Fundamentals of Artificial Neural Networks

Fakhri Karray

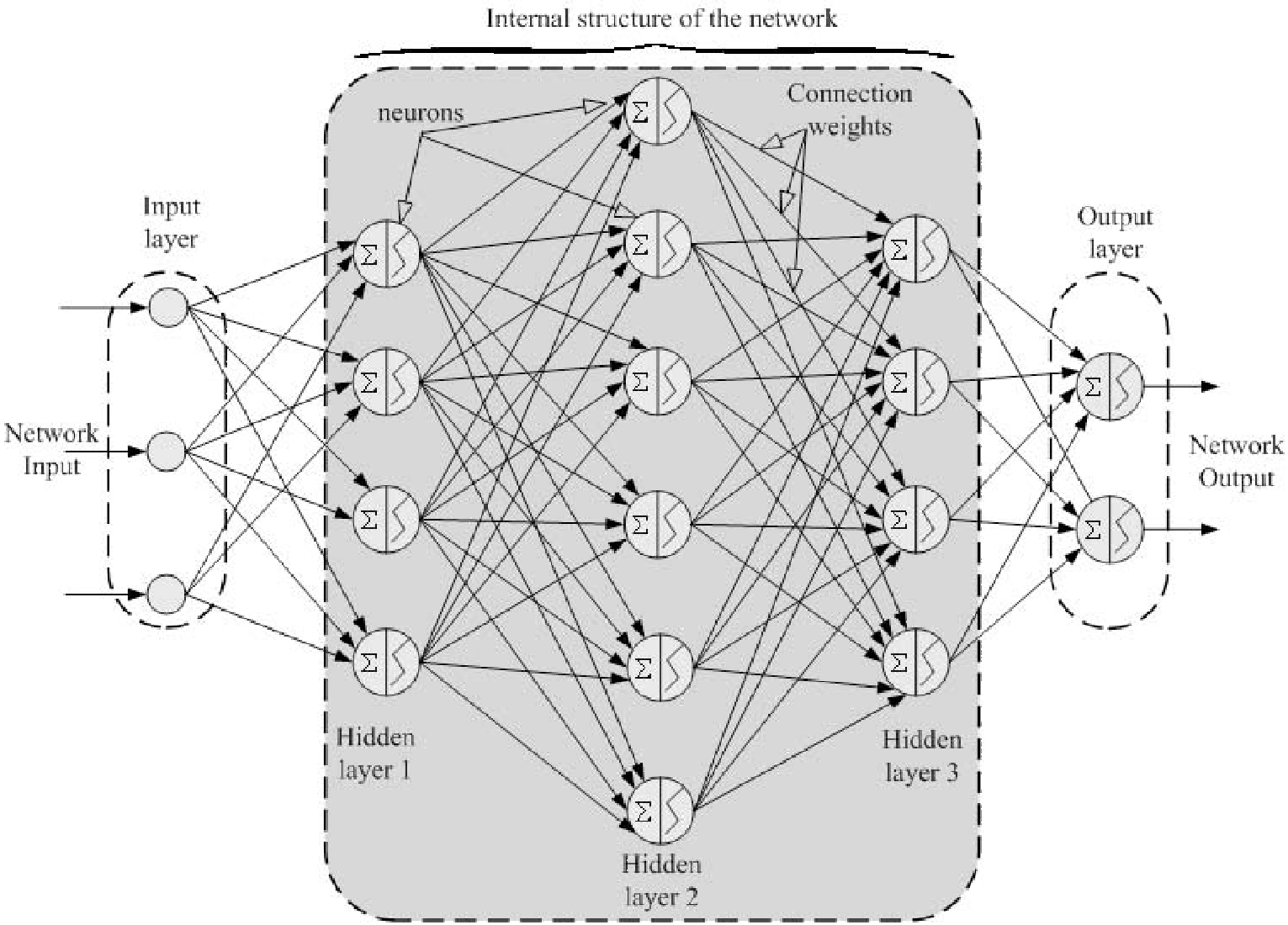
University of Waterloo

Outline

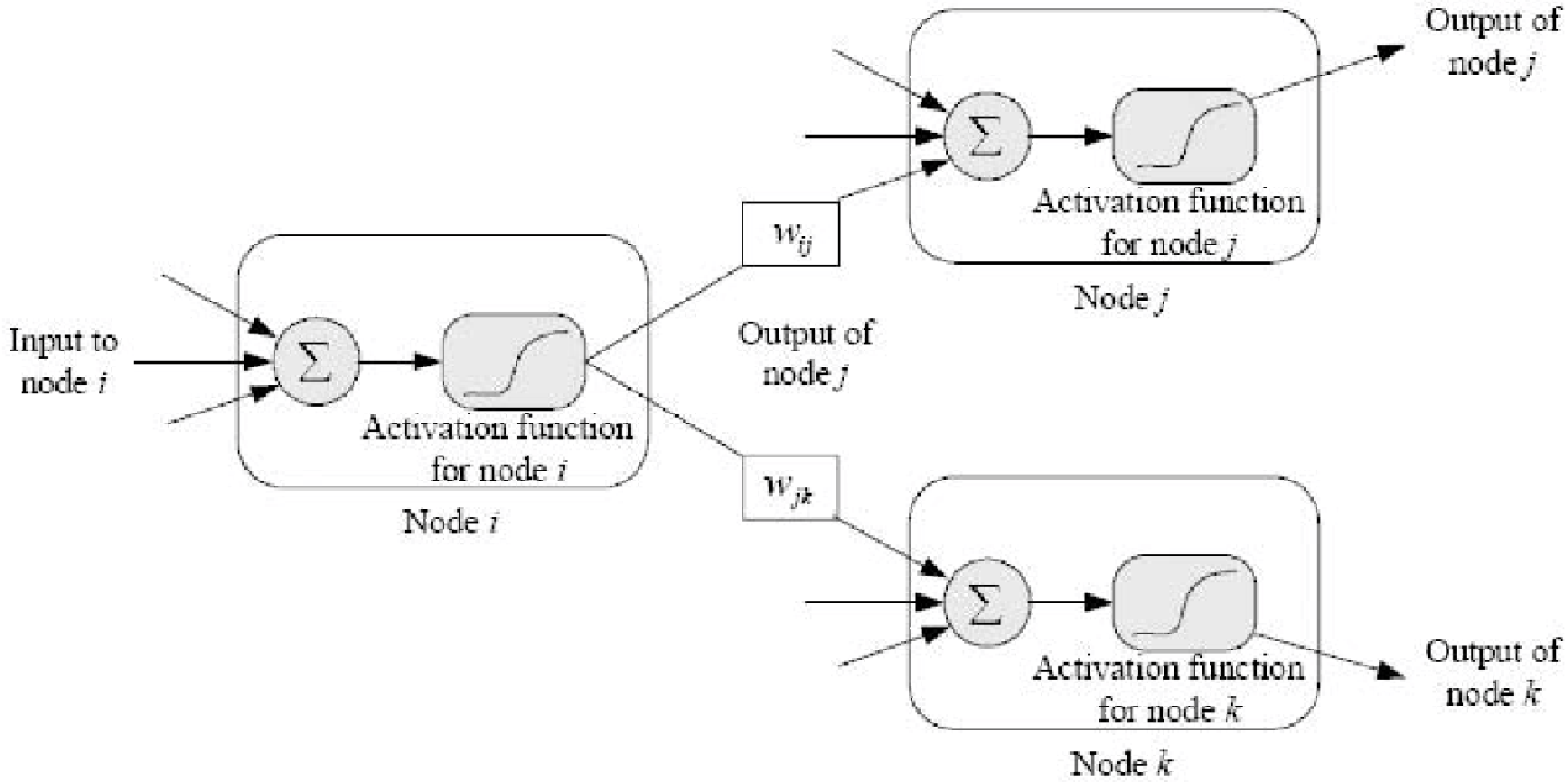
Introduction

- Artificial Neural Networks (ANNs) are physical cellular systems, which can acquire, store and utilize experiential knowledge.
- ANNs are a set of parallel and distributed computational elements classified according to topologies, learning paradigms and at the way information flows within the network.
- ANNs are generally characterized by their:
  - Architecture
  - Learning paradigm
  - Activation functions

Typical Representation of a Feedforward ANN



Interconnections Between Neurons



A Brief History

- ANNs have been originally designed in the early forties for pattern classification purposes.
  $\Rightarrow$ They have evolved so much since then.
- ANNs are now used in almost every discipline of science and technology:
  - from Stock Market Prediction to the design of Space Station frame,
  - from medical diagnosis to data mining and knowledge discovery,
  - from chaos prediction to control of nuclear plants.

Features of ANNs

ANN are classified according to the following:

