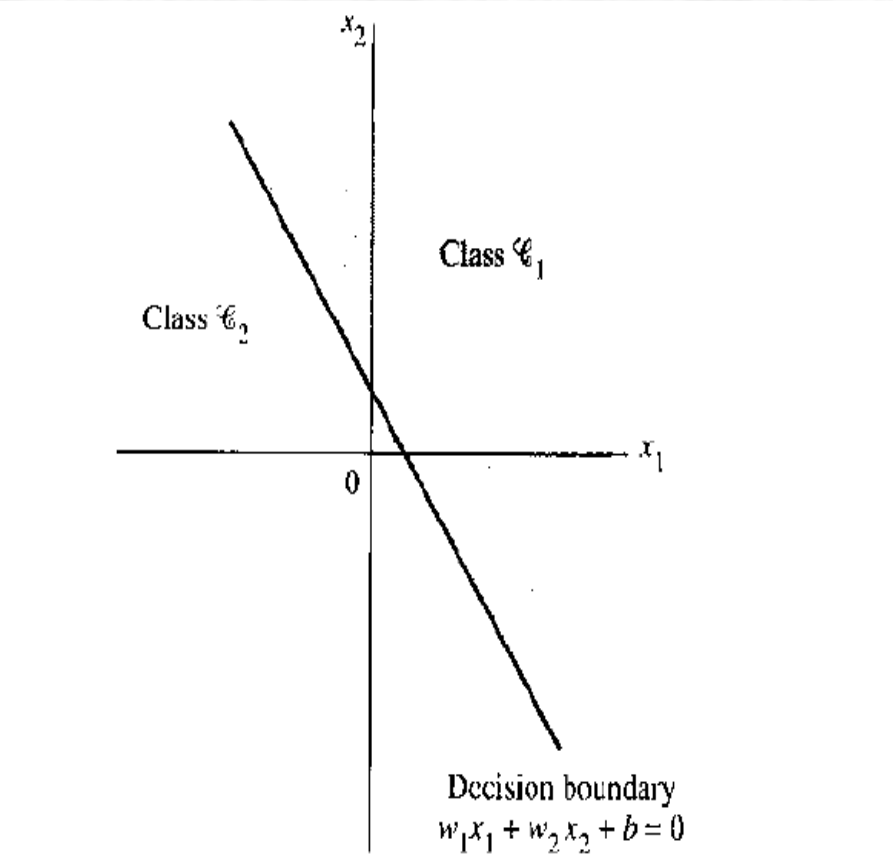
IV. SINGLE LAYER STRUCTURES

4.1. Perceptrons

- The perceptron is the simplest form of a neural network used for the classification of a special type of patterns said to be **linearly separable**,
- It was introduced by Rosenblatt (1958),
- Activation function in the form of a strict threshold function
- It may form a classifier for input pattern belonging into one of two pattern as classes

Single Neuron Perceptron



- Works well with linearly separable classes
- Linearly separable classes are: classes that can be separated by a hyperplane

  A straight line is a hyperplane in 2 D space
- w^TX-b $>0 \Rightarrow X \in C1$ *(class1)*
- w^TX-b $<0 \Rightarrow X \in C2$ *(class2)*
- w^TX-b $=0 \Rightarrow$ *X on the line*

Decision Hyperplanes and Linear Separability

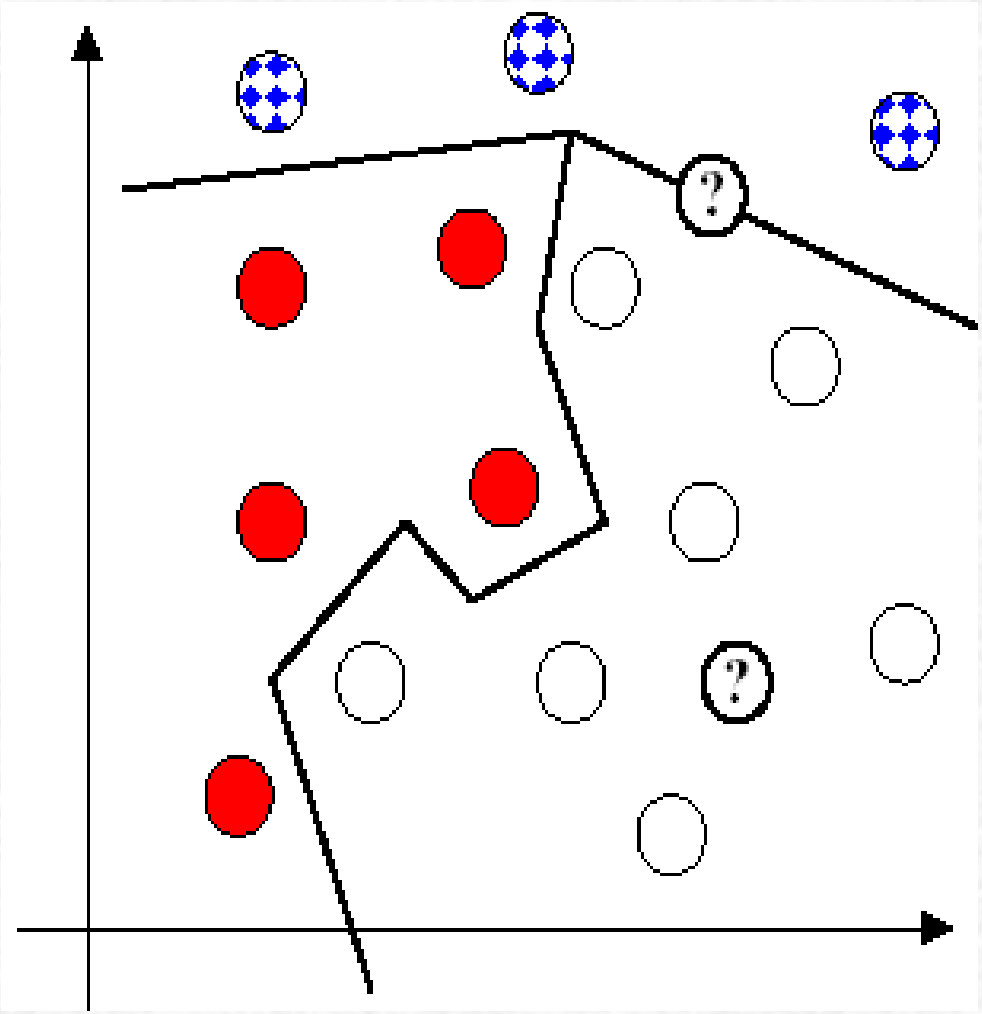
- If we have two inputs, then the weights define a decision boundary that is a one dimensional straight line in the two dimensional ***input space*** of possible input values.
- If we have *n* inputs, the weights define a decision boundary that is an *n-1* dimensional ***hyperplane*** in the *n* dimensional input space:

  $w_1X_1 + w_2X_2 + \ldots + w_nX_n - b = 0$

  The perceptron classifies input vectors according to which side of the hyperplane they lie based on the priori information

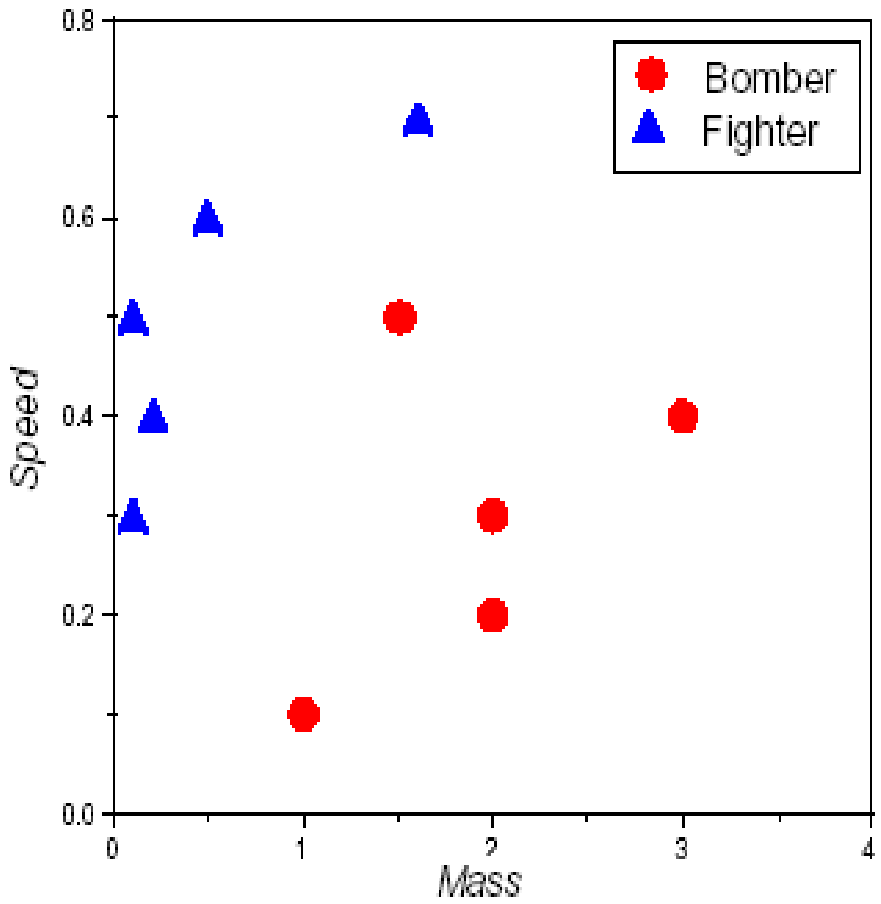
- This hyperplane is clearly still linear (i.e. straight/flat) and can still only divide the space into two regions. We still need more complex transfer functions, or more complex networks, to deal with XOR type problems.
- Problems with input patterns which can be classified using a single hyperplane are said to be ***linearly separable***. Problems (such as XOR) which cannot be classified in this way are said to be ***non-linearly separable***.
- Regardless of the form of nonlinearity used, a single layer perceptron can perform pattern classification only on linearly separable patterns (i.e. sufficiently separated from each other)

General Decision Boundaries



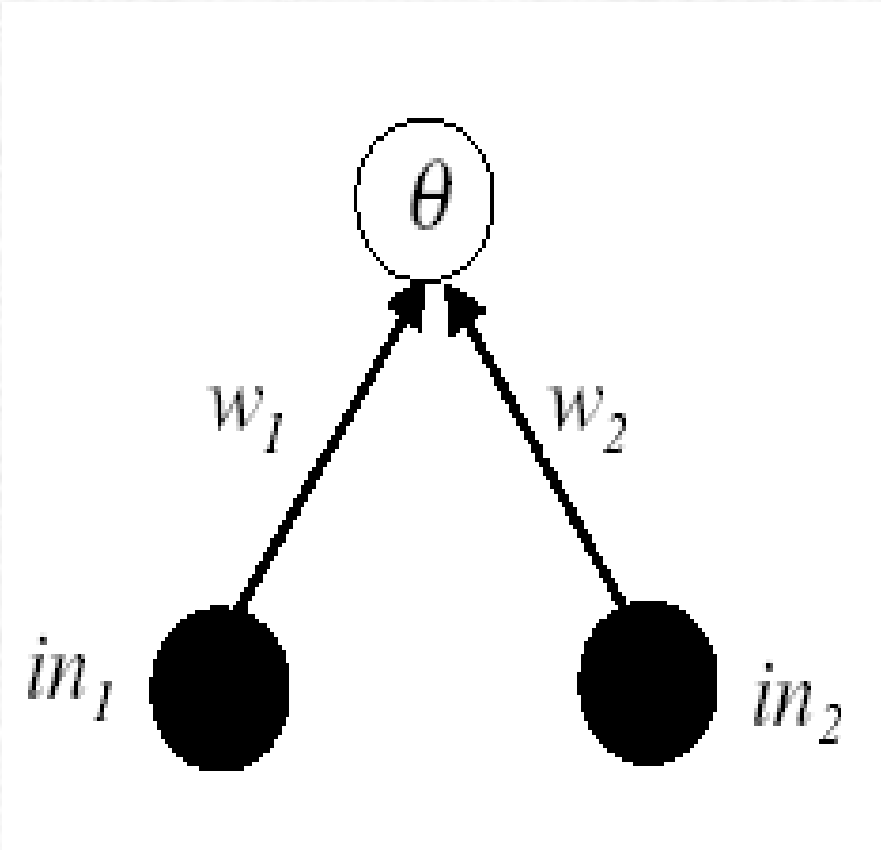
- Generally we will be interested in non-binary input patterns, and we will expect our neural networks to form complex decision boundaries,
- We may wish to classify many types of input (such as the three shown here).
- Are they linearly separable?

