Module 2

Adaline Madaline

Syllabus

Training Techniques for ANNs 10 CO2

2.1 Introduction to supervised and unsupervised learning, Adaline and Madaline

2.2 Hebbian learning, Perceptron Learning, Delta learning rule, Widrow Hoff learning, Winner take all Learning Rule , Out star learning

2.3 Multilayer Feedforward Network, Error Back Propagation Training, Learning factors.

Adaptive Linear Neuron (Adaline)

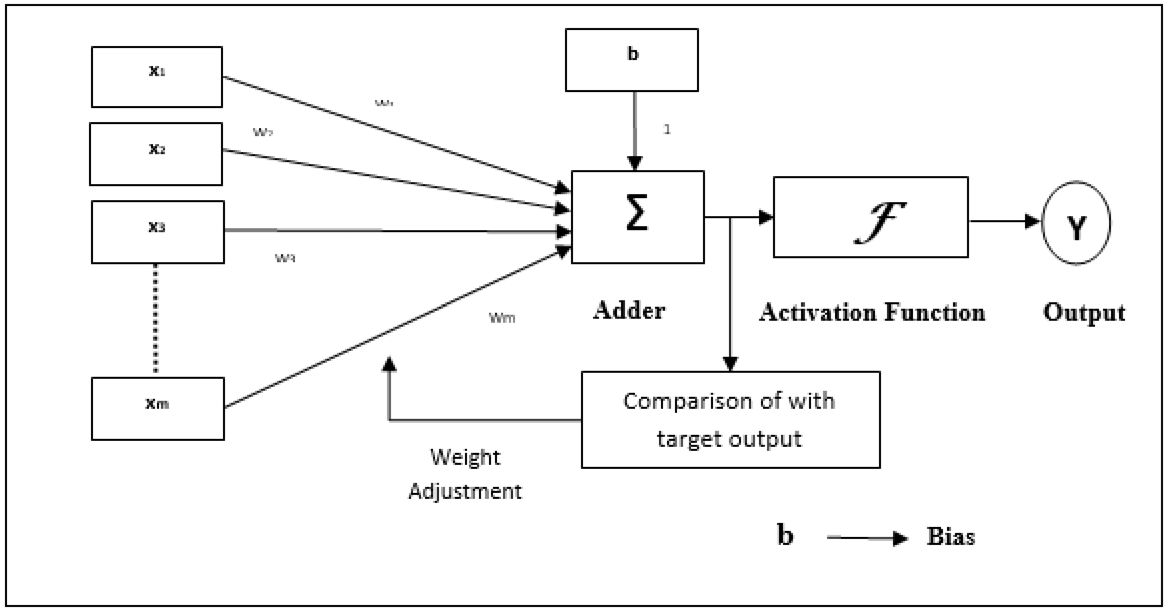
Adaline which stands for Adaptive Linear Neuron, is a network having a single linear unit.

It was developed by Widrow and Hoff in 1960.

Some important points about Adaline are as follows –

- It uses bipolar activation function.
- It tries to minimize the Mean-Squared Error (MSE) between the actual output and the desired/target output.
- The weights and the bias are adjustable.

Architecture



Algorithm

Step 1: Initialize the following to start the training –Weights, Bias, Learning rate α

Step 2: While the stopping condition is False do steps 3 to 7.

Step 3: for each training set perform steps 4 to 6.

Step 4: Set activation of input unit $x_i = s_i$ for (i=1 to n).

Step 5: compute net input to output unit $y_{in} = \sum w_i x_i + b$

Here, b is the bias and n is the total number of neurons.

Step 6: Update the weights and bias for i=1 to n

$$w_i(new) = w_i(old) + \alpha(t - y_{in})x_i$$
$$b(new) = b(old) + (t - y_{in})$$

and calculate $error : (t - y_{in})^2$

Step 7: Test the stopping condition. The stopping condition may be when the weight changes at a low rate or no change.

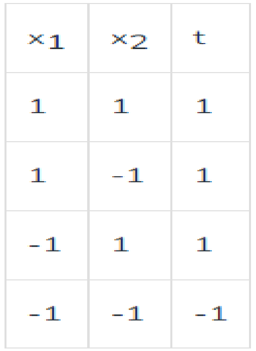
Problem:Design OR gate using Adaline Network.

Solution :

Initially, all weights are assumed to be small random values, say 0.1, and set learning rule to 0.1.

Also, set the least squared error to 2.

The weights will be updated until the total error is greater than the least squared error.

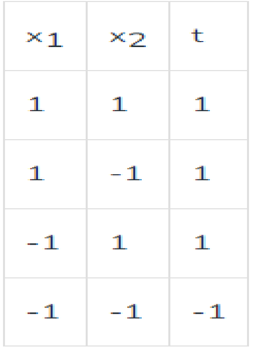


x_1	x_2	t
1	1	1
1	-1	1
-1	1	1
-1	-1	-1

Calculate the net input $y_{in} = \sum w_i x_i + b$

(when $x_1=x_2=1$) $y_{in} = 0.1 \times 1 + 0.1 \times 1 + 0.1 = 0.3$

Now compute, $(t-y_{in})=(1-0.3)=0.7$, calculate the error $error = (t - y_{in})^2 = 0.7^2 = 0.49$



x1	x2	t
1	1	1
1	-1	1
-1	1	1
-1	-1	-1

Now, update the weights and bias

$$w_i(new) = w_i(old) + \alpha(t - y_{in})x_i$$
$$b(new) = b(old) + (t - y_{in})$$

$$w_1(new) = 0.1 + 0.1(1 - 0.3)1 = 0.17$$
$$w_2(new) = 0.1 + 0.1(1 - 0.3)1 = 0.17$$

$$b(new) = 0.1 + 0.1(1 - 0.3) = 0.17$$

x_1	x_2	t	y_{in}	$(t-y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.1)	w_2 (0.1)	b (0.1)	$(t-y_{in})$^2
1	1	1	0.3	0.7	0.07	0.07	0.07	0.17	0.17	0.17	0.49
1	-1	1	0.17	0.83	0.083	-0.083	0.083	0.253	0.087	0.253	0.69
-1	1	1	0.087	0.913	-0.0913	0.0913	0.0913	0.1617	0.1783	0.3443	0.83
-1	-1	-1	0.0043	-1.0043	0.1004	0.1004	-0.1004	0.2621	0.2787	0.2439	1.01

This is epoch 1 where the total error is 0.49 + 0.69 + 0.83 + 1.01 = 3.02 so more epochs will run until the total error becomes less than equal to the least squared error i.e 2.

```
# Predict from the evaluated weight and bias of adaline
def prediction(X,w,b):
        y=[]
        for i in range(X.shape[0]):
                x = X[i]
                y.append(sum(w*x)+b)
        return y
prediction(x,w,b)
```

```
[array([1.0192042]),
array([0.99756877]),
array([0.99756877]),
array([-0.99756877])]
```