Architecture

Feedforward
Recurrent

Activation Functions

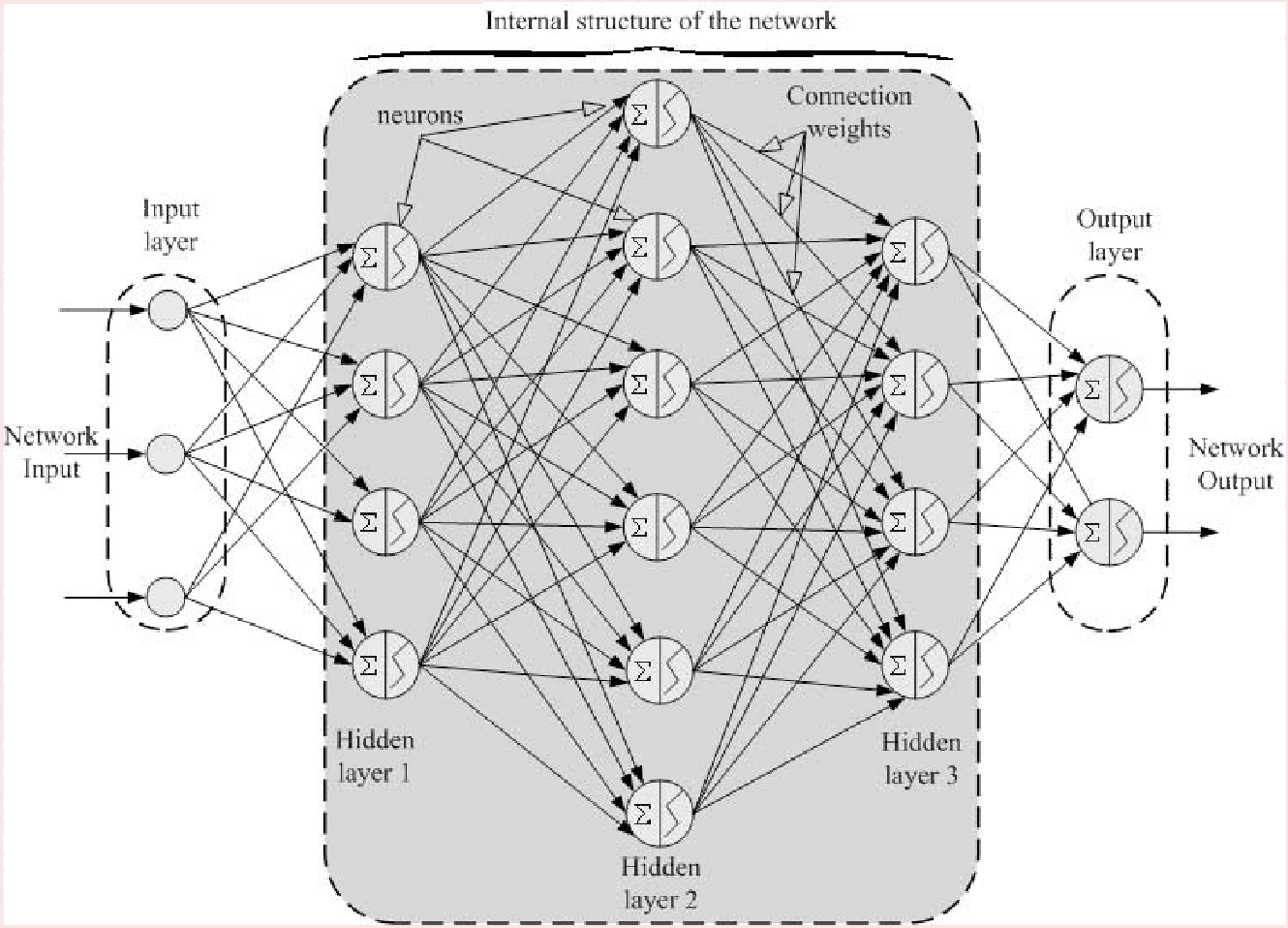
Binary
Continuous

Learning Paradigms

Supervised
Unsupervised
Hybrid

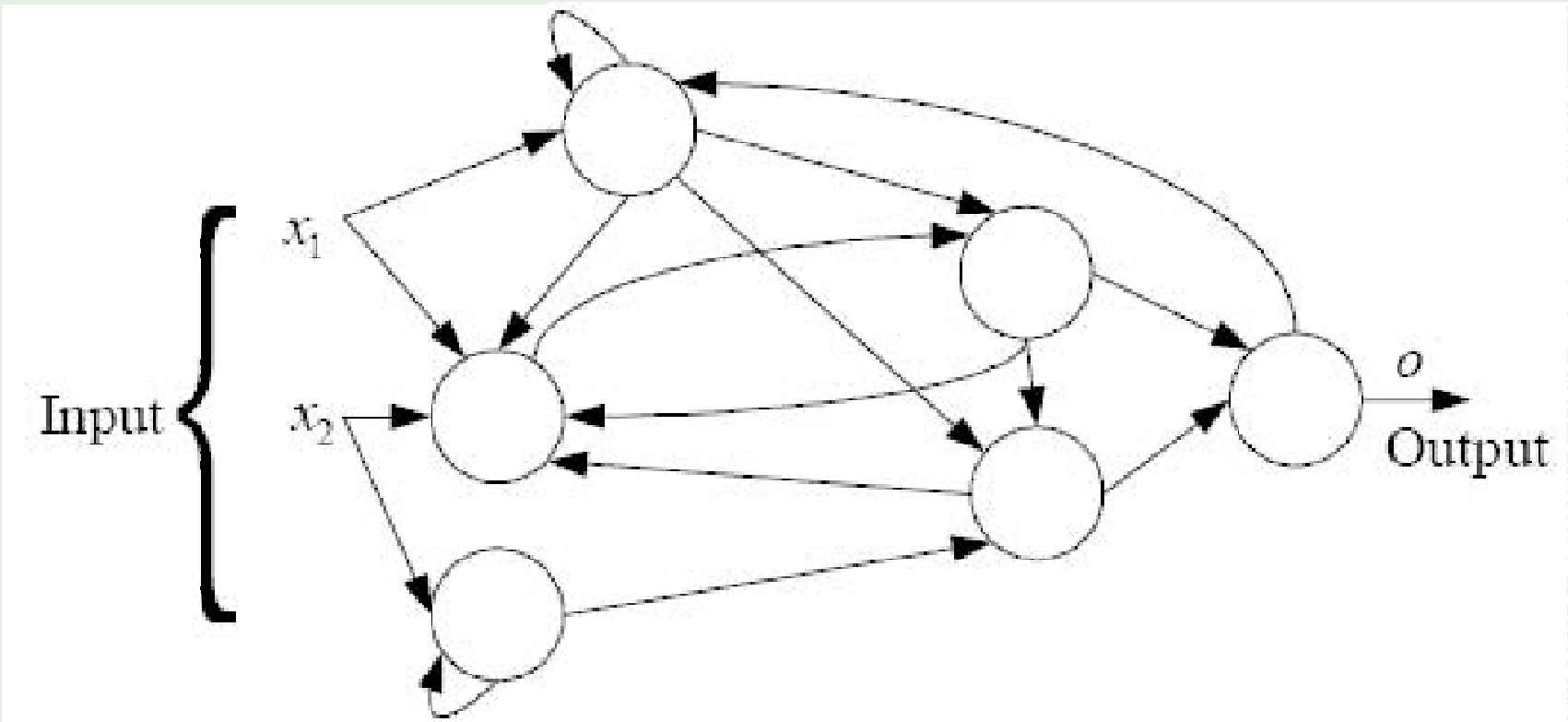
Neural Network Topologies

Feedforward Flow of Information



Neural Network Topologies (cont.)

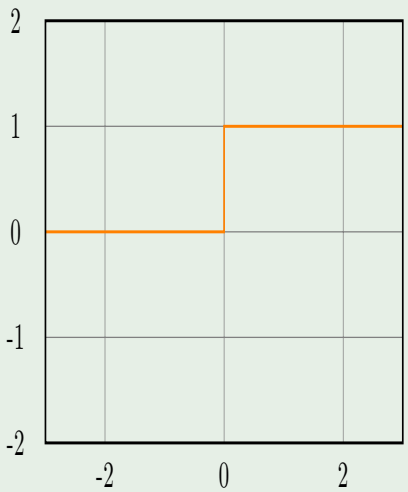
Recurrent Flow of Information



Binary Activation Functions

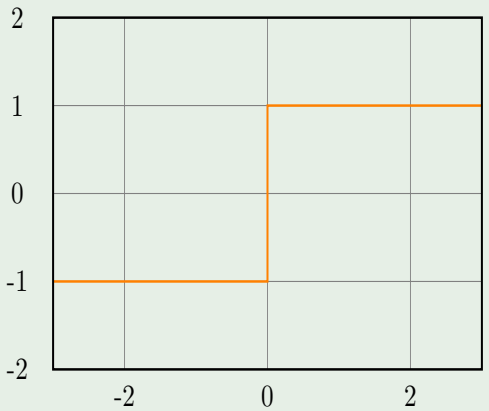
Step Function

$$\text{step}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



Signum Function

$$\text{sigum}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{otherwise} \end{cases}$$

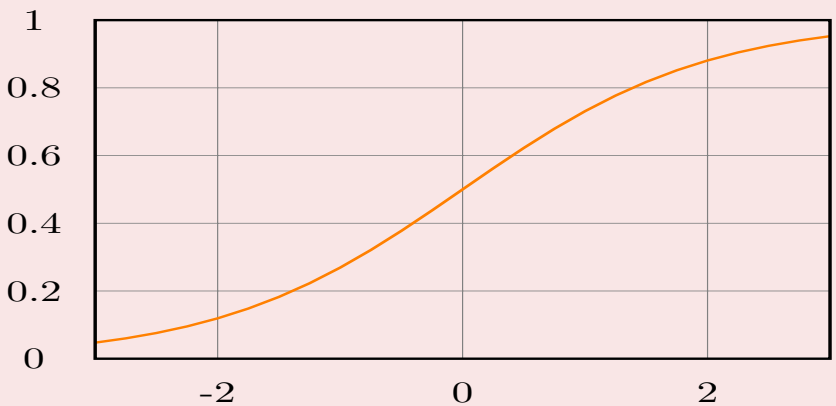


Differentiable Activation Functions

Differentiable functions

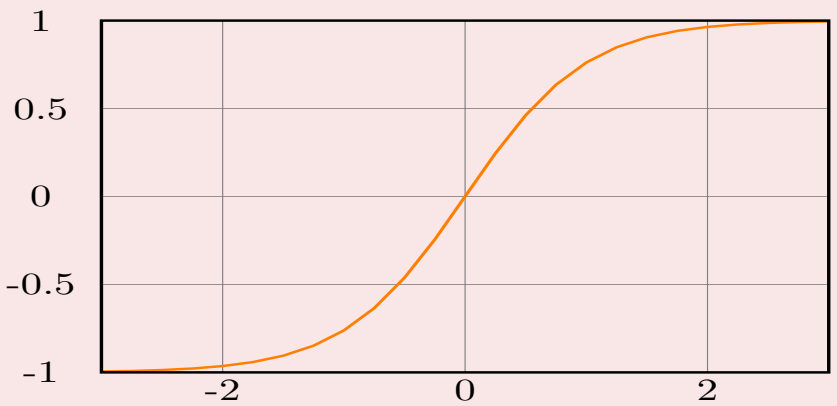
Sigmoid function

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



Hyperbolic tangent

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

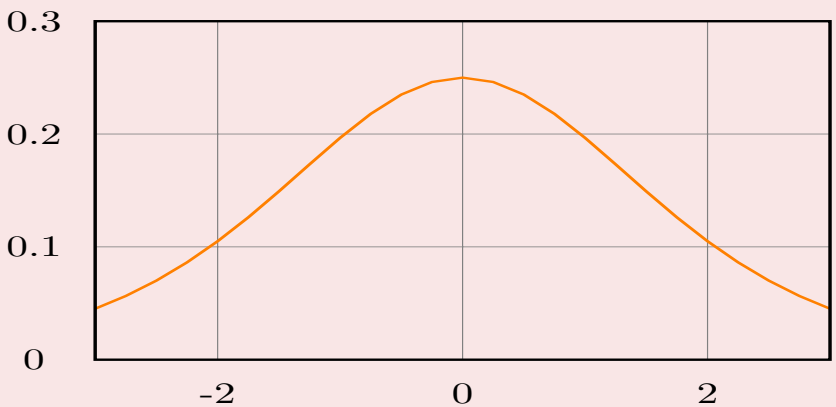


Differentiable Activation Functions (cont.)

Differentiable functions

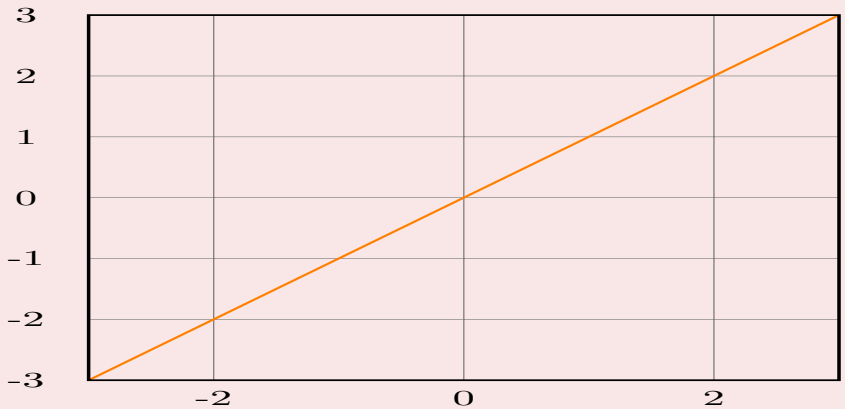
Sigmoid derivative

$$\text{sigderiv}(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$



Linear function

$$\text{lin}(x) = x$$



Learning Paradigms

Supervised Learning

- Multilayer perceptrons
- Radial basis function networks
- Modular neural networks
- LVQ (learning vector quantization)

