

Neural Networks

Outline:

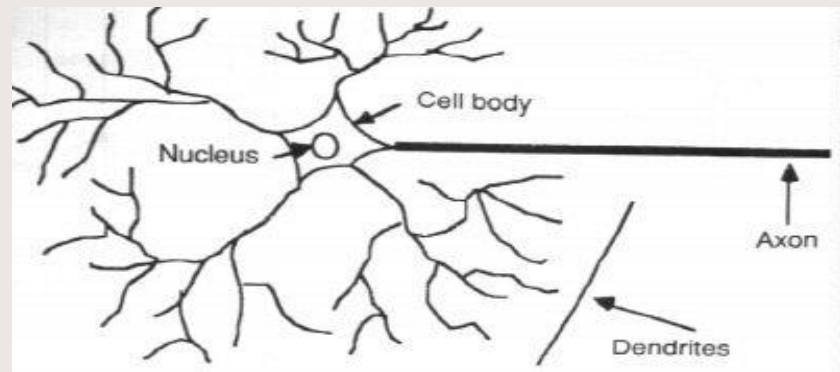
1. Introduction to Neural Networks
2. Perceptrons
3. Introduction to Backpropagation
4. Associative Memory: Hopfield Nets
5. Summary

Introduction

- A **neural network** can be defined as a model of reasoning based on the human brain. The brain consists of a densely interconnected set of nerve cells, or basic information-processing units, called neurons.

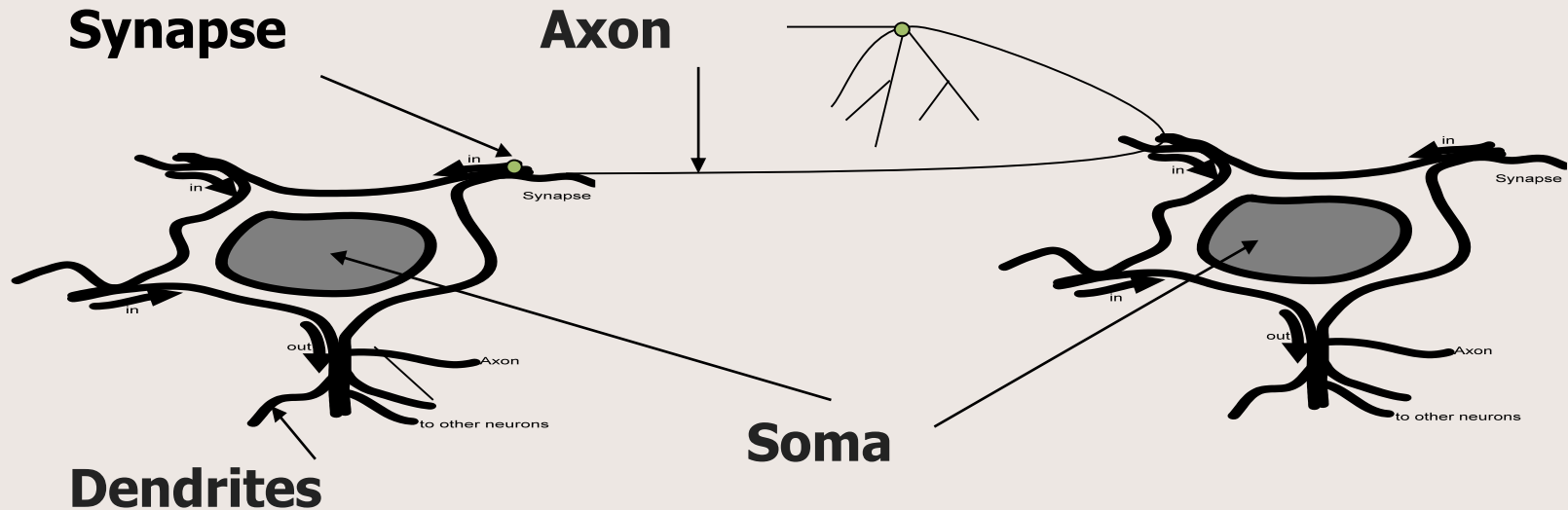
Neuron

- The cell that perform information processing in the brain.



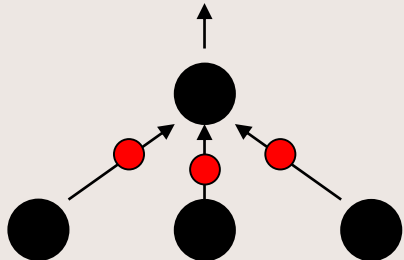
- Neural Networks can be :
 - Biological models
 - Artificial models
- Desire to produce artificial systems capable of sophisticated computations **similar to the human brain.**

Biological Neural Network



- A neuron consists of a cell body, **soma**, a number of fibers called **dendrites**, and a single long fiber called the **axon**. While dendrites branch into a network around the soma, the axon stretches out to the dendrites and somas of other neurons. The connections between neurons are realized in the synapses. Signals are propagated from one neuron to another by complex electrochemical reactions.

How the brain works

- Each neuron receives inputs from other neurons
 - Some neurons also connect to receptors
 - Cortical neurons use spikes to communicate
 - The timing of spikes is important
 - The effect of each input line on the neuron is controlled by a synaptic weight
 - The weights can be positive or negative
- 
- The synaptic weights **adapt** so that the whole network learns to perform useful computations
 - Recognizing objects, understanding language, making plans, controlling the body
 - Human brain has about 10^{11} (10 billions) neurons. Each connected with other through (about) 10^4 weights (interconnections).
 - A huge number of weights can affect the computation in a very short time. Much better bandwidth than Pentium Processor.

Human Brain

- Properties of the brain
 - It can learn, reorganize itself from experience
 - It adapts to the environment
 - It is robust and fault tolerant

Human Brain vs. Computer

- Computers require hundreds of cycles to simulate a firing of a neuron.
- The brain can fire all the neurons in a single step.
Parallelism
- Serial computers require billions of cycles to perform some tasks but the brain takes less than a second.
e.g. Face Recognition

Future Goal : Combine parallelism of the brain with the switching speed of the computer.

Analogy between Biological and Artificial Neural Networks

Biological NN	Artificial NN
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

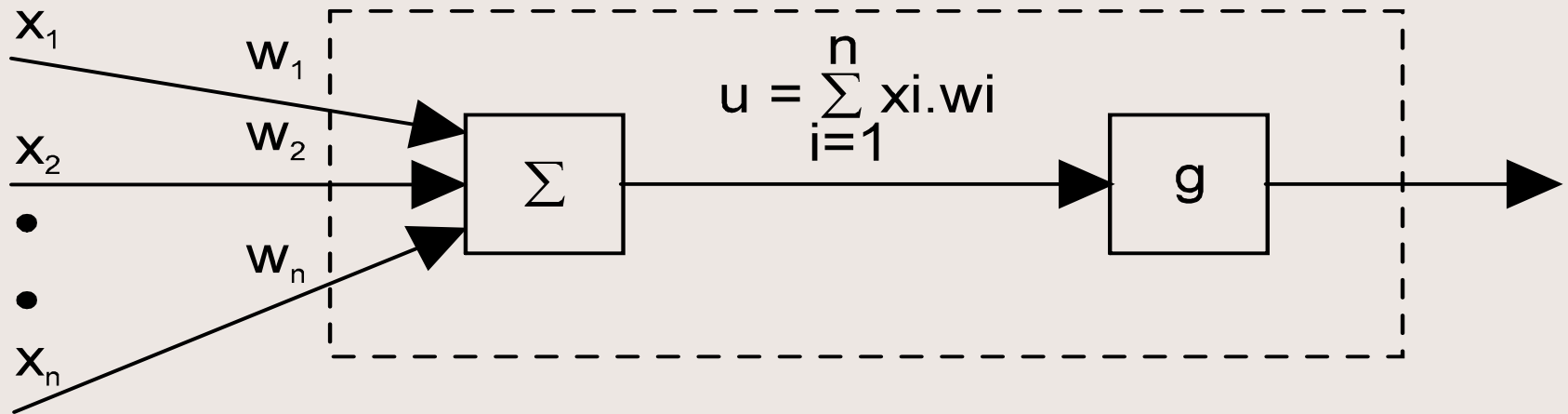
Artificial Neural Networks (ANN)

- A mathematical model to solve engineering problems
 - Group of highly connected neurons to realize compositions of non linear functions
- Tasks
 - Classification
 - Discrimination
 - Estimation
- Types of networks
 - Feed forward Neural Networks
 - Recurrent Neural Networks

An ANN system is characterized by

- its ability to learn;
- its dynamic capability;
- its interconnectivity.

Artificial Neuron and Rosenblatt's Single Layer Perceptron



A model of an artificial neuron.

Perceptron is the simplest form of a neural network. It consists of a single neuron with adjustable synaptic weights and a hard limiter activation function (threshold).

Perceptron's Learning Algorithm

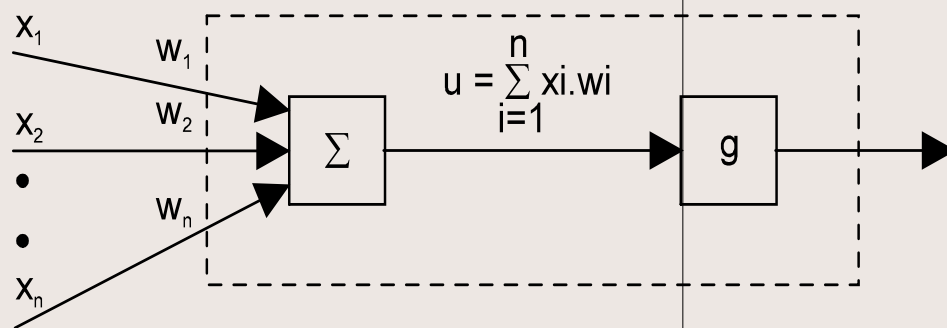
MSE (Minimum Squared Error) by Steepest Descent Minimization Procedure:

$$E = \frac{1}{2} (D - d)^2$$

$$D = \sum_{i=0}^n w_i x_i$$

d = desired output for a sample

$$\Rightarrow \frac{\partial E}{\partial w_i} = (D - d) x_i$$



Step 1: Pick starting weight $w_1, w_1, w_1, \dots, w_n$ and choose a positive constant α (learning rate).

