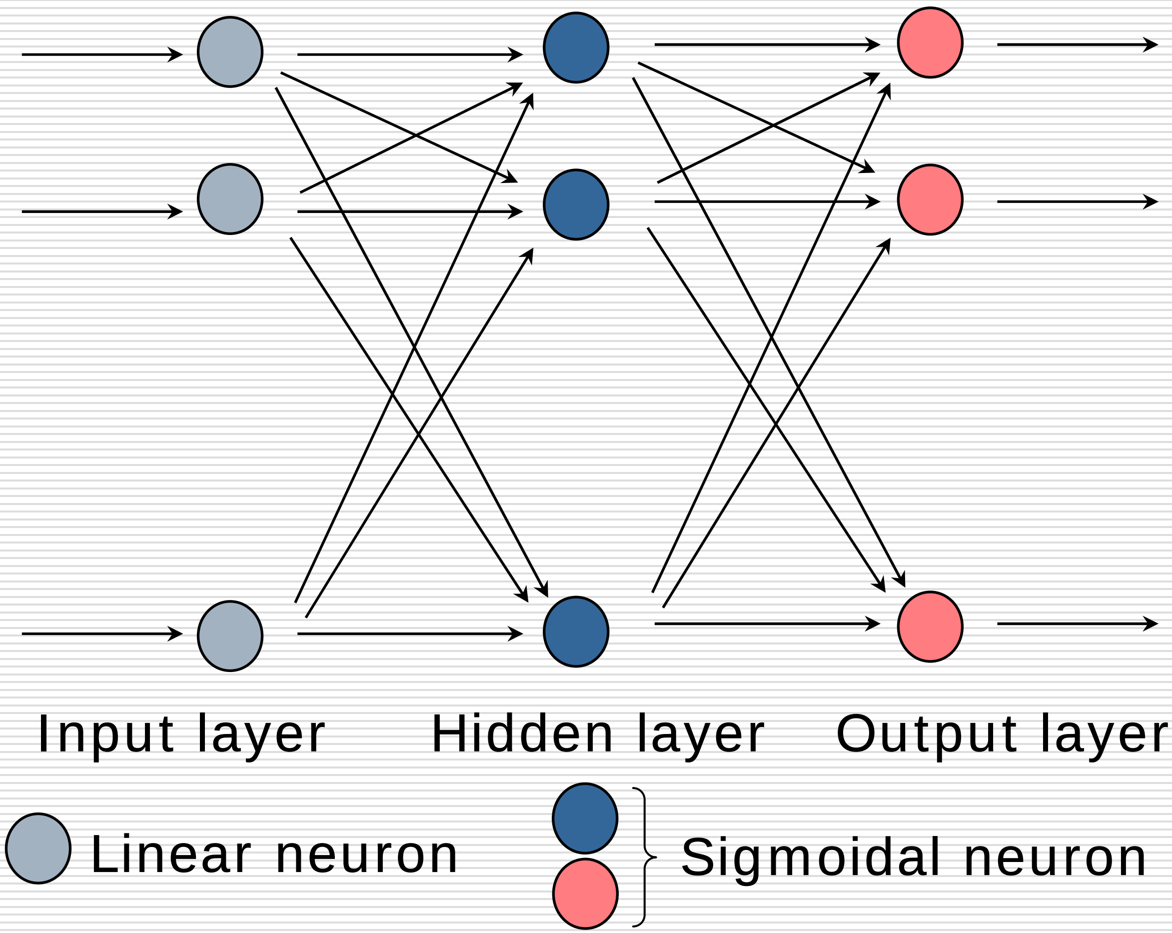
Chapter 6

Supervised Learning II: Backpropagation and Beyond

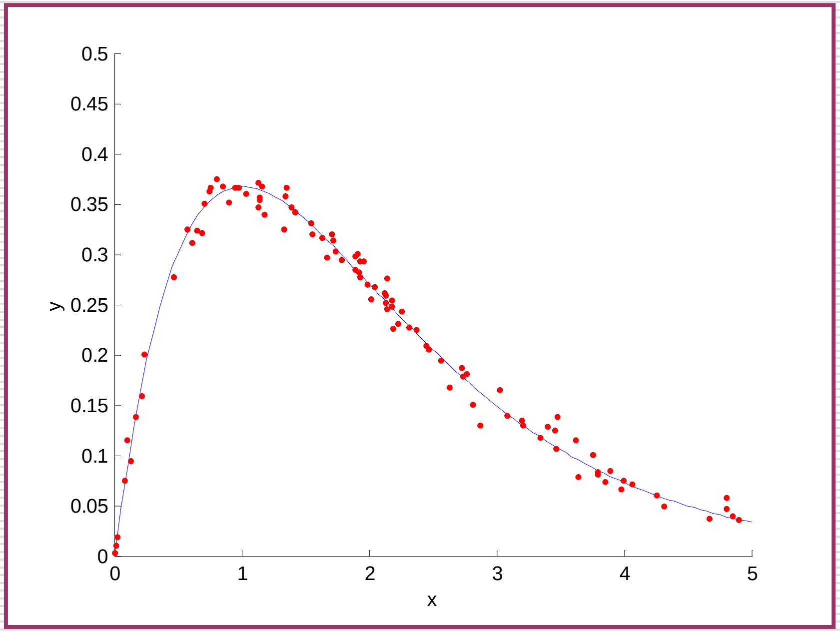
Neural Networks: A Classroom Approach
Satish Kumar
Department of Physics & Computer Science
Dayalbagh Educational Institute (Deemed University)

Multilayered Network Architectures



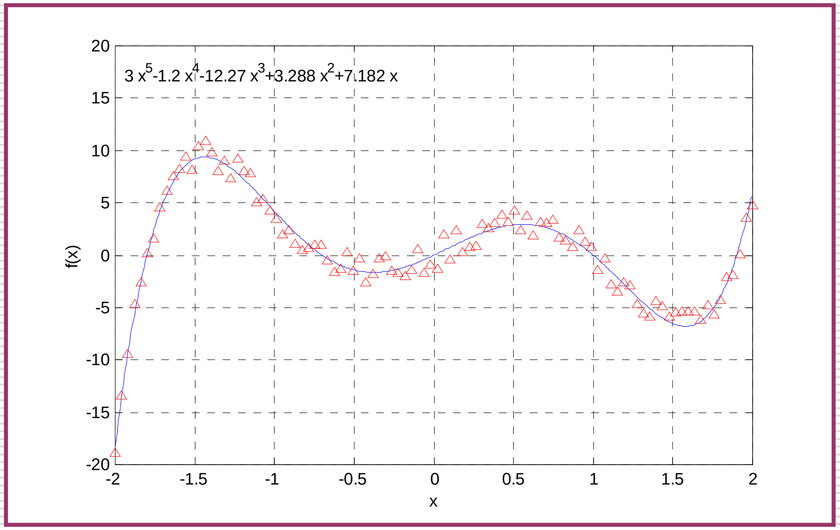
Approximation and Generalization

- What kind of network is required to learn with sufficient accuracy a function that is represented by a finite data set?
- Does the trained network predict values correctly on unseen inputs?



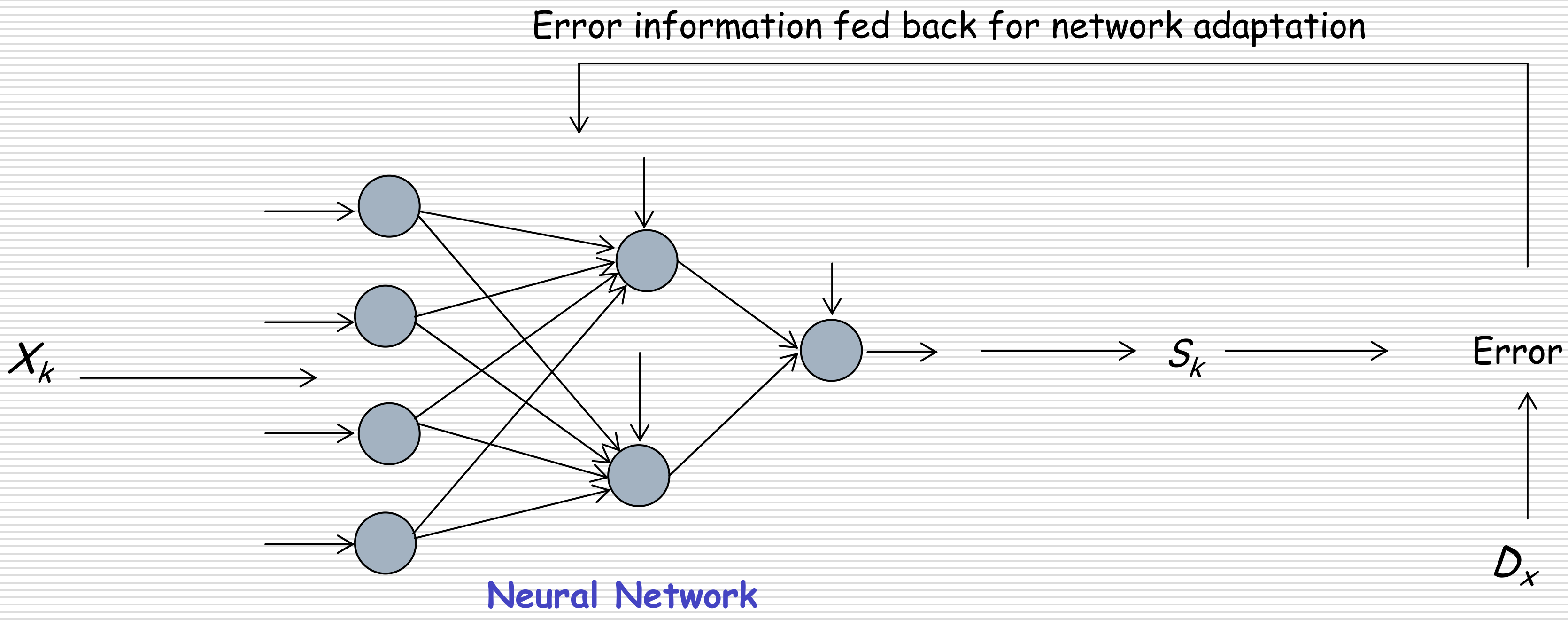
Function Described by Discrete Data

- Assume a set of Q training vector pairs: T = (X_k, D_k) k=1…Q $X_k \in R^n$, $D_k \in R^p$, where D_k is a vector response desired when input X_k is presented as input to the network.



$$\mathcal{T} = \{(X_k, D_k)\}_{k=1}^{Q}$$

Supervised Learning Procedure



Backpropagation Weight Update Procedure

1. Select a pattern X_k from the training set T present it to the network.
2. Forward Pass: Compute activations and signals of input, hidden and output neurons in that sequence.
3. Error Computation: Compute the error over the output neurons by comparing the generated outputs with the desired outputs.
4. Compute Weight Changes: Use the error to compute the change in the hidden to output layer weights, and the change in input to hidden layer weights such that a global error measure gets reduced.

Backpropagation Weight Update Procedure

5. Update all weights of the network.

Hidden to output layer weights

$$w_{hj}^{k+1} = w_{hj}^{k} + \Delta w_{hj}^{k}$$

Input to hidden layer weights

$$w_{ih}^{k+1} = w_{ih}^{k} + \Delta w_{ih}^{k}$$

6. Repeat Steps 1 through 5 until the global error falls below a predefined threshold.

Square Error Function

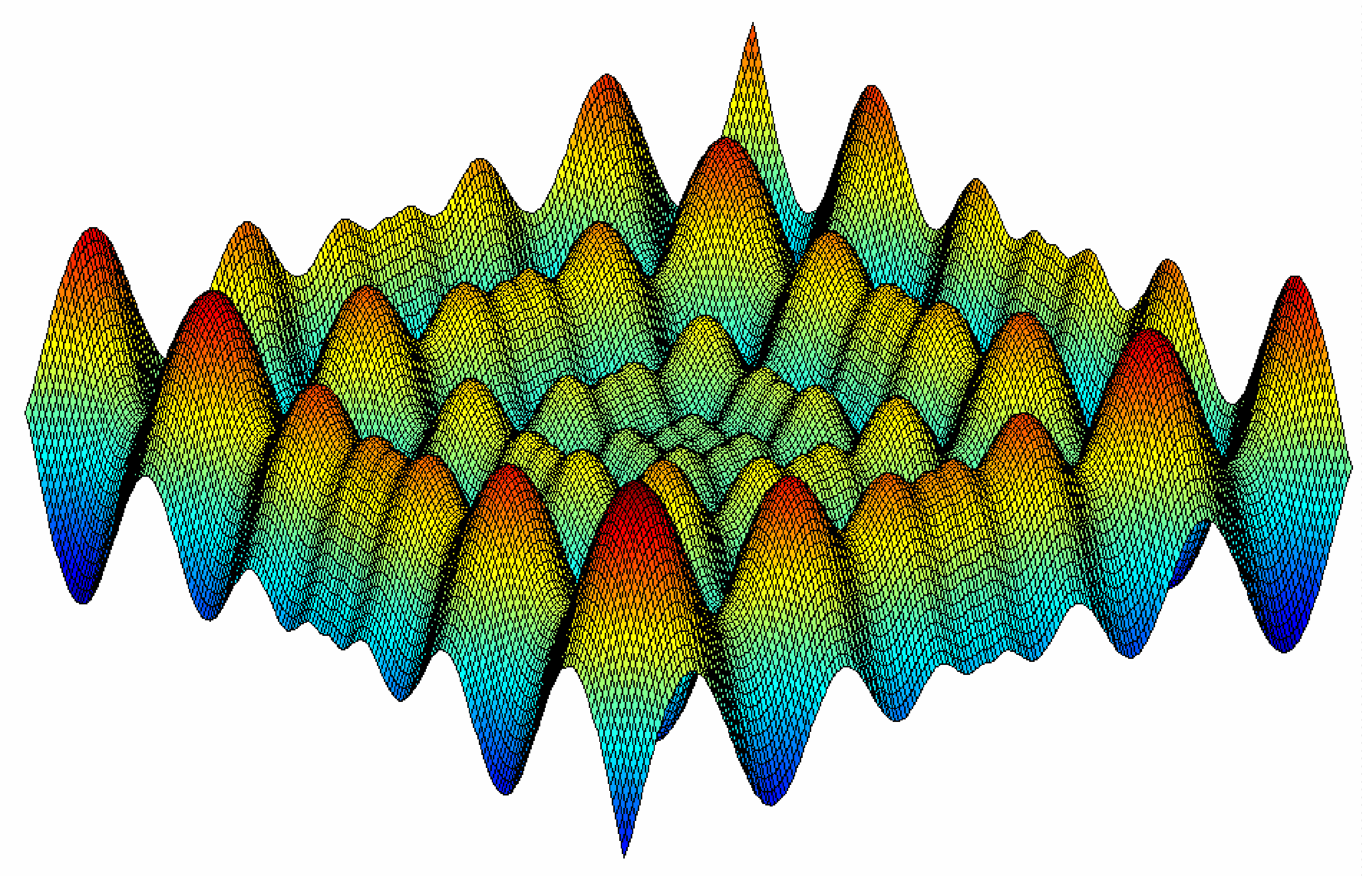
- The instantaneous summed squared error ε_k is the sum of the squares of each individual output error e_j^k, scaled by one-half:

$$E_k = (e_1^k, \dots, e_p^k)^T = \left(d_1^k - \mathcal{S}(y_1^k), \dots, d_p^k - \mathcal{S}(y_p^k)\right)^T$$