Perceptrons for Decision



- How do we know if a simple Perceptron is powerful enough to solve a given problem?
- If it can't even do XOR, why should we expect it to be able to deal with our aeroplane classification example, or even more complex tasks?

Decision Boundaries in Two Dimensions



- It is forming decision boundaries between classes. Remember, the network output is:
- *out* =sgn(*w*1*x*1+*w*2*x*2-q)
- The decision boundary (between *out*=0 and *out*= 1) is at *w*1*x*1+*w*2*x*2-b *=0*
- i.e. along the straight line:

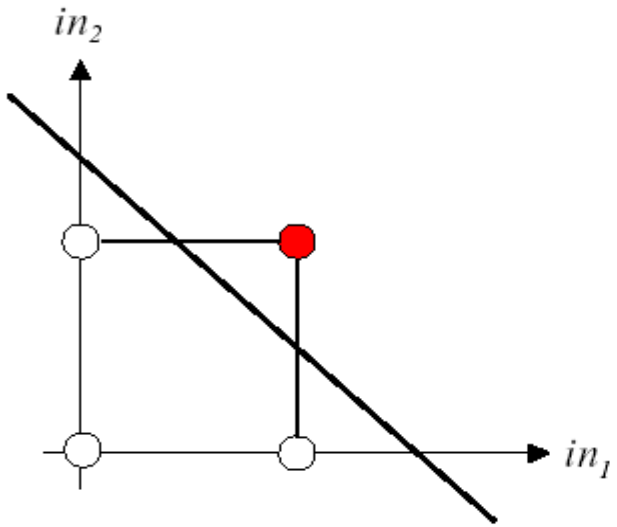
  x2=-(w1/w2)x1+(b/w2)
- So, in two dimensions the decision boundaries are always straight lines.

Decision Boundaries for AND and OR

We can easily plot the decision boundaries we found by inspection last lecture:

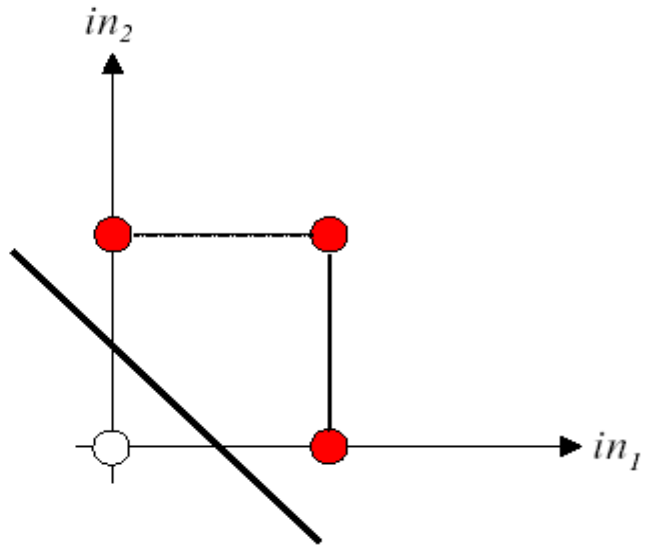
AND

$w_1 = 1,\ w_2 = 1,\ \theta = 1.5$



OR

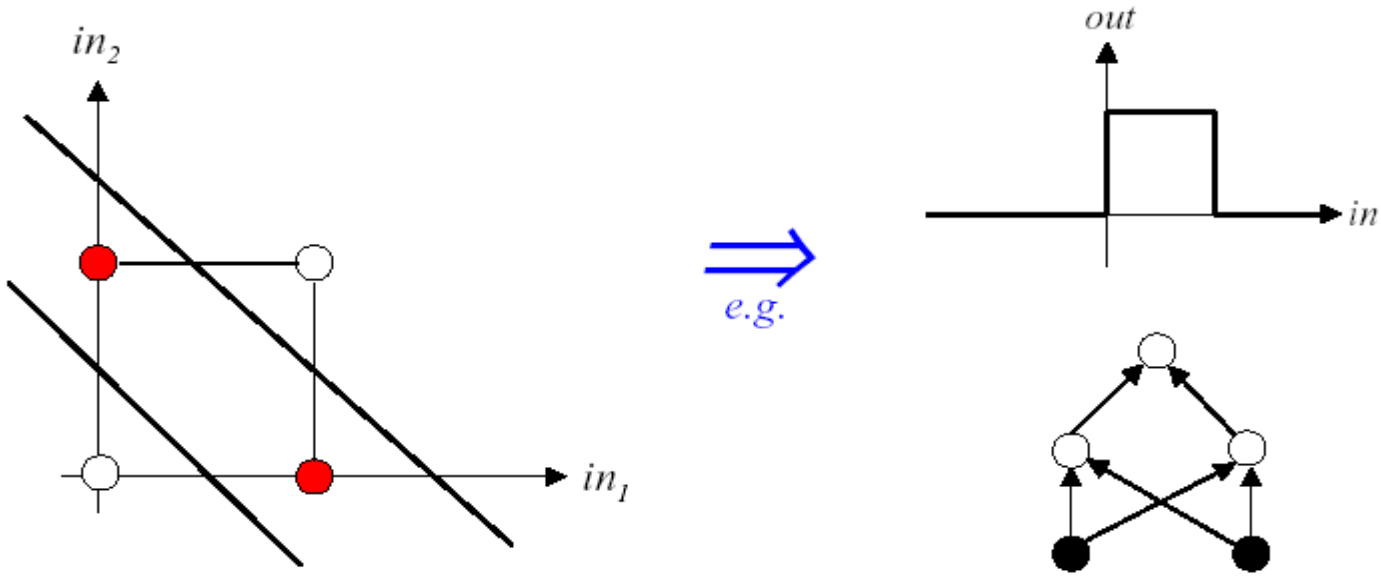
$w_1 = 1,\ w_2 = 1,\ \theta = 0.5$



The extent to which we can change the weights and thresholds without changing the output decisions is now clear.