Unsupervised Learning

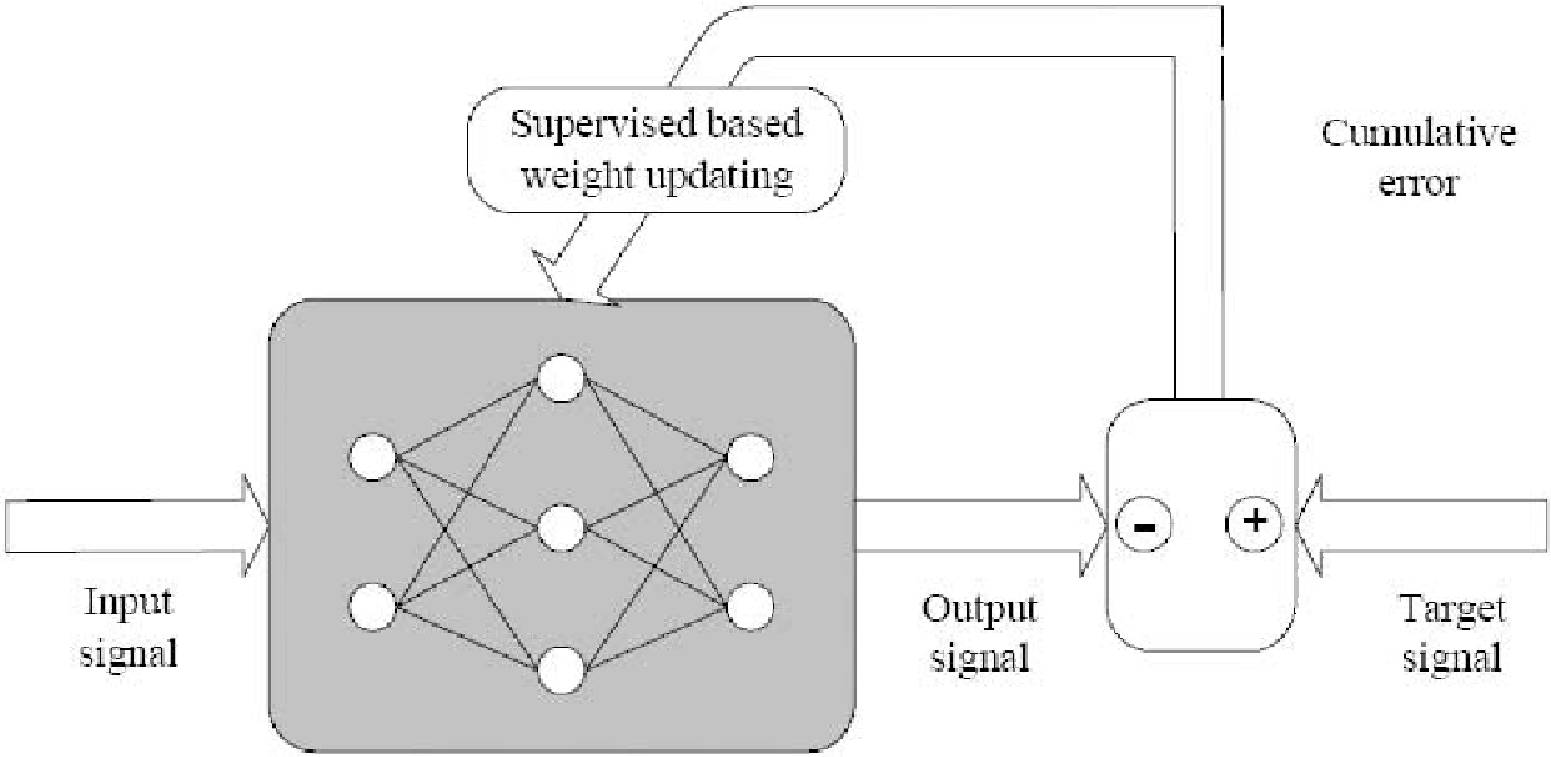
- Competitive learning networks
- Kohonen self-organizing networks
- ART (adaptive resonant theory)

Others

- Autoassociative memories (Hopfield networks)

Supervised Learning

- Training by example; i.e., priori known desired output for each input pattern.
- Particularly useful for feedforward networks.



Supervised Learning (cont.)

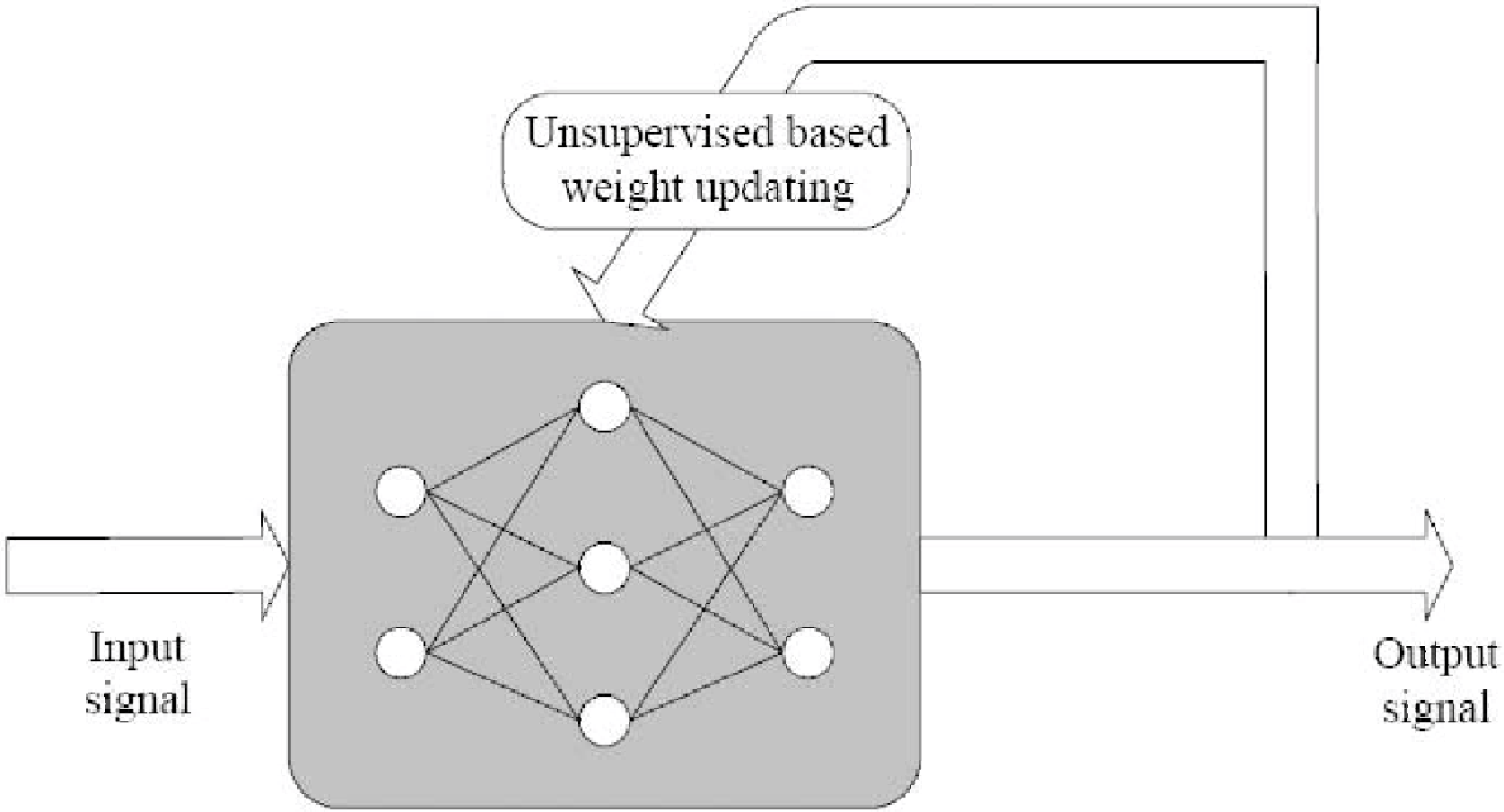
Training Algorithm

1. Compute error between desired and actual outputs
2. Use the error through a learning rule (e.g., gradient descent) to adjust the network's connection weights
3. Repeat steps 1 and 2 for input/output patterns to complete one epoch
4. Repeat steps 1 to 3 until maximum number of epochs is reached or an acceptable training error is reached

Unsupervised Learning

- No priori known desired output.
- In other words, training data composed of input patterns only.
- Network uses training patterns to discover emerging collective properties and organizes the data into clusters.

Unsupervised Learning: Graphical Illustration



Unsupervised Learning (cont.)

Unsupervised Training

1. Training data set is presented at the input layer
2. Output nodes are evaluated through a specific criterion
3. Only weights connected to the winner node are adjusted
4. Repeat steps 1 to 3 until maximum number of epochs is reached or the connection weights reach steady state

Rationale

- Competitive learning strengths the connection between the incoming pattern at the input layer and the winning output node.
- The weights connected to each output node can be regarded as the center of the cluster associated to that node.

Unsupervised Learning (cont.)

Unsupervised Training

1. Training data set is presented at the input layer
2. Output nodes are evaluated through a specific criterion
3. Only weights connected to the winner node are adjusted
4. Repeat steps 1 to 3 until maximum number of epochs is reached or the connection weights reach steady state

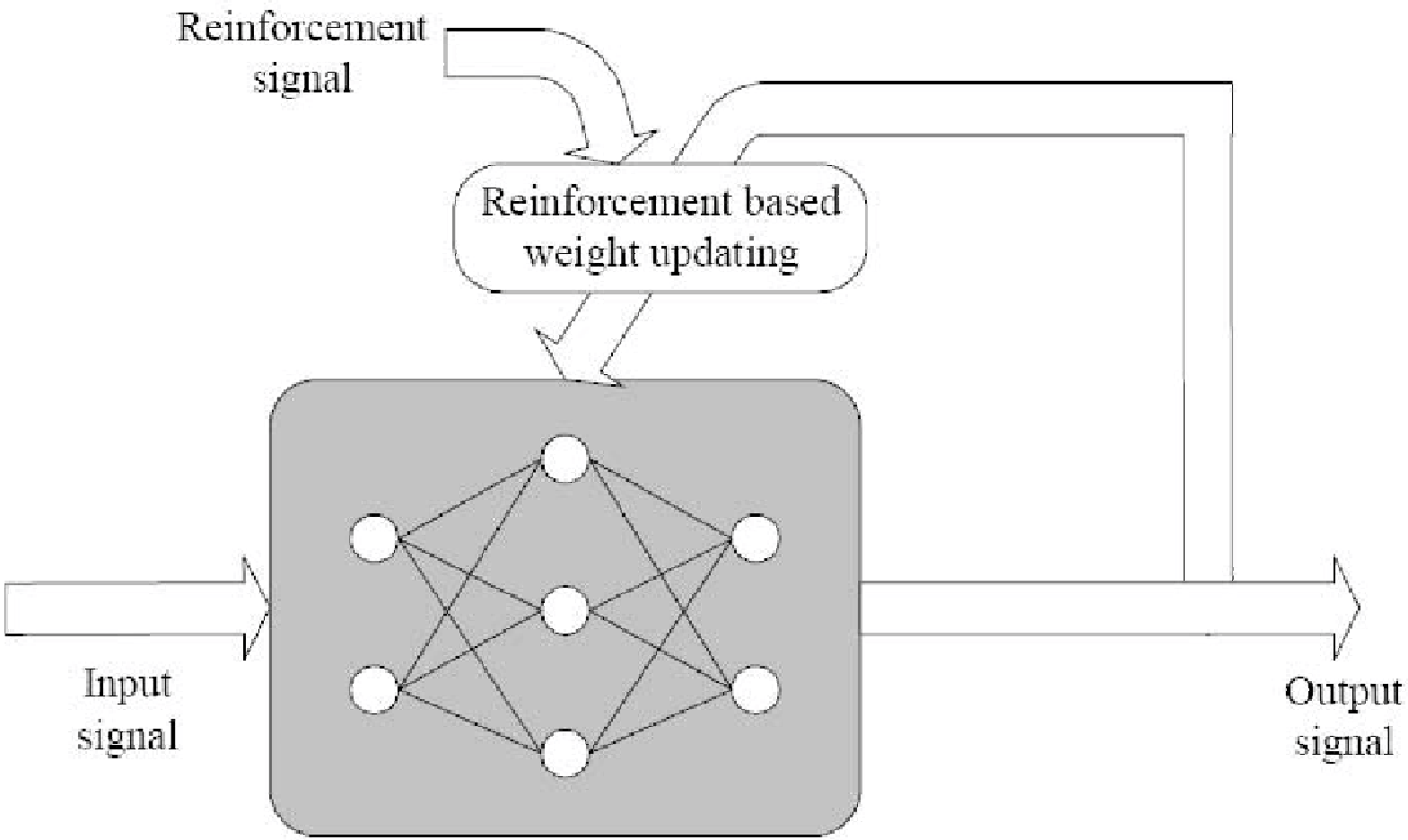
Rationale

- Competitive learning strengths the connection between the incoming pattern at the input layer and the winning output node.
- The weights connected to each output node can be regarded as the center of the cluster associated to that node.