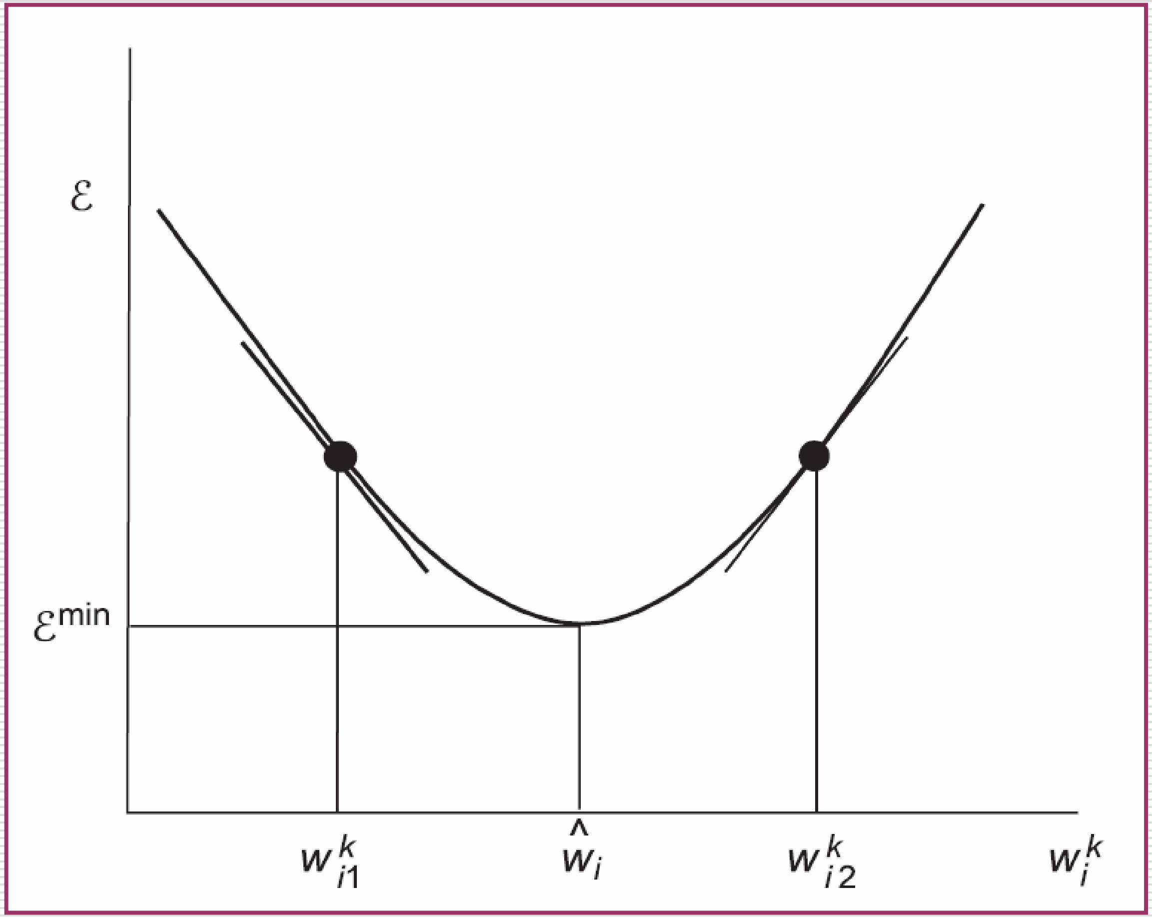
$$\mathcal{E}_k = \frac{1}{2} \sum_{j=1}^{p} \left(d_j^k - \mathcal{S}(y_j^k)\right)^2 = \frac{1}{2} E_k^T E_k$$

$$\mathcal{E} = \frac{1}{Q} \sum_{k=1}^{Q} \mathcal{E}_k$$

Error Surface



Gradient Descent Procedure



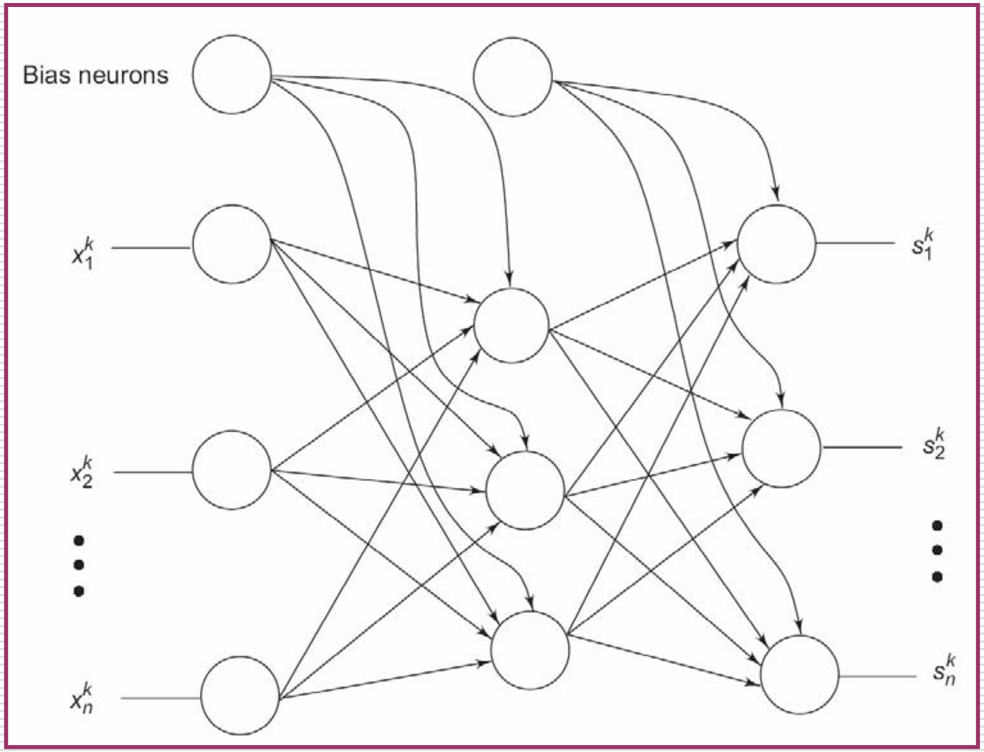
Recall: Gradient Descent Update Equation

- If $\frac{\partial \mathcal{E}}{\partial w_i^k} > 0$, $(w_i^k > \hat{w}_i)$, <u>decrease</u> w_i^k
- If $\frac{\partial \mathcal{E}}{\partial w_i^k} < 0$, $(w_i^k < \hat{w}_i)$ <u>increase</u> w_i^k

- It follows logically therefore, that the weight component should be updated in proportion with the negative of the gradient as follows:

$$w_i^{k+1} = w_i^k + \eta\left(-\frac{\partial \mathcal{E}}{\partial w_i^k}\right) \qquad i = 0, 1, \ldots, n$$

Neuron Signal Functions



- Input layer neurons are linear.

$$\mathcal{S}(x) = x$$

- Hidden and output layer neurons are sigmoidal.

$$\mathcal{S}(x) = \frac{1}{1 + e^{-\lambda x}}$$

- A training data set is assumed to be given which will be used to train the network.

$$\mathcal{T} = \{(X_k, D_k)\}_{k=1}^{Q}$$

Notation for Backpropagation Algorithm Derivation

	Input		*Hidden*		*Output*
Number of neurons	$n+1$		$q+1$		p
Signal function	linear		sigmoidal		sigmoidal
Neuron index range	$i=0,\ldots,n$		$h=0,\ldots,q$		$j=1,\ldots,p$
Activation	x_i		z_h		y_j
Signal	$\mathcal{S}(x_i)$		$\mathcal{S}(z_h)$		$\mathcal{S}(y_j)$
Weights (including bias)		$\rightarrow w_{ih} \rightarrow$		$\rightarrow w_{hj} \rightarrow$	

The General Idea Behind Iterative Training…

- Employ the gradient of the pattern error in order to reduce the global error over the entire training set.
- Compute the error gradient for a pattern and use it to change the weights in the network.
- Such weight changes are effected for a sequence of training pairs $(X_1, D_1), (X_2, D_2), \ldots, (X_k, D_k), \ldots$ picked from the training set.
- Each weight change perturbs the existing neural network slightly, in order to reduce the error on the pattern in question.

Square Error Performance Function

- The k^{th} training pair (X_k, D_k) then defines the instantaneous error:
  - $E_k = D_k - S(Y_k)$ where
  - $E_k = (e_1^k, \ldots, e_p^k)$
  - $= (d_1^k - S(y_1^k), \ldots, d_p^k - S(y_p^k))$
- The instantaneous summed squared error E_k is the sum of the squares of each individual output error e_j^k, scaled by one-half:

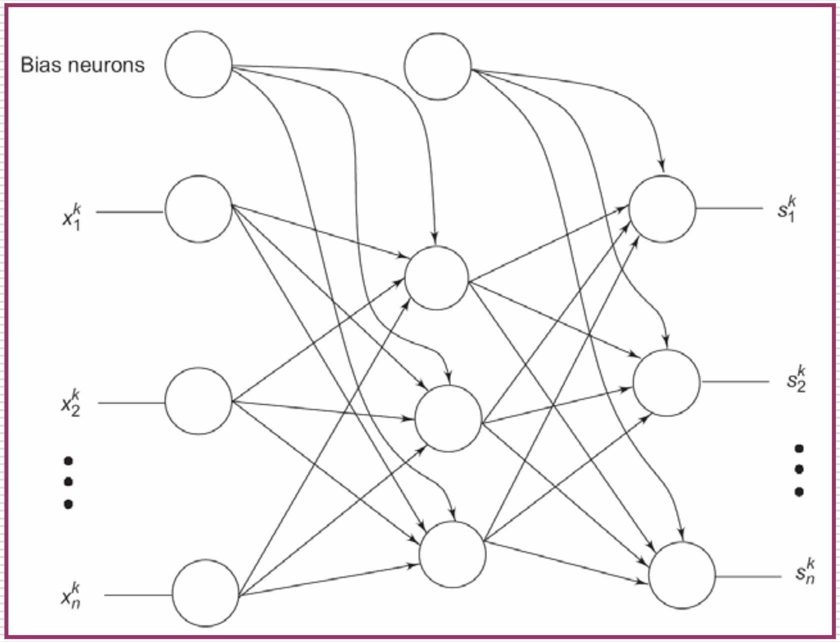
$$\mathcal{E}_k = \frac{1}{2} \sum_{j=1}^{p} \left(d_j^k - \mathcal{S}(y_j^k) \right)^2$$

The Difference Between Batch and Pattern Update

$$\text{Batch Update} \Rightarrow \left[\left. \begin{matrix} X_1 \rightarrow \mathcal{N}^{1} \rightarrow \mathcal{E}_1 \\ X_2 \rightarrow \mathcal{N}^{1} \rightarrow \mathcal{E}_2 \\ \vdots \\ X_Q \rightarrow \mathcal{N}^{1} \rightarrow \mathcal{E}_Q \end{matrix} \right\} \rightarrow \mathcal{E} \rightarrow \nabla\mathcal{E} \rightarrow \Delta W \right] \rightarrow \mathcal{N}^{2}$$

$$\text{Pattern Update} \Rightarrow \left[\begin{matrix} X_1 \rightarrow \widehat{\mathcal{N}}^{1} \rightarrow \mathcal{E}_1 \rightarrow \nabla\mathcal{E}_1 \rightarrow \Delta W_1 \rightarrow \widehat{\mathcal{N}}^{2} \\ X_2 \rightarrow \widehat{\mathcal{N}}^{2} \rightarrow \mathcal{E}_2 \rightarrow \nabla\mathcal{E}_2 \rightarrow \Delta W_2 \rightarrow \widehat{\mathcal{N}}^{3} \\ \vdots \\ X_Q \rightarrow \widehat{\mathcal{N}}^{Q} \rightarrow \mathcal{E}_Q \rightarrow \nabla\mathcal{E}_Q \rightarrow \Delta W_Q \rightarrow \widehat{\mathcal{N}}^{Q+1} \end{matrix} \right] \rightarrow \widehat{\mathcal{N}}^{Q+1}$$