Step 2: Present samples 1 to N repeatedly to the classifier. For each pattern of input sample, compute D .

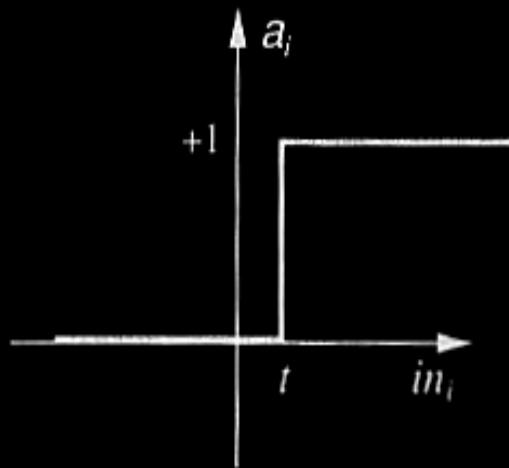
Step3: Update the weight w_i by $w_i - \alpha \frac{\partial E}{\partial w_i}$ for all i

Step4: Repeat steps2 and 3 until the weights cease to change significantly.

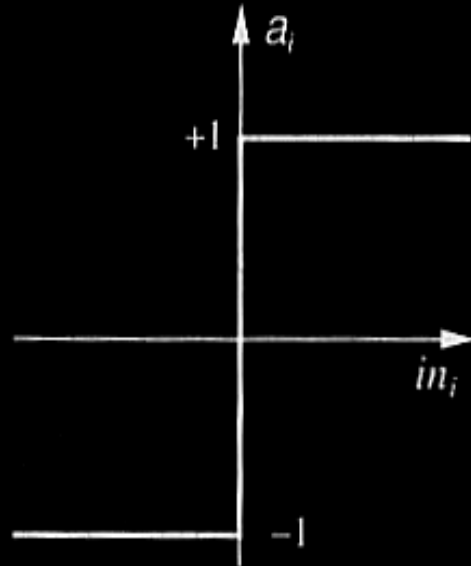
Activation Function

- Use different functions to obtain different models.
- 3 most common choices :
 - 1) Step function
 - 2) Sign function
 - 3) Sigmoid function
- An output of 1 represents firing of a neuron down the axon.

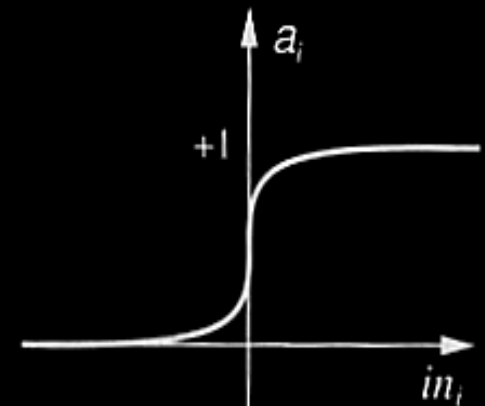
Activation Function



(a) Step function



(b) Sign function



(c) Sigmoid function

Figure 19.5 Three different activation functions for units.

$$\text{step}_t(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases} \quad \text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \quad \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Learning

- The procedure that consists in estimating the parameters of neurons so that the whole network can perform a specific task
- Types of learning
 - The supervised learning
 - The unsupervised learning
- The Learning process (supervised)
 - Present the network a number of inputs and their corresponding outputs
 - See how closely the actual outputs match the desired ones
 - Modify the parameters to better approximate the desired outputs.

Learning

A neuron learns because it is adaptive.

- **SUPERVISED LEARNING:** The connection strengths of a neuron are modifiable depending on the input signal receives, its output value and a pre-determined or desired response. The desired response is sometimes called *teacher response*. The difference between the desired response and the actual output is called the *error* signal.

- **UNSUPERVISED LEARNING:** In some cases the teacher's response is not available and no error signal is available to guide the learning. When no teacher's response is available the neuron, if properly configured, will modify its weight based only on the input and/or output.

Zurada (1992:59-63)

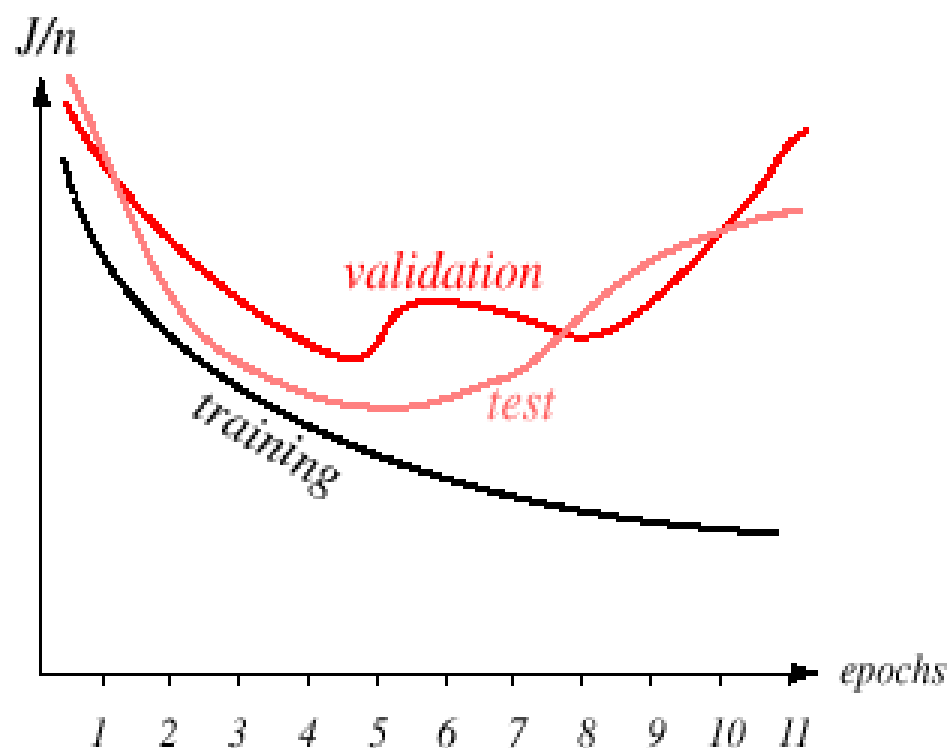


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^n J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

Unsupervised learning

- Idea : group typical input data in function of resemblance criteria un-known a priori
- Data clustering
- No need of a teacher
 - The network finds itself the correlations between the data
 - Examples of such networks :
 - Kohonen self-organized feature maps (SOFM)

- Learning Curves

- Before training starts, the error on the training set is high; through the learning process, the error becomes smaller
- The error per pattern depends on the amount of training data and the expressive power (such as the number of weights) in the network
- The average error on an independent test set is always higher than on the training set, and it can decrease as well as increase
- A validation set is used in order to decide when to stop training ; we do not want to overfit the network and decrease the power of the classifier generalization

“we stop training at a minimum of the error on the validation set”

Rosenblatt's Perceptrons

Consider the following set of training vectors x_1 , x_2 , and x_3 , which are to be used in training a Rosenblatt's perceptron together with the weights etc. Show how the learning proceeds:

$$x_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}; x_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}; x_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

with the desired responses and weights initialized as:

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}; w = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}; c = 0.1$$

Rosenblatt's Perceptrons

Learning cycles for the perceptron above. Note the initial weights are selected at random:

$$w^{1t} = [1 \ -1 \ 0 \ 0.5]$$

Step 1. Input is x_1 and the desired output is d_1 :

$$net^1 = w^{1t} x_1 = [1 \ -1 \ 0 \ 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 2.5$$

$$y = \text{sgn}(net^1) = 1$$

Correction in this step is necessary since $d_1 = -1$ and the input of the perceptron in response to x_1 is 3.5; the error signal is $r \approx -1 - 1 = -2$

Rosenblatt's Perceptrons

Step 1 (contd). The updated weight vector is

$$w^2 = w^1 + 0.1 (-1 \ -1) * x_1$$

$$w^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + 0.1 (-1 \ -1) * \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$$

$$w^{2t} = [0.8 \ -0.6 \ 0 \ 0.7]$$

Rosenblatt's Perceptrons

Step 1 (contd). The updated weight vector is

$$\begin{aligned}w^2 &= w^1 + 0.1(-1 \ -1) * x_1 \\w^2 &= \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ +0.2 \end{bmatrix} = \begin{bmatrix} 1-0.2 \\ -1+0.4 \\ 0 \\ 0.5+0.2 \end{bmatrix} \\w^{2t} &= [0.8 \ -0.6 \ 0 \ 0.7]\end{aligned}$$

Rosenblatt's Perceptrons

Step 2. The updated weight vector w_2

$$w^{2t} = [0.8 \quad -0.6 \quad 0 \quad 0.7]$$

$$net^2 = w^{2t} x_2 = [0.8 \quad -0.6 \quad 0 \quad 0.7] \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} = -1.6$$

Correction in this step is not necessary since $d_2 = -1$ and the output of the perceptron in response to x_2 is $sgn(-1.6) = -1$; the error signal is $r \approx -1 - (-1) = 0$.

Rosenblatt's Perceptrons

Step 3. Input is x_3 and the desired output is d_3 :

$$w^{3t} = w^{2t} = [0.8 \ -0.6 \ 0 \ 0.7]$$

$$net^3 = w^{3t} x_3 = [-1 \ 1 \ 0.5 \ -1] \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} = -2.1$$

Correction in this step is necessary since $d_3=+1$ and the output of the perceptron in response to x_3 is $sgn(-2.1)=-1$; the error signal is $r \approx -1-(-1) = 2$.