Reinforcement Learning

- Reinforcement learning mimics the way humans adjust their behavior when interacting with physical systems (e.g., learning to ride a bike).
- Network's connection weights are adjusted according to a qualitative and not quantitative feedback information as a result of the network's interaction with the environment or system.
- The qualitative feedback signal simply informs the network whether or not the system reacted "well" to the output generated by the network.

Reinforcement Learning: Graphical Representation



Reinforcement Learning

Reinforcement Training Algorithm

1. Present training input pattern network
2. Qualitatively evaluate system's reaction to network's calculated output
   - If response is "Good", the corresponding weights led to that output are strengthened
   - If response is "Bad", the corresponding weights are weakened.

Fundamentals of ANNs

Late 1940's : McCulloch Pitt Model (by McCulloch and Pitt)

Late 1950's – early 1960's : Perceptron (by Roseblatt)

Mid 1960's : Adaline (by Widrow)

Mid 1970's : Back Propagation Algorithm - BPL I (by Werbos)

Mid 1980's : BPL II and Multi Layer Perceptron (by Rumelhart and Hinton)

Fundamentals of ANNs

Late 1940's : McCulloch Pitt Model (by McCulloch and Pitt)

Late 1950's – early 1960's : Perceptron (by Roseblatt)

Mid 1960's : Adaline (by Widrow)

Mid 1970's : Back Propagation Algorithm - BPL I (by Werbos)

Mid 1980's : BPL II and Multi Layer Perceptron (by Rumelhart and Hinton)

Fundamentals of ANNs

Late 1940's : McCulloch Pitt Model (by McCulloch and Pitt)

Late 1950's – early 1960's : Perceptron (by Roseblatt)

Mid 1960's : Adaline (by Widrow)

Mid 1970's : Back Propagation Algorithm - BPL I (by Werbos)

Mid 1980's : BPL II and Multi Layer Perceptron (by Rumelhart and Hinton)

Fundamentals of ANNs

Late 1940's : McCulloch Pitt Model (by McCulloch and Pitt)

Late 1950's – early 1960's : Perceptron (by Roseblatt)

Mid 1960's : Adaline (by Widrow)

Mid 1970's : Back Propagation Algorithm - BPL I (by Werbos)

Mid 1980's : BPL II and Multi Layer Perceptron (by Rumelhart and Hinton)

Fundamentals of ANNs

Late 1940's : McCulloch Pitt Model (by McCulloch and Pitt)

Late 1950's – early 1960's : Perceptron (by Roseblatt)

Mid 1960's : Adaline (by Widrow)

Mid 1970's : Back Propagation Algorithm - BPL I (by Werbos)

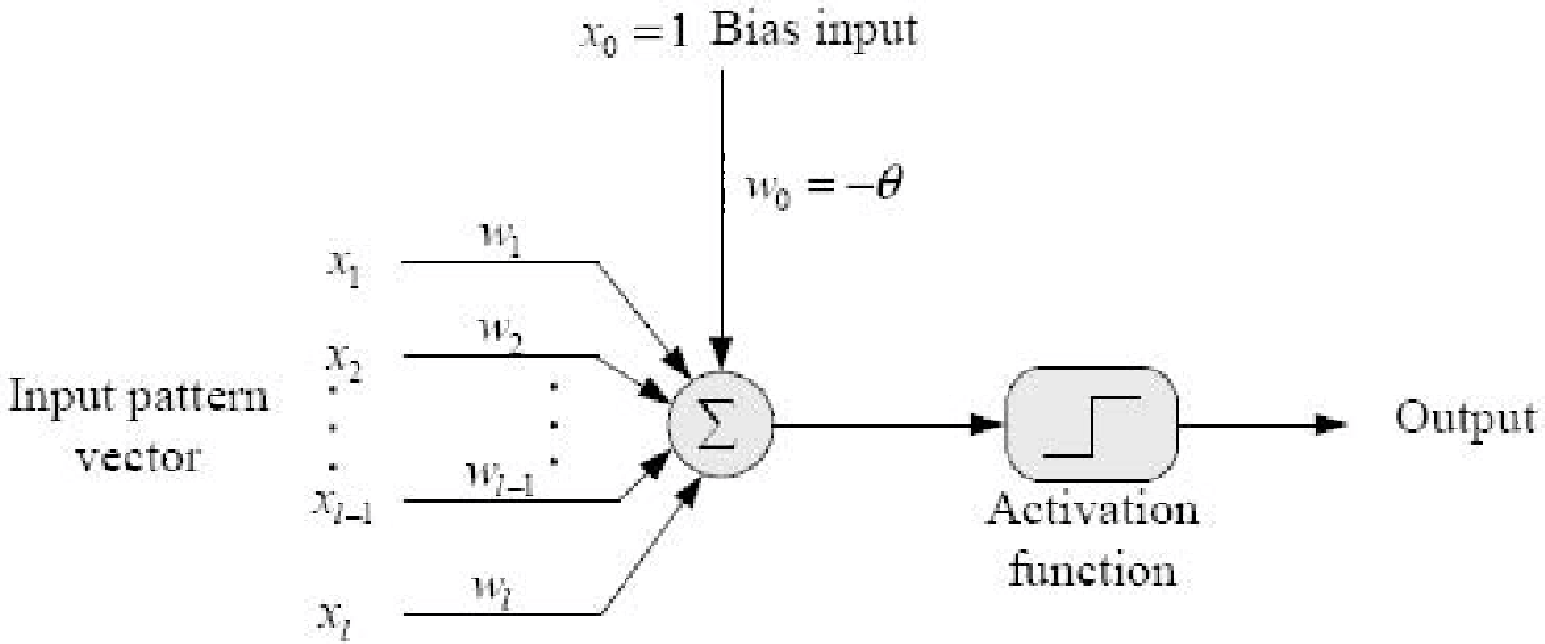
Mid 1980's : BPL II and Multi Layer Perceptron (by Rumelhart and Hinton)

McCulloch-Pitts Model

Overview

- First serious attempt to model the computing process of the biological neuron.
- The model is composed of one neuron only.
- Limited computing capability.
- No learning capability.

McCulloch-Pitts Model: Architecture



McCulloch-Pitts Models (cont.)

Functionality

1. I input signals presented to the network: $x_1, x_2, \ldots, x_I$.
2. I hard-coded weights, $w_1, w_2, \ldots, w_I$, and bias θ, are applied to compute the neuron's net sum: $\sum_{i=1}^{I} w_i I_i - \theta$.
3. A binary activation function f is applied to the neuron's net sum to calculate the node's output o: $o = f\left(\sum_{i=1}^{I} w_i x_i - \theta\right)$.

McCulloch-Pitts Models (cont.)

Remarks

- It is sometimes simpler and more convenient to introduce a virtual input $x_0 = 1$ and assigning its corresponding weight $w_0 = -\theta$. Then,

$$o = f\left(\sum_{i=0}^{I} w_i x_i\right) \text{ with } x_0 = 1, w_0 = -\theta$$

- Synaptic weights are not updated due to the lack of a learning mechanism.

Perceptron

Overview

- Uses supervised learning to adjust its weights in response to a comparative signal between the network's actual output and the target output.
- Mainly designed to classify linearly separable patterns.

Definition: Linear Separation

Patterns are linearly separable means that there exists a hyperplanar multidimensional decision boundary that classifies the patterns into two classes.

Perceptron

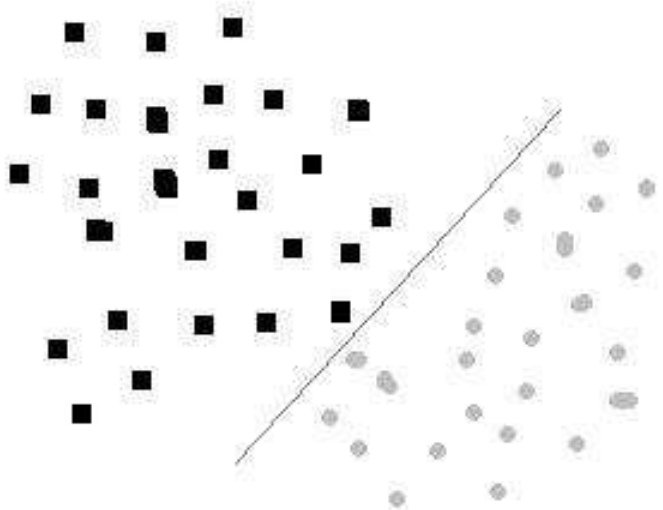
Overview

- Uses supervised learning to adjust its weights in response to a comparative signal between the network's actual output and the target output.
- Mainly designed to classify linearly separable patterns.

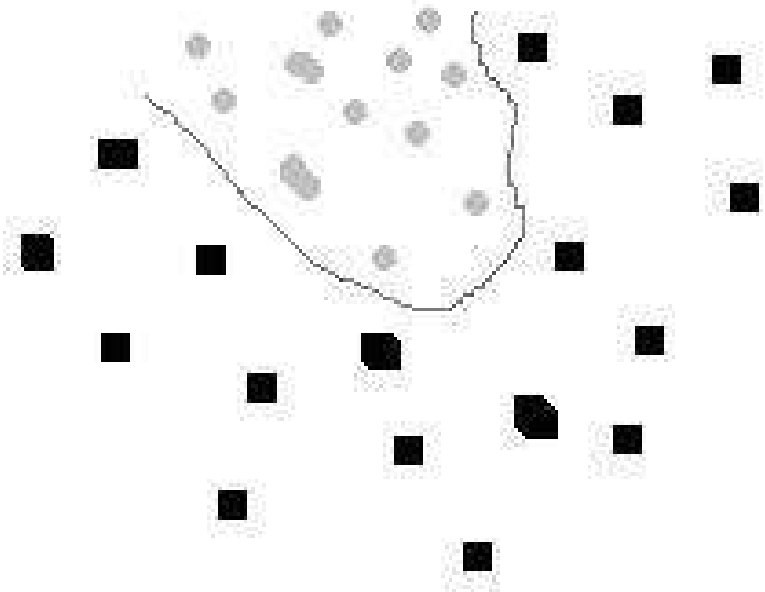
Definition: Linear Separation

Patterns are linearly separable means that there exists a hyperplanar multidimensional decision boundary that classifies the patterns into two classes.