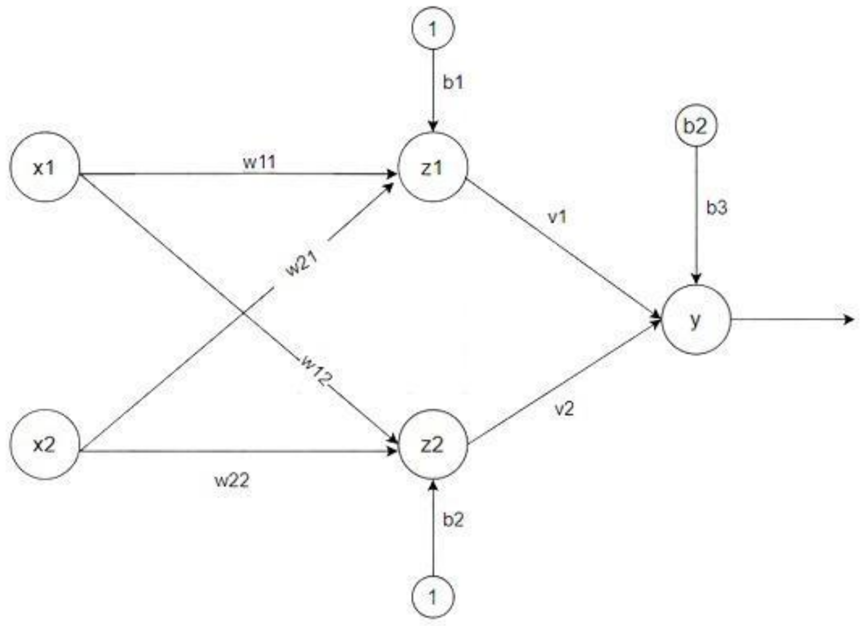
Output:

```
Error : [2.33228319]
Error : [1.09355784]
Error : [0.73680883]
Error : [0.50913731]
Error : [0.35233593]
Error : [0.24384625]
Error : [0.16876305]
Error : [0.11679891]
Error : [0.08083514]
Error : [0.05594504]
Error : [0.0387189]
Error : [0.02679689]
Error : [0.01854581]
Error : [0.01283534]
Error : [0.00888318]
Error : [0.00614795]
Error : [0.00425492]
Error : [0.00294478]
Error : [0.00203805]
Error : [0.00141051]
Error : [0.0009762]
weight : [0.01081771 0.01081771 0.98675106] Bias :
[0.01081771]
```

Madaline (Multiple Adaptive Linear Neuron) :

- The Madaline(supervised Learning) model consists of many Adaline in parallel with a single output unit. The Adaline layer is present between the input layer and the Madaline layer hence Adaline layer is a hidden layer. The weights between the input layer and the hidden layer are adjusted, and the weight between the hidden layer and the output layer is fixed.
- It may use the majority vote rule, the output would have an answer either true or false. Adaline and Madaline layer neurons have a bias of '1' connected to them. use of multiple Adaline helps counter the problem of non-linear separability.

Architecture



- There are three types of a layer present in Madaline
- First input layer contains all the input neurons, the Second hidden layer consists of an adaline layer, and weights between the input and hidden layers are adjustable and the third layer is the output layer the weights between hidden and output layer is fixed they are not adjustable.

Algorithm

Step 1: Initialize weight and set learning rate α.

$v_1=v_2=0.5$, $b=0.5$

other weight may be a small random value.

Step 2: While the stopping condition is False do steps 3 to 9.

Step 3: for each training set perform steps 4 to 8.

Step 4: Set activation of input unit xi = si for (i=1 to n).

Step 5: compute net input of Adaline unit

$z_{in1} = b_1 + x_1 w_{11} + x_2 w_{21}$

$z_{in2} = b_2 + x_1 w12 + x_2 w_{22}$

Step 6: for output of remote Adaline unit using activation function given below:

Activation function f(z) $= 1 \text{ if } z \geq 0 \ (and) \ (-1) \text{ if } z < 0$

z1=f(zin1)

z2=f(zin2)

Algorithm

Step 7: Calculate the net input to output.

$y_{in} = b_3 + z_1v_1 + z_2v_2$

Apply activation to get the output of the net

$y=f(y_{in})$

Step 8: Find the error and do weight updation

if $t \neq y$ then t=1 update weight on z(j) unit whose next input is close to 0.

if $t = y$ no updation

$w_{ij}(new) = w_{ij}(old) + \alpha(t-z_{inj})x_i$

$b_j(new) = b_j(old) + \alpha(t-z_{inj})$

if t=-1 then update weights on all unit z_k which have positive net input

Step 9: Test the stopping condition; weights change all number of epochs.

Training Techniques of ANN

(Module 2.2 of syllabus)

Outline

Hebbian learning,
Perceptron Learning,
Delta learning rule,
Widrow Hoff learning,
Winner take all Learning Rule ,
Out star learning