Rosenblatt's Perceptrons

Step 3 (contd). The updated weight vector is

$$w^{4t} = w^{3t} + 0.1 * (1 + 1) * x_3$$

$$w^4 = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + 0.2 * \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$$

$$w^{4t} = [0.6 \ -0.4 \ 0.1 \ 0.5]$$

Rosenblatt's Perceptrons

Step 4. Input is x_1 and the desired output is d_1 :

$$w^{4t} = [0.6 \quad -0.4 \quad 0.1 \quad 0.5]$$

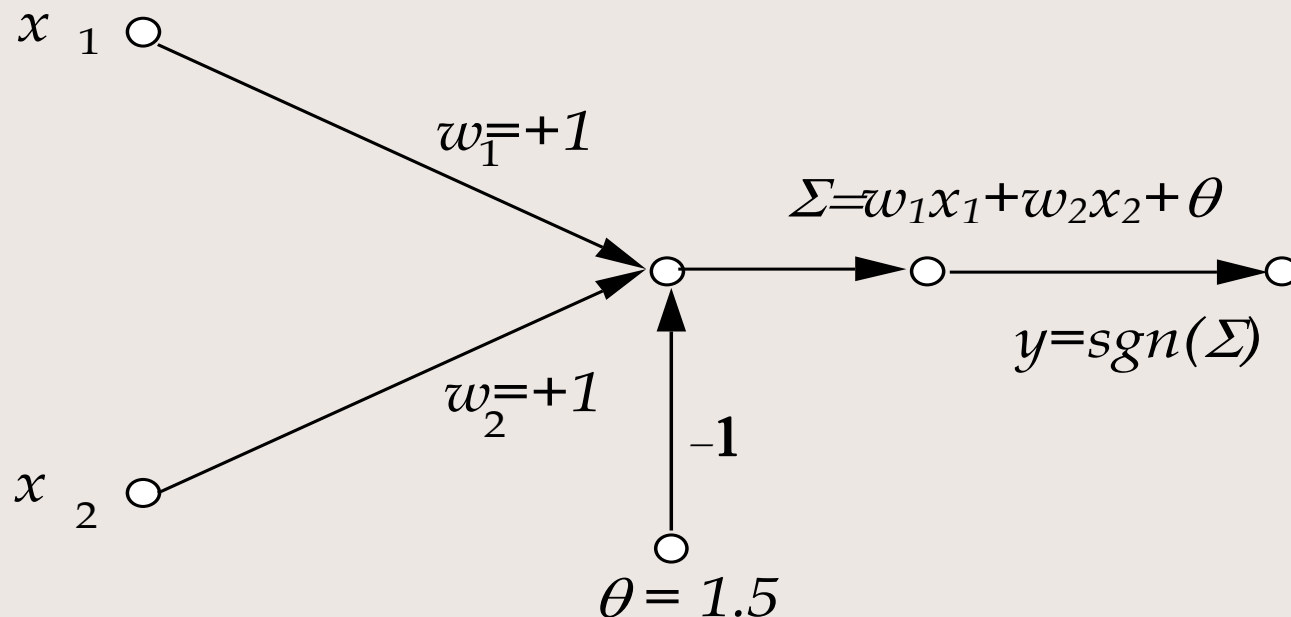
$$net^4 = w^{4t} * x_1 = [0.6 \quad -0.4 \quad 0.1 \quad 0.5] * \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 0.9$$

Correction in this step is necessary since $d_1 = -1$ and the output of the perceptron in response to x_3 is $sgn(0.9) = +1$; the error signal is $r \approx -1 - (1) = -2$; recall that net_1 was 2.5 and r was -3.5

Rosenblatt's Perceptron

A single layer perceptron can carry out a number can perform a number of logical operations which are performed by a number of computational devices.

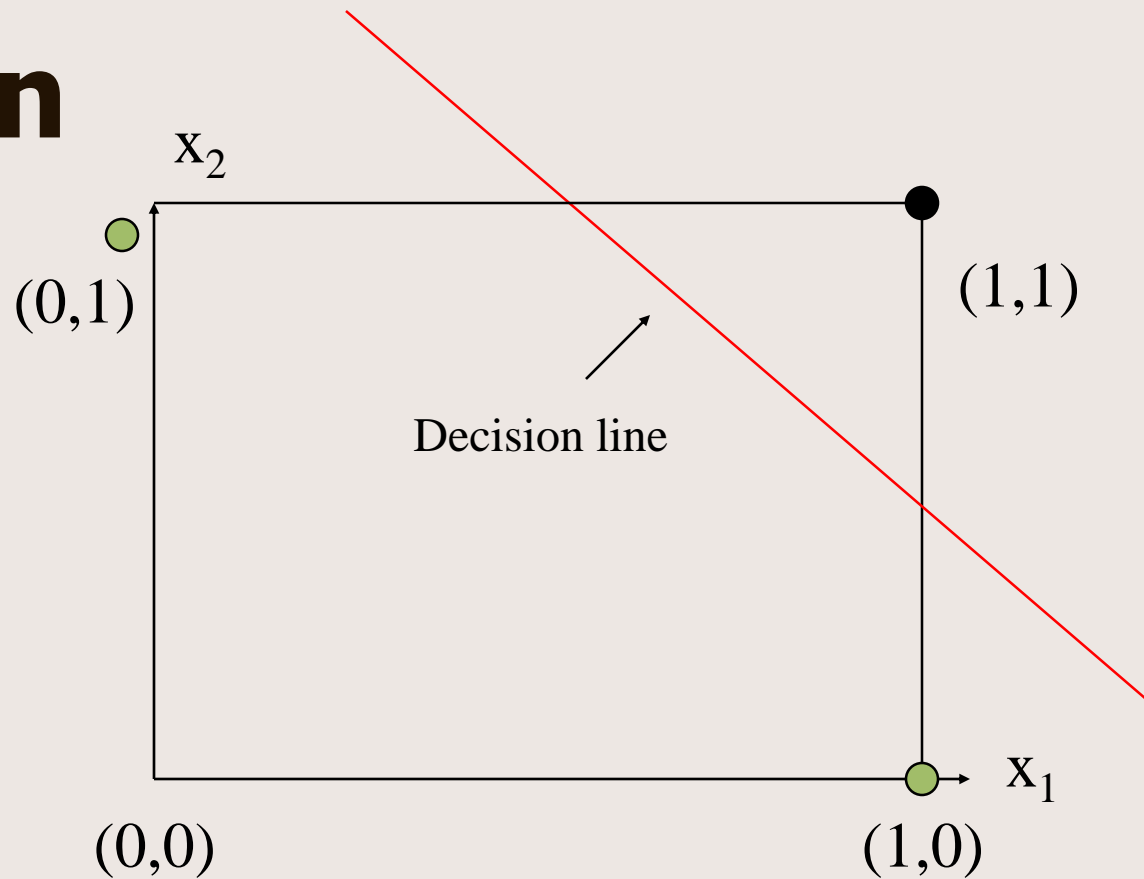
The perceptron below performs the AND operation:



Rosenblatt's Perceptron

Definition of AND

Input		Output
X_1	X_2	Y
0	0	0
0	1	0
1	0	0



Denotes 1

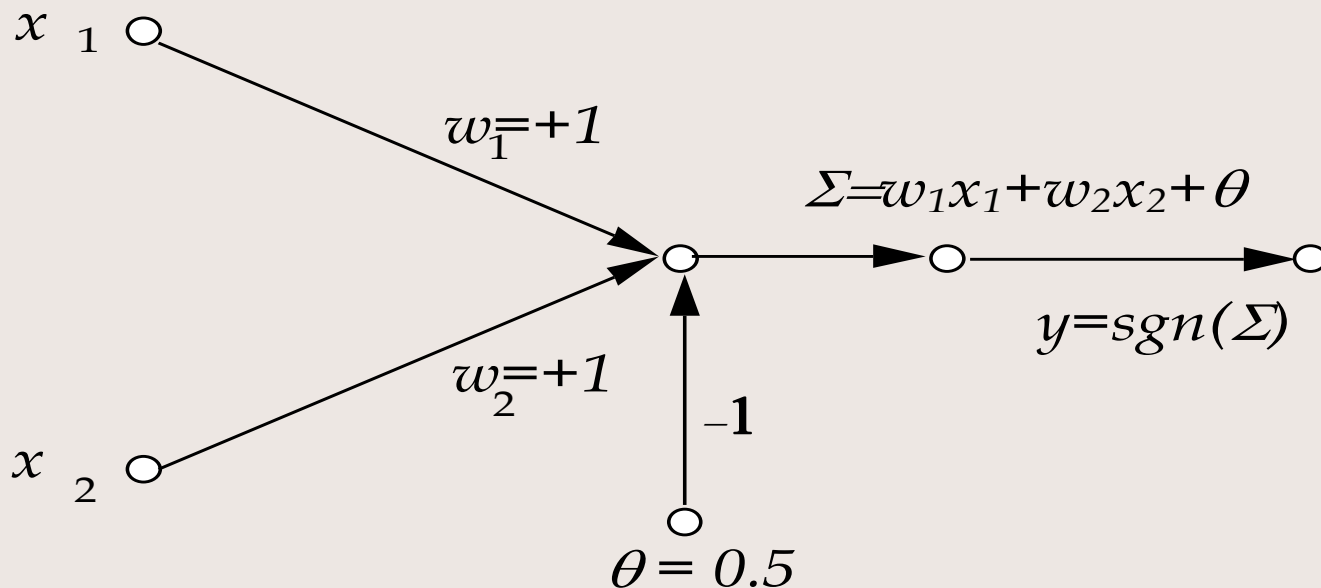


Denotes 0

Rosenblatt's Perceptron

A single layer perceptron can carry out a number can perform a number of logical operations which are performed by a number of computational devices.