Linearly Separable Patterns



Non-Linearly Separable Patterns

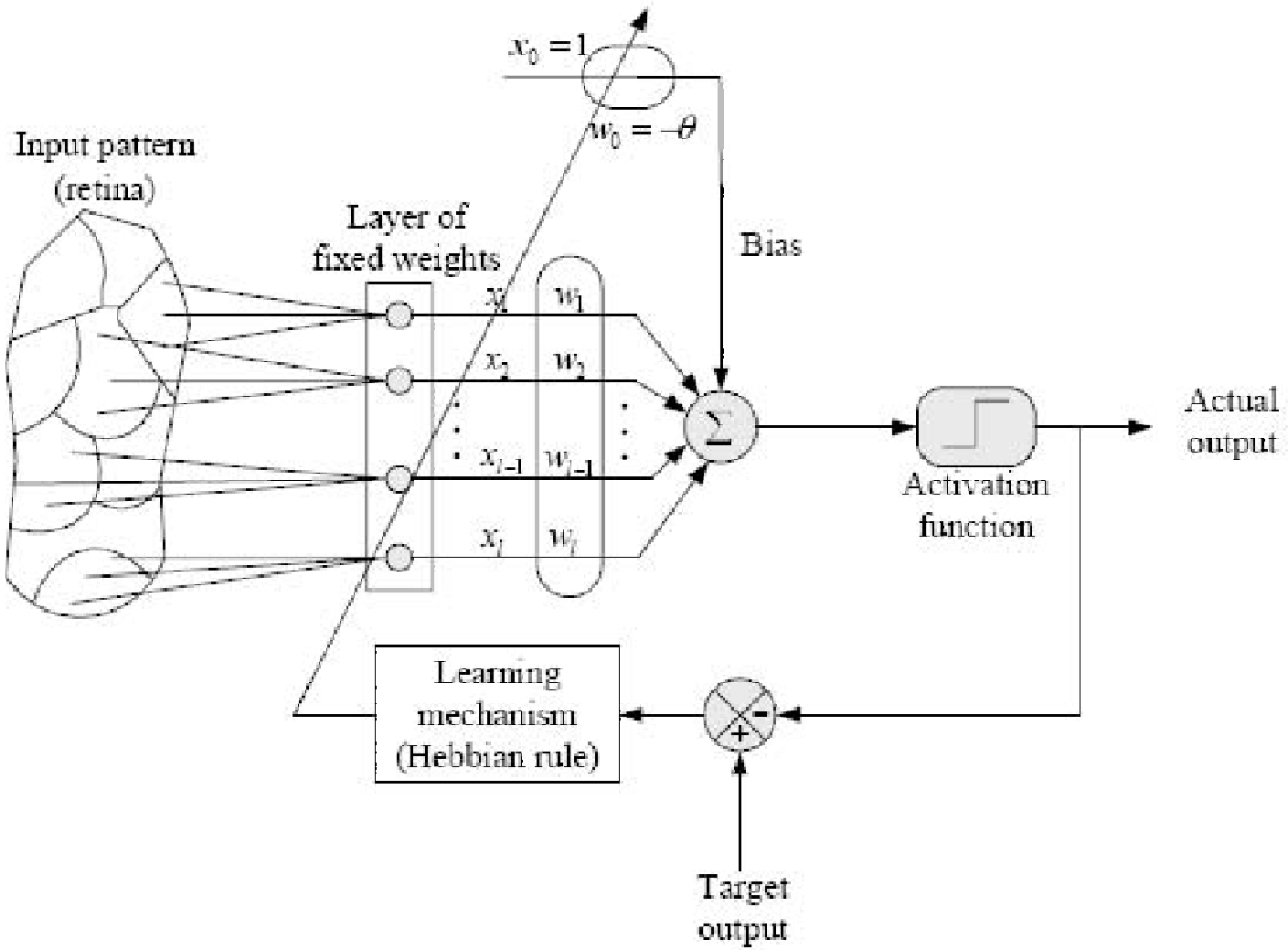


Perceptron

Remarks

- One neuron (one output)
- I input signals: $x_1, x_2, \ldots, x_I$
- Adjustable weights $w_1, w_2, \ldots, w_I$, and bias θ
- Binary activation function; i.e., step or hard limiter function

Perceptron: Architecture

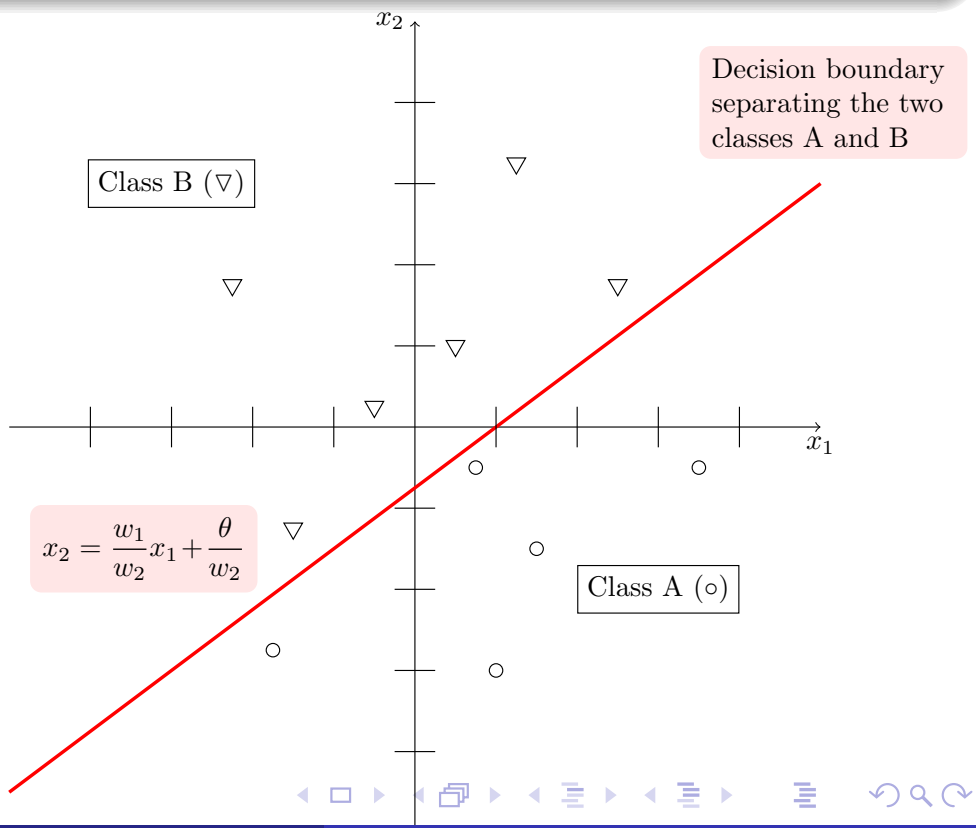


Perceptron (cont.)

Perceptron Convergence Theorem

If the training set is linearly separable, there exists a set of weights for which the training of the Perceptron will converge in a finite time and the training patterns are correctly classified.

In the two-dimensional case, the theorem translates into finding the line defined by $w_1x_1 + w_2x_2 - \theta = 0$, which adequately classifies the training patterns.



Training Algorithm

1. Initialize weights and thresholds to small random values.
2. Choose an input-output pattern $(x^{(k)}, t^{(k)})$ from the training data.
3. compute the network's actual output $o^{(k)} = f\left(\sum_{i=1}^{I} w_i x_i^{(k)} - \theta\right)$.
4. Adjust the weights and bias according to the Perceptron learning rule: $\Delta w_i = \eta[t^{(k)} - o^{(k)}]x_i^{(k)}$, and $\Delta\theta = -\eta[t^{(k)} - o^{(k)}]$, where $\eta \in [0, 1]$ is the Perceptron's learning rate.

   If f is the the signum function, this becomes equivalent to:

$$\Delta w_i = \begin{cases} 2\eta t^{(k)} x_i^{(k)} & \text{, if } t^{(k)} \neq o^{(k)} \\ 0 & \text{, otherwise} \end{cases} \qquad \Delta\theta = \begin{cases} -2\eta t^{(k)} & \text{, if } t^{(k)} \neq o^{(k)} \\ 0 & \text{, otherwise} \end{cases}$$

5. If a whole epoch is complete, then pass to the following step; otherwise go to Step 2.
6. If the weights (and bias) reached steady state ($\Delta w_i \approx 0$)through the whole epoch, then stop the learning; otherwise go through one more epoch starting from Step 2.

Example

Problem Statement

- Classify the following patterns using $\eta = 0.5$:

  Class (1) with target value (-1) :$T = [2, 0]^T, U = [2, 2]^T, V = [1, 3]^T$

  Class (2) with target value $(+1)$:$X = [-1, 0]^T, Y = [-2, 0]^T, Z = [-1, 2]^T$

- Let the **initial weights** be $w_1 = -1, w_2 = 1, \theta = -1$.

- Thus, **initial boundary** is defined by $x_2 = x_1 - 1$.

Example

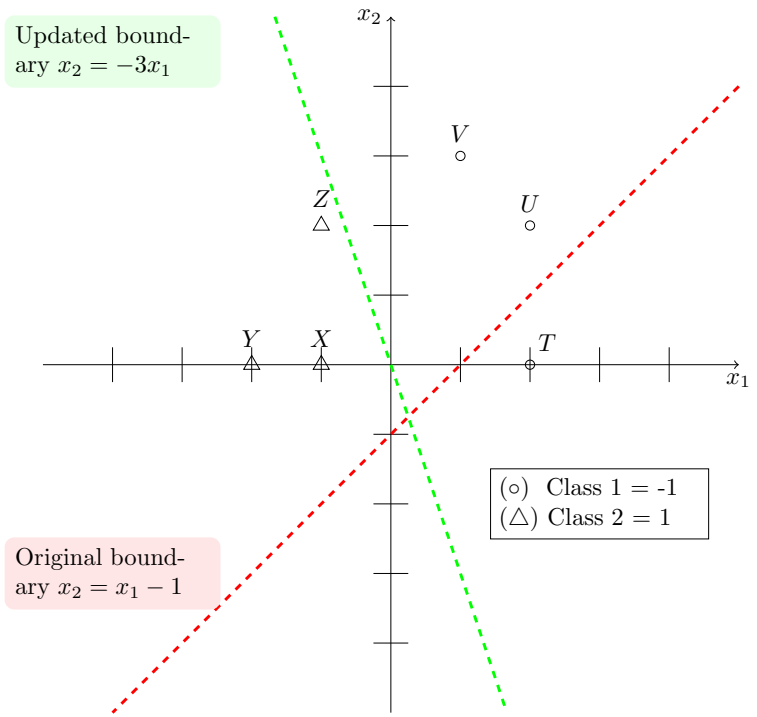
Solution

- T properly classified, but not U and V.
- Hence, training is needed.
- Let us start by selecting pattern U.

$$\text{sgn}(2 \times (-1) + 2 \times (1) + 1) = 1 \quad \Rightarrow \Delta w_1 = \Delta w_2 = -1 \times (2) = -2, \quad \Rightarrow \Delta\theta = +1$$

- **Updated boundary** is defined by $x_2 = -3x_1$.
- All patterns are now properly classified.

Example: Graphical Solution



Perceptron (cont.)

Remarks

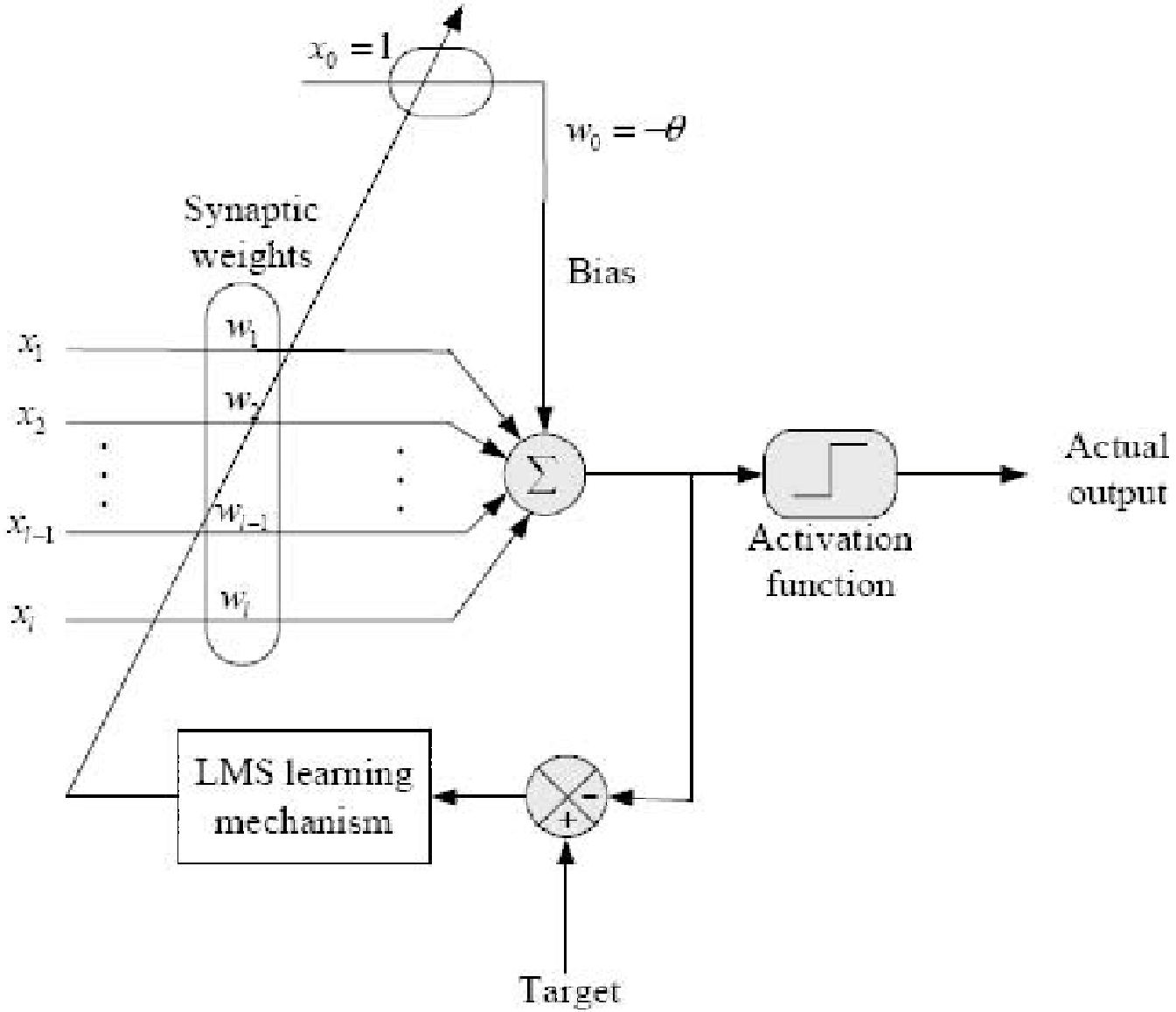
- Simple-layer perceptrons suffer from two major shortcomings:
  1. Cannot separate linearly non-separable patterns.
  2. Lack of generalization: once trained, it cannot adapt its weights to a new set of data.