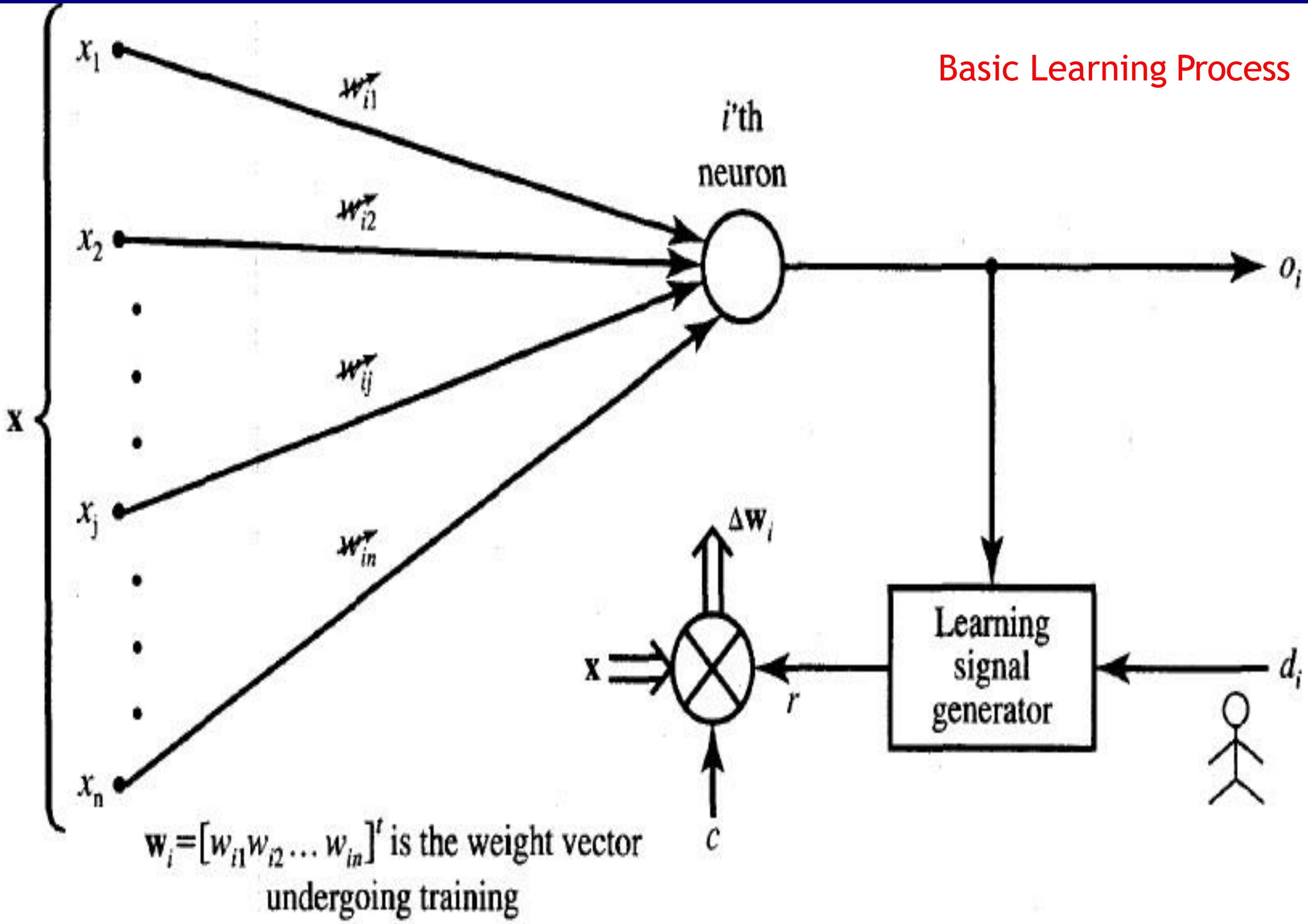
Neural Network Learning Rules

- A neuron →adaptive element.
- Weights are modifiable depending on the input signal it receives, its output value and the associated teacher response.

- In some cases the teacher signal is not available and no error information can be used, thus the neuron will modify its weights based only on the input/output as per unsupervised learning.
- A general rule adapted in neural network studies: the weight vector i.e.

$$\mathbf{w}_i = \begin{bmatrix} w_{i1} & w_{i2} & \cdots & w_{in} \end{bmatrix}^t$$

Basic Learning Process



$\mathbf{w}_i = [w_{i1} w_{i2} \ldots w_{in}]^t$ is the weight vector undergoing training

- Increase in weights proportion to the input x and learning signal r.
- Input learning signal r is a function of w_i, x and summations of the teacher's signal d_i.

$$r = r(\mathbf{w}_i, \mathbf{x}, d_i)$$

- The increment of the weight vector w_i produced by the learning step at time t according to the general rule is given by:

$$\Delta\mathbf{w}_i(t) = cr\left[\mathbf{w}_i(t), \mathbf{x}(t), d_i(t)\right]\mathbf{x}(t)$$

- C → positive number called the learning constant that determines rate of learner.

- The weight vector adapted at time t becomes at the next instant or learning step.

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + cr\left[\mathbf{w}_i(t), \mathbf{x}(t), d_i(t)\right]\mathbf{x}(t)$$

- For the $_k{}^{th}$ step we thus have

$$\mathbf{w}_i^{k+1} = \mathbf{w}_i^k + cr(\mathbf{w}_i^k, \mathbf{x}^k, d_i^k)\mathbf{x}^k$$

- Now continuous time learning can be expressed as

$$\frac{d\mathbf{w}_i(t)}{dt} = cr\mathbf{x}(t)$$

Learning Rules

Hebbian Learning Rule

Hebbian Learning Rule

- Unsupervised learning
- Learning signal = neuron’ output (Hebb.- 1949).
- We have

$$r \triangleq f(\mathbf{w}_i^t \mathbf{x})$$

- The increment Δw_i of the weight vector becomes

$$\Delta \mathbf{w}_i = cf(\mathbf{w}_i^t \mathbf{x})\mathbf{x}$$

$$f(net_i) = f(w_i^t . x) = o_i$$

$$net_i = w_i^t . x$$

- Single weight adapted using following increment

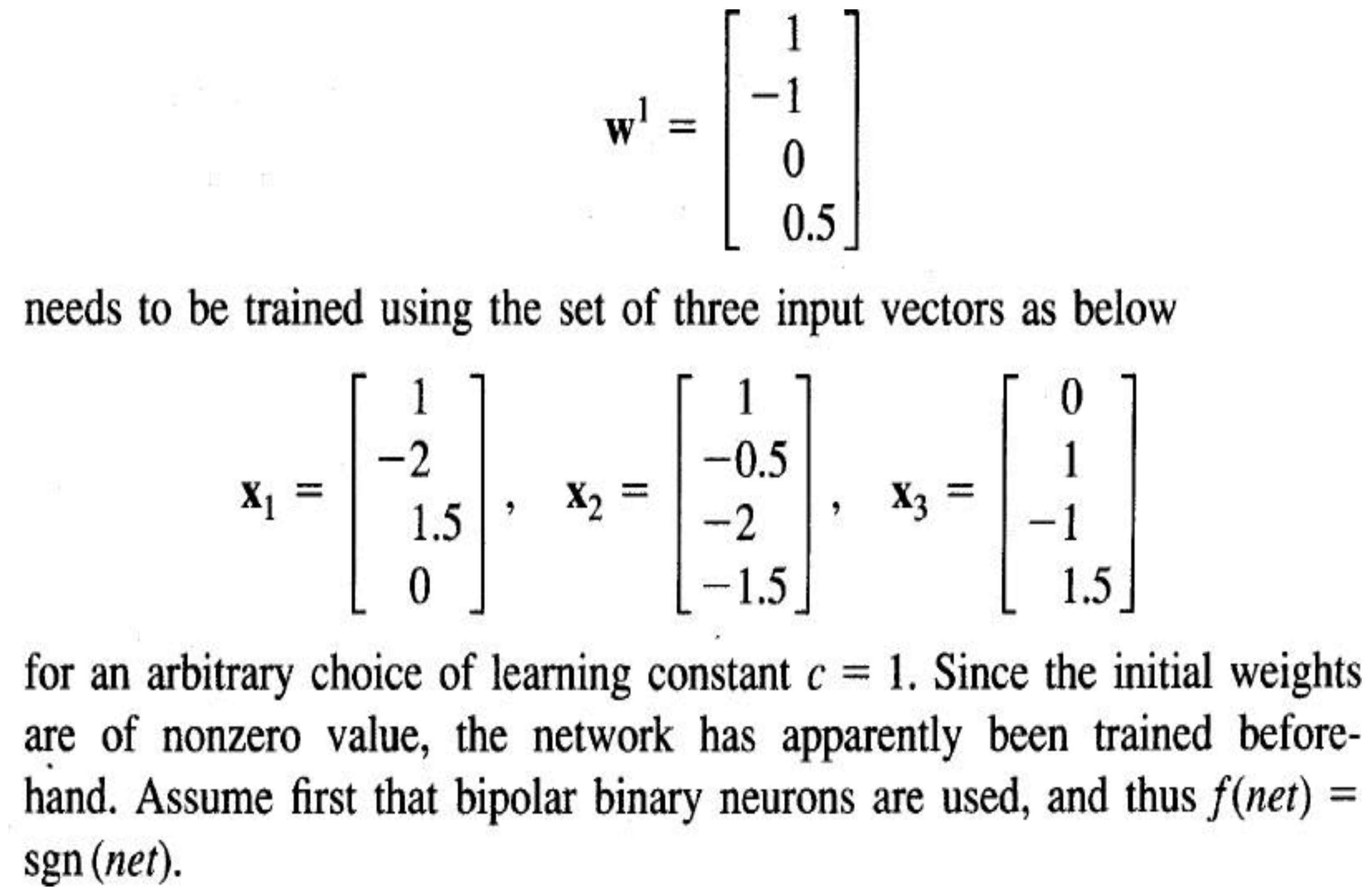
$$\Delta w_{ij} = cf(\mathbf{w}_i^t \mathbf{x})x_j$$