Decision Boundaries for XOR

The difficulty in dealing with XOR is beginning to look obvious. We need two straight lines to separate the different outputs/decisions:



There are two obvious remedies: either change the transfer function so that it has more than one decision boundary, or use a more complex network that is able to generate more complex decision boundaries.

Learning and Generalization

- There are two important aspects of the network's operation to consider:

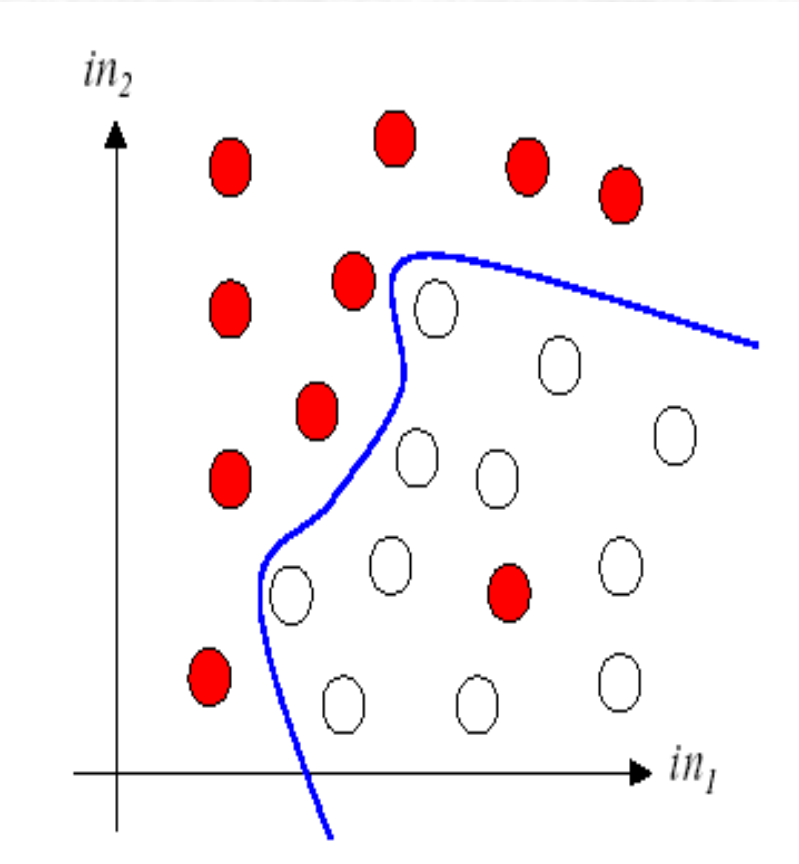
  Learning The network must learn decision surfaces from a set of ***training patterns*** so that these training patterns are classified correctly.

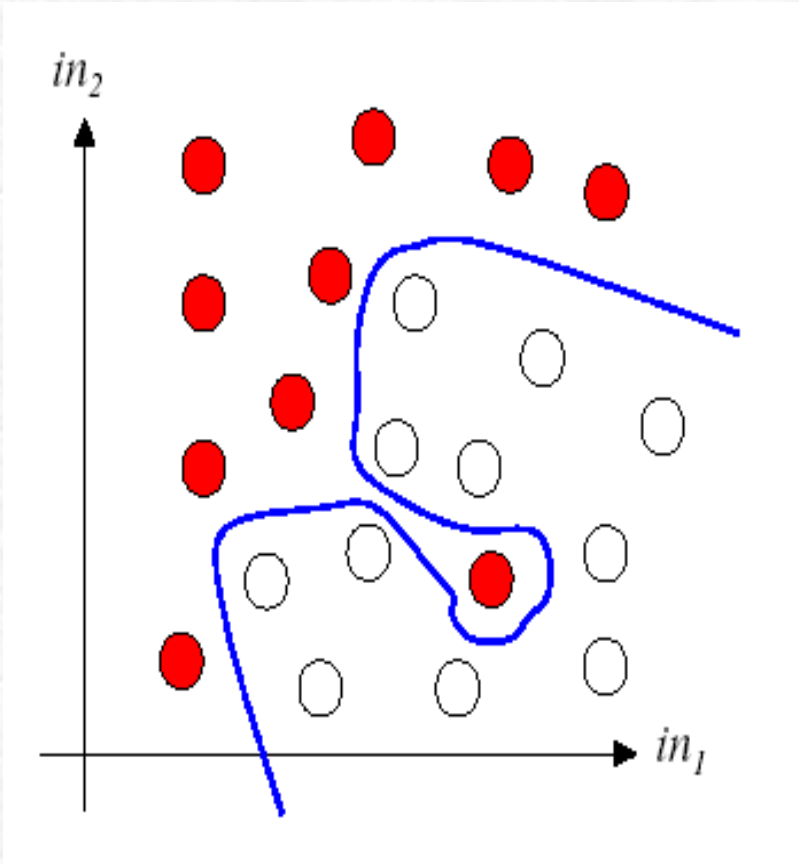
  Generalization After training, the network must also be able to generalize, i.e. correctly classify ***test patterns*** it has never seen before.

- Often we wish the network to learn perfectly, and then generalize well. The training data may contain errors (e.g. noise in the experimental determination of the input values, or incorrect classifications). In this case, learning the training data perfectly may make the generalisation worse. There is an important ***tradeoff*** between learning and generalization that arises quite generally.