Adaline (Adaptive Linear Neuron)

Overview

- More versatile than the Perceptron in terms of generalization.
- More powerful in terms of weight adaptation.
- An Adaline is composed of a linear combiner, a binary activation function (hard limiter), and adaptive weights.

Adaline: Graphical Illustration



Adaline (cont.)

Learning in an Adaline

- Adaline adjusts its weights according to the least mean squared (LMS) algorithm (also known as the Widrow-Hoff learning rule) through gradient descent optimization.
- At every iteration, the weights are adjusted by an amount proportional to the gradient of the cumulative error of the network $E(w)$.
  $\Rightarrow \Delta w = -\eta \nabla_w E(w)$

Adaline (cont.)

Learning in an Adaline (cont.)

- The network's cumulative error $E(w)$ for all patterns $(x^{(k)}, t^{(k)})$, $k = 1, 2, \ldots, n$. This is the error between the desired response $t^{(k)}$ and the linear combiner's output $(\sum_i w_i x_i^{(k)} - \theta)$.

$$E(w) = \sum_k \left[t^{(k)} - \left(\sum_i w_i x_i^{(k)} - \theta \right) \right]^2$$

- Hence, individual weights are updated as:

$$\Delta w_i = \eta \left(t^{(k)} - \sum_i w_i x_i^{(k)} \right) x_i^{(k)}.$$

Adaline (cont.)

Training Algorithm

1. Initialize weights and thresholds to small random values.
2. Choose an input-output pattern $(x^{(k)}, t^{(k)})$ from the training data.
3. Compute the linear combiner's output $r^{(k)} = \sum_{i=1} w_i x_i^{(k)} - \theta$.
4. Adjust the weights (and bias) according to the LMS rule as: $\Delta w_i = \eta \left(t^{(k)} - \sum_i w_i x_i^{(k)} \right) x_i^{(k)}$, where $\eta \in [0, 1]$ being the learning rate.
5. If a whole epoch is complete, then pass to the following step; otherwise go to Step 2.
6. If the weights (and bias) reached steady state ($\Delta w_i \approx 0$) through the whole epoch, then stop the learning; otherwise go through one more epoch starting from Step 2.

Adaline (cont.)

Advantages of the LMS Algorithm

- Easy to implement.
- Suitable for generalization, which is a missing feature in the Perceptron.

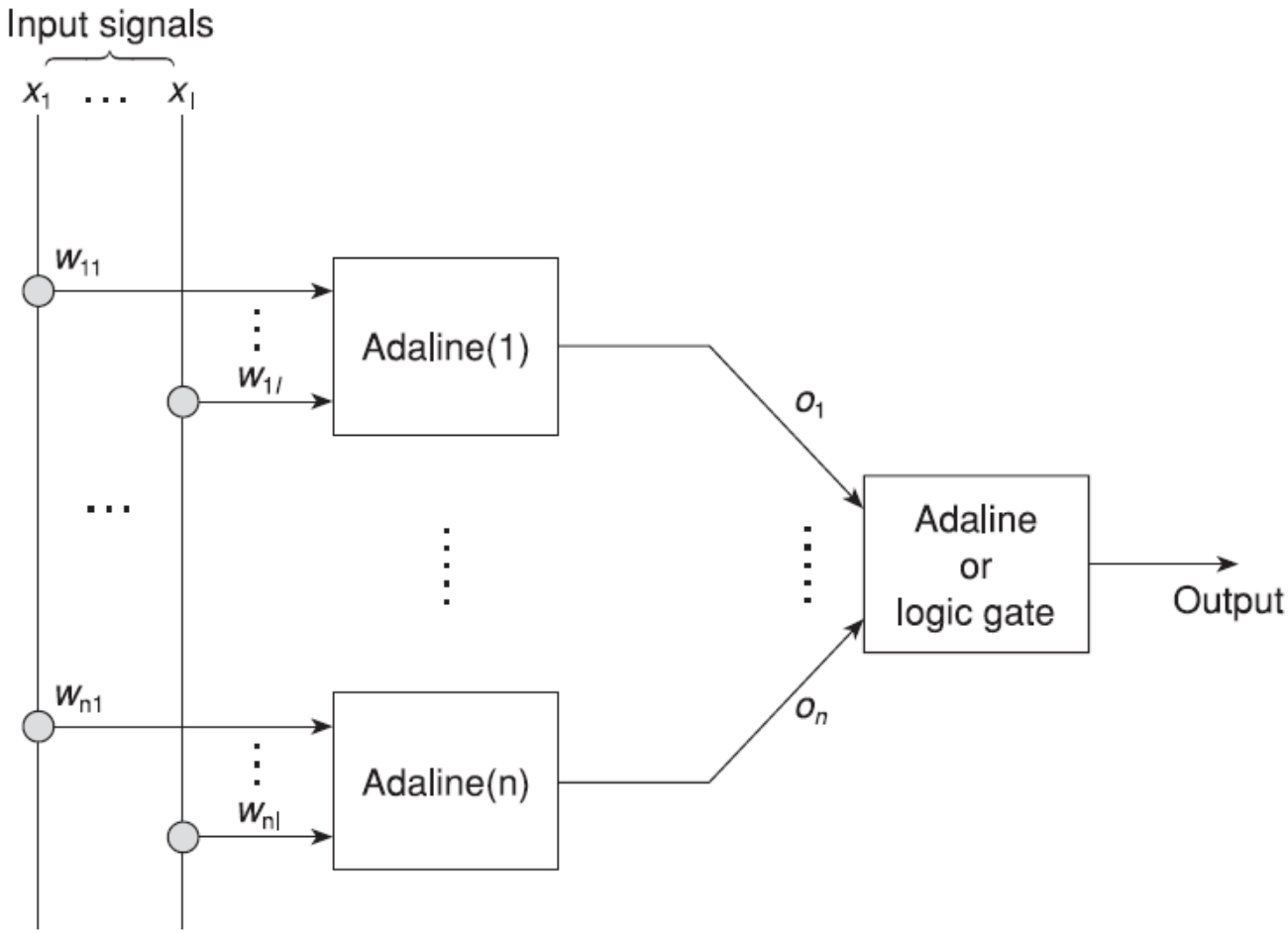
Madaline

Shortcoming of Adaline

The adaline, while having attractive training capabilities, suffers also (similarly to the perceptron) from the inability to train patterns belonging to nonlinearly separable spaces.

- Researchers have tried to circumvent this difficulty by setting cascade layers of adaline units.
- When first proposed, this seemingly attractive idea did not lead to much improvement due to the lack of an existing learning algorithm capable of adequately updating the synaptic weights of a cascade architecture of perceptrons.
- Other researchers were able to solve the nonlinear separability problem by combining in parallel a number of adaline units called a madaline.

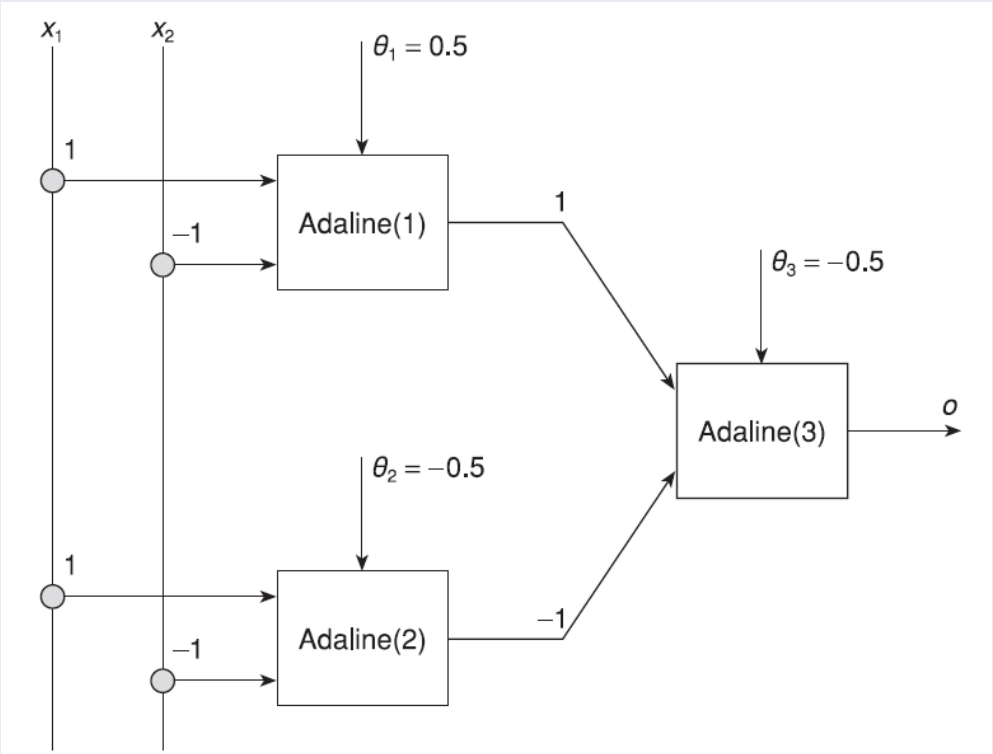
Madaline: Graphical Representation



Madaline: Example

- Solving the XOR logic function by combining in parallel two adaline units using the AND logic gate.

Graphical Solution



Related Binary Table

x_1	x_2	$o = x_1 XOR x_2$
0	0	1
0	1	-1
1	0	-1
1	1	1

Madaline (cont.)

Remarks

- Despite the successful implementation of the adaline and the madaline units in a number of applications, many researchers conjectured that to have successful connectionist computational tools, neural models should involve a topology with a number of cascaded layers.
- Schematics of the madaline implementation of the backpropagation learning algorithm to neural network models composed of multiplelayers of perceptrons.