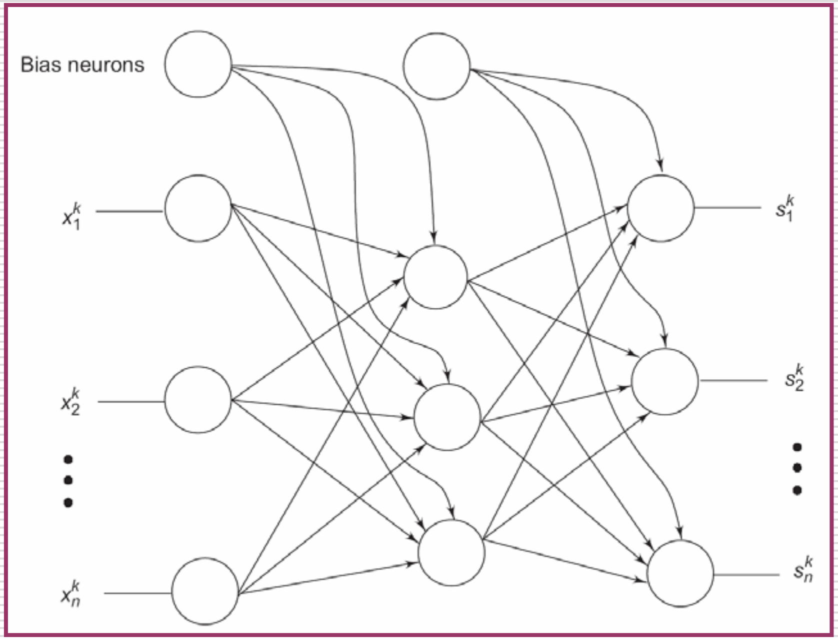
Derivation of BP Algorithm: Forward Pass-Input Layer



$$\mathcal{S}(x_i^k) = x_i^k, \quad i = 1, \dots, n$$

$$\mathcal{S}(x_0^k) = x_0^k = 1$$

Derivation of BP Algorithm: Forward Pass-Hidden Layer

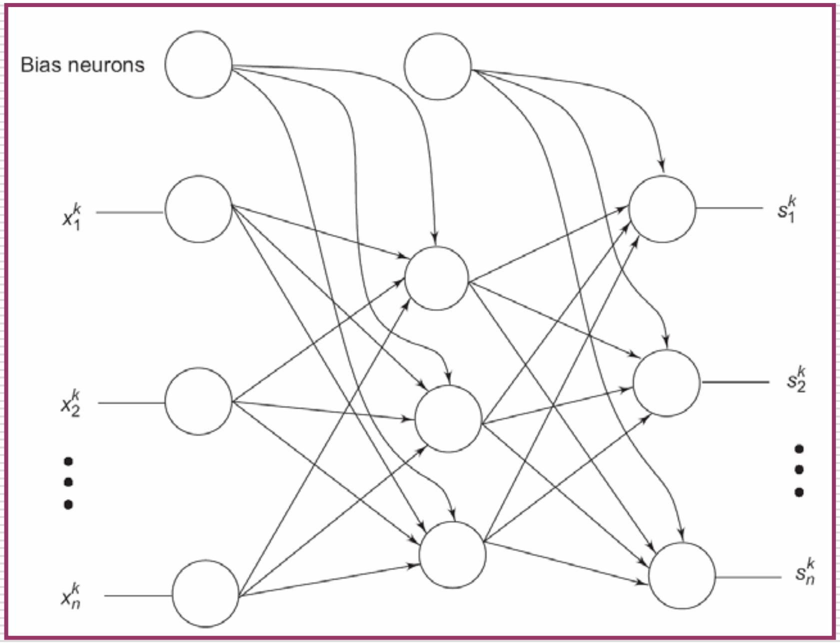


$$z_h^k = \sum_{i=0}^{n} w_{ih}^k \mathcal{S}(x_i^k) = \sum_{i=0}^{n} w_{ih}^k x_i^k, \ h = 1, \dots, q$$

$$\mathcal{S}(z_h^k) = \frac{1}{1 + e^{-z_h^k}}, \ h = 1, \dots, q$$

$$\mathcal{S}(z_0^k) = 1, \ \forall k$$

Derivation of BP Algorithm: Forward Pass-Output Layer



$$y_j^k = \sum_{h=0}^{q} w_{hj}^k \mathcal{S}(z_h^k), \quad j = 1, \ldots, p$$

$$\mathcal{S}(y_j^k) = \frac{1}{1 + e^{-y_j^k}}, \quad j = 1, \ldots, p$$

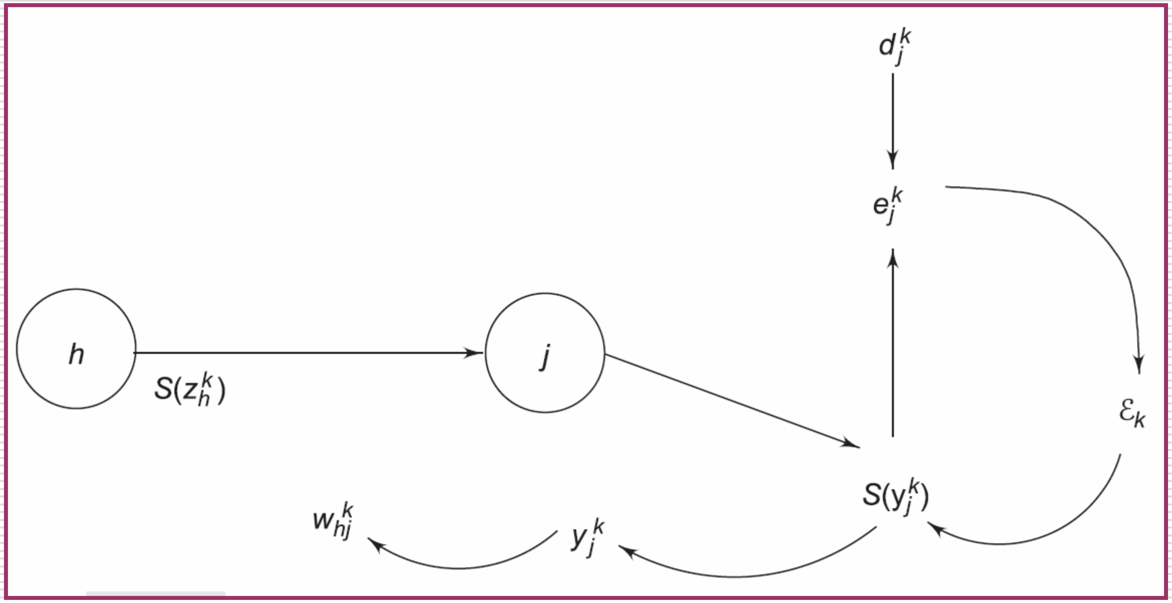
Recall the Gradient Descent Update Equation

- A weight gets updated based on the negative of the error gradient with respect to the weight

$$
\begin{aligned}
w_{hj}^{k+1} &= w_{hj}^{k} + \Delta w_{hj}^{k} \\
&= w_{hj}^{k} + \eta\left(-\frac{\partial \mathcal{E}_k}{\partial w_{hj}^{k}}\right)
\end{aligned}
$$

$$
\begin{aligned}
w_{ih}^{k+1} &= w_{ih}^{k} + \Delta w_{ih}^{k} \\
&= w_{ih}^{k} + \eta\left(-\frac{\partial \mathcal{E}_k}{\partial w_{ih}^{k}}\right)
\end{aligned}
$$

Derivation of BP Algorithm: Computation of Gradients



$$\frac{\partial \mathcal{E}_k}{\partial w_{hj}^k} = \frac{\partial \mathcal{E}_k}{\partial \mathcal{S}(y_j^k)} \frac{\partial \mathcal{S}(y_j^k)}{\partial y_j^k} \frac{\partial y_j^k}{\partial w_{hj}^k}$$

Derivation of BP Algorithm: Computation of Gradients

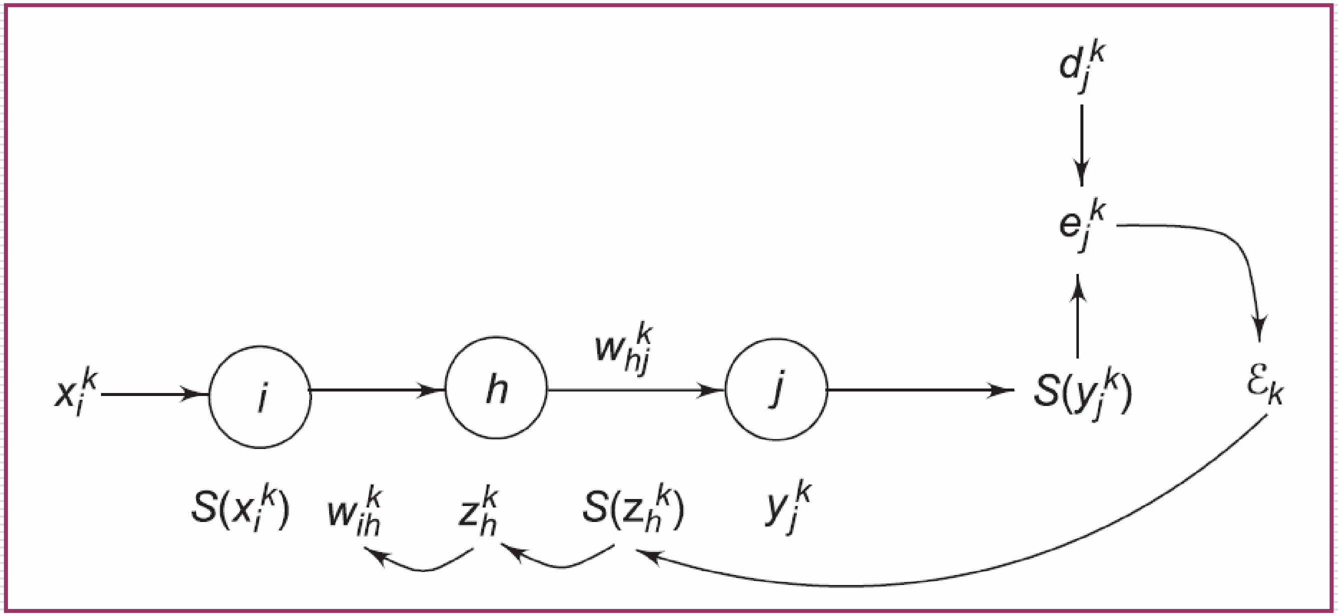
$$\frac{\partial \mathcal{E}_k}{\partial \mathcal{S}(y_j^k)} = -\left(d_j^k - \mathcal{S}(y_j^k)\right) = -e_j^k$$

$$\frac{\partial \mathcal{S}(y_j^k)}{\partial y_j^k} = \mathcal{S}'(y_j^k) = \mathcal{S}(y_j^k)\left(1 - \mathcal{S}(y_j^k)\right)$$

$$\frac{\partial y_j^k}{\partial w_{hj}^k} = \mathcal{S}(z_h^k)$$

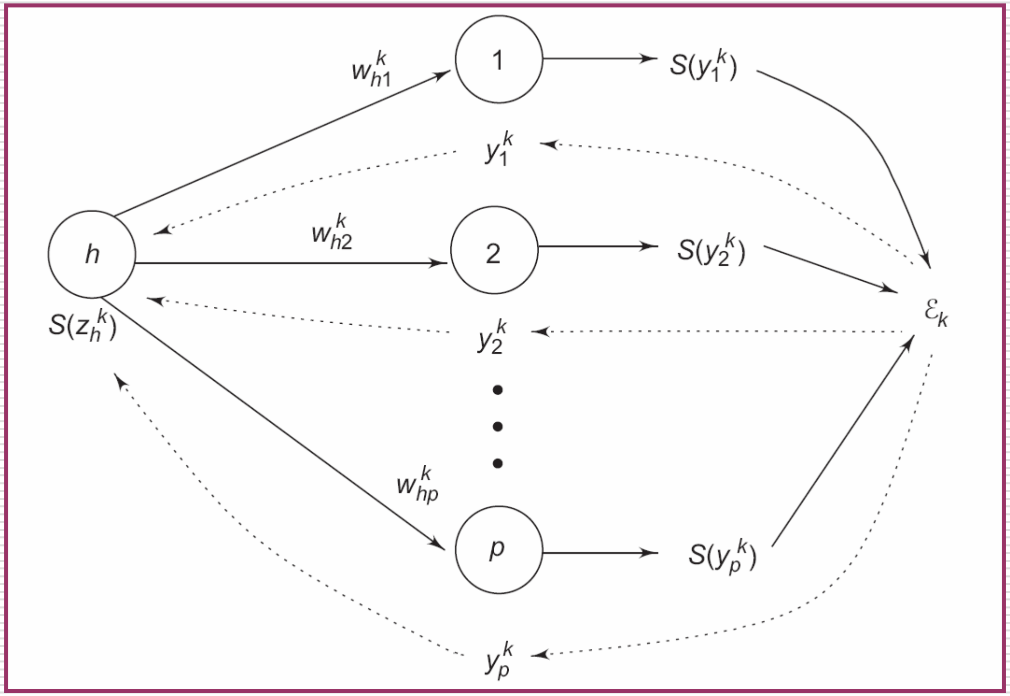
$$\begin{aligned}\frac{\partial \mathcal{E}_k}{\partial w_{hj}^k} &= -e_j^k \mathcal{S}'(y_j^k)\mathcal{S}(z_h^k) \\ &= -\delta_j^k \mathcal{S}(z_h^k)\end{aligned}$$

Derivation of BP Algorithm: Computation of Gradients



$$\frac{\partial \mathcal{E}_k}{\partial w_{ih}^k} = \frac{\partial \mathcal{E}_k}{\partial \mathcal{S}(z_h^k)} \frac{\partial \mathcal{S}(z_h^k)}{\partial z_h^k} \frac{\partial z_h^k}{\partial w_{ih}^k}$$

Derivation of BP Algorithm: Computation of Gradients



$$\frac{\partial \mathcal{E}_k}{\partial \mathcal{S}(z_h^k)} = \sum_{j=1}^{p} \left\{ \frac{\partial \mathcal{E}_k}{\partial y_j^k} \frac{\partial y_j^k}{\partial \mathcal{S}(z_h^k)} \right\}$$

Derivation of BP Algorithm: Computation of Gradients

$$\frac{\partial \mathcal{E}_k}{\partial w_{ih}^k} = \sum_{j=1}^{p} \left\{ \frac{\partial \mathcal{E}_k}{\partial y_j^k} \; \frac{\partial y_j^k}{\partial \mathcal{S}(z_h^k)} \right\} \; \mathcal{S}'(z_h^k)\, \mathcal{S}(x_i^k)$$

$$= \sum_{j=1}^{p} \left\{ \frac{\partial \mathcal{E}_k}{\partial \mathcal{S}(y_j^k)} \; \frac{\partial \mathcal{S}(y_j^k)}{\partial y_j^k} \; \frac{\partial y_j^k}{\partial \mathcal{S}(z_h^k)} \right\} \; \mathcal{S}'(z_h^k)\, \mathcal{S}(x_i^k)$$

$$= \sum_{j=1}^{p} \left\{ -e_j^k \;\; \mathcal{S}'(y_j^k) \;\; w_{hj}^k \right\} \mathcal{S}'(z_h^k)\, x_i^k$$

$$= - \boxed{\underbrace{\sum_{j=1}^{p} \left(\delta_j^k w_{hj}^k\right)}_{\text{error backpropagation}} \quad \mathcal{S}'(z_h^k)}\, x_i^k$$

$$e_h^k = \sum_{j=1}^{p} \delta_j^k w_{hj}^k$$

$$\delta_h^k = e_h^k \mathcal{S}'(z_h^k)$$

$$\boxed{\frac{\partial \mathcal{E}_k}{\partial w_{ih}^k} = -\delta_h^k \; x_i^k}$$