Generalization in Classification

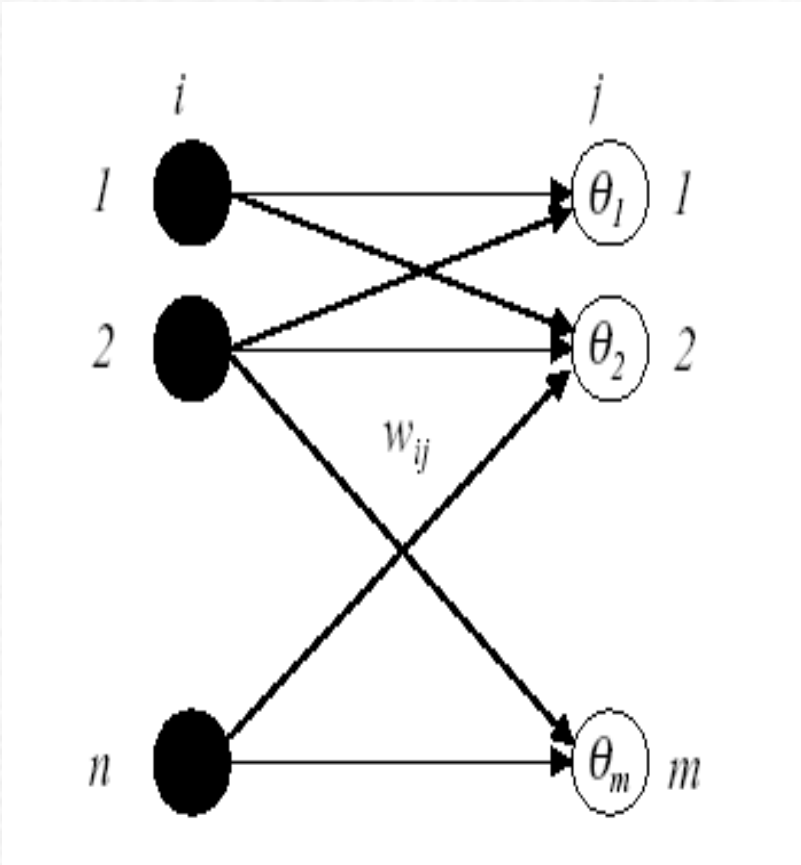
Suppose the task of our network is to learn a decision boundary:





Multi Neuron Perceptrons

A hyperplane is associated with a neuron and a network simultaneously classifies an input vector relative to all of the hyperplanes



- $\mathbf{y}=[y_1 \quad y_m]^T$
- $\mathbf{b}=[b1 \quad bm]^T$
- $\mathbf{x}=[x1 \quad xn]$
- $\mathbf{W} \in R^{mxn}$
- $\mathbf{y}=\text{sgn}(\mathbf{W}\mathbf{x}^T-\mathbf{b})$

  The i th raw of the weight matrix defines the vector which is normal to the ith hyperplane

Perceptron Learning

- For simple Perceptrons performing classification, we have seen that the decision boundaries are hyperplanes, and we can think of ***learning*** as the process of shifting around the hyperplanes until each training pattern is classified correctly.
- Somehow, we need to formalize that process of "shifting around" into a systematic algorithm that can easily be implemented on a computer.
- The "shifting around" can conveniently be split up into a number of small steps.

Error Correction Rules- Algorithm

If the network weights at time n are $wij(n)$, then the shifting process corresponds to moving them by an amount $\Delta\, wij(t)$ so that at time $n+1$ we have weights $wij\,(n+1)= wij(n)+ \Delta w\, ij$ (n)

- Initialization: Set w(0)=0
- Activation and Computing Actual Response: Apply x(n) and t(n) (target or desired response)

  $\mathbf{y}=\mathrm{sgn}(\mathbf{W}\mathbf{x}^{T}-\mathbf{b}) \in \{-1,+1\}$
- Adaptation: Update the weight vector

  $\Delta w\,(n)= \eta[t(n)-y(n)]x(n)$ where η is the

  $\Delta b\,(n)= \eta[t(n)-y(n)]$ learning rate $(0<\eta<1)$
- Increment n by one, go back to step 2

Convergence of Perceptron Learning

- The weight changes Δwij need to be applied repeatedly – for each weight *wij* in the network and for each training pattern in the training set. One pass through all the weights for the whole training set is called one ***epoch*** of training.
- Eventually, when all the network outputs match the targets for all the training patterns, all the Δwij will be zero and the process of training will cease. We then say that the training process has ***converged*** to a solution.

Batch mode Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\vec{w}]$
2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}]$

Incremental mode Gradient Descent:
Do until satisfied

- For each training example d in D
  1. Compute the gradient $\nabla E_d[\vec{w}]$
  2. $\vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}]$

$$E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$E_d[\vec{w}] \equiv \frac{1}{2}(t_d - o_d)^2$$

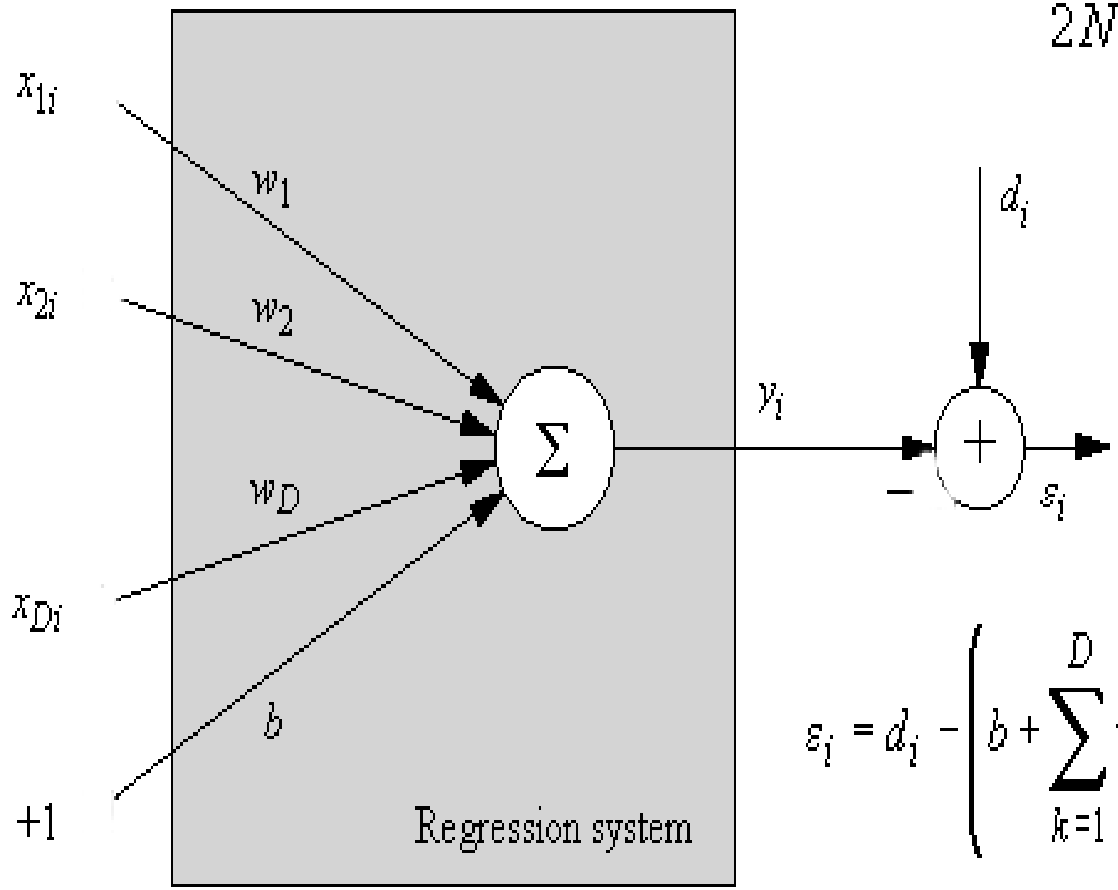
Incremental Gradient Descent can approximate *Batch Gradient Descent* arbitrarily closely if η made small enough