$$\Delta w_{ij} = co_i x_j, \quad \text{for } j = 1, 2, \ldots, n$$

Hebbian Learning Rule

- Requires the weight initialization at small random values around $w_i = 0$
- Represents a purely feed forward unsupervised learning.
- The rule states that "If the cross product of output and input or oscillation term $o_i x_j$ is positive then this will result in an increase of weight w_{ij} otherwise the weight decreases"

- Example:- Hebbian learning with binary and continuous activation functions of a very simple network is to be trained using an initial weight vector and three inputs



$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

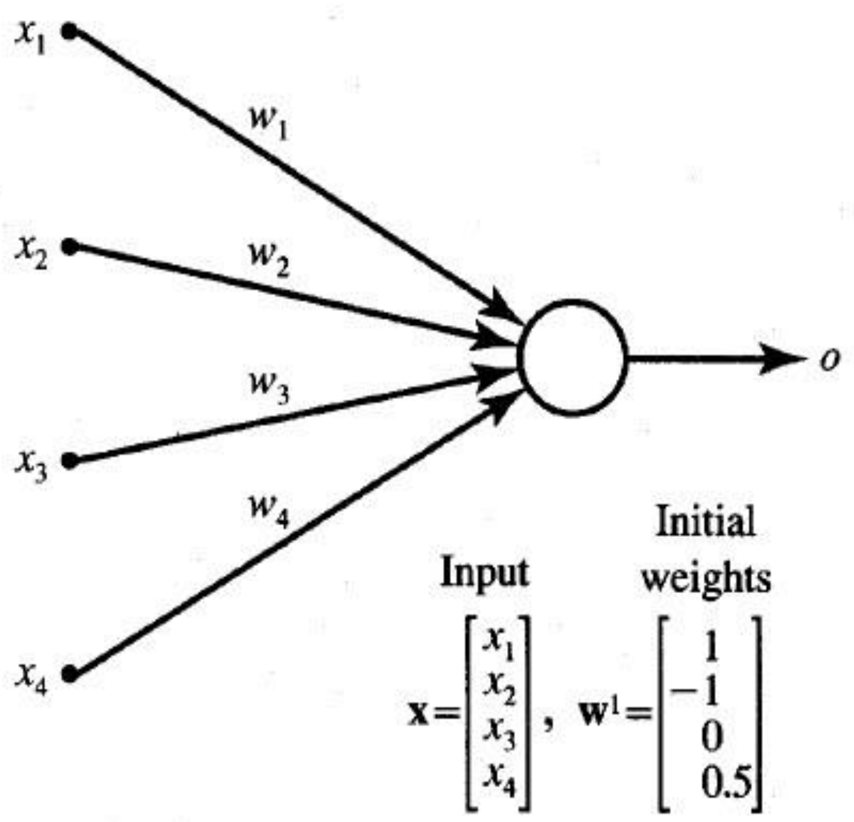
needs to be trained using the set of three input vectors as below

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix}$$

for an arbitrary choice of learning constant $c = 1$. Since the initial weights are of nonzero value, the network has apparently been trained beforehand. Assume first that bipolar binary neurons are used, and thus $f(net) = \operatorname{sgn}(net)$.

Step 1 Input $\mathbf{x}_1$ applied to the network results in activation net^1 as below:

$$net^1 = \mathbf{w}^{1t}\mathbf{x}_1 = \begin{bmatrix} 1 & -1 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = 3$$



x_1
x_2
x_3
x_4
w_1
w_2
w_3
w_4
o
Input
Initial weights
$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$, $\mathbf{w}^1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$

Figure 2.22 Network for training in Examples 2.4 through 2.6.

The updated weights are

$$\mathbf{w}^2 = \mathbf{w}^1 + \operatorname{sgn}(net^1)\mathbf{x}_1 = \mathbf{w}^1 + \mathbf{x}_1$$

and plugging numerical values we obtain

$$\mathbf{w}^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 1.5 \\ 0.5 \end{bmatrix}$$

where the superscript on the right side of the expression denotes the number of the current adjustment step.