Generalized Delta Rule: *Momentum*

- Increases the rate of learning while maintaining stability

$$\Delta w_{hj}^{k} = \eta \delta_j^k \mathcal{S}(z_h^k) + \alpha \Delta w_{hj}^{k-1}$$

$$\Delta w_{ih}^{k} = \eta \delta_h^k x_i^k + \alpha \Delta w_{ih}^{k-1}$$

How Momentum Works

- Momentum should be less than 1 for convergent dynamics.
- If the gradient has the same sign on consecutive iterations the net weight change increases over those iterations accelerating the descent.
- If the gradient has different signs on consecutive iterations then the net weight change decreases over those iterations and the momentum decelerates the weight space traversal. This helps avoid oscillations.

$$\Delta w_{hj}^0 = 0$$

$$\Delta w_{hj}^1 = \eta \delta_j^1 s_h^1$$

$$\begin{aligned} \Delta w_{hj}^2 &= \eta \delta_j^2 s_h^2 + \alpha \Delta w_{hj}^1 \\ &= \eta \delta_j^2 s_h^2 + \alpha \eta \delta_j^1 s_h^1 \end{aligned}$$

$$\begin{aligned} \Delta w_{hj}^3 &= \eta \delta_j^3 s_h^3 + \alpha \Delta w_{hj}^2 \\ &= \eta \delta_j^3 s_h^3 + \alpha(\eta \delta_j^2 s_h^2 + \alpha \eta \delta_j^1 s_h^1) \\ &= \eta \delta_j^3 s_h^3 + \alpha \eta \delta_j^2 s_h^2 + \alpha^2 \eta \delta_j^1 s_h^1 \end{aligned}$$

$$\boxed{\Delta w_{hj}^k = \eta \sum_{t=1}^{k} \alpha^{k-t} \delta_j^t s_h^t = -\eta \sum_{t=1}^{k} \alpha^{k-t} \frac{\partial \mathcal{E}_t}{\partial w_{hj}^t}}$$

Derivation of BP Algorithm: Finally…!

1. *For hidden to output layer weights:*

$$\begin{aligned} w_{hj}^{k+1} &= w_{hj}^k + \Delta w_{hj}^k \\ &= w_{hj}^k + \eta\left(-\frac{\partial \mathcal{E}_k}{\partial w_{hj}^k}\right) \\ &= w_{hj}^k + \eta \delta_j^k \mathcal{S}(z_h^k) \end{aligned}$$

2. *For input to hidden layer weights:*

$$\begin{aligned} w_{ih}^{k+1} &= w_{ih}^k + \Delta w_{ih}^k \\ &= w_{ih}^k + \eta\left(-\frac{\partial \mathcal{E}_k}{\partial w_{ih}^k}\right) \\ &= w_{ih}^k + \eta \delta_h^k x_i^k \end{aligned}$$

Backpropagation Algorithm: Operational Summary

Given	A training set $\mathcal{T}$ comprising vectors $X_k \in \mathbb{R}^n$ and desired output vectors $D_k \in \mathbb{R}^p$ and an n–q–p architecture neural network $\mathcal{N}$
Initialize	↬ Randomize weights w^1_{ih} to small values, set $\Delta w^0_{ih} = 0, \ i = 0, \ldots, n; \quad h = 1, \ldots, q$ ↬ Randomize weights w^1_{hj} to small values, set $\Delta w^0_{hj} = 0, \ h = 0, \ldots, q; \quad j = 1, \ldots, p$ ↬ Set $k = 1$, η, α, and the error tolerance τ as desired.
Iterate	⭮ Repeat { ⇝ Select a training pair $(X_k, D_k) \in \mathcal{T}$ ⇝ Compute signals on forward pass in the following sequence: $\mathcal{S}(x^k_i) = x^k_i, \quad i = 1, \ldots, n$ $\mathcal{S}(x^k_0) = 1$ $z^k_h = \sum_{i=0}^{n} w^k_{ih} x^k_i, \quad h = 1, \ldots, q$ $\mathcal{S}(z^k_h) = \frac{1}{1+\exp(-z^k_h)}, \quad h = 1, \ldots, q$ $\mathcal{S}(z^k_0) = 1$ $y^k_j = \sum_{h=0}^{q} w^k_{hj} \mathcal{S}(z^k_h), \quad j = 1, \ldots, p$ $\mathcal{S}(y^k_j) = \frac{1}{1+\exp(-y^k_j)}, \quad j = 1, \ldots, p$

Backpropagation Algorithm: Operational Summary(contd.)

$\rightsquigarrow$ Compute deltas/errors at output neurons:

$$\delta_j^k = (d_j^k - \mathcal{S}(y_j^k))\mathcal{S}'(y_j^k) \qquad j = 1, \ldots, p$$
$$\Delta w_{hj}^k = \eta \delta_j^k \mathcal{S}(x_h^k) \qquad h = 0, \ldots, q; \; j = 1, \ldots, p$$

$\rightsquigarrow$ Compute deltas/errors at hidden neurons:

$$\delta_h^k = \left(\sum_{j=1}^{p} \delta_j^k w_{hj}^k \right) \mathcal{S}'(z_h^k) \qquad h = 1, \ldots, q$$
$$\Delta w_{ih}^k = \eta \delta_h^k x_i^k \qquad i = 0, \ldots, n; \; h = 1, \ldots, q$$

$\rightsquigarrow$ Update weights:

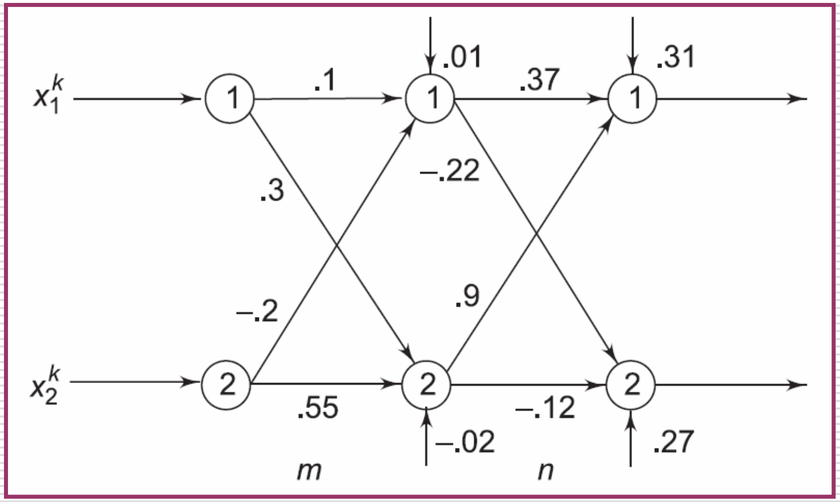
$$w_{hj}^{k+1} = w_{hj}^k + \Delta w_{hj}^k + \alpha \Delta w_{hj}^{k-1} \qquad h = 0, \ldots, q; \; j = 1, \ldots, p$$
$$w_{ih}^{k+1} = w_{ih}^k + \Delta w_{ih}^k + \alpha \Delta w_{ih}^{k-1} \qquad i = 0, \ldots, n; \; h = 1, \ldots, q$$

$\rightsquigarrow$ Collect pattern error $\mathcal{E}_k$

} until $(\mathcal{E}_{av} = \frac{1}{Q} \sum_{k=1}^{Q} \mathcal{E}_k < \tau)$

Hand-worked Example

Pattern Index	x_1^k	x_2^k	d_1^k	d_2^k
1	0.5	–0.5	0.9	0.1
2	–0.5	0.5	0.1	0.9



Forward Pass 1/Backprop Pass 1

k	x_1^k	x_2^k	$\mathcal{S}(x_1^k)$	$\mathcal{S}(x_2^k)$	z_1^k	z_2^k	$\mathcal{S}(z_1^k)$	$\mathcal{S}(z_2^k)$	y_1^k	y_2^k	$\mathcal{S}(y_1^k)$	$\mathcal{S}(y_2^k)$
1	.5	−.5	.5	−.5	.16	−.145	.5399	.4638	.9271	.0955	.7164	.5238

$$e_1^1 = d_1^1 - s_1^1 = 0.9 - 0.7164 = 0.1836$$

$$e_2^1 = d_2^1 - s_2^1 = 0.1 - 0.5238 = -0.4238$$

$$\delta_1^1 = 0.1836 \times 0.7164(1 - 0.7164) \quad = \quad 0.0373$$

$$\delta_2^1 = -0.4238 \times 0.5238(1 - 0.5238) = -0.1057$$

$$\delta H_1^1 = \left(0.0373 \times 0.37 + (-0.1057 \times -0.22)\right) \times 0.5399(1 - 0.5399) = 0.0092$$

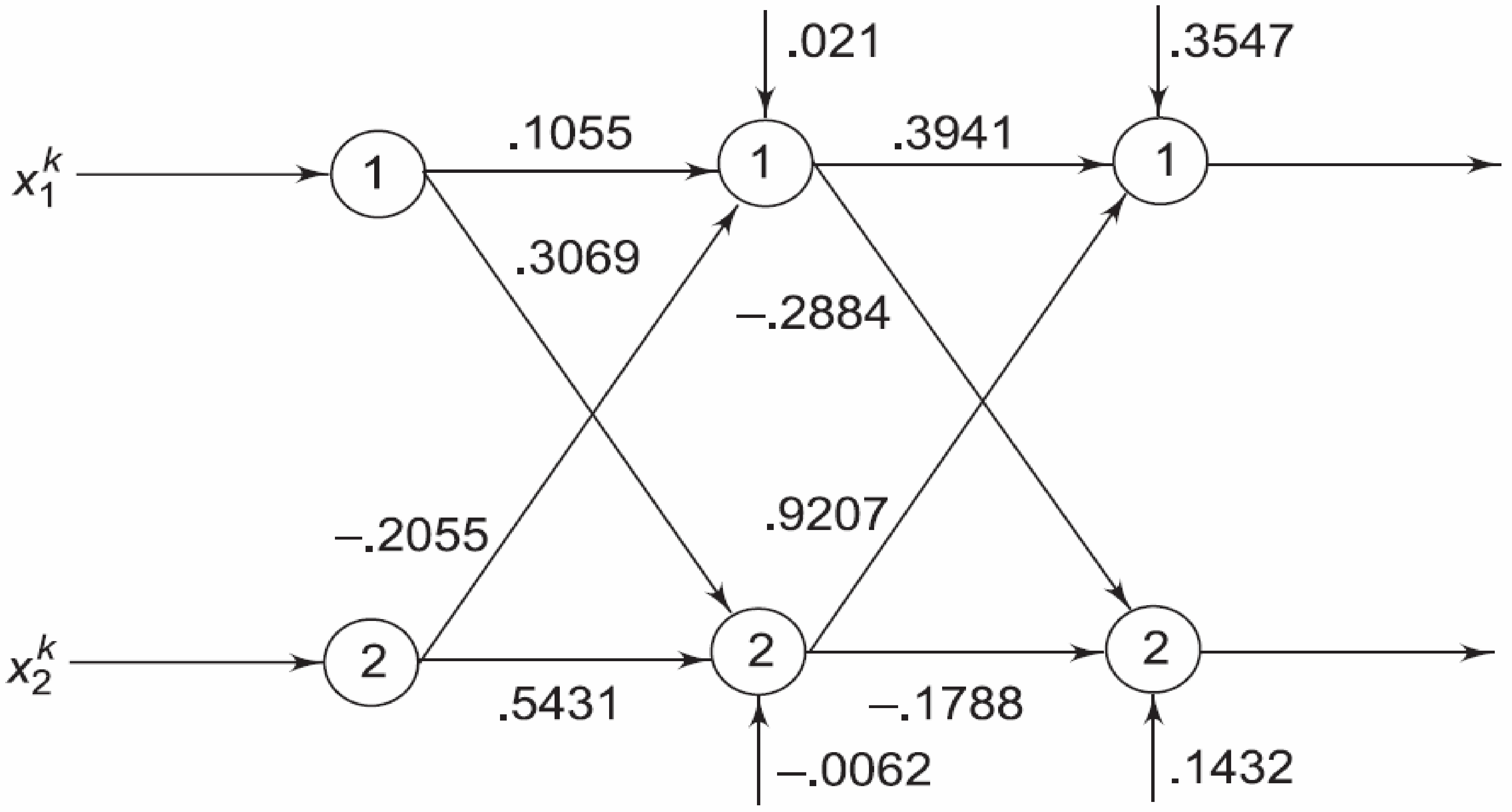
$$\delta H_2^1 = \left(0.0373 \times 0.9 + (-0.1057 \times -0.12)\right) \times 0.4638(1 - 0.4638) = 0.0115$$