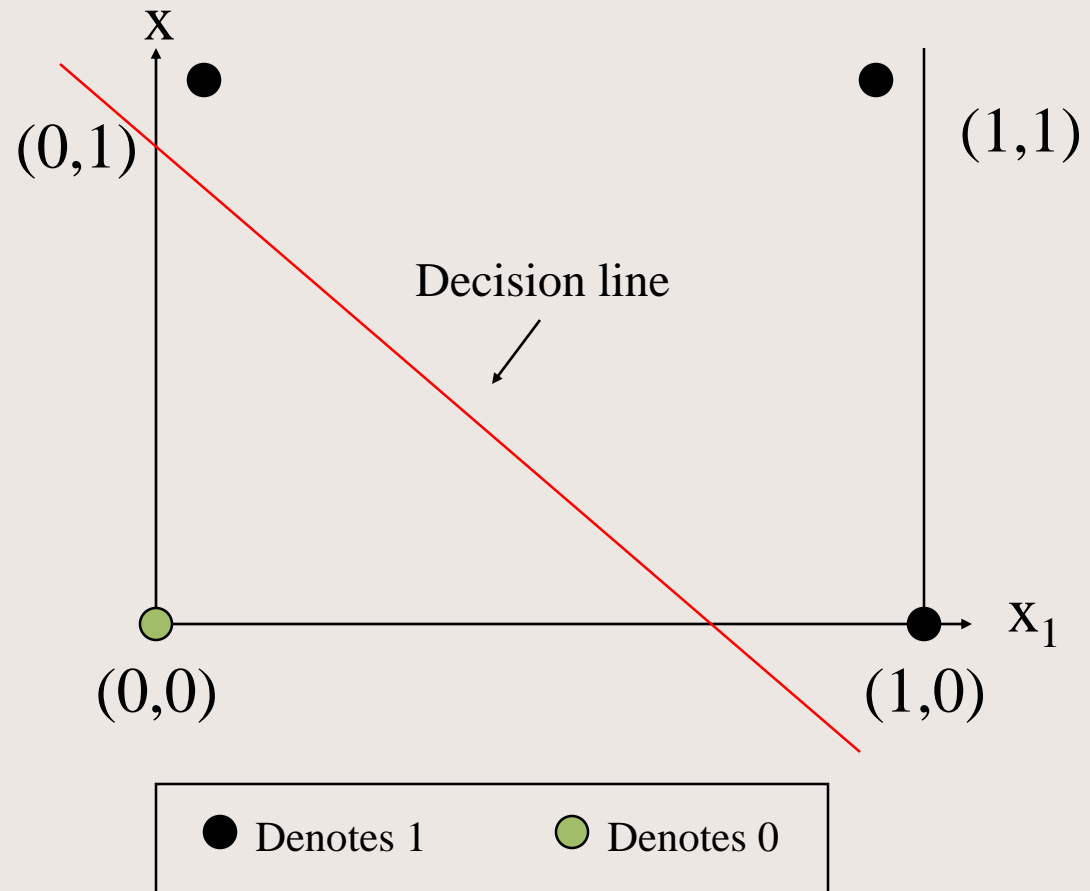
The perceptron below performs the OR operation:



Rosenblatt's Perceptron

Definition of OR

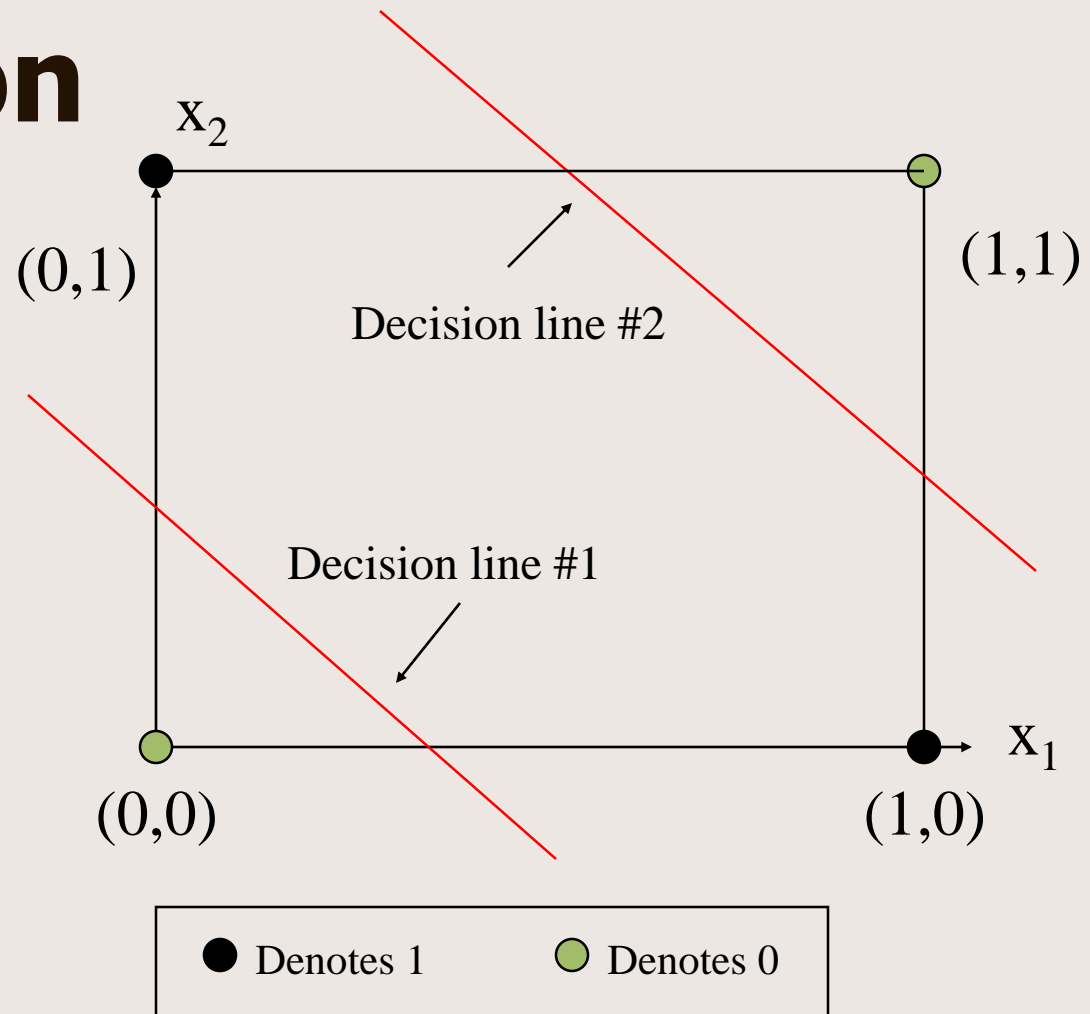
Input		Output
X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	1



Rosenblatt's Perceptron

Definition of XOR

Input		Output
X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

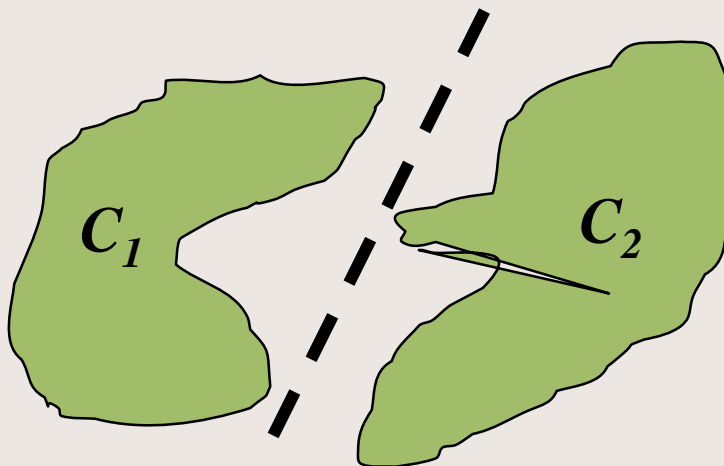


Rosenblatt's Perceptron

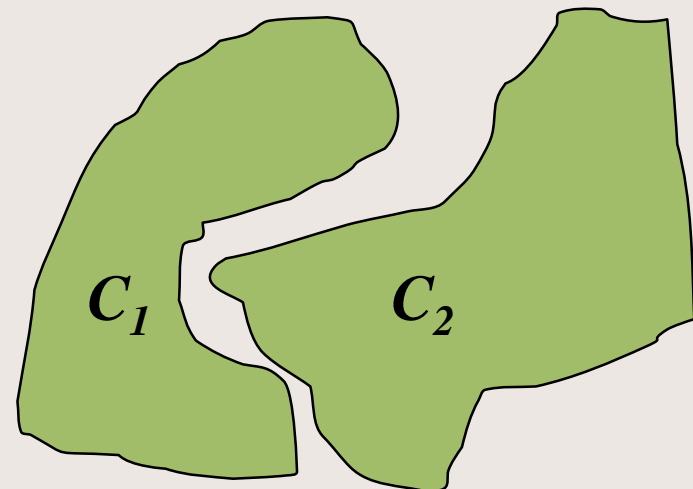
A single layer perceptron can carry out a number can perform a number of logical operations which are performed by a number of computational devices.

However, the single layer perceptron cannot perform the *exclusive-OR* or *XOR* operation. The reason is that a single layer perceptron can only classify two classes, say C_1 and C_2 , should be sufficiently separated from each other to ensure the decision surface consists of a hyperplane.

Linearly separable classes



Linearly non-separable classes



Rosenblatt's Perceptron

- **The XOR 'problem'**

- The simple perceptron cannot learn a linear decision surface to separate the different outputs, *because no such decision surface exists.*
- Such a non-linear relationship between inputs and outputs as that of an XOR-gate are used to simulate vision systems that can tell whether a line drawing is connected or not, and in separating figure from ground in a picture.

Rosenblatt's Perceptron

- **The XOR 'problem'**

- For simulating the behaviour of an XOR-gate we need to draw *elliptical decision surfaces* that would encircle two '1' outputs: A simple perceptron is unable to do so.
- Solution? Employ two separate line-drawing stages.