Step 2 This learning step is with $\mathbf{x}_2$ as input:

$$net^2 = \mathbf{w}^{2t}\mathbf{x}_2 = \begin{bmatrix} 2 & -3 & 1.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} = -0.25$$

The updated weights are

$$\mathbf{w}^3 = \mathbf{w}^2 + \operatorname{sgn}(net^2)\mathbf{x}_2 = \mathbf{w}^2 - \mathbf{x}_2 = \begin{bmatrix} 1 \\ -2.5 \\ 3.5 \\ 2 \end{bmatrix}$$

Step 3 For input $\mathbf{x}_3$, we obtain in this step

$$net^3 = \mathbf{w}^{3t}\mathbf{x}_3 = \begin{bmatrix} 1 & -2.5 & 3.5 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix} = -3$$

The updated weights are

$$\mathbf{w}^4 = \mathbf{w}^3 + \operatorname{sgn}(net^3)\mathbf{x}_3 = \mathbf{w}^3 - \mathbf{x}_3 = \begin{bmatrix} 1 \\ -3.5 \\ 4.5 \\ 0.5 \end{bmatrix}$$

- Revisiting the Hebbian learning example with continuous bipolar activation function f(net), using input x_1 and initial weight w^1 we obtain neuron output values and updated weights for $\lambda=1$.
- The only difference compared with the previous case is that instead of f(net)=sigm(net). Now the neurons response is computed-

Step 1

$$f(net^1) = 0.905$$

$$\mathbf{w}^2 = \begin{bmatrix} 1.905 \\ -2.81 \\ 1.357 \\ 0.5 \end{bmatrix}$$

Subsequent training steps result in weight vector adjustment as below:

Step 2

$$f(net^2) = -0.077$$

$$\mathbf{w}^3 = \begin{bmatrix} 1.828 \\ -2.772 \\ 1.512 \\ 0.616 \end{bmatrix}$$

Step 3

$$f(net^3) = -0.932$$

$$\mathbf{w}^4 = \begin{bmatrix} 1.828 \\ -3.70 \\ 2.44 \\ -0.783 \end{bmatrix}$$

Comparison of learning using discrete and continuous activation functions indicates that the weight adjustments are tapered for continuous $f(net)$ but are generally in the same direction. ■

Perceptron Learning Rule

Perceptron Learning Rule

- Learning signal = ***difference*** between the ***desired and the actual neuron's response*** (ROSENBLATT-1958).
- **Supervised** learning
- Learning signal=

$$r \overset{\Delta}{=} d_i - o_i$$

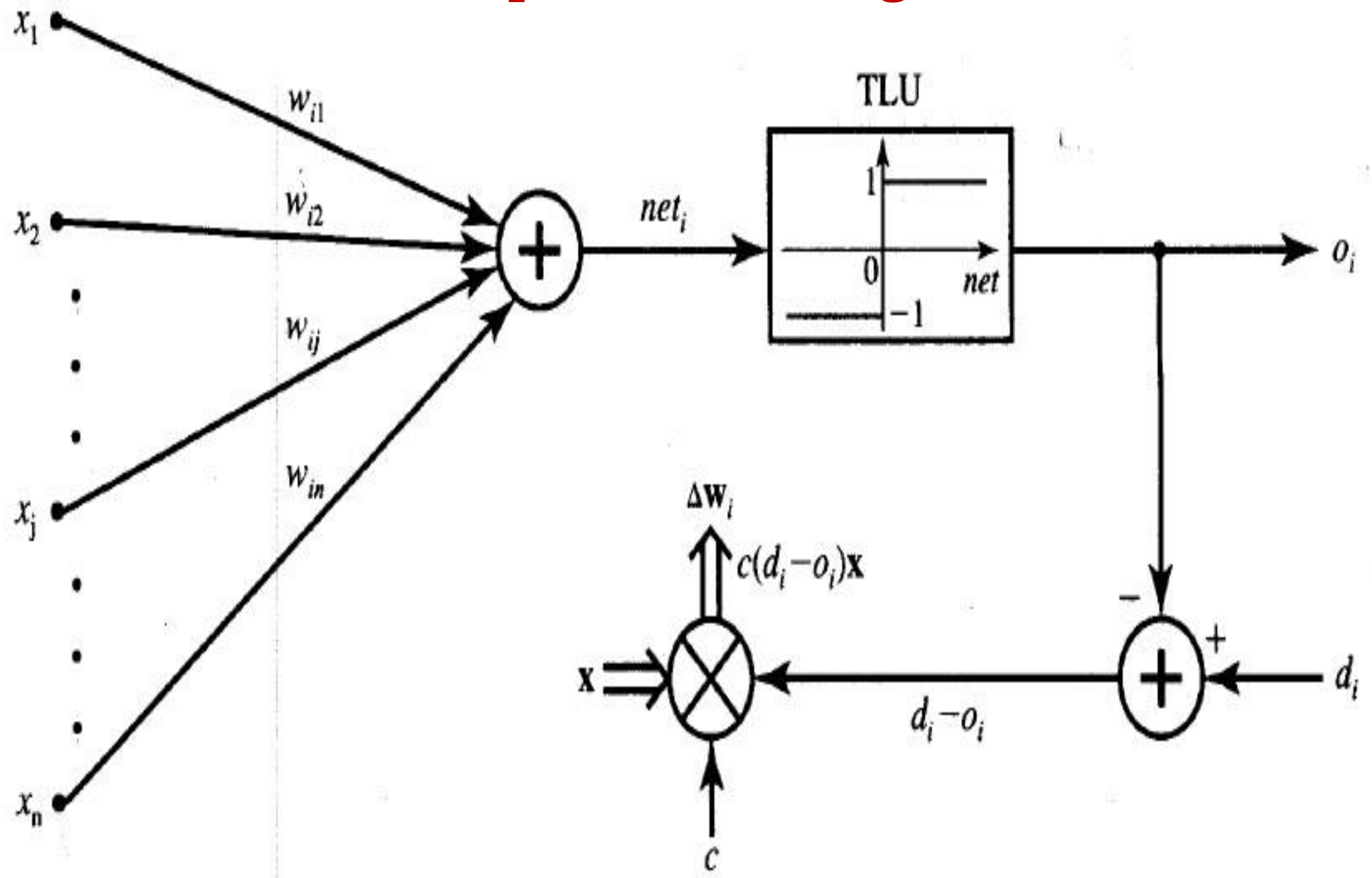
where $o_i = \operatorname{sgn}(\mathbf{w}_i^t \mathbf{x})$, and d_i is the desired response

Weight adjustments in this method, $\Delta \mathbf{w}_i$ and Δw_{ij}, are obtained as follows

$$\Delta \mathbf{w}_i = c\left[d_i - \operatorname{sgn}(\mathbf{w}_i^t \mathbf{x})\right]\mathbf{x}$$

$$\Delta w_{ij} = c\left[d_i - \operatorname{sgn}(\mathbf{w}_i^t \mathbf{x})\right]x_j, \quad \text{for } j = 1, 2, \ldots, n$$

Perceptron Learning Rule



Perceptron Learning Rule

Perceptron Learning Rule

- ***Note-1:-*** This rule is applicable only for binary neuron response and the relationship express the rule for the binary case.
- ***Note-2:-*** Under this rule weight is only adjusted if and only if o_i is incorrect.
- ***Note-3:-*** Weight can be initialized at any value since the desired response is either +1 or -1, the weight adjustment reduce to

$$\Delta \mathbf{w}_i = \pm 2c\mathbf{x}$$

- +ve sign is applicable when

$$d_i = 1, \text{ and } \operatorname{sgn}(\mathbf{w}^t\mathbf{x}) = -1,$$

- -ve sign is applicable when

$$d_i = -1, \text{ and } \operatorname{sgn}(\mathbf{w}^t\mathbf{x}) = 1.$$

Delta Learning Rule

Continuous Perceptron Learning rule

Delta Learning Rule

- Valid for the continuous activation functions i.e.