Weight Changes: Pass 1

$$\begin{aligned}
\Delta n_{01}^{1} &= 1.2 \times 0.0373 \times 1.0 &=& \quad 0.0447 \\
\Delta n_{11}^{1} &= 1.2 \times 0.0373 \times 0.5399 &=& \quad 0.0241 \\
\Delta n_{21}^{1} &= 1.2 \times 0.0373 \times 0.4638 &=& \quad 0.0207 \\
\Delta n_{02}^{1} &= 1.2 \times -0.1057 \times 1.0 &=& \, -0.1268 \\
\Delta n_{12}^{1} &= 1.2 \times -0.1057 \times 0.5399 &=& \, -0.0684 \\
\Delta n_{22}^{1} &= 1.2 \times -0.1057 \times 0.4638 &=& \, -0.0588 \\
\Delta m_{01}^{1} &= 1.2 \times 0.0092 \times 1.0 &=& \quad 0.011 \\
\Delta m_{11}^{1} &= 1.2 \times 0.0092 \times 0.5 &=& \quad 0.0055 \\
\Delta m_{21}^{1} &= 1.2 \times 0.0092 \times -0.5 &=& \, -0.0055 \\
\Delta m_{02}^{1} &= 1.2 \times 0.0115 \times 1.0 &=& \quad 0.0138 \\
\Delta m_{12}^{1} &= 1.2 \times 0.0115 \times 0.5 &=& \quad 0.0069 \\
\Delta m_{22}^{1} &= 1.2 \times 0.0115 \times -0.5 &=& \, -0.0069
\end{aligned}$$

$$\begin{aligned}
n_{01}^{2} &= 0.31 + 0.0447 = 0.3547 \\
n_{11}^{2} &= 0.37 + 0.0241 = 0.3941 \\
n_{21}^{2} &= 0.9 + 0.0207 = 0.9207 \\
n_{02}^{2} &= 0.27 - 0.1268 = 0.1432 \\
n_{12}^{2} &= -0.22 - 0.0684 = -0.2884 \\
n_{22}^{2} &= -0.12 - 0.0588 = -0.1788 \\
m_{01}^{2} &= 0.01 + 0.011 = 0.021 \\
m_{11}^{2} &= 0.1 + 0.0055 = 0.1055 \\
m_{21}^{2} &= -0.2 - 0.0055 = -0.2055 \\
m_{02}^{2} &= -0.02 + 0.0138 = -0.0062 \\
m_{12}^{2} &= 0.3 + 0.0069 = 0.3069 \\
m_{22}^{2} &= 0.55 - 0.0069 = 0.5431
\end{aligned}$$

Network N^2 after first Iteration



Forward Pass 2/Backprop Pass 2

k	x_1^k	x_2^k	$\mathcal{S}(x_1^k)$	$\mathcal{S}(x_2^k)$	z_1^k	z_2^k	$\mathcal{S}(z_1^k)$	$\mathcal{S}(z_2^k)$	y_1^k	y_2^k	$\mathcal{S}(y_1^k)$	$\mathcal{S}(y_2^k)$
2	–.5	.5	–.5	.5	–.1345	.1119	.4664	.5279	1.0245	–.0856	.7358	.4786

$$e_1^2 = d_1^2 - s_1^2 = 0.1 - 0.7358 = -0.6358$$

$$e_2^2 = d_2^2 - s_2^2 = 0.9 - 0.4786 = 0.4214$$

$$\delta_1^2 = -0.6358 \times 0.7358(1 - 0.7358) = -0.1235$$

$$\delta_2^2 = 0.4214 \times 0.4786(1 - 0.4786) = 0.1051$$

$$\delta H_1^2 = \left(-0.1235 \times 0.3941 + 0.1051 \times -0.2884\right) \times 0.4664(1 - 0.4664) = -0.0196$$

$$\delta H_2^2 = \left(-0.1235 \times 0.9207 + 0.1051 \times -0.1788\right) \times 0.5279(1 - 0.5279) = -0.033$$

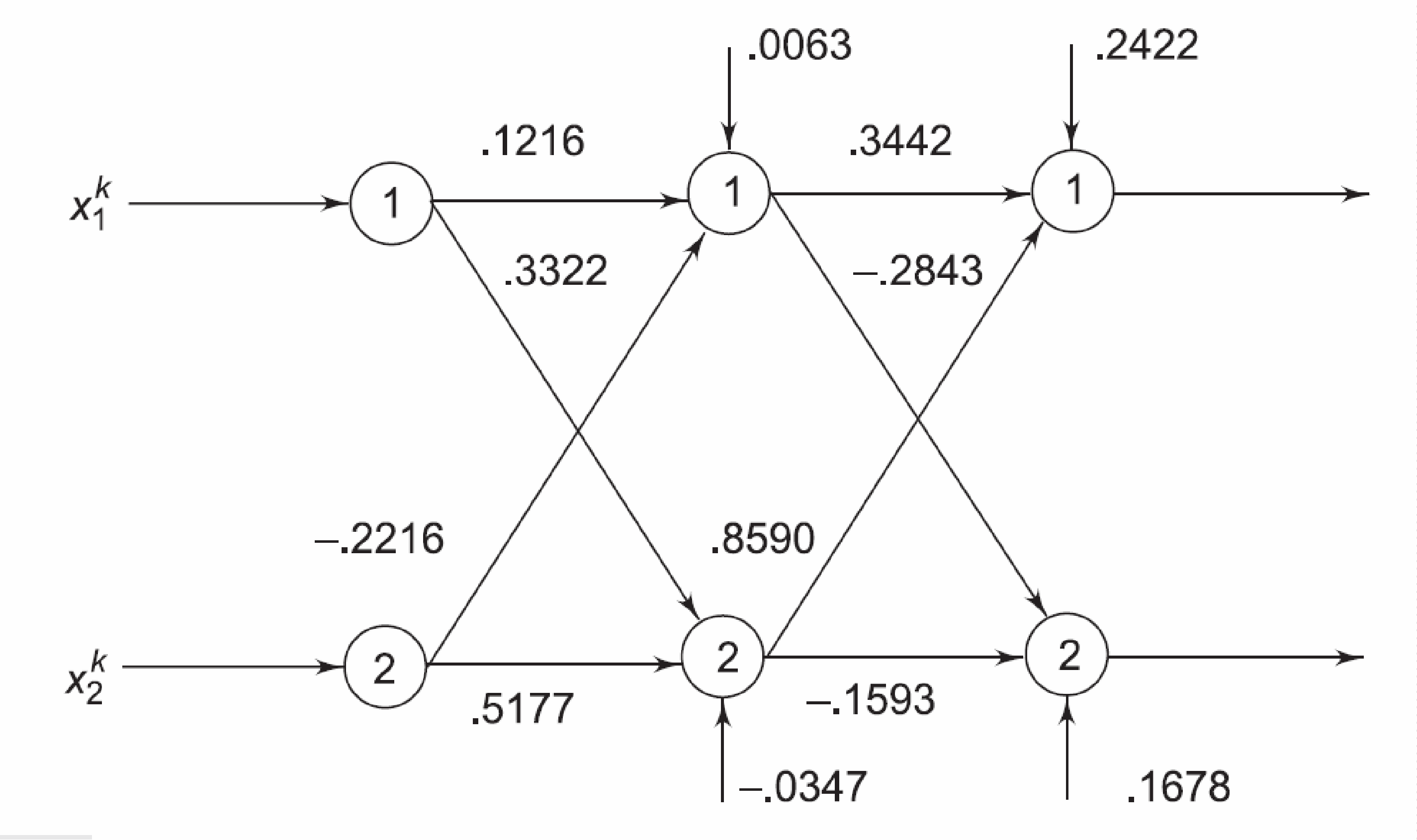
Weight Changes: Pass 2

$$\Delta n^2_{01} = 1.2 \times -.1235 \times 1.0 = -0.1482$$
$$\Delta n^2_{11} = 1.2 \times -.1235 \times 0.4664 = -0.0691$$
$$\Delta n^2_{21} = 1.2 \times -.1235 \times 0.5279 = -0.0782$$
$$\Delta n^2_{02} = 1.2 \times 0.1051 \times 1.0 = 0.1261$$
$$\Delta n^2_{12} = 1.2 \times 0.1051 \times 0.4664 = 0.0588$$
$$\Delta n^2_{22} = 1.2 \times 0.1051 \times 0.5279 = 0.0665$$
$$\Delta m^2_{01} = 1.2 \times -.0196 \times 1.0 = -0.0235$$
$$\Delta m^2_{11} = 1.2 \times -.0196 \times -0.5 = 0.0117$$
$$\Delta m^2_{21} = 1.2 \times -.0196 \times 0.5 = -0.0117$$
$$\Delta m^2_{02} = 1.2 \times -.033 \times 1.0 = -0.0396$$
$$\Delta m^2_{12} = 1.2 \times -.033 \times -0.5 = 0.0198$$
$$\Delta m^2_{22} = 1.2 \times -.033 \times 0.5 = -0.0198$$

$$n^3_{11} = 0.3941 - 0.0691 + 0.8 \times 0.0241 = 0.3442$$
$$n^3_{21} = 0.9207 - 0.0782 + 0.8 \times 0.0207 = 0.8590$$
$$n^3_{01} = 0.3547 - 0.1482 + 0.8 \times 0.0447 = 0.2422$$
$$n^3_{12} = -0.2884 + 0.0588 + 0.8 \times -0.0684 = -0.2843$$
$$n^3_{22} = -0.1788 + 0.0665 + 0.8 \times -0.0588 = -0.1593$$
$$n^3_{02} = 0.1432 + 0.1261 + 0.8 \times -0.1268 = 0.1678$$

$$m^3_{11} = 0.1055 + 0.0117 + 0.8 \times 0.0055 = 0.1216$$
$$m^3_{21} = -0.2055 - 0.0117 + 0.8 \times -0.0055 = -0.2216$$
$$m^3_{01} = 0.021 - 0.0235 + 0.8 \times 0.011 = 0.0063$$
$$m^3_{12} = 0.3069 + 0.0198 + 0.8 \times 0.0069 = 0.3322$$
$$m^3_{22} = 0.5431 - 0.0198 + 0.8 \times -0.0069 = 0.5177$$
$$m^3_{02} = -0.0062 - 0.0396 + 0.8 \times 0.0138 = -0.0347$$

Network N^3 after second Iteration

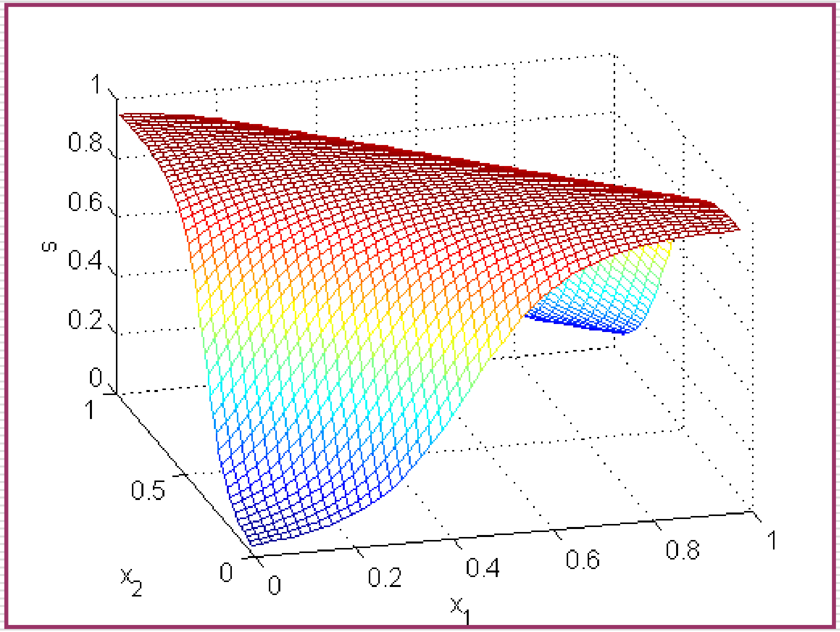


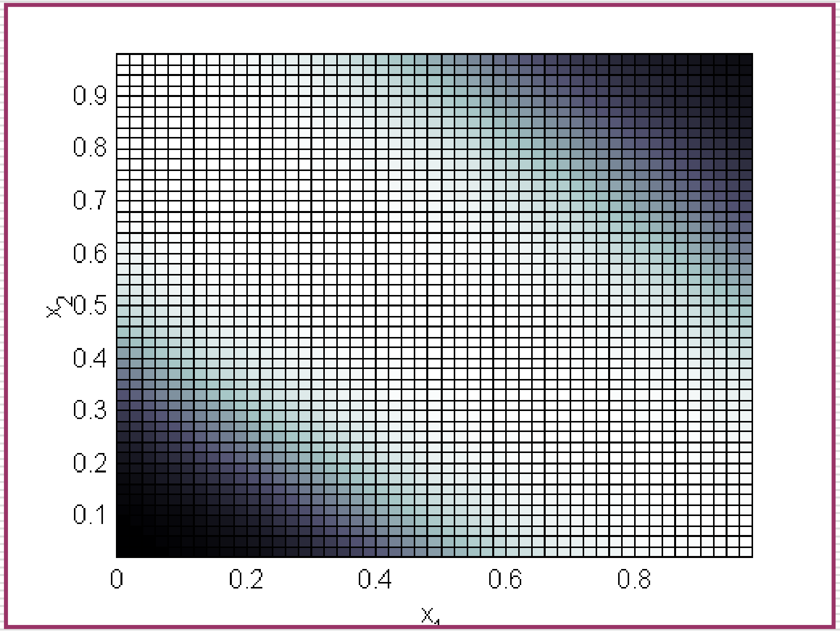
MATLAB Simulation Example 1
Two Dimensional XOR Classifier

- Specifying a 0 or 1 desired value does not make sense since a sigmoidal neuron can generate a 0 or 1 signal only at an activation value $-\infty$ or ∞. So it is never going to quite get there.
- The values 0.05, 0.95 are somewhat more reasonable representatives of 0 and 1.
- Note that the inputs can still be 0 and 1 but the desired values must be changed keeping in mind the signal range.