Rosenblatt's Perceptron

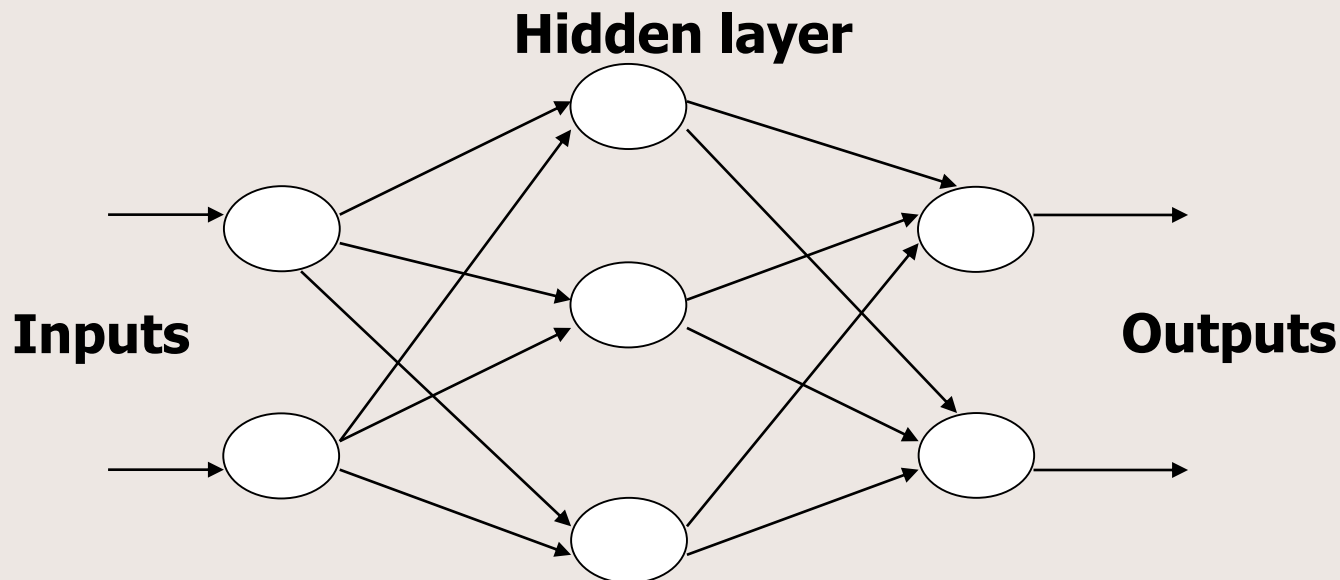
- **The XOR 'solution'**

- The multilayer perceptron designed to solve the XOR problem has a serious problem.
- **The perceptron convergence theorem does not extend to multilayer perceptrons. The perceptron learning algorithm can adjust the weights between the inputs and outputs, but it cannot adjust weights *between* perceptrons.**
- For this we have to wait for the backpropagation learning algorithms.

Rosenblatt's Perceptron

Multilayer Perceptron

The perceptron built around a single neuron is limited to performing pattern classification with only two classes (hypotheses). By expanding the output (computation) layer of perceptron to include more than one neuron, it is possible to perform classification with more than two classes, but the classes have to be separable.



Back-propagation Algorithm: Supervised Learning

- Backpropagation (**BP**) is amongst the 'most popular algorithms for ANNs': it has been estimated by Paul Werbos, the person who first worked on the algorithm in the 1970's, that between 40% and 90% of the real world ANN applications use the BP algorithm. Werbos traces the algorithm to the psychologist Sigmund Freud's theory of *psychodynamics*. Werbos applied the algorithm in *political forecasting*.
- David Rumelhart, Geoffery Hinton and others applied the BP algorithm in the 1980's to problems related to supervised learning, particularly *pattern recognition*.
- The most useful example of the BP algorithm has been in dealing with problems related to *prediction* and *control*.

Back-propagation Algorithm

- A a back-propagation neural network is a multi-layer network and the layers are fully connected, that is, every neuron in each layer is connected to every other neuron in the adjacent forward layer.

BASIC DEFINITIONS

1. Backpropagation is a procedure for efficiently calculating the derivatives of some output quantity of a non-linear system, with respect to all inputs and parameters of that system, through calculations proceeding *backwards* from outputs to inputs.
2. Backpropagation is any technique for adapting the weights or parameters of a nonlinear system by somehow using such derivatives or the equivalent.

According to Paul Werbos there is no such thing as a “backpropagation network”, he used an ANN design called a *multilayer perceptron*.

Back-propagation Algorithm

Paul Werbos provided a 'rule for updating the weights of a multi-layered network undergoing **supervised learning**. It is the weight adaptation rule which is called *back propagation*.

Typically, a fully connected feed forward network is used to be trained using the BP algorithm: activation in such networks travels in a direction from the input to the output layer and the units in one layer are connected to every other unit in the next layer.

There are two *sweeps* of the fully connected network: *forward sweep* and *backward sweep*.

Back-propagation Algorithm

There are two *sweeps* of the fully connected network: *forward sweep* and *backward sweep*.

Forward Sweep: This sweep is similar to any other feed-forward ANN – the input stimuli is given to the network, the network computes the weighted average from all the input units and then passes the average through a *squash (activation)* function. The ANN generates an output subsequently.

The ANN may have a number of *hidden layers*, for example, the multi-net perceptron, and the output from each hidden layer becomes the input to the next layer forward.

Back-propagation Algorithm

There are two *sweeps* of the fully connected network: *forward sweep* and *backward sweep*.

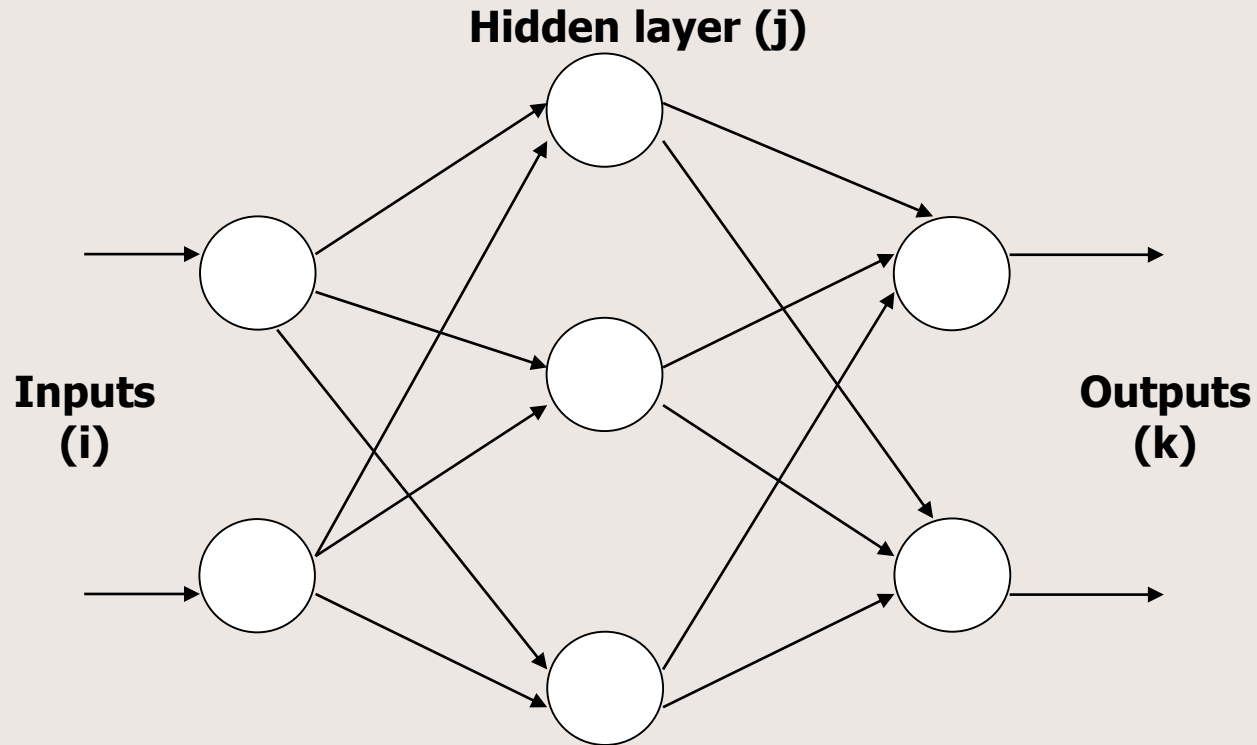
Backwards Sweep: This sweep is similar to the forward sweep, except what is 'swept' are the error values. These values essentially are the differences between the actual output and a desired output.

$$(\Rightarrow e_j = d_j - o_j)$$

The ANN may have a number of *hidden layers*, for example, the multi-net perceptron, and the output from each hidden layer becomes the input to the next layer forward.

In the backward sweep output unit sends errors back to the first proximate hidden layer which in turn passes it onto the next hidden layer. No error signal is sent to the input units.

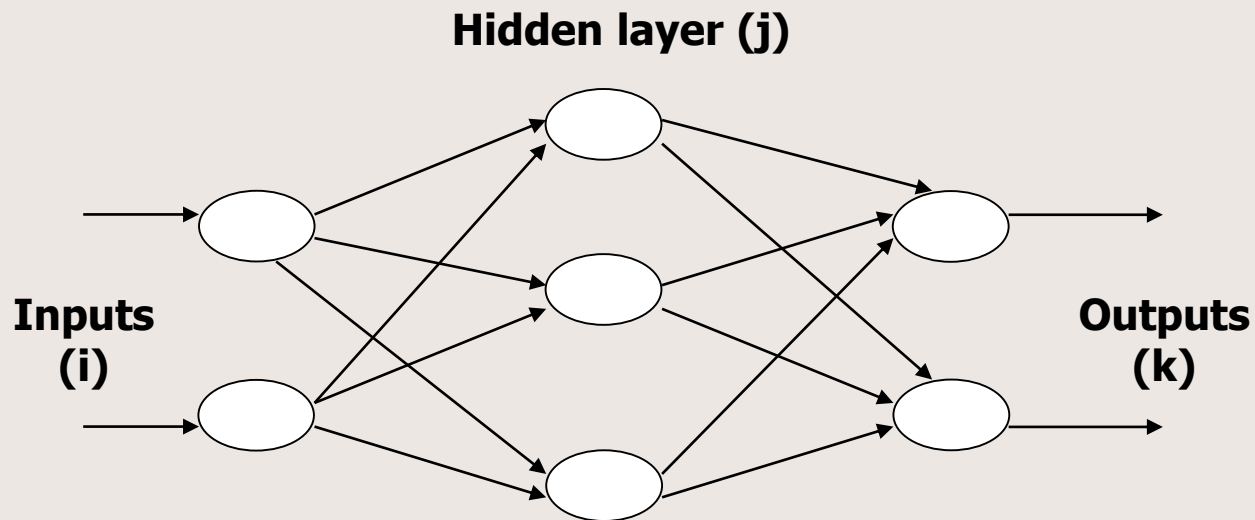
Back-propagation Algorithm



- To derive the back-propagation learning law, let us consider the three-layer network shown in **Fig.** The indices, i, j, k , here refer to neurons in the input, hidden and output layers, respectively.