4.2. Adaptive Linear Combiner (ADALINE)

- Its output is a linear combination of its inputs so hard limit is replaced by a linear element (ADALINE)
- Adaptation algorithm (LMS-Least Mean Square- Algorithm or often called the Widrow Hoff Delta rule) automatically adjusts the weights
- Adaline is the basic building block in many neural networks together with a cascaded hard limited quantizer to produce ±1 output

  yk=sgn(sk)

$$J = \frac{1}{2N}\sum_{i}\left(d_i - \sum_{k=0}^{D} w_k x_{ik}\right)^2$$



$$\varepsilon_i = d_i - \left(b + \sum_{k=1}^{D} w_k x_{ik}\right) = d_i - \sum_{k=0}^{D} w_k x_{ik} \qquad i = 1...N$$

Wiener Solution

- The mean square error (MSE) Averaging over Ensemble yields

$$J = \frac{1}{2N}\sum_i \left(d_i - \sum_{k=0}^{D} w_k x_{ik} \right)^2$$

$$\frac{\partial J}{\partial w_j} = -\frac{1}{N}\sum_i x_{ij}\left(d_i - \sum_{k=0}^{D} w_k x_{ik} \right) = 0 \qquad \text{for } j = 0 \ldots D$$

$$\sum_i x_{ij} d_i = \sum_{k=0}^{D} w_k \sum_i x_{ik} x_{ij} \qquad j = 0, 1, \ldots D$$

$$P_j = \frac{1}{N}\sum_i x_{ij} d_i$$

$$R_{kj} = \frac{1}{N}\sum_i x_{ik} x_{ij}$$

$$\mathbf{p} = \mathbf{R}\mathbf{w}^* \quad \text{or} \quad \mathbf{w}^* = \mathbf{R}^{-1}\mathbf{p}$$

The LMS Algorithm

- LMS algorithm is based on the use of instantaneous values for the cost function

$$\nabla \mathbf{J} = \left[\frac{\partial J}{\partial w_0}, \ldots, \frac{\partial J}{\partial w_D} \right]^T$$

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla \mathbf{J}(k)$$

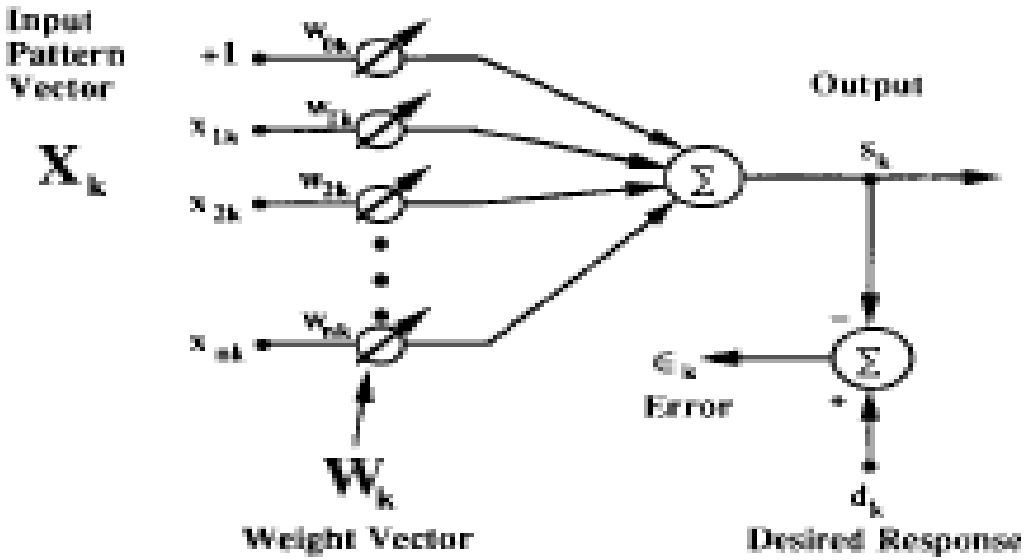
$$\nabla J(k) = \frac{\partial}{\partial w(k)} J = \frac{\partial}{\partial w(k)} \frac{1}{2N} \sum_i \varepsilon_i^2 \approx \frac{1}{2} \frac{\partial}{\partial w(k)} \left[\varepsilon^2(k) \right] = -\varepsilon(k) x(k)$$

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta\, \varepsilon(k) \mathbf{x}(k)$$

The LMS Algorithm offers some important features

- Simplicity of implementation
- Ability to operate in an unknown environment
- Ability to track the variations of input statistics
- The fundamental constraint of adapting the weights using LMS is a single step size μ. The speed of adaptation is controlled by the smallest eigenvalue, while the largest step size is constrained by the inverse of the largest eigenvalue. $0 < \mu_{max} < 2/\lambda_{max}$
- That is, the step size μ must be smaller than the inverse of the largest eigenvalue of the autocorrelation matrix,R.