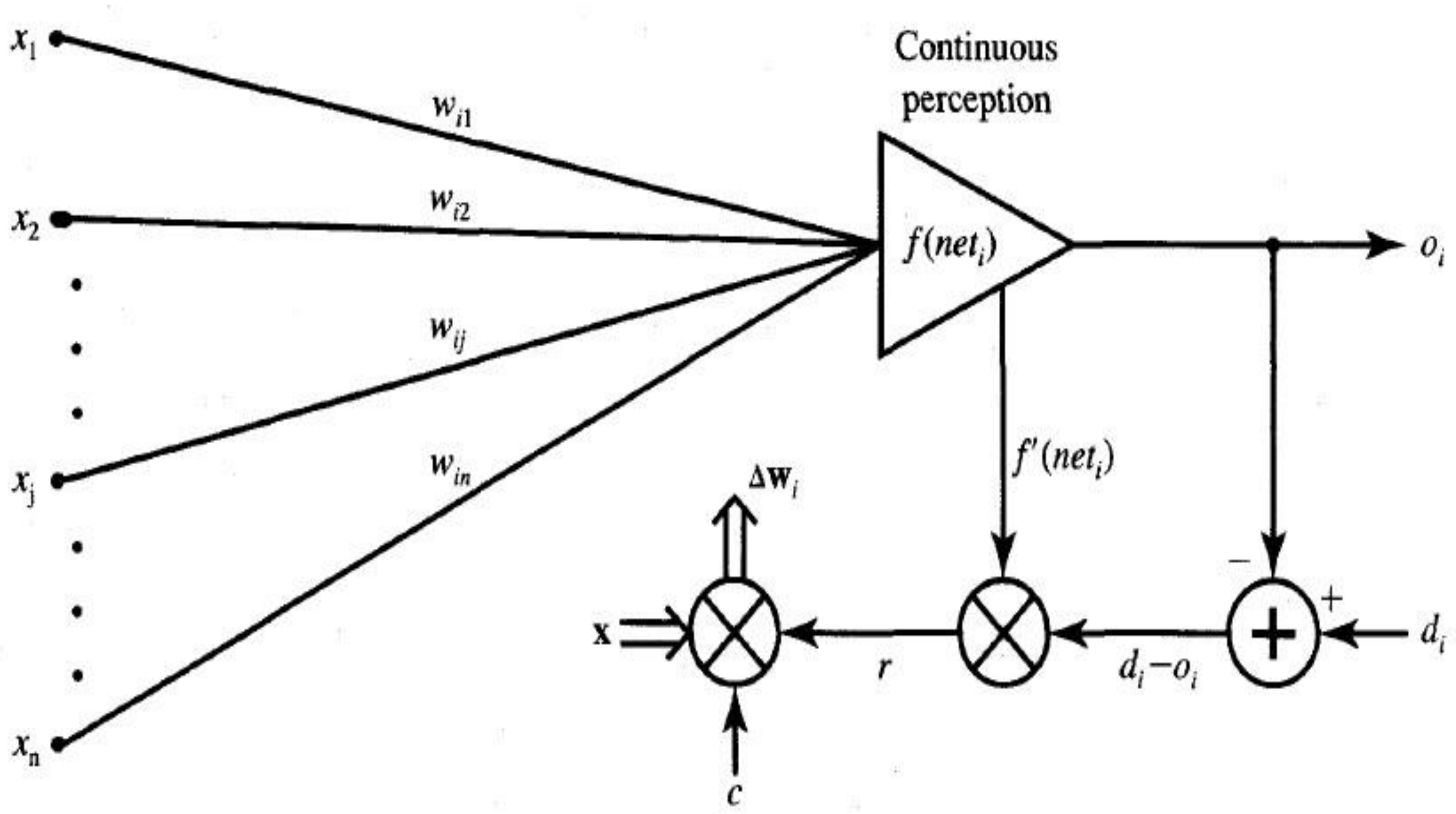
$$f(net) \triangleq \frac{2}{1+\exp(-\lambda net)} - 1 \qquad \text{...Eqn. (A)}$$

$$f(net) \triangleq \operatorname{sgn}(net) = \begin{cases} +1, & net > 0 \\ -1, & net < 0 \end{cases} \qquad \text{..Eqn. (B)}$$

- Supervised learning
- Learning signal for this rule called delta and is defined as:

$$r \triangleq [d_i - f(\mathbf{w}_i^t\mathbf{x})]f'(\mathbf{w}_i^t\mathbf{x})$$

Delta Learning Rule



Delta Learning Rule

- $f'(\mathbf{w}_i^t\mathbf{x})$ → derivative of activation function f(net) for $net = \mathbf{w}_i^t\mathbf{x}.$
- Readily derived from the condition of least-square error between o_i and d_i.
- Calculating the gradient vector w.r.t. w_i of square error defined as:

$$E \triangleq \frac{1}{2}(d_i - o_i)^2$$

which is equivalent to

$$E = \frac{1}{2}\left[d_i - f(\mathbf{w}_i^t\mathbf{x})\right]^2$$

we obtain the error gradient vector value

$$\nabla E = -(d_i - o_i)f'(\mathbf{w}_i^t\mathbf{x})\mathbf{x}$$

Delta Learning Rule

- -ve sign is present because we want to move the weight vector in the direction that decrease (E).

- The gradient rectifies the direction that produces the steepest increase in E. The –ve of this vector therefore gives the direction of steepest decrease.