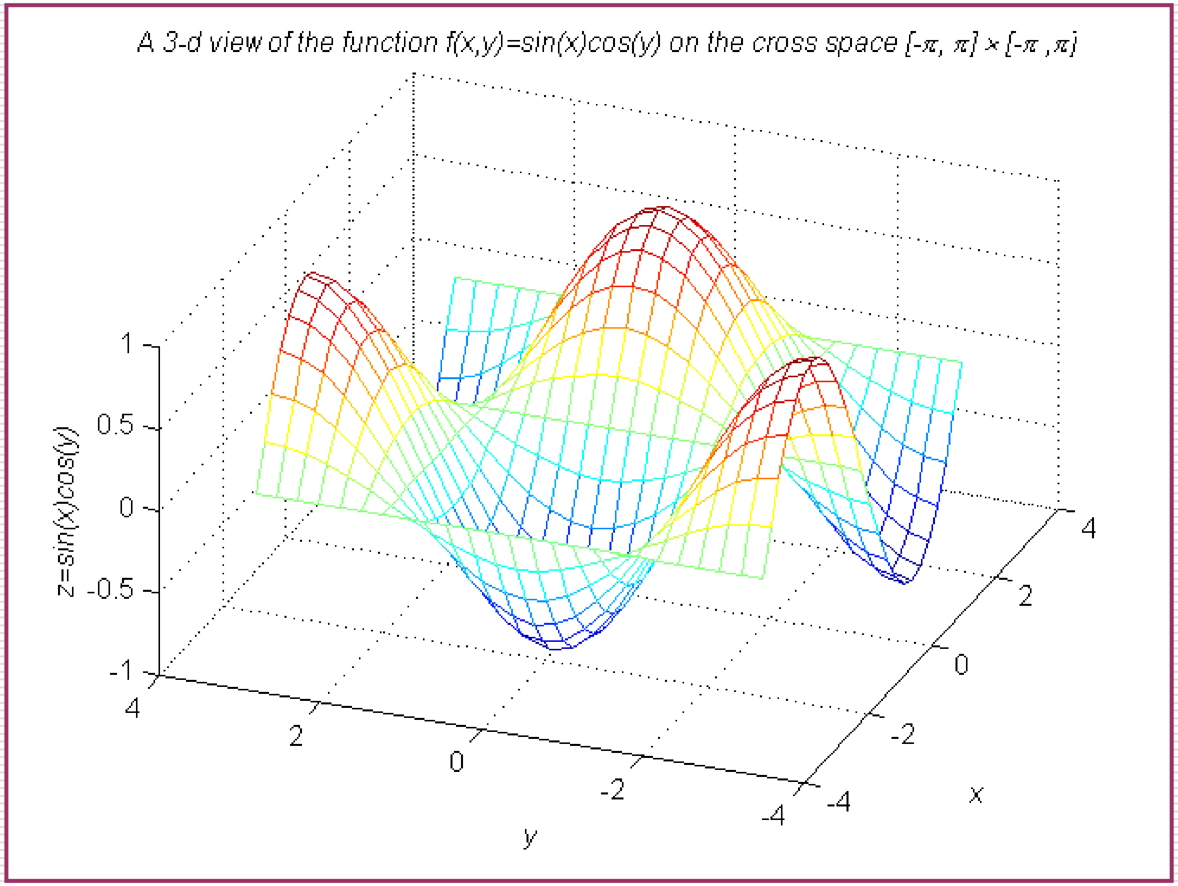
x_1	x_2	$f_\oplus$
0.05	0.05	0.05
0.05	0.95	0.95
0.95	0.05	0.95
0.95	0.95	0.05

Generalization Surface, Grayscale Map of the Network Response



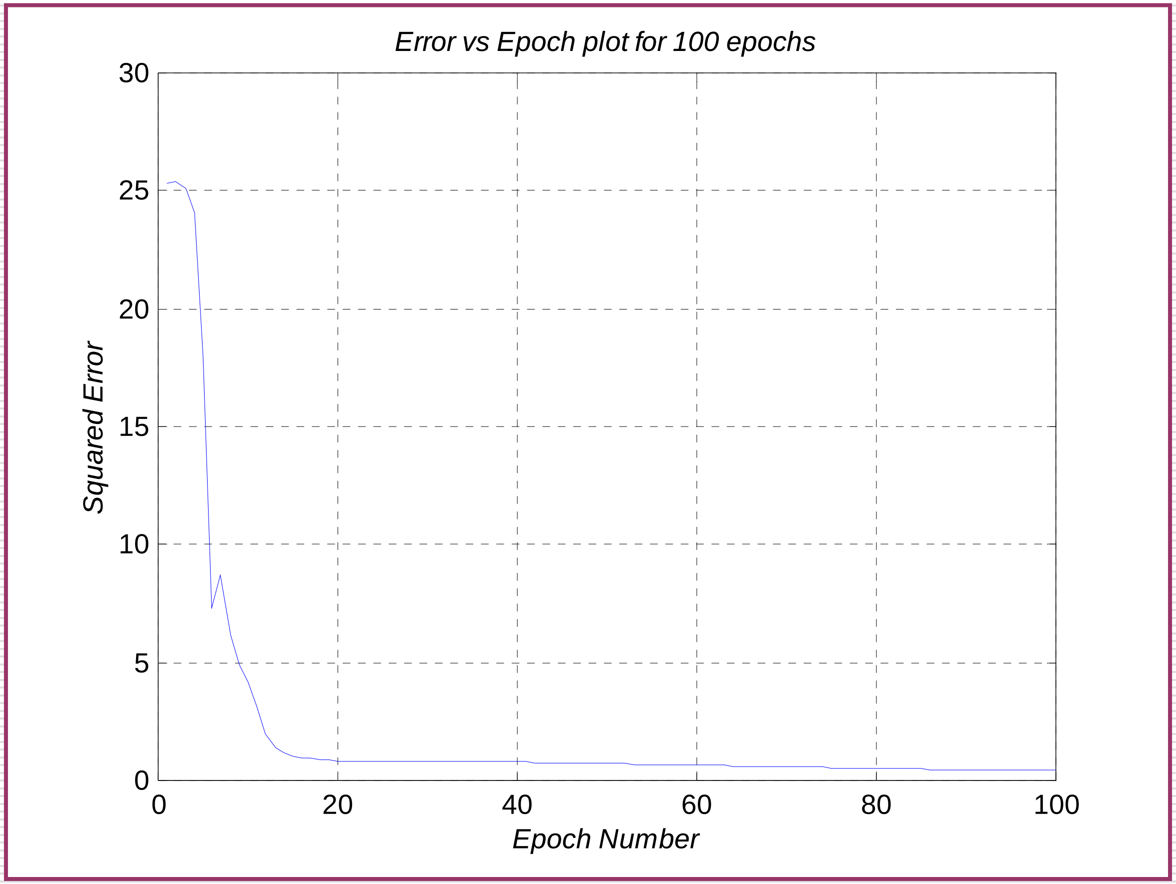


MATLAB Simulation 2: Function Approximation

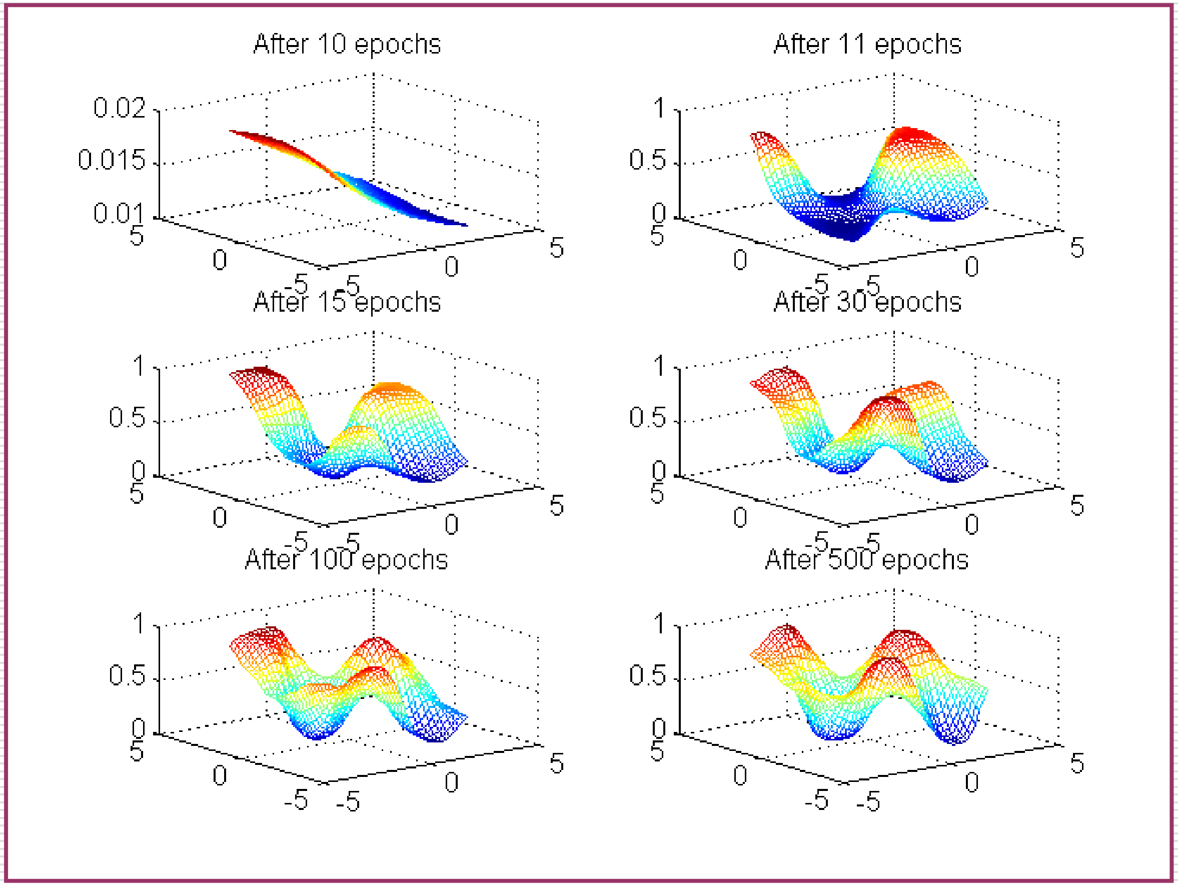


Parameter	*Number / Value*
Input nodes	2
Hidden nodes–Layer 1	10
Hidden nodes–Layer 2	10
Number of patterns	625
Learning rate, η	0.9
Momentum, α	0.6
Gain Scale, λ	1.0
Tolerance, τ	0.05

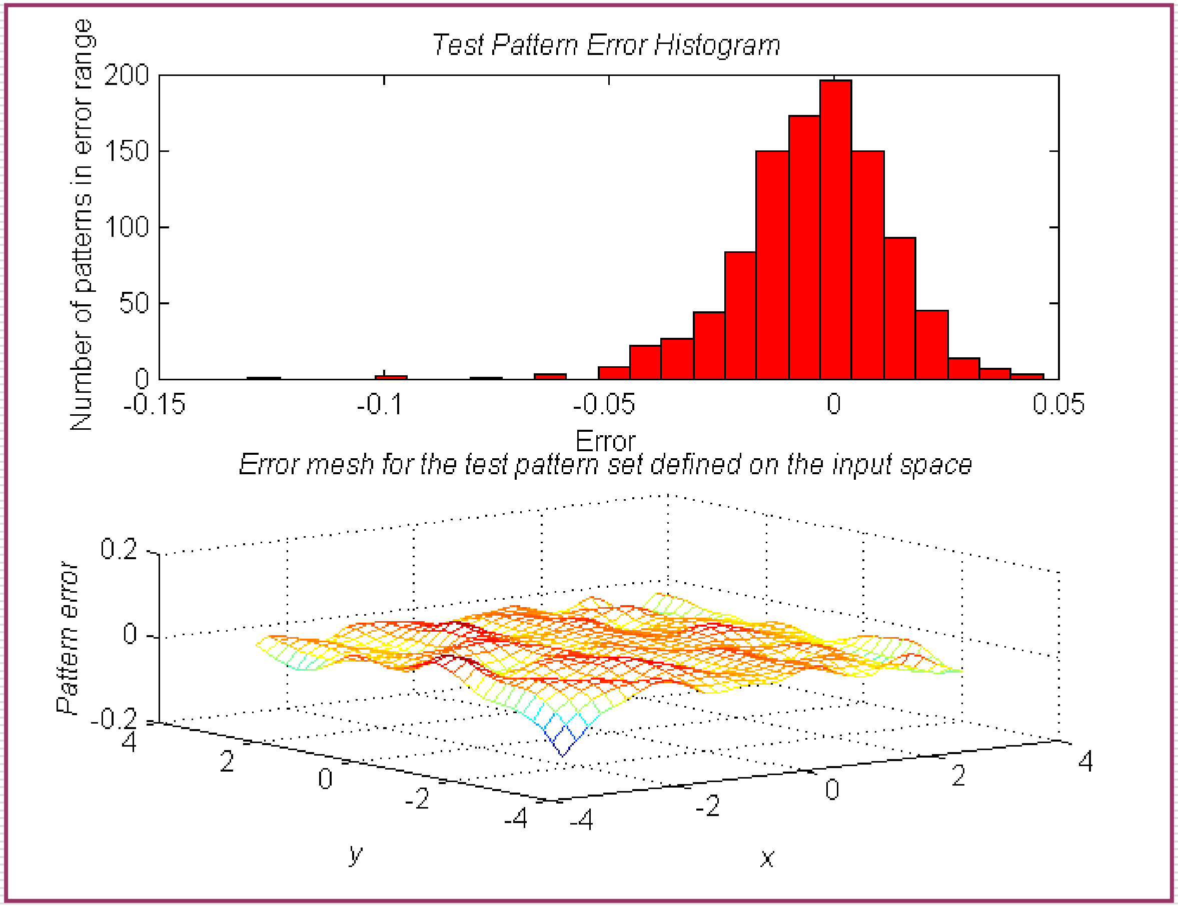
MATLAB Simulation 2: Error vs Epochs



MATLAB Simulation 2: Simulation Snapshots



MATLAB Simulation 2: Error Histogram and Error Mesh



MATLAB Code for Backprop

```
pattern=[0.1 0.1  0.1
          0.1 .95 .95
         .95 0.1 .95
         .95 .95 0.1];
eta = 1.0;          % Learning rate
alpha = 0.7;        % Momentum
tol = 0.001;        % Error tolerance
Q = 4;              % Total no. of the patterns to be input
n = 2; q = 2; p = 1; % Architecture
Wih = 2 * rand(n+1,q) - 1;          % Input-hidden weight matrix
Whj = 2 * rand(q+1,p) - 1;          % Hidden-output weight matrix
DeltaWih = zeros(n+1,q);            % Weight change matrices
DeltaWhj = zeros(q+1,p);
DeltaWihOld = zeros(n+1,q);
DeltaWhjOld = zeros(q+1,p);
```

MATLAB Code for Backprop - 2

```
Si = [ones(Q,1) pattern(:,1:2)]; % Input signals
D = pattern(:,3);             % Desired values
Sh = [1 zeros(1,q)];          % Hidden neuron signals
Sy = zeros(1,p);              % Output neuron signals
deltaO = zeros(1,p);          % Error-slope product at output
deltaH = zeros(1,q+1);        % Error-slope product at hidden
sumerror = 2*tol;              % To get in to the loop
while (sumerror > tol)         % Iterate
sumerror = 0;
for k = 1:Q
   Zh = Si(k,:) * Wih;                   % Hidden activations
   Sh = [1 1./(1 + exp(-Zh))];           % Hidden signals
   Yj = Sh * Whj;                        % Output activations
   Sy = 1./(1 + exp(-Yj));               % Output signals
   Ek = D(k) - Sy;                       % Error vector
   deltaO = Ek .* Sy .* (1 - Sy);        % delta output
```

MATLAB Code for Backprop - 3

```
    for h = 1:q+1
        DeltaWhj(h,:) = deltaO * Sh(h);      % Delta W:hidden-output
    end
    for h = 2:q+1 % delta hidden
       deltaH(h)=(deltaO*Whj(h,:)')*Sh(h)*(1-Sh(h));
    end
    for i = 1:n+1 % Delta W:input-hidden
       DeltaWih(i,:) = deltaH(2:q+1) * Si(k,i);
    end % Update weights
    Wih = Wih + eta * DeltaWih + alpha * DeltaWihOld;
    Whj = Whj + eta * DeltaWhj + alpha * DeltaWhjOld;
    DeltaWihOld = DeltaWih; DeltaWhjOld = DeltaWhj; % Store changes
    sumerror = sumerror + sum(Ek.^2); % Compute error
  end
  sumerror %Print epoch error
end
```

Practical Considerations: Pattern or Batch Mode Training

- Pattern Mode:
  - ↓ Present a single pattern
  - ↓ Compute local gradients
  - ↓ Change the network weights
- Given Q training patterns $\{X_i, D_i\}_{i=1}^{Q}$, and some initial neural network N_0, pattern mode training generates a sequence of Q neural networks $N_1, \ldots, N_Q$ over one epoch of training.
- Batch Mode (true gradient descent) :
  - ↓ Collect the error gradients over an entire epoch
  - ↓ Change the weights of the initial neural network N_0 in one shot.