Back-propagation Algorithm

- Input signals are propagated through the network from left to right, and error signals from right to left. The symbol w_{ij} denotes the weight for the connection between neuron i in the input layer and neuron j in the hidden layer, and the symbol w_{jk} the weight between neuron j in the hidden layer and neuron k in the output layer.
- To propagate error signals, we start at the output layer and work backward to the hidden layer.



Back-propagation Algorithm

Step 1: Initialize all the weights to small random values and choose a positive constant α (learning rate).

Step 2: Present samples 1 to N repeatedly to the classifier.

Forward sweep:

For each input sample, calculate the output of a hidden layer neuron at the p^{th} training cycle,

$$y_j(p) = R\left(\sum_{i=1}^n x_i(p)w_{ij}(p)\right)$$

and also at the output layer,

$$y_k(p) = R\left(\sum_{j=1}^m y_j(p)w_{jk}(p)\right)$$

Sigmoid function:

$$R = \frac{1}{1 + e^{-x}}$$

$$\Rightarrow \frac{dR}{dx} = R(1 - R)$$

Back-propagation Algorithm

Step 3: Back-propagation step, Compute error signals at the output and the hidden layers,

$$e_k(p) = y_k(p) - d_k$$

$$\delta_k(p) = y_k(p)[1 - y_k(p)]e_k(p)$$

$$\delta_j(p) = y_j(p)[1 - y_j(p)] \sum_{k=1}^l \delta_k(p) w_{jk}(p)$$

Step 4: Update the weights. By the following MSE algorithm:

$$w_{jk}(p+1) = w_{jk}(p) - \alpha \delta_k(p) y_j(p)$$

$$w_{ij}(p+1) = w_{ij}(p) - \alpha \delta_j(p) x_i(p)$$

Step 5: Repeat steps 2 and 4 until the weights cease to change significantly.

Hebbian Learning

DONALD HEBB, a Canadian psychologist, was interested in investigating **PLAUSIBLE MECHANISMS FOR LEARNING AT THE CELLULAR LEVELS IN THE BRAIN**. (Donald Hebb's (1949) *The Organisation of Behaviour*. New York: Wiley).

HEBB's POSTULATE: When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

Hebbian Learning

Hebbian Learning laws Cause weight changes in response to events within a processing element that happen simultaneously. The learning laws in this category are characterized by their completely local - both in space and in time-character.

Hebbian Learning

A simple form of Hebbian Learning Rule

$$w_{k j}^{new} = w_{k j}^{old} + \eta y_k x_j ,$$

where η is the so-called *rate of learning* and x and y are the input and output respectively.

This rule is also called the *activity product rule*.

Hebbian Learning

A simple form of Hebbian Learning Rule

If there are "m" pairs of vectors,

$$(\bar{x}_1, \bar{y}_1) \cdots (\bar{x}_m, \bar{y}_m)$$

to be stored in a network, then the training sequence will change the weight-matrix, W, from its initial value of ZERO to its final state by simply adding together all of the incremental weight change caused by the "m" applications of Hebb's law:

$$W = \bar{y}_1 \bar{x}_1^T + \bar{y}_2 \cdot \bar{x}_2^T + \cdots + \bar{y}_n \cdot x_n^T$$

Hebbian Learning

A worked example: Consider the Hebbian learning of three input vectors:

$$x^{(1)} = \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix}; \quad x^{(2)} = \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix}; \quad x^{(3)} = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix}$$

in a network with the following initial weight vector:

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

Hebbian Learning

The worked example shows that with discrete $f(net)$ and $\eta = 1$, the weight change involves **ADDING** or **SUBTRACTING** the entire input pattern vectors to and from the weight vectors respectively.

Consider the case when the activation function is a continuous one. For example, take the bipolar continuous activation function:

$$f(net) = \frac{2}{1 + \exp(-\lambda * net)} - 1;$$

where $\lambda > 0$.

Hebbian Learning

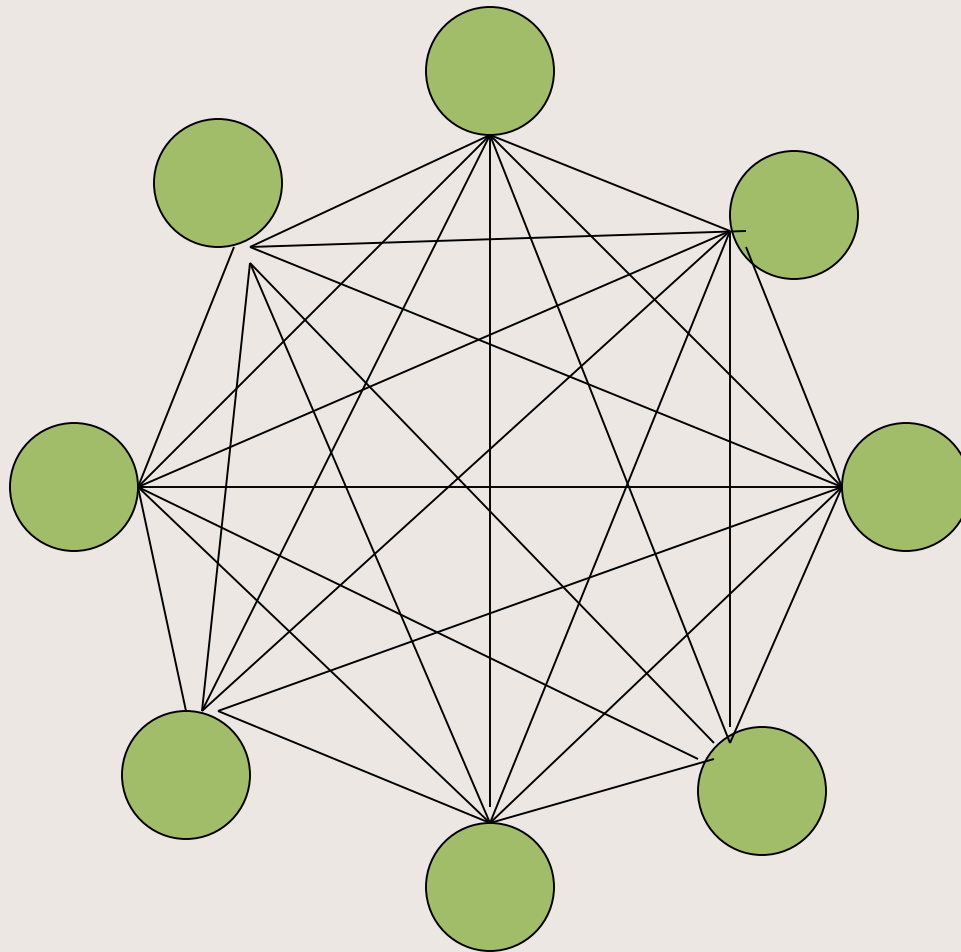
The worked example shows that with bipolar continuous activation function indicates that the weight adjustments are tapered for the continuous function but are generally in the same direction:

Vector	Discrete Bipolar $f(net)$	Continuous Bipolar $f(net)$
$x^{(1)}$	1	0.905
$x^{(2)}$	-1	-0.077
$x^{(3)}$	-1	-0.932

Hopfield Network

- The Hopfield network consists of a set of N interconnected neurons which update their activation values asynchronously and independently of other neurons.
- All neurons are both input and output neurons. The activation values are binary (+1, -1)

Hopfield Network



Hopfield Network

- The state of the system is given by the activation values $y = (y_k)$.
- The net input $s_k(t+1)$ of a neuron k at cycle $(t+1)$ is a weighted sum

$$s(t+1) = \sum_{j \neq k} y_j(t) w_{jk} + b_k$$

Hopfield Network

- A threshold function is applied to obtain the output

$$y_k(t+1) = \text{sgn}(s_k(t+1))$$

Hopfield Network

- A neuron k in the net is stable at time t I.e.

$$y_k(t) = \text{sgn}(s_k(t-1))$$

- A state is state if all the neurons are stable

Hopfield Networks

- If $w_{jk} = w_{kj}$ the behavior of the system can be described with an energy function

$$\mathcal{E} = -\frac{1}{2} \sum_j \sum_{j \neq k} y_j y_k w_{jk} - \sum_k b_k y_k$$