Case Study: Binary Classification Using Perceptron

- We need to train the network using the following set of input and desired output training vectors:

$$(x^{(1)} = [1, -2, 0, -1]^T; t^{(1)} = -1),$$
$$(x^{(2)} = [0, 1.5, -0.5, -1]^T; t^{(2)} = -1),$$
$$(x^{(3)} = [-1, 1, 0.5, -1]^T; t^{(3)} = +1),$$

- Initial weight vector $w^{(1)} = [1, -1, 0, 0.5]^T$
- Learning rate $\eta = 0.1$

Epoch 1

Introducing the first input vector $x^{(1)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(1)} &= sgn(w^{(1)^T} x^{(1)}) \\ &= sgn([1, -1, 0, 0.5][1, -2, 0, -1]^T) \\ &= +1 \neq t^{(1)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(2)} &= w^{(1)} + \eta[t^{(1)} - o^{(1)}]x^{(1)} \\ &= w^{(1)} + 0.1(-2)x^{(1)} \\ &= [0.8, -0.6, 0, 0.7]^T \end{aligned}$$

Epoch 1

Introducing the first input vector $x^{(2)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(2)} &= sgn(w^{(2)^T} x^{(2)}) \\ &= sgn([0.8, -0.6, 0, 0.7][0, 1.5, -0.5, -1]^T) \\ &= -1 = t^{(2)}, \end{aligned}$$

- Updating weight vector

$$w^{(3)} = w^{(2)}$$

Epoch 1

Introducing the first input vector $x^{(3)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(3)} &= sgn(w^{(3)^T} x^{(3)}) \\ &= sgn([0.8, -0.6, 0, 0.7][-1, 1, 0.5, -1]^T) \\ &= -1 \neq t^{(3)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(4)} &= w^{(3)} + \eta[t^{(3)} - o^{(3)}]x^{(3)} \\ &= w^{(3)} + 0.1(2)x^{(3)} \\ &= [0.6, -0.4, 0.1, 0.5]^T \end{aligned}$$

Epoch 2

We reuse the training set $(x^{(1)}, t^{(1)}), (x^{(2)}, t^{(2)})$ and $(x^{(3)}, t^{(3)})$ as $(x^{(4)}, t^{(4)}), (x^{(5)}, t^{(5)})$ and $(x^{(6)}, t^{(6)})$, respectively.

Introducing the first input vector $x^{(4)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(4)} &= sgn(w^{(4)^T} x^{(4)}) \\ &= sgn([0.6, -0.4, 0.1, 0.5][1, -2, 0, -1]^T) \\ &= +1 \neq t^{(4)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(5)} &= w^{(4)} + \eta[t^{(4)} - o^{(4)}]x^{(4)} \\ &= w^{(4)} + 0.1(-2)x^{(4)} \\ &= [0.4, 0, 0.1, 0.7]^T \end{aligned}$$

Epoch 2

Introducing the first input vector $x^{(5)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(5)} &= sgn(w^{(5)^T} x^{(5)}) \\ &= sgn([0.4, 0, 0.1, 0.7][0, 1.5, -0.5, -1]^T) \\ &= -1 = t^{(5)}, \end{aligned}$$

- Updating weight vector

$$w^{(6)} = w^{(5)}$$

Epoch 2

Introducing the first input vector $x^{(6)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(6)} &= sgn(w^{(6)^T} x^{(6)}) \\ &= sgn([0.4, 0, 0.1, 0.7][-1, 1, 0.5, -1]^T) \\ &= -1 \neq t^{(6)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(7)} &= w^{(6)} + \eta[t^{(6)} - o^{(6)}]x^{(6)} \\ &= w^{(6)} + 0.1(2)x^{(6)} \\ &= [0.2, 0.2, 0.2, 0.5]^T \end{aligned}$$

Epoch 3

We reuse the training set $(x^{(1)}, t^{(1)}), (x^{(2)}, t^{(2)})$ and $(x^{(3)}, t^{(3)})$ as $(x^{(7)}, t^{(7)}), (x^{(8)}, t^{(8)})$ and $(x^{(9)}, t^{(9)})$, respectively.

Introducing the first input vector $x^{(7)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(7)} &= sgn(w^{(7)^T} x^{(7)}) \\ &= sgn([0.2, 0.2, 0.2, 0.5][1, -2, 0, -1]^T) \\ &= -1 = t^{(7)}, \end{aligned}$$

- Updating weight vector

$$w^{(8)} = w^{(7)}$$

Epoch 3

Introducing the first input vector $x^{(8)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(8)} &= sgn(w^{(8)^T} x^{(8)}) \\ &= sgn([0.2, 0.2, 0.2, 0.5][0, 1.5, -0.5, -1]^T) \\ &= -1 = t^{(8)}, \end{aligned}$$

- Updating weight vector

$$w^{(9)} = w^{(8)}$$

Epoch 3

Introducing the first input vector $x^{(9)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(9)} &= sgn(w^{(9)^T} x^{(9)}) \\ &= sgn([0.2, 0.2, 0.2, 0.5][-1, 1, 0.5, -1]^T) \\ &= -1 \neq t^{(9)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(10)} &= w^{(9)} + \eta[t^{(9)} - o^{(9)}]x^{(9)} \\ &= w^{(9)} + 0.1(2)x^{(9)} \\ &= [0, 0.4, 0.3, 0.3]^T \end{aligned}$$

Epoch 4

We reuse the training set $(x^{(1)}, t^{(1)}), (x^{(2)}, t^{(2)})$ and $(x^{(3)}, t^{(3)})$ as $(x^{(10)}, t^{(10)}), (x^{(11)}, t^{(11)})$ and $(x^{(12)}, t^{(12)})$, respectively.

Introducing the first input vector $x^{(10)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(10)} &= sgn(w^{(10)^T} x^{(10)}) \\ &= sgn([0, 0.4, 0.3, 0.3][1, -2, 0, -1]^T) \\ &= -1 = t^{(10)}, \end{aligned}$$

- Updating weight vector

$$w^{(11)} = w^{(10)}$$

Epoch 4

Introducing the first input vector $x^{(11)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(11)} &= sgn(w^{(11)^T} x^{(11)}) \\ &= sgn([0, 0.4, 0.3, 0.3][0, 1.5, -0.5, -1]^T) \\ &= +1 \neq t^{(11)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(12)} &= w^{(11)} + \eta[t^{(11)} - o^{(11)}]x^{(11)} \\ &= w^{(11)} + 0.1(-2)x^{(11)} \\ &= [0, 0.1, 0.4, 0.5]^T \end{aligned}$$

Epoch 4

Introducing the first input vector $x^{(12)}$ to the network

- Computing the output of the network

$$\begin{aligned} o^{(12)} &= sgn(w^{(12)^T} x^{(12)}) \\ &= sgn([0, 0.1, 0.4, 0.5][-1, 1, 0.5, -1]^T) \\ &= -1 \neq t^{(12)}, \end{aligned}$$

- Updating weight vector

$$\begin{aligned} w^{(13)} &= w^{(12)} + \eta[t^{(12)} - o^{(12)}]x^{(12)} \\ &= w^{(12)} + 0.1(2)x^{(12)} \\ &= [-0.2, 0.3, 0.5, 0.3]^T \end{aligned}$$

Final Weight Vector

- Introducing the input vectors for another epoch will result in **no change to the weights** which indicates that $w^{(13)}$ is the solution for this problem;
- Final weight vector: $w = [w_1, w_2, w_3, w_4] = [-0.2, 0.3, 0.5, 0.3]$.

Major Classes of Neural Networks

Outline

- Multi-Layer Perceptrons (MLPs)
- Radial Basis Function Network
- Kohonen's Self-Organizing Network
- Hopfield Network

Multi-Layer Perceptrons (MLPs)

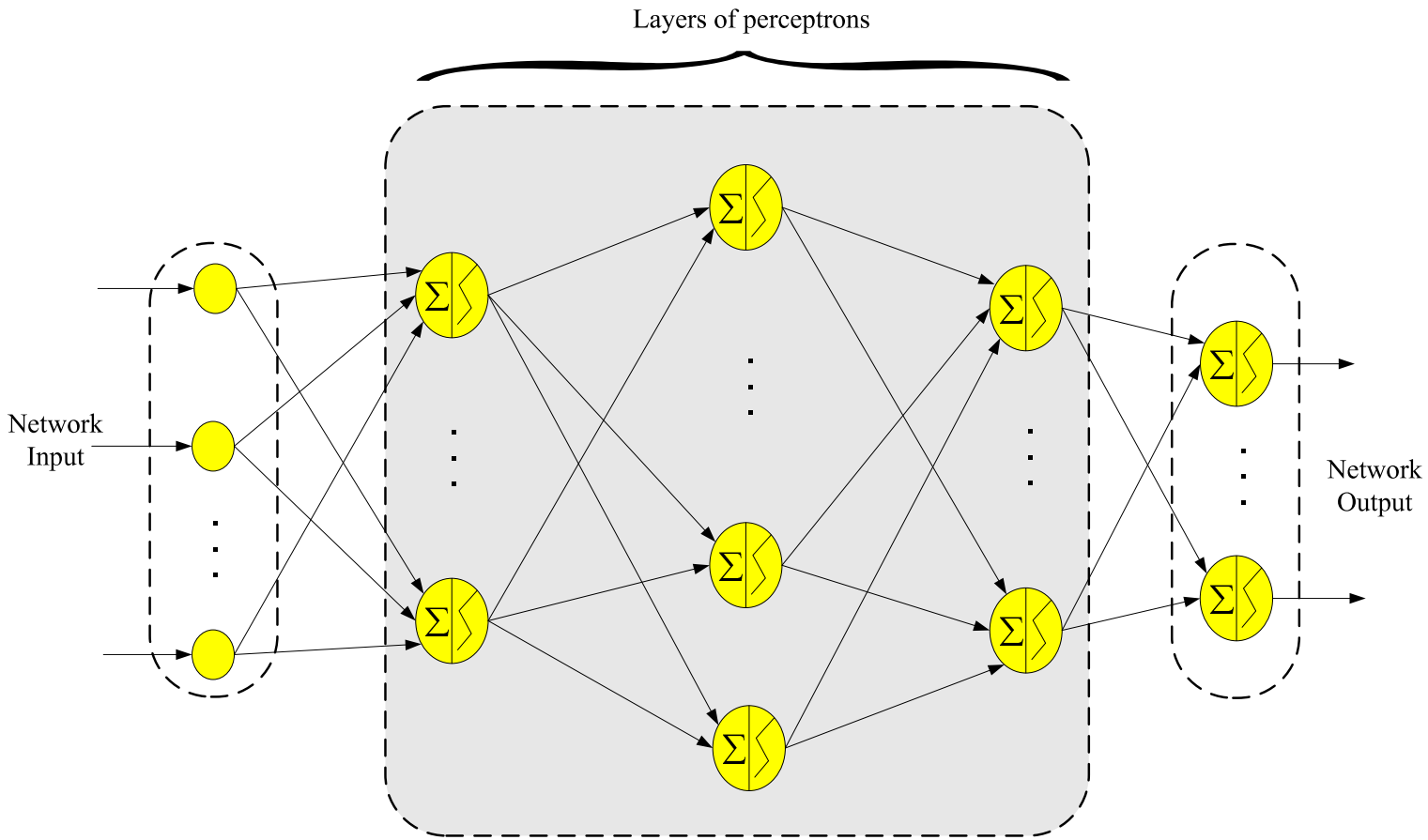
Background

- The perceptron lacks the important capability of recognizing patterns belonging to non-separable linear spaces.
- The madaline is restricted in dealing with complex functional mappings and multi-class pattern recognition problems.
- The multilayer architecture first proposed in the late sixties.

Background (cont.)

- MLP re-emerged as a solid connectionist model to solve a wide range of complex problems in the mid-eighties.
- This occurred following the reformulation of a powerful learning algorithm commonly called the Back Propagation Learning (BPL).
- It was later implemented to the multilayer perceptron topology with a great deal of success.

Schematic Representation of MLP Network



Backpropagation Learning Algorithm (BPL)

- The backpropagation learning algorithm is based on the **gradient descent technique** involving the **minimization of the network cumulative error**.

$$E(k) = \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

  - i represents i-th neuron of the output layer composed of a total number of q neurons.