Practical Considerations: When Do We Stop Training?

1. Compare absolute value of squared error averaged over one epoch, E_{av}, with a training tolerance, typically 0.01 or as low as 0.0001.
2. Alternatively use the absolute rate of change of the mean squared error per epoch.
3. Stop the training process if the Euclidean norm of the error gradient falls below a sufficiently small threshold. (Requires computation of the gradient at the end of each epoch.)
4. Check the generalization ability of the network. The network generalizes well if it is able to predict correct or near correct outputs for unseen inputs.
   - Partition the data set T into two subsets: $T_{training}$ (used for estimation) and T_{test} (used for evaluation of the network)
   - $T_{training}$ is divided into $T_{learning}$ and $T_{validation}$. ($T_{validation}$ might comprise 20 – 30 per cent of the patterns in $T_{training}$.
   - Use $T_{learning}$ to train the network using backpropagation.
   - Evaluate network performance at the end of each epoch using $T_{validation}$.
   - Stop the training process when the error on $T_{validation}$ starts to rise.

Practical Considerations: Use a Bipolar Signal Function

- Introducing a bipolar signal function such as the hyperbolic tangent function can cause a significant speed up in the network convergence.
- Specifically, $S(x) = a \tanh(\lambda x)$ with $a = 1.716$ and $\lambda = 0.66$ being suitable values.
- The use of this function comes with the added advantage that the range of valid desired signals extends to $[-1 + \varepsilon, 1 - \varepsilon]$ where $\varepsilon > 0$.

Practical Considerations: Weight Initialization

- Choose small random values within some interval $[-\varepsilon, +\varepsilon]$. (Identical initial values can lead to network paralysis—the network learns nothing.)
- Avoid very small ranges of weight randomization—may lead to very slow learning initially.
- Incorrect choice of weights might lead to network saturation where weight changes are almost negligible over consecutive epochs.
  - May be incorrectly interpreted as a local minimum.
  - Signal values are close to the 0 or 1; signal derivatives are infinitesimally small.
  - Weight changes are negligibly small.
  - Small weight changes allow the neuron to escape from *incorrect saturation* only after a very long time.
  - Randomization of network weights helps avoid these problems.
- For bipolar signal functions it is useful to randomize weights depending on individual neuron *fan-in,* f_i : randomized in the interval $(-2.4/f_i, 2.4/f_i)$

Practical Considerations: Check the Input and Target Ranges

- Given a logistic signal function which ranges in the interval (0,1) the desired outputs of patterns in the entire training set should lie in an interval $[0 + \varepsilon, 1 - \varepsilon]$ where $\varepsilon > 0$ is some small number.
- Desired values of 0 and 1 causes weights to grow increasingly large in order to generate these limiting values of the output.
- To generate a 0 and 1 requires a $-\infty$ or ∞ activation which can be accomplished by increasing the values of weights.
- Algorithm cannot be expected to converge if desired outputs lie outside the interval [0,1].
- If one were to use a hyperbolic tangent signal function with the range [–1.716,+1.716] , then target values of –1, 0 or +1 would be perfectly acceptable.

Practical Considerations: Adjusting Learning Rates

- For small learning rates, convergence to the local minimum in question is guaranteed but may lead to long training times.
- If network learning is non-uniform, and we stop before the network is trained to an error minimum, some weights will have reached their final "optimal" values; others may not have.
  - In such a situation, the network might perform well on some patterns and very poorly on others.
- If we assume that the error function can be approximated by a quadratic then we can make the following observations.
  - An optimal learning rate reaches the error minimum in a single learning step.
  - Rates that are lower take longer to converge to the same solution.
  - Rates that are larger but less than twice the optimal learning rate converge to the error minimum but only after much oscillation.
  - Learning rates that are larger than twice the optimal value will diverge from the solution.

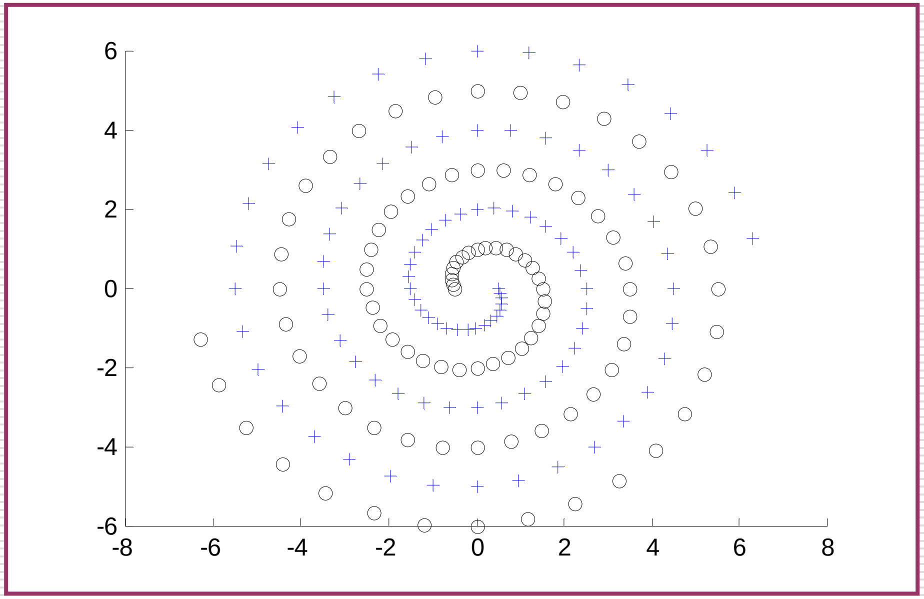
Practical Considerations: Selection of a Network Architecture

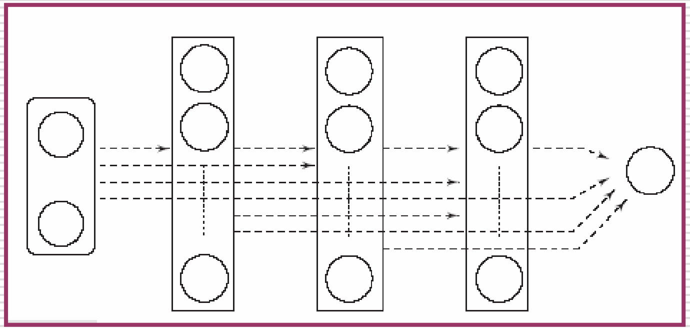
- A three-layered network can approximate any continuous function.
- Problem with multilayered nets using one hidden layer:
  - neurons tend to interact with each other globally
  - interactions make it difficult to generate approximations of arbitrary accuracy.
- With two hidden layers the curve-fitting process is easier:
  - The first hidden layer extracts local features of the function (as binary threshold neurons partition the input space into regions.)
  - Global features are extracted in the second hidden layer.

Practical Considerations: Cross Validation

- Divide the data set into a training set $T_{training}$ and a test set T_{test} .
- Subdivide $T_{training}$ into two subsets: one to train the network $T_{learning}$, and one to validate the network $T_{validation}$.
- Train different network architectures on $T_{learning}$ and evaluate their performance on $T_{validation}$.
- Select the best network.
- Finally, retrain this network architecture on $T_{training}$.
- Test for generalization ability using T_{test}.

Backpropagation: Two-Spirals Problem





- Solved by Lang and Witbrock 2-5-5-5-1 architecture 138 weights.
- In the Lang-Witbrock network each layer of neurons is connected to *every* succeeding layer.

Structure Growing Algorithms

- Approach 1:
  - Starts out with a large number of weights in the network and gradually prunes them.
  - Idea: eliminate weights that are least important.
  - Examples: Optimal Brain Damage, Optimal Brain Surgeon, Hinton weight decay procedure.
- Approach 2:
  - Starts out with a minimal architecture which is made to *grow* during training.
  - Examples: Tower algorithm, Pyramid algorithm, cascade correlation algorithm.

Structure Growing Algorithms: Tower Algorithm

- Train a single TLN using the pocket learning algorithm with a ratchet.
  - If there are n inputs, we have to train $n + 1$ weights.
- Freeze the weights of the TLN.
- Now add a new TLN to the system.
  - This TLN gets all the original n inputs, as well as an additional input from the first TLN trained.
- Train the $n + 2$ weights of this TLN using the pocket learning algorithm with ratchet.
- Continue this process until no further improvement in classification accuracy is achieved.
- Freeze the weights of the TLN.
  - Each added TLN is guaranteed to correctly classify a greater number of input patterns
- The tower algorithm is guaranteed to classify linearly non-separable pattern sets with an arbitrarily high probability provided
  - enough TLNs are added
  - enough training iterations are provided to train each incrementally added TLN.

Structure Growing Algorithms: Pyramid Algorithm

- Similar to the tower algorithm
- Each added TLN receives inputs from the n original inputs
- The added TLN receives inputs from *all* previously added TLNs.
- Training is done using the pocket algorithm with ratchet.

Structure Growing Algorithms: Cascade Correlation Algorithm

- Assume a minimal network structure: n input neurons; p output neurons with full feedforward connectivity, the requisite bias connections, and no hidden neurons.

- This network is trained using backpropagation learning (or the Quickprop algorithm) until no further reduction in error takes place.
- Errors at output neurons are computed over the entire pattern set.
- Next, a single hidden neuron is added. This neuron receives $n + 1$ inputs including the bias and is not connected to the p output neurons.
- The weights of this hidden neuron are adjusted using BP or Quickprop
- The goal of training is to maximize the correlation C between the signal of the hidden neuron and the residual output error.

$$C = \sum_{j=1}^{p} \left| \sum_{k=1}^{Q} \left(\mathcal{S}(z_h^k) - \mathcal{S}_{av}\right)\left(\delta_j^k - \Delta_j\right) \right|$$

Quickprop: Fast Relative of Backprop

- Works with second order error derivative information instead of only the usual first order gradients.
- Based on two "risky" assumptions:
  - The error function E is a parabolic function of any weight w_{ij}.
  - The change in the slope of the error curve is independent of other concurrent weight changes.

- The slope of the error function is thus linear.
- The algorithm pushes the weight w_{ij} directly to a value that minimizes the parabolic error.
- To compute this weight, we require the previous value of the gradient $\partial E/\partial w_{ij}^{k-1}$ and the previous weight change.

$$\Delta w_{ij}^{k} = \frac{\frac{\partial \mathcal{E}}{\partial w_{ij}^{k}}}{\frac{\partial \mathcal{E}}{\partial w_{ij}^{k-1}} - \frac{\partial \mathcal{E}}{\partial w_{ij}^{k}}} \Delta w_{ij}^{k-1} = \alpha_{ij}^{k} \Delta w_{ij}^{k-1}$$

$$\Delta w_{ij}^{k} = -\eta \; I\left[\frac{\partial \mathcal{E}}{\partial w_{ij}^{k}} \Delta w_{ij}^{k-1} > 0\right] \frac{\partial \mathcal{E}}{\partial w_{ij}^{k}} + \alpha_{ij}^{k} \Delta w_{ij}^{k-1}$$

Universal Function Approximation

- Kolmogorov proved that any continuous function f defined on an n-dimensional cube is representable by sums and superpositions of continuous functions of exactly one variable:

$$f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \phi_q \left[\sum_{p=1}^{n} \psi_{pq}(x_p) \right]$$

Universal Approximation Theorem

Let $\phi(\cdot)$ be a non-constant, bounded, and monotone-increasing continuous function. Let $\mathbb{I}^n$ denote the n-dimensional unit hypercube $[0, 1]^n$, and the space of continuous functions on $\mathbb{I}^n$ be denoted by $C(\mathbb{I}^n)$. Then given any function $F \in C(\mathbb{I}^n)$ and $\epsilon > 0$, there exists an integer p and sets of real constants α_j, θ_j, and w_{ij}, where $i = 1, \ldots, n$ and $j = 1, \ldots, p$ such that we may define

$$f(X, W) = \sum_{j=1}^{p} \alpha_j \, \phi\left(\sum_{i=1}^{n} w_{ij} x_i - \theta_j\right) \quad X \in \mathbb{I}^n, W \in \mathbb{R}^{n \times p}$$

as an approximate realization of the function $F(X)$ where

$$|f(X, W) - F(X)| < \epsilon$$

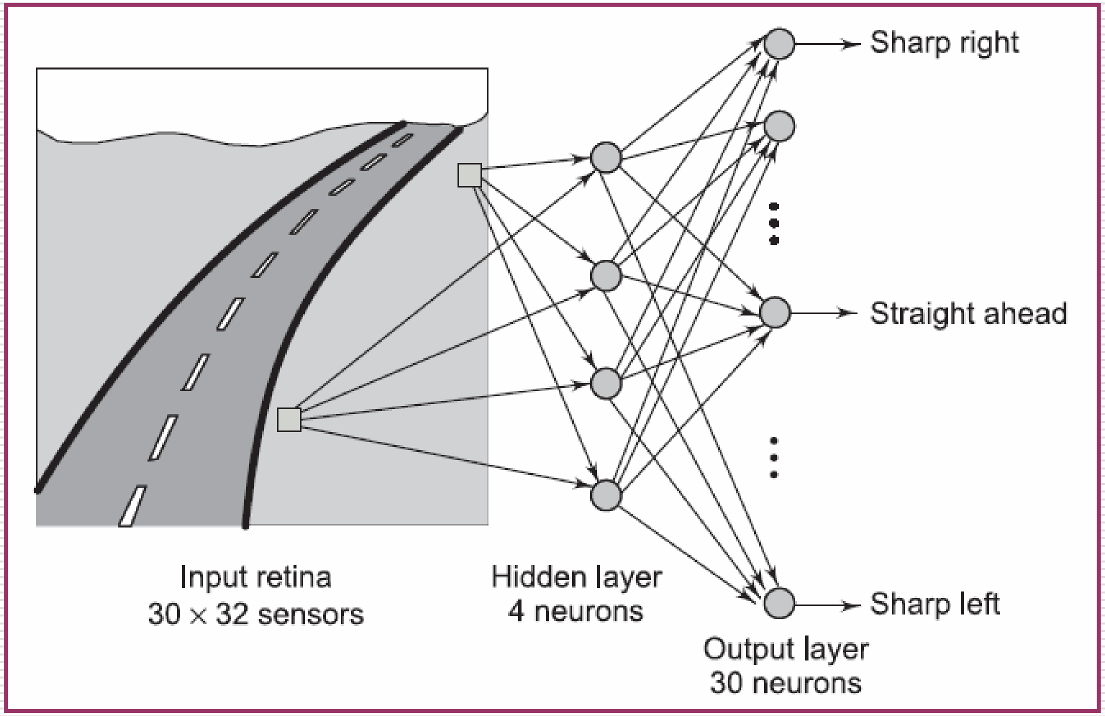
for all $X \in \mathbb{I}^n$.