- This kind of network has stable limit points

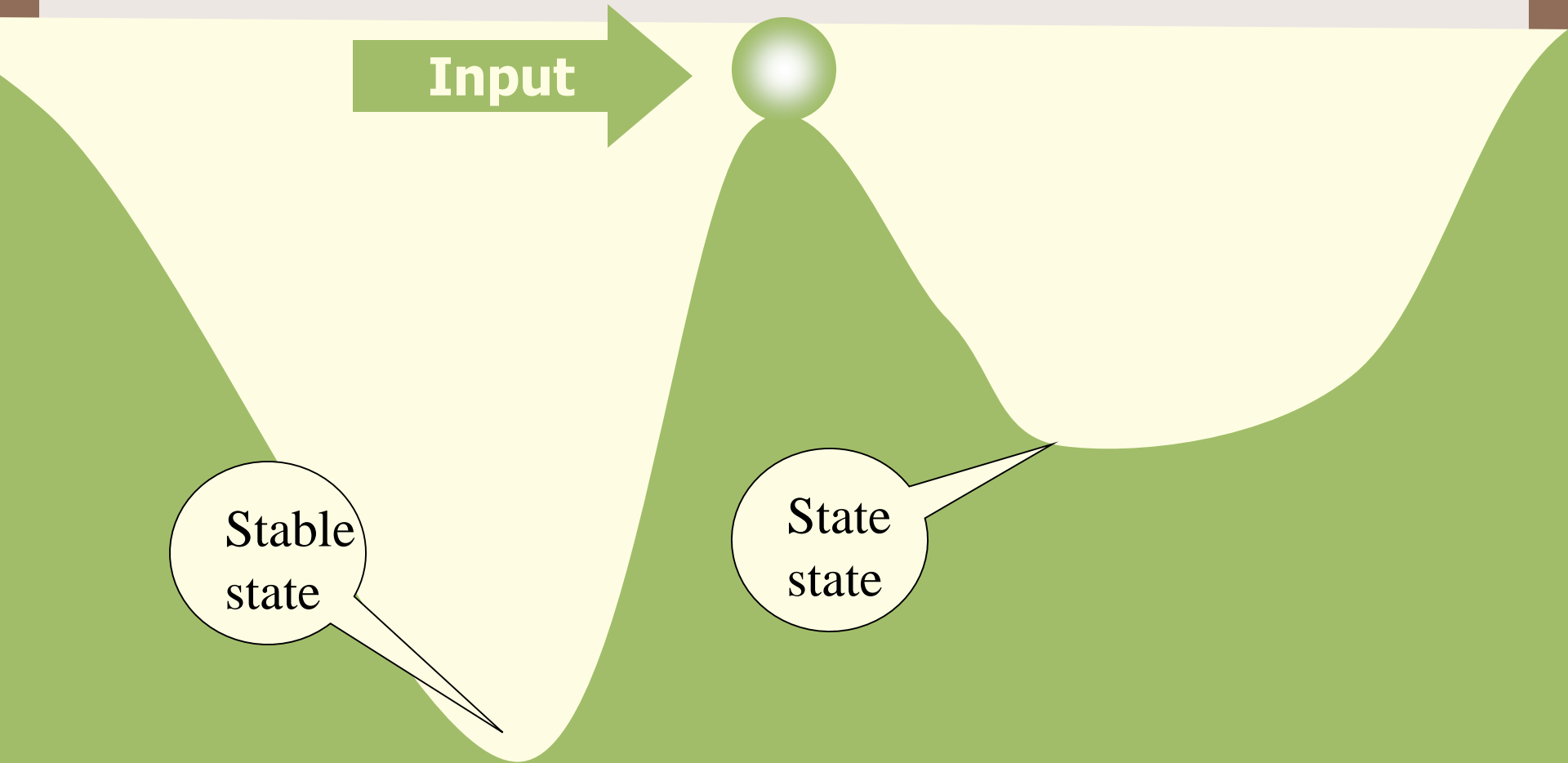
Hopfield net. applications

- A primary application of the Hopfield network is an associative memory.
- The states of the system corresponding with the patterns which are to be stored in the network are stable.
- These states can be seen as 'dips' in energy space.

Hopfield Networks

- It appears, however, that the network gets saturated very quickly, and that about $0.15N$ memories can be stored before recall errors become severe.

Hopfield Networks



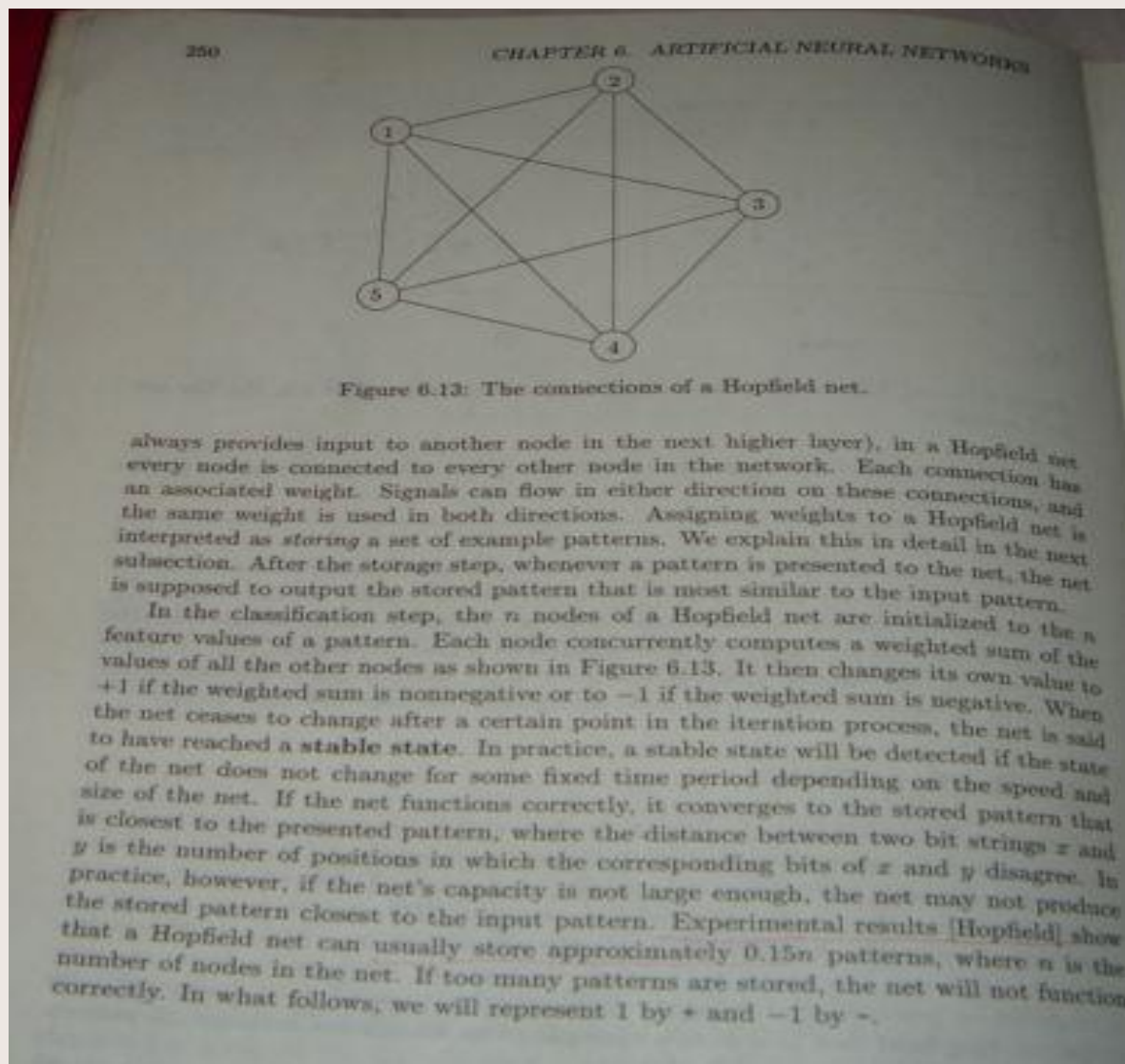
Hopfield Nets

6.5 Hopfield Nets

A distinctive feature of biological brains is their ability to recognize a pattern given only an imperfect representation of it. A person can recognize a friend that he or she has not seen for years given only a poor quality photograph of that person; the memory of that friend can then evoke other related memories. This process is an example of *associative memory*. A memory or pattern is accessed by associating it with another pattern, which may be imperfectly formed. By contrast, ordinary computer memory is retrieved by precisely specifying the location or address where the information is stored. A Hopfield net is a simple example of an associative memory. A pattern consists of an n -bit sequence of 1s and -1s.

While the nets discussed in the previous sections are strictly feed-forward (a node

Hopfield Nets



Hopfield Nets

Example 6.6 A Hopfield net.

Suppose that a ten-node Hopfield net has stored the patterns

$$\mathbf{x}^{(1)} = ++++++, \quad \mathbf{x}^{(2)} = +-+++-, \quad \mathbf{x}^{(3)} = +++-----$$

If the new pattern $++++-----$ is presented to the net, the stored pattern $++++-----$ should be retrieved because the distance between $++++-----$ and $++++-----$ is 1 while the distance between $++++-----$ and $+-+++-$ is 4 and the distance between $+-+++-$ and $++++-----$ is 3.

Although using a nearest neighbor algorithm to compute the stored pattern closest to the input pattern might be more straightforward, one attractive feature of the Hopfield net is that all the nodes in the net can operate asynchronously in parallel. Suppose that the nodes asynchronously compute the weighted sum of the other nodes and that they recompute and change their states at completely random times, such that this occurs on the average once every time unit. We say the node fires when it reads the states of the other nodes, updates its state after recomputing its (possibly) new value, and sends its recomputed state to the other nodes. Note that the state of a node does not necessarily change when the node fires, because the recomputed state may be the same as its previous state. In general the probability that two or more nodes fire at exactly the same time is 0, so the firings can be listed in order of their occurrence. However, even though the firings occur in sequential order, they occur more quickly than they would occur if a conventional computer with a single processor were used. Entry ij in the matrix in Figure 6.14 represents the connection weight between nodes i and j after the three patterns from Example 6.6 are stored. Example 6.7 shows how the values of the net change at successive iterations during a retrieval step.

Example 6.7 Iterations of a Hopfield net.

The following table shows the successive firings of the Hopfield net having the weight matrix defined in Figure 6.14 and input pattern $+++-----$.

Hopfield Nets

232

CHAPTER 6. ARTIFICIAL NEURAL NETWORKS

	1	2	3	4	5	6	7	8	9	10
1	0	1	3	1	3	-1	1	-1	1	-1
2	1	0	1	3	1	-1	-1	1	-1	1
3	3	1	0	1	3	-1	1	-1	1	-1
4	1	3	1	0	1	1	-1	1	-1	1
5	3	1	3	1	0	-1	1	-1	1	-1
6	-1	1	-1	1	-1	0	1	3	1	3
7	1	-1	1	-1	1	1	0	1	3	1
8	-1	1	-1	1	-1	3	1	0	1	3
9	1	-1	1	-1	1	1	3	1	0	1
10	-1	1	-1	1	-1	3	1	3	1	0

Figure 6.14: The weight matrix of the Hopfield net in Example 6.7.

Iteration Number	Node	Fire Time	Current Node Values
1	7	0.123	+++++++
2	10	0.134	+++++++
3	5	0.328	+++++++
4	10	0.441	+++++++
5	4	0.471	+++++++
6	8	0.524	+++++++

In this case, the net converges to the stored pattern +++++++, which is the closest stored pattern to the input pattern. This example required five iterations for convergence. However, the firings of the nodes occur randomly, so another run might require a different number of iterations to reach a stable state because the firing sequence may be different for another run.