- It is designed to **update the weights in the direction of the gradient descent of the cumulative error**.

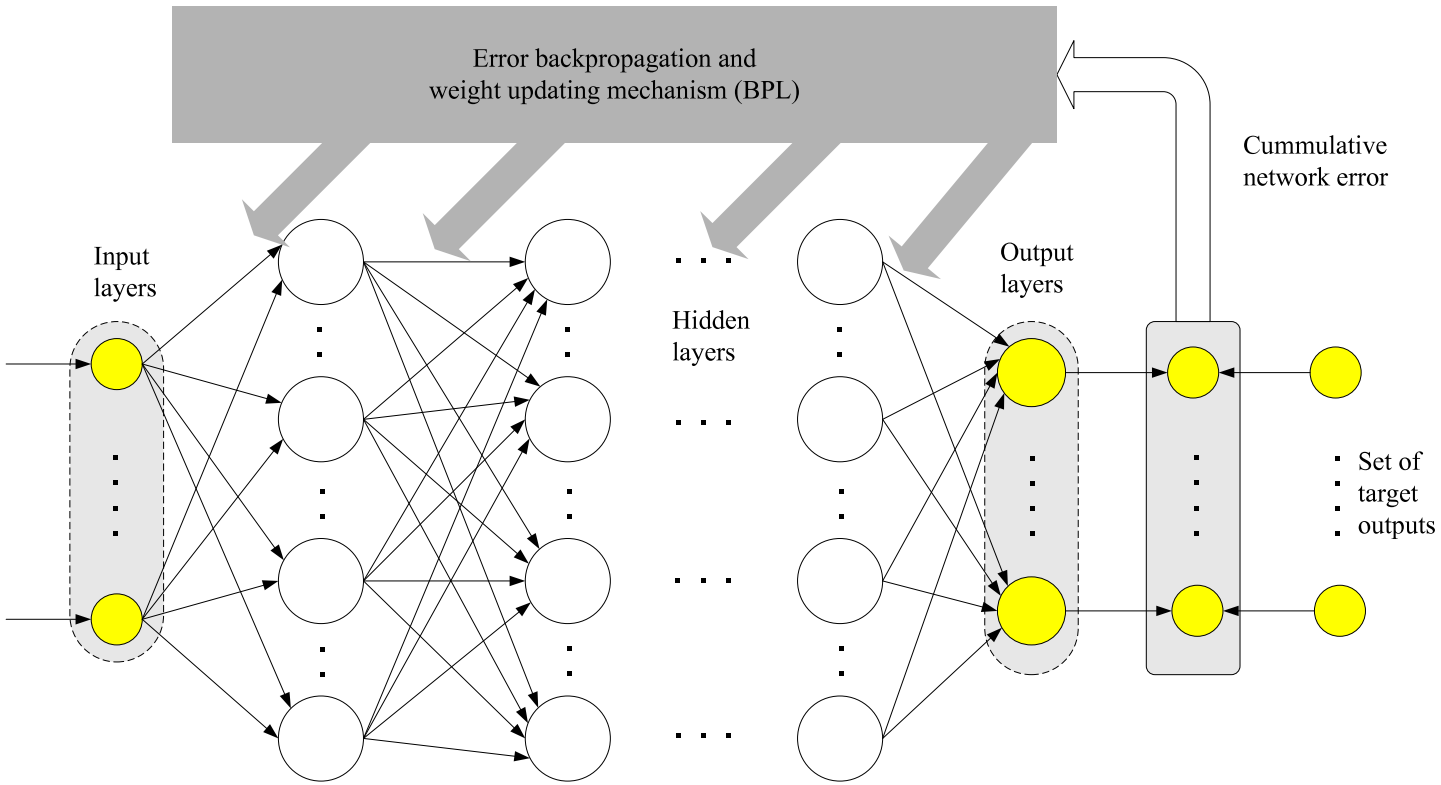
Backpropagation Learning Algorithm (cont.)

A Two-Stage Algorithm

1. First, patterns are presented to the network.
2. A feedback signal is then propagated backward with the main task of updating the weights of the layers connections according to the back-propagation learning algorithm.

BPL: Schematic Representation

- Schematic Representation of the MLP network illustrating the notion of error back-propagation



Backpropagation Learning Algorithm (cont.)

Objective Function

- Using the **sigmoid function** as the activation function for all the neurons of the network, we define E_c as

$$E_c = \sum_{k=1}^{n} E(k) = \frac{1}{2} \sum_{k=1}^{n} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

Backpropagation Learning Algorithm (cont.)

- The formulation of the **optimization problem** can now be stated as **finding the set of the network weights** that minimizes E_c or $E(k)$.

Objective Function: Off-Line Training

$$min_w E_c = min_w \frac{1}{2} \sum_{k=1}^{n} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

Objective Function: On-Line Training

$$min_w E(k) = min_w \frac{1}{2} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

BPL: On-Line Training

- Objective Function: $min_w E(k) = min_w \frac{1}{2} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$

Updating Rule for Connection Weights

$$\Delta w^{(l)} = -\eta \frac{\partial E(k)}{\partial w^l},$$

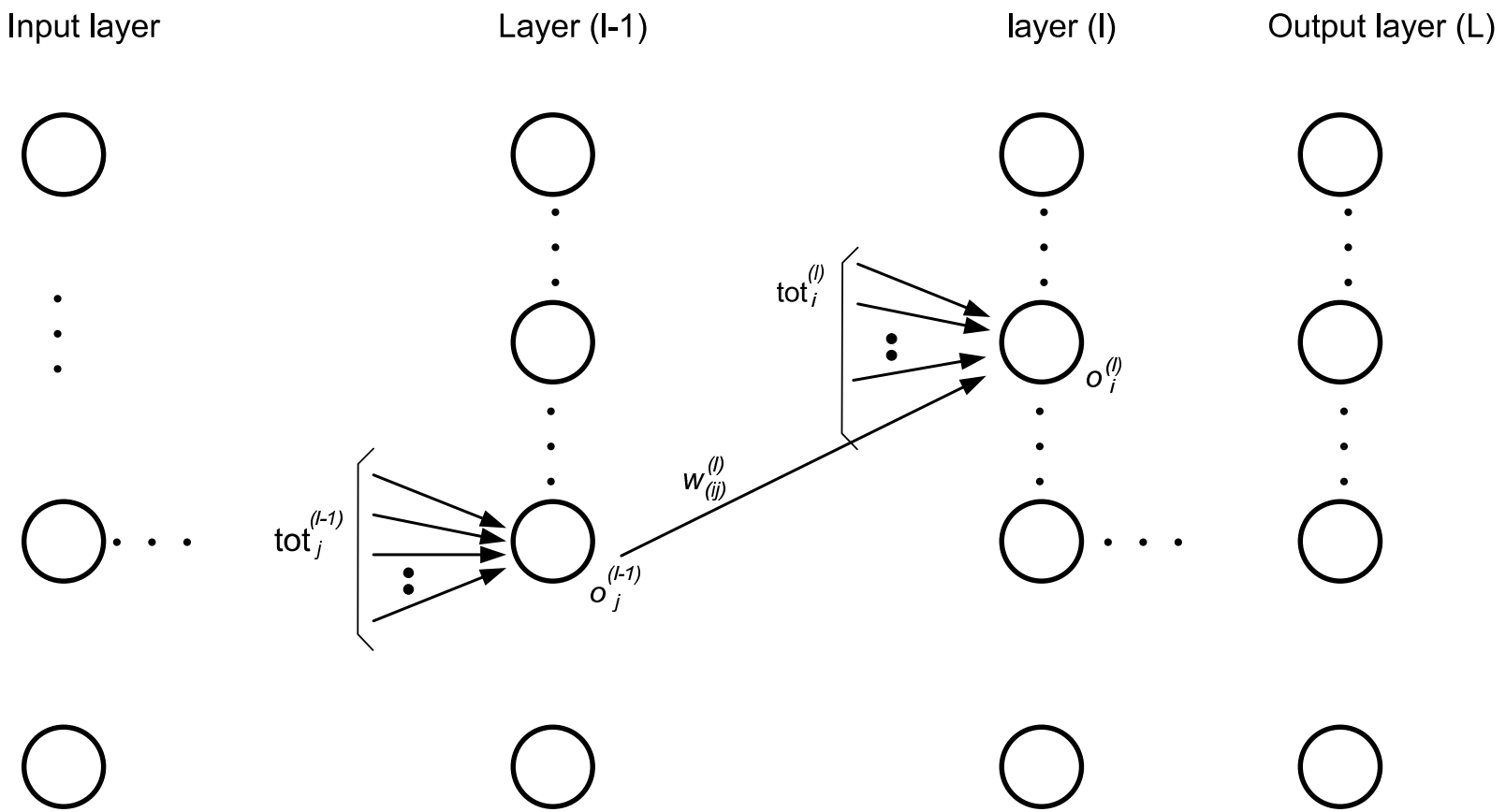
- l is layer (l-th) and η denotes the learning rate parameter,
- $\Delta w_{ij}^{(l)}$: the weight update for the connection linking the node j of layer $(l-1)$ to node i located at layer l.

BPL: On-Line Training (cont.)

Updating Rule for Connection Weights

- o_j^{l-1}: the output of the neuron j at layer $l-1$, the one located just before layer l,
- tot_i^l: the sum of all signals reaching node i at hidden layer l coming from previous layer $l-1$.

Illustration of Interconnection Between Layers of MLP



Interconnection Weights Updating Rules

- $\Delta w^{(l)} = \Delta w_{ij}^{(l)} = -\eta[\frac{\partial E(k)}{\partial o_i^{(l)}}][\frac{\partial o_i^{(l)}}{\partial tot_i^{(l)}}][\frac{\partial tot_i^{(l)}}{\partial w_{ij}^{(l)}}]$
- For the case where the layer (l) is the output layer (L):
  $\Delta w_{ij}^{(L)} = \eta[t_i - o_i^{(L)}][f'(tot)_i^{(L)}]o_j^{(L-1)}; \quad f'(tot)_i^{(l)} = \frac{\partial f(tot_i^{(l)})}{\partial tot_i^{(l)}}$
- By denoting $\delta_i^{(L)} = [t_i - o_i^{(L)}][f'(tot)_i^{(L)}]$ as being the **error signal** of the i-th node of the output layer, the weight update at layer (L) is as follows: $\Delta w_{ij}^{(L)} = \eta\delta_i^{(L)} o_j^{(L-1)}$
- In the case where f is the sigmoid function, the error signal becomes expressed as:
  $\delta_i^L = [(t_i - o_i^{(L)})o_i^{(L)}(1 - o_i^{(L)})]$

Interconnection Weights Updating Rules (cont.)

- Propagating the error backward now, and for the case where (l) represents a hidden layer ($l < L$), the expression of $\Delta w_{ij}^{(l)}$ becomes given by: $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$, where $\delta_i^{(l)} = f'(tot)_i^{(l)} \sum_{p=1}^{n_l} \delta_p^{l+1} w_{pi}^{l+1}$.
- Again when f is taken as the sigmoid function, $\delta_i^{(l)}$ becomes expressed as: $\delta_i^{(l)} = o_i^{(l)}(1 - o_i^{(l)}) \sum_{p=1}^{n_l} \delta_p^{l+1} w_{pi}^{l+1}$.

Updating Rules: Off-Line Training

- The weight update rule:

$$\Delta w^{(l)} = -\eta \frac{\partial E_c}{\partial w^l}.$$

- All previous steps outlined for developing the on-line update rules are reproduced here with the exception that $E(k)$ becomes replaced with E_c.

- In both cases though, once the network weights have reached steady state values, the training algorithm is said to converge.

Required Steps for Backpropagation Learning Algorithm

- Step 1: Initialize weights and thresholds to small random values.
- Step 2: Choose an input-output pattern from the training input-output data set $(x(k), t(k))