Applications of BP: Steering Autonomous Vehicles

- The primary objective is to steer a robot vehicle like Carnegie Mellon University's (CMU) Navlab, which is equipped with motors on the steering wheel, brake and accelerator pedal thereby enabling computer control of the vehicles' trajectory.

Applications of BP: Steering Autonomous Vehicles-ALVINN

- ALVINN
  - (autonomous land vehicle in a neural network)



ALVINN Network Architecture

- Input to the system is a 30 × 32 neuron "retina"
- Video images are projected onto the retina.
- Each of these 960 input neurons is connected to four hidden layer neurons which are connected to 30 output neurons.
- Output neurons represent different steering directions—the central neuron being the "straight ahead" and the first and last neurons denoting "sharp left" and "sharp right" turns of 20 m radius respectively.
- To compute an appropriate steering angle, an image from a video camera is reduced to 30 × 32 pixels and presented to the network.
- The output layer activation profile is translated to a steering command using a *center of mass* around the hill of activation surrounding the output neuron with the largest activation.
- Training of ALVINN involves presentation of video images as a person drives the vehicle using the steering angle as the desired output.

CMU NAVLAB and ALVINN

- ALVINN runs on two SUNSPARC stations on board Navlab and training on the fly takes about two minutes.
- During this time the vehicle is driven over a 1/4 to 1/2 mile stretch of the road and ALVINN is presented about 50 images, each transformed 15 times to generate 750 images.
- The ALVINN system successfully steers NAVLAB in a variety of weather and lighting conditions. With the system capable of processing 10 images/second Navlab can drive at speeds up to 55 mph, five times faster than any other connectionist system.
- On highways, ALVINN has been trained to navigate at up to 90 mph!

Reinforcement Learning: The Underlying Principle

- If an action of a system is followed by a satisfactory response, then strengthen the tendency to produce that action.
- Evaluate the success and failure of a neuron to produce a desired response.
  - If success: encourage the neuron to respond in the same way by supplying a reward
  - Otherwise supply a penalty
- Requires the presence of an external critic that evaluates the response within an environment.

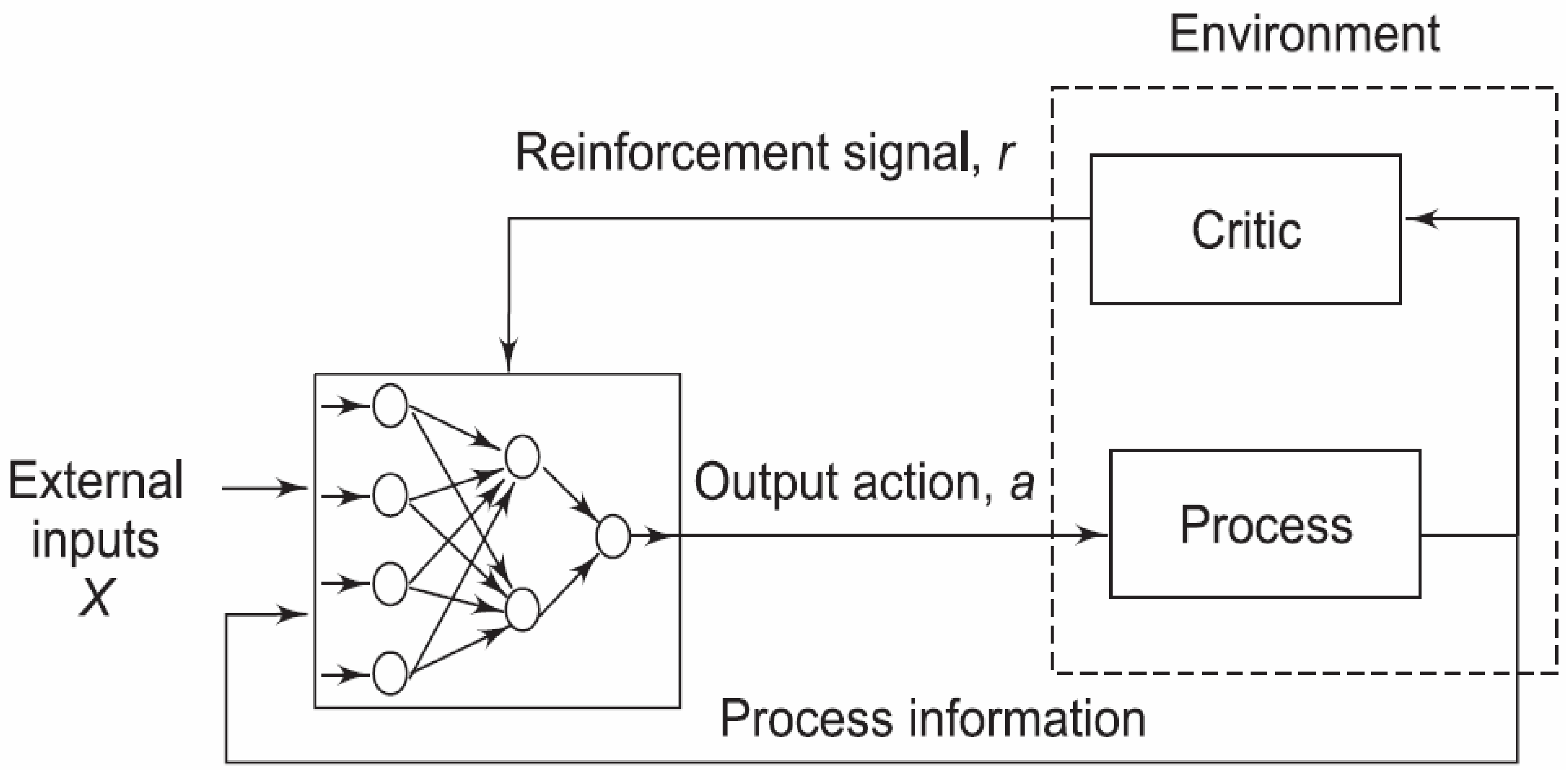
Types of Reinforcement Learning

- Reinforcement learning algorithms generally fall into one of three categories:
  - Non-associative
  - Associative
  - Sequential

Three Basic Components

- A critic which sends the neural network a reinforcement signal whose value at any time k, is a measure of the “goodness” of the behaviour of the process at that point of time.
- A learning procedure in which the network updates its parameters—that determine its actions—based on this coarse information.
- The generation of another action, and the subsequent repetition of the above two operations.

Architecture of Reinforcement Learning Networks



Nonassociative Reinforcement Learning

- Extensively studied as a part of learning automata theory
- Assume that the learning system has m possible actions which we denote by a_i , $i = 1 \ldots m$.
- The effect of these actions on the binary (or bipolar) success–failure reinforcement signal can be modelled as a collection of probabilities which are denoted by P_i—which is the probability of success given that the learning system generated an action a_i.
- Objective: Maximize the probability of receiving a "success"— perform an action a_j such that the probability $P_j = \max(P_i)$, $i = 1 \ldots m$.

Associative Reinforcement Learning

- Assume that at time instant k stimulus vector X_k buffers the system.
- System selects an action $a_k = a_j$ through a procedure that usually depends on X_k.
- Upon execution of this action, the critic provides its reinforcement signals: “success” with probability $P_j(X_k)$ and “failure” with probability $1 - P_j(X_k)$.
- Objective: Maximize the success probability—at all subsequent time instants k the learning system executes action $a_k = a_j$, such that $P_j(X_k) = \max(P_i(Xk))$ $i = 1 \ldots m$.

Associative Reinforcement Learning Rules

- Consider the j^{th} neuron in a field that receives an input stimulus vector $X_k = (x_0^k, \ldots, x_n^k)$ at time k in addition to the critic's reinforcement signal, r_k.
- Let $W_k = (w_{0j}^k, \ldots, w_{nj}^k)$ and $a_k = s_j^k$ respectively denote the neuronal weight vector and action (neuron signal), and let y_j^k denote the neuronal activation at time k.

$$y_j^k = \sum_{i=0}^{n} w_{ij}^k x_i^k$$

Associative Search Unit

- Extension of the Hebbian learning rule.
- Neuron signal function is assumed to be a probabilistic function of its activation.

$$s_j^k = \begin{cases} 1 & \text{with probability } P(y_j^k) \\ 0,\ -1 & \text{with probability } 1 - P(y_j^k) \end{cases}$$

$$P\left[s_j^k = +1\right] = P(y_j^k) = \frac{1}{1 + \exp(-2\beta y_j^k)}$$

- where $y_j^k = \sum_i w_{ij}^k x_i^k$

Associative Search Neuron Weight Updated

- This is essentially the Hebbian learning rule with the reinforcement signal acting as an additional modulatory factor.
- $\Delta W_k = \eta r_k s_j^{k-\tau} X_{k-\tau}$ where we assume that the critic takes a (discrete) time τ to evaluate the output action, and $r_k \in \{1,-1\}$ such that +1 denotes a success and –1 a failure.
- As before, $\eta > 0$ is the learning rate.
- The interpretation of this rule is as follows:
  - if the neuron fires a signal $s_j^k = +1$ in response to an input X_k, and this action is followed by "success", then change the weights so that the neuron will be more likely to fire a +1 signal in the presence of X_k.
- The converse is true for failure reinforcement.

Selective Bootstrapping

- The neuron signal $s_j^k \in \{0, 1\}$ is computed as the deterministic threshold of the activation, y_j^k.
- It receives a reinforcement signal, r_k, and updates its weights according to a *selective bootstrap* rule:

$$\Delta W_k = \begin{cases} \eta(s_j^k - y_j^k)X_k & \text{if } r_k = \text{ reward} \\ \eta(1 - s_j^k - y_j^k)X_k & \text{if } r_k = \text{ penalty} \end{cases}$$

- The reinforcement signal r_k simply evaluates s_j^k.
- When s_j^k produces a "success", the LMS rule is applied with a desired value s_j^k ← *positive bootstrap adaptation*
- when s_j^k produces a "failure", the LMS rule is applied with a desired value $1 - s_j^k$. ← *negative bootstrap adaptation*

Associative Reward-Penalty Neurons

- The ARP neuron combines stochasticity with selective bootstrapping.

$$\Delta w_{ij}^k = \begin{cases} \eta(s_j^k - E[s_j^k])x_i^k & \text{if } r_k = +1 \text{ (reward)} \\ \lambda\eta(-s_j^k - E[s_j^k])x_i^k & \text{if } r_k = -1 \text{ (penalty)} \end{cases}$$

- $E[s_j^k] = (+1)P(y_j^k) + (-1)1 - P(y_j^k) = \tanh \beta y_j^k$
- Asymmetry is important: asymptotic performance improves as λ approaches zero.
- If binary rather than bipolar neurons are used, the $-s_j^k$ in the penalty case is replaced by $1 - s_j^k$.
- $E[s_j^k]$ then represents a probability of getting a 1.

Reinforcement Learning Networks

- Networks of *ARP* neurons have been used successfully in both supervised and associative reinforcement learning tasks in feedforward architectures.
- Supervised learning:
  - output layer neurons learn as in standard error backpropagation
  - hidden layer neurons learn according to the *ARP* rule.
  - the reinforcement signal is defined to increase with a decrease in the output error.
  - Hidden neurons learn simultaneously using this reinforcement signal.
- If the entire network is involved in an associative reinforcement learning task, then all the neurons which are *ARP* neurons receive a common reinforcement signal.
  - *self-interested* or *hedonistic neurons*
  - attempt to achieve a global purpose through individual maximization of the reinforcement signal *r*.

Observations on Reinforcement Learning

- A critical aspect of reinforcement learning is its stochasticity.
- A critic is an abstract process model employed to evaluate the actions of learning networks.
- A reinforcement signal need not be just a two-state success/failure signal. It can be a signal that takes on real values in which case the objective of learning is to maximize its expected value.
- The critic's signal does not suggest which action is the best; it is only evaluative in nature. No error gradient information is available, and this is an important aspect in which reinforcement learning differs from supervised learning.

Observations on Reinforcement Learning (contd.)

- There must be a variety in the process that generates outputs.
  - Permits the varied effect of alternative outputs to be compared following which the best can be selected.
  - Behavioural variety is referred to as *exploration*
  - Randomness plays an important role.
- Involves a trade-off between *exploitation* and *exploration*
  - Network learning mechanism has to exploit what it has already learnt to obtain a consistently high success rate
  - At the same time it must explore the unknown in order to learn more.
  - These are conflicting requirements, and reinforcement learning algorithms need to carefully balance them.