The Storage and Retrieval Algorithms

There are two distinct phases in using a Hopfield net. The first phase uses a **storage algorithm** to compute the matrix of connection weights from the patterns to be stored. This is not an iterative adaptive process; the weight between nodes i and j depends only on the similarity between bits i and j in the set of patterns to be stored. Let

$$\mathbf{x}^{(p)} = (x_1^{(p)}, x_2^{(p)}, \dots, x_n^{(p)})$$

Hopfield Nets

be the p th pattern that is stored in the set, where $1 \leq p \leq m$. The weight between nodes i and j is calculated as

$$w_{ij} = \begin{cases} \sum_{p=1}^m x_i^{(p)} x_j^{(p)} & i \neq j \\ 0 & i = j \end{cases} \quad (6.17)$$

Thus, the weight between node i and node j is the number of times that the i th component of a stored pattern is the same as the j th component of that same pattern, minus the number of times that they differ, summed over all the stored patterns.

For example, to obtain the weight matrix entry in the second row and the sixth column in Figure 6.14, note that $x_1^{(1)} = 1$, $x_6^{(1)} = 1$, $x_1^{(2)} = -1$, $x_6^{(2)} = -1$, $x_1^{(3)} = 1$, and $x_6^{(3)} = -1$, so (6.17) gives

$$w_{26} = (1)(1) + (-1)(-1) + (1)(-1) = 1.$$

The second phase of using a Hopfield net is the retrieval algorithm. Let the input pattern to be retrieved be

$$\mathbf{y} = (y_1, y_2, \dots, y_n).$$

The processing begins by initializing the value of node i to y_i . Next the nodes begin firing. When node i fires, u_i is set to

$$\text{sgn} \left(\sum_{j=1}^n w_{ij} y_j \right), \quad (6.18)$$

where $\text{sgn } x = +1$, if $x \geq 0$, and $\text{sgn } x = -1$, if $x < 0$. These calculations are repeated until the states stop changing. To see how an input pattern can be iteratively updated to retrieve a stored pattern, we rewrite the sum in (6.18) as

$$\sum_{j=1}^n w_{ij} y_j = \sum_{j=1, j \neq i}^n \sum_{p=1}^m x_i^{(p)} x_j^{(p)} y_j = \sum_{p=1}^m x_i^{(p)} \left(\sum_{j=1, j \neq i}^n x_j^{(p)} y_j \right). \quad (6.19)$$

Now if the input pattern \mathbf{y} is close to the stored pattern $\mathbf{x}^{(p)}$, the quantity in parentheses in (6.19) will be close to n . If \mathbf{y} and $\mathbf{x}^{(p)}$ are very different from each other, this quantity will be close to $-n$. The pros and cons are then summed for each pattern. If $x_i^{(p)}$ is $+1$ and the quantity in parentheses is positive, then the $+1$ is probably correct, so a positive number is contributed to the sum in (6.19). If the quantity in parentheses is negative, then the $+1$ is probably incorrect, so a negative number is contributed to the sum in (6.19). The situation is reversed if $x_i^{(p)}$ is -1 . In either case, if (6.19) is nonnegative, the most likely value of $x_i^{(p)}$ is $+1$, so $x_i^{(p)}$ is changed to $+1$. If (6.19) is negative, $x_i^{(p)}$ is changed to -1 .

Hopfield Nets

In order to show that the net eventually converges, we define

$$E = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{ij} y_i y_j, \quad (6.20)$$

which is called the *Lapapunov function* of the net. We claim that when a node, say k , fires, E does not increase. To see this, first suppose that node k fires and y_k changes from $+1$ to -1 . We may rewrite E as

$$E = y_k \sum_{j=1}^n w_{kj} y_j + \frac{1}{2} \sum_{i=1, i \neq k}^n \sum_{j=1, j \neq k}^n w_{ij} y_i y_j.$$

If we let ΔE denote the original value of E (when $y_k = +1$) minus the new value of E (when $y_k = -1$) and Δy denote the original value of y_k ($+1$) minus the new value of y_k (-1), then

$$\Delta E = \Delta y \sum_{j=1}^n w_{kj} y_j = 2 \left(\sum_{j=1}^n w_{kj} y_j \right). \quad (6.21)$$

Since y_k changed from $+1$ to -1 , the term in brackets is nonnegative. Therefore $\Delta E \geq 0$. Thus if node k fires and y_k changes from $+1$ to -1 , E does not increase.

Similarly, if node k fires and y_k changes from -1 to $+1$,

$$\Delta E = -2 \left(\sum_{j=1}^n w_{kj} y_j \right).$$

In this case, since y_k changed from -1 to $+1$, the term in brackets is negative. Therefore $\Delta E > 0$. Thus if node k fires and y_k changes from -1 to $+1$, E decreases.

Since the weights w_{ij} are integers and each y_i is ± 1 , it follows from (6.21) that when E decreases, it decreases by at least 2. Further, since E is bounded below, it cannot decrease indefinitely. (A crude lower bound for E is $-\frac{1}{2}n^2W$, where W is the maximum of $\{|w_{ij}|\}$. The term n^2 occurs because there are n^2 summands in (6.20).)

Suppose, by way of contradiction, that nodes continue to fire and the y_i continue to change. Then E will never increase. Furthermore, eventually nodes will change from -1 to $+1$, and E will, therefore, decrease infinitely often. Since this is impossible, it follows that eventually the net will converge to a **stable state**, that is, a state that will not change as the nodes continue to fire. The stable state to which the net converges may not be unique, and it may not even be one of the patterns that were stored (see Problem 6.14). Such situations arise when the number of patterns stored is too large compared with the number of nodes in the network. The following example shows what happens when too many patterns are stored in a net.

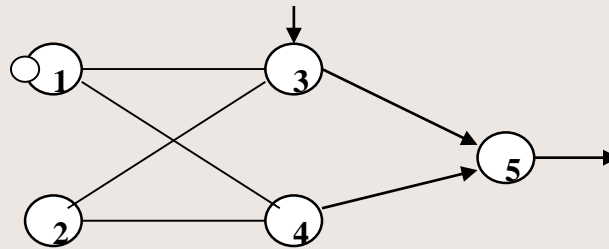
Neural Net Summary

ANN is used in variety of applications:

- Pattern recognition
- Fault diagnosis
- Monitoring patients in medical settings
- Character recognition
- Fraud detection
- Adaptive filtering
- and so on

Problems

1. (a) Define the weight adaptation rule described in the neural networks' literature as *back-propagation* in terms of the two patterns of computation used in the implementation of the rule: the *forward* and *backward* pass.
- (b) Consider a multi-layer feed forward network with a 2-2-1 architecture:



This network is to be trained using the back propagation rule on input pattern x , and a target output, d of 0.9.

$$x^T = [0.1 \ 0.9]$$

Perform a complete *forward*- and *backward-pass* computation using the input x and the target d for one cycle. Compute the outputs of the hidden layer neurons 3 and 4 and for the output neuron 5. Compute the error for the *output node* and for the *hidden nodes*.

Examples

4. (a) A self-organising map has been trained with an input consisting of eight Gaussian clouds with unit variance but difference centres located at eight points around a cube: $(0,0,0,0,0,0,0,0)$, $(4,0,0,0,0,0,0,0)$... $(0,4,4,0,0,0,0,0)$ (Figure 1). This 3-dimensional input structure was represented by a two dimensional lattice of 10 by 10 neurons output layer of a Kohonen self-organising map (SOM). Table 1 shows the feature map.

1	2	3	4	5	6	7	8	9	10	
6	6	6	6	5	5	5	5	5	5	1
6	6	6	5	5	5	5	5	5	1	2
6	7	6	8	8	8	5	5	1	1	3
7	7	7	7	8	8	8	5	1	1	4
7	7	7	7	8	8	8	4	1	1	5
7	7	7	7	8	8	4	4	1	1	6
6	7	7	7	7	4	4	4	4	1	7
2	2	2	3	3	3	4	4	4	1	8
2	2	2	3	3	3	3	4	4	4	9
2	2	2	3	3	3	3	4	4	4	10

Table 1: Feature Map provided by an SOM

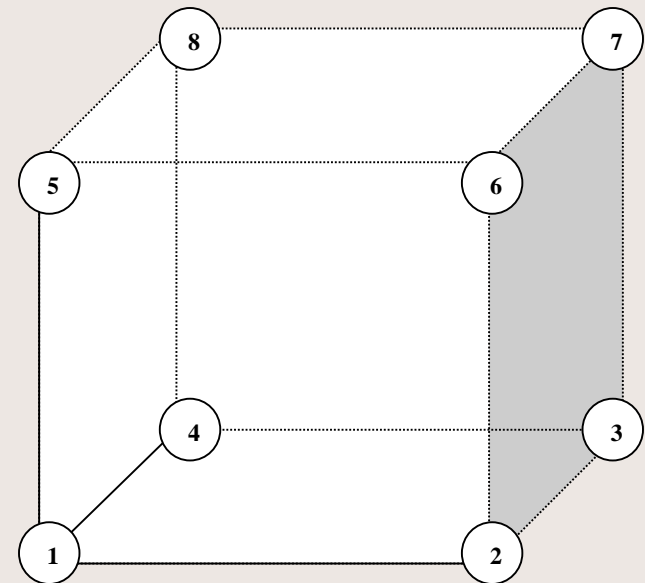


Fig. 1: Input Pattern

Examples

4. (a) *cont.* Do you think that an SOM has classified the Gaussian clouds correctly? Draw the clusters out in the feature map. What conclusions can be drawn from the statement that ‘SOM algorithm preserves the topological relationship that exist in the input space’?

(b) A *self-organised feature map* (SOFM) with three input units and four output units is to be trained using the four training vectors:

$$\mathbf{x}_1^T = [0.80.70.4]; \mathbf{x}_2^T = [0.60.90.9]; \\ \mathbf{x}_3^T = [0.30.40.1] \text{ \& } \mathbf{x}_4^T = [0.10.10.3]$$

The initial weight matrix is:

$$\begin{bmatrix} 0.50.4 \\ 0.60.2 \\ 0.80.5 \end{bmatrix};$$

The *initial radius* is *zero* and the learning rate η is set to 0.5. Calculate the weight changes during the first cycle through the training data taking the training vectors in the order given above.