Adaptive Linear Combiner



Adaptive linear combiner.

Adaline

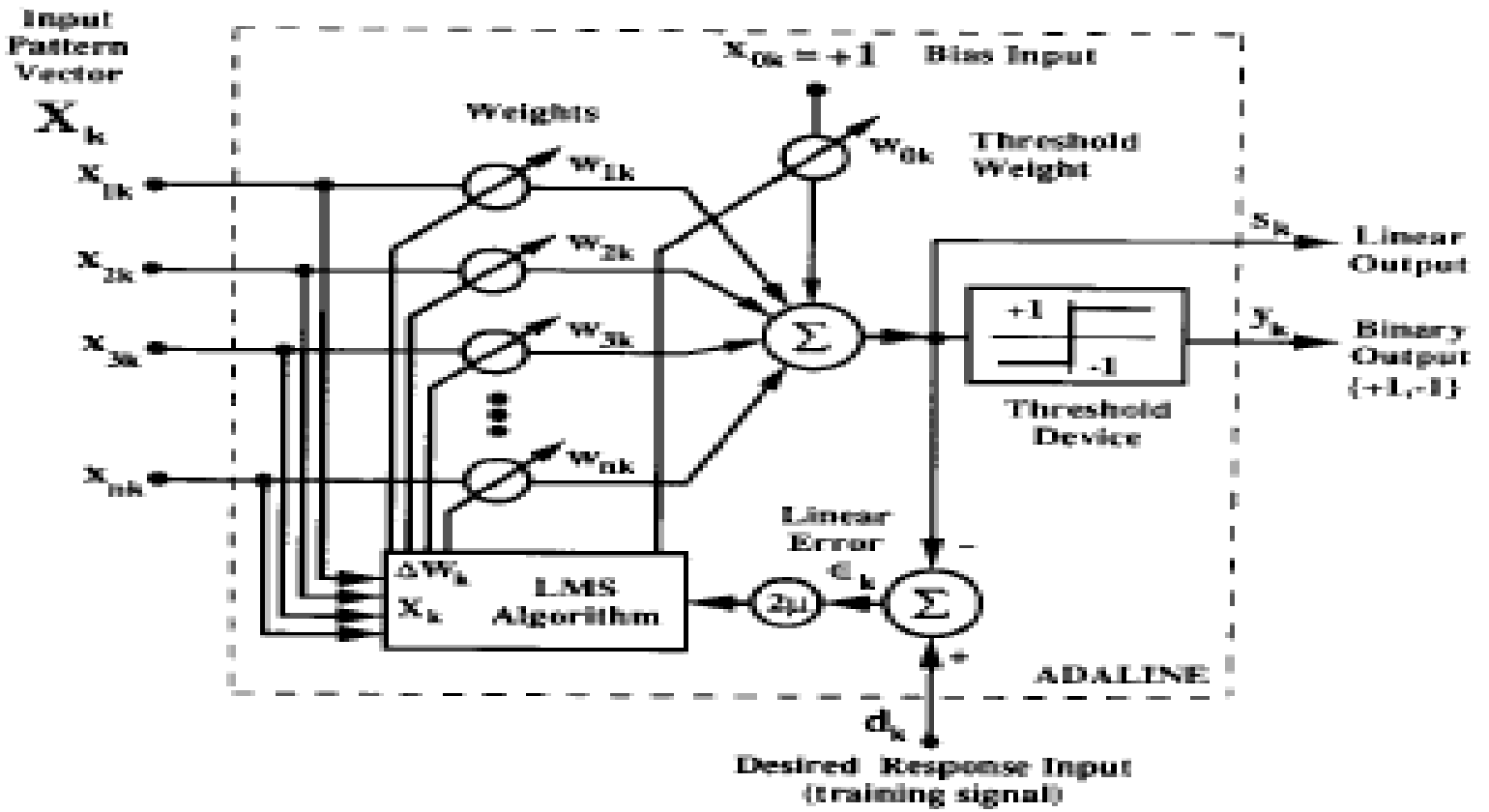
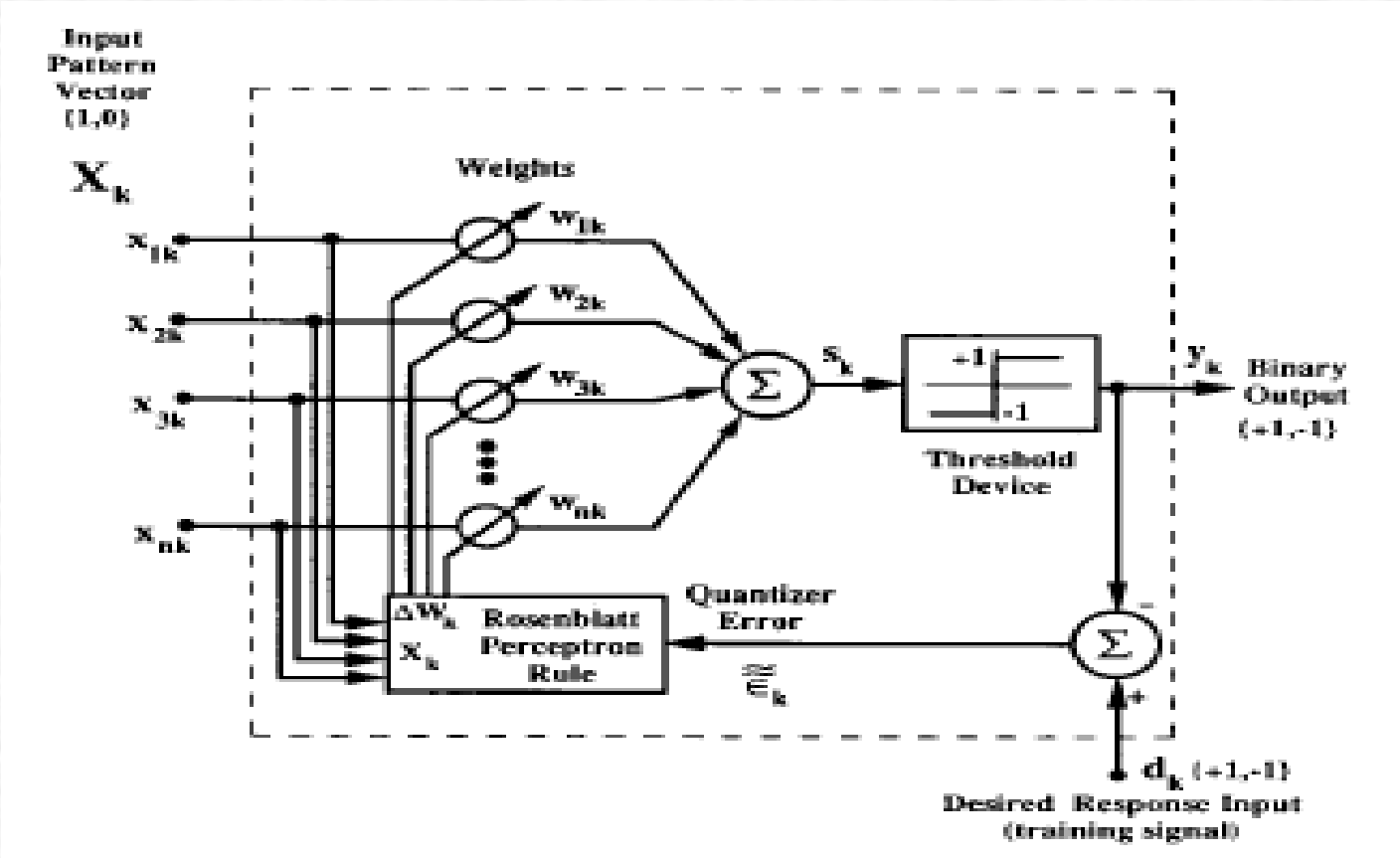
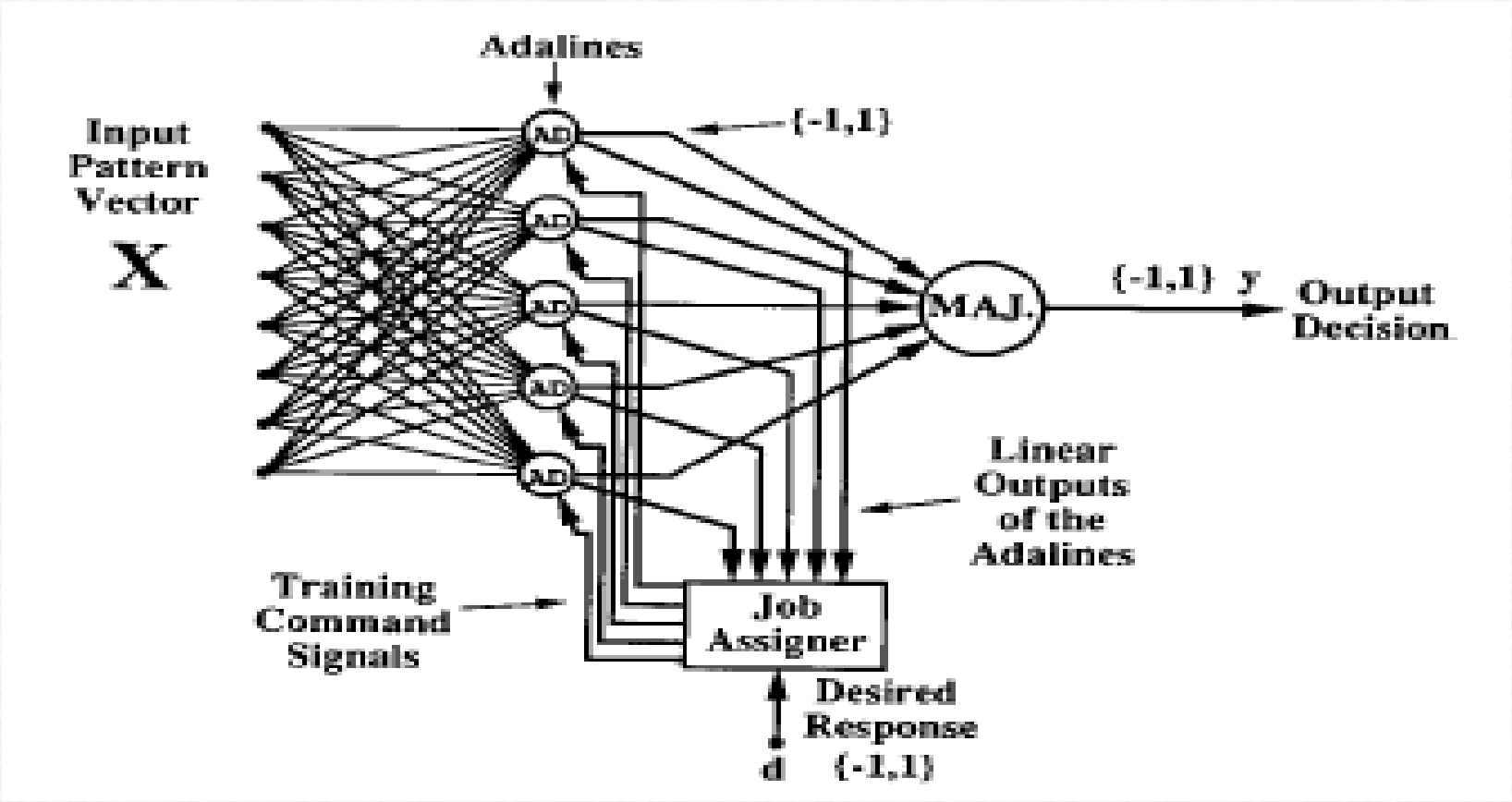


Fig. 2. Adaptive linear element (Adaline).

Adaptive Threshold Element of the Perceptron



A Five-Adaline Example of the Madaline I Architecture



Two Layer Madaline II Atchitecture