The components of the gradient vector are

$$\frac{\partial E}{\partial w_{ij}} = -(d_i - o_i)f'(\mathbf{w}_i^t\mathbf{x})x_j, \quad \text{for } j = 1, 2, \ldots, n$$

Delta Learning Rule

- Minimization of error requires the weight changes to be in the gradient direction, therefore

$$\Delta \mathbf{w}_i = -\eta \nabla E$$ [Where η is a +ve constant]

- From the above equations

$$\Delta \mathbf{w}_i = \eta(d_i - o_i) f'(net_i)\mathbf{x}$$

- Or for the single weight the adjustment will be

$$\Delta w_{ij} = \eta(d_i - o_i) f'(\text{net}_i) x_j, \quad \text{for } j = 1, 2, \ldots, n$$

- Note: weight adjustment is computed based on minimization of the squared error

Delta Learning Rule

- Considering the general learning rule and plugging in the learning signed as defined in

$$\Delta \mathbf{w}_i = c(d_i - o_i)f'(\text{net}_i)\mathbf{x}$$

$$\Delta w_i = \frac{1}{2}(d_i - o_i)(1 - o_i^2)c.x$$

- Since c and η have been assumed to be arbitrary constant.
- Weights are initialized at any value for this method of training.

$$f'(net) = \frac{1}{2}(1 - o^2)$$

For Bipolar activation function

$$f'(net) = o(1 - o)$$

For Unipolar activation function

Winner-Take-All Learning Rule

Competitive Learning Rule

Winner-Take-All Learning Rule

- Winner-Take-All learning rule differs from the other rules.

- Used for unsupervised learning, rather it is used for learning statistical properties of inputs.

- Learning is based on the concept that mth neuron has the maximum response due to input x. That neuron is declared as winner.

$$W_m = [w_{m1}, w_{m2}, \ldots\ldots\ldots\ldots\ldots w_{mn}]^t$$

- In the following figure
  - 1) neurons are arranged in a layer of permits.
  - 2) adjusted weights are highlighted.

Winner-Take-All Learning Rule

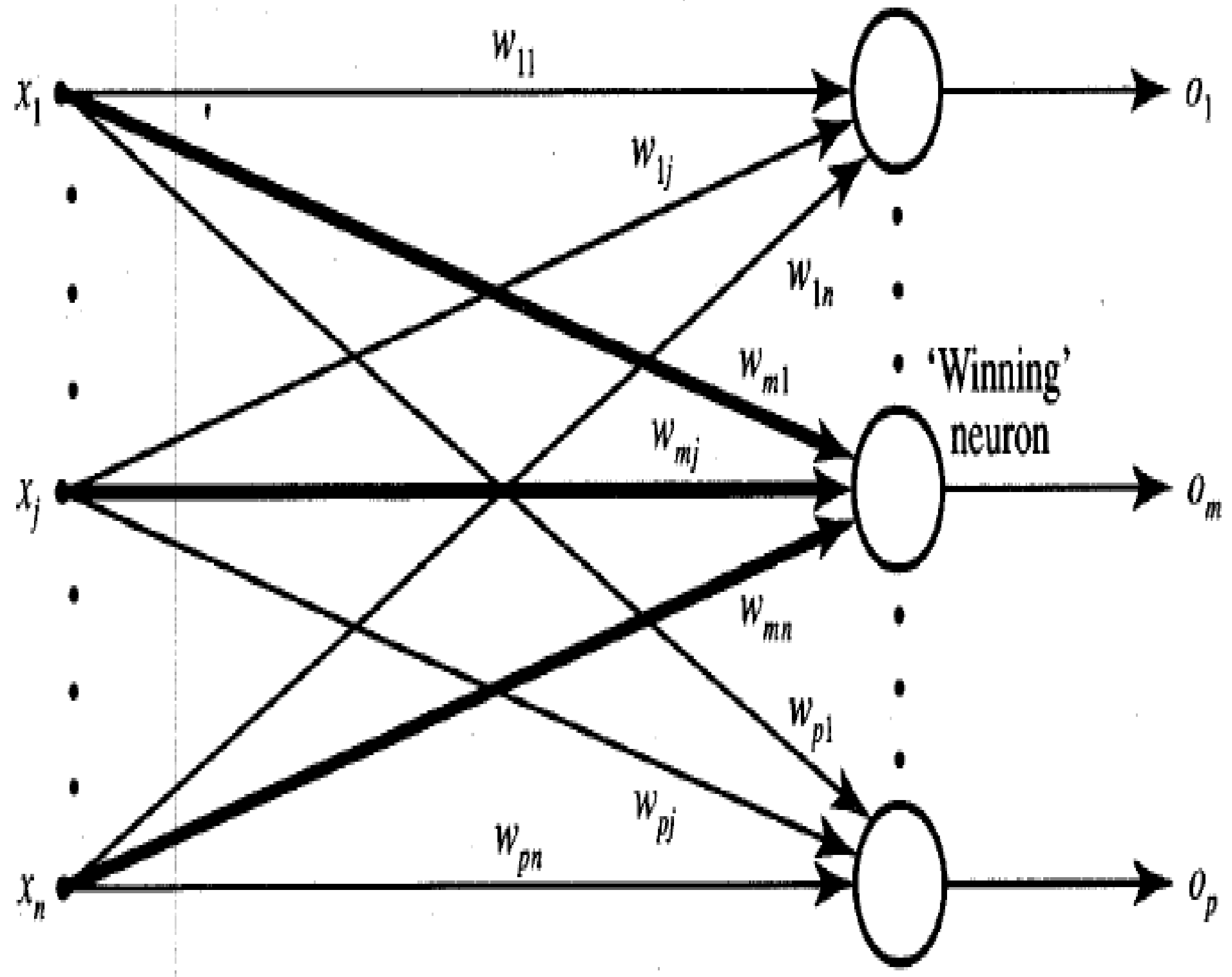


Fig: Winner Take All Learning Rule

Winner-Take-All Learning Rule

containing weights highlighted in the figure is the only one adjusted in the given unsupervised learning step. Its increment is computed as follows

$$\Delta \mathbf{w}_m = \alpha(\mathbf{x} - \mathbf{w}_m)$$

or, the individual weight adjustment becomes

$$\Delta w_{mj} = \alpha(x_j - w_{mj}), \quad \text{for } j = 1, 2, \ldots, n$$

where $\alpha > 0$ is a small learning constant, typically decreasing as learning progresses. The winner selection is based on the following criterion of maximum activation among all p neurons participating in a competition: [illegible]

$$\mathbf{w}_m^t \mathbf{x} = \max_{i=1,2,\ldots,p} (\mathbf{w}_i^t \mathbf{x})$$

Outstar Learning Rule

Outstar Learning Rule

- Works for layer of neurons.
- Supervised learning.
- Allows the network to extract statistical properties of the input and output signal.
- The adjustment weight computed as follows:

$$\Delta \mathbf{w}_j = \beta(\mathbf{d} - \mathbf{w}_j)$$

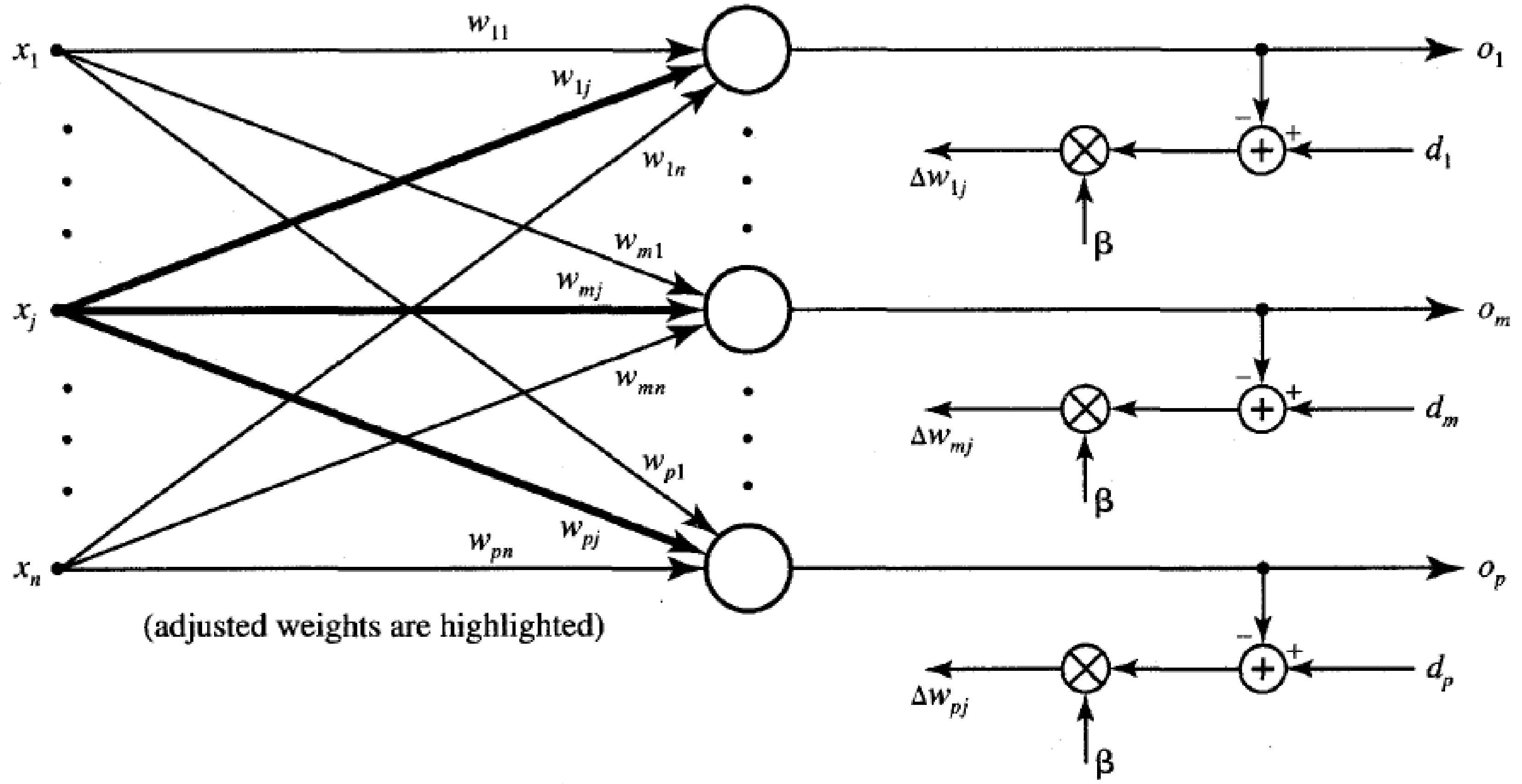
or, the individual weight adjustments are

$$\Delta w_{mj} = \beta(d_m - w_{mj}), \quad \text{for } m = 1, 2, \ldots, p$$

Outstar Learning Rule

Note that in contrast to any learning rule discussed so far, the adjusted weights are fanning out of the j'th node in this learning method and the weight vector is defined accordingly as

$$\mathbf{w}_j \overset{\Delta}{=} \begin{bmatrix} w_{1j} & w_{2j} & \cdots & w_{pj} \end{bmatrix}^t$$



(adjusted weights are highlighted)

LMS or Widrow-Hoff

LMS or Widrow-Hoff

- Least Mean Square error (LMS or Widrow-Hoff)
- Supervised training
- Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration
- An approximate steepest descent algorithm, in which the performance index is mean square error.
- Widely used today in many signal processing applications.
- Precursor to the back propagation algorithm for multilayer networks.

LMS or Widrow-Hoff

- Special case of Delta Learning rule

$$net_i = w_i^t x$$

$$\Delta w_i = C\left(d_i - o_i\right) f' net_i \; x$$

$$net_i = 1$$

$$\Delta w_i = C\left(d_i - o_i\right) x$$

$$\Delta w_i = C\left(d_i - w_i^t x\right) x$$

LMS or Widrow-Hoff

Mean Square Error

- Network output compared to the target.
- **The error** is calculated as the difference between the target output and the network output.
- Try to minimize the average of the sum of these errors.

$$mse = \frac{1}{Q}\sum_{k=1}^{Q} e(k)^2 = \frac{1}{Q}\sum_{k=1}^{Q} (t(k) - a(k))^2$$

- The **LMS** algorithm adjusts the weights and biases of the linear network so as to **minimize this mean square error.**

Summary of learning rules and their properties.

Learning rule	Single weight adjustment Δw_{ij}	Initial weights	Learning	Neuron characteristics	Neuron / Layer
Hebbian	$co_i x_j$ $j = 1, 2, \ldots, n$	0	U	Any	Neuron
Perceptron	$c\left[d_i - \operatorname{sgn}(\mathbf{w}_i^t\mathbf{x})\right] x_j$ $j = 1, 2, \ldots, n$	Any	S	Binary bipolar, or Binary unipolar*	Neuron
Delta	$c(d_i - o_i)f'(net_i)x_j$ $j = 1, 2, \ldots, n$	Any	S	Continuous	Neuron
Widrow-Hoff	$c(d_i - \mathbf{w}_i^t\mathbf{x})x_j$ $j = 1, 2, \ldots, n$	Any	S	Any	Neuron
Correlation	$cd_i x_j$ $j = 1, 2, \ldots, n$	0	S	Any	Neuron
Winner-take-all	$\Delta w_{mj} = \alpha(x_j - w_{mj})$ m-winning neuron number $j = 1, 2, \ldots, n$	Random Normalized	U	Continuous	Layer of p neurons
Outstar	$\beta(d_i - w_{ij})$ $i = 1, 2, \ldots, p$	0	S	Continuous	Layer of p neurons

c, α, β are positive learning constants
S — supervised learning, U — unsupervised learning
* — Δw_{ij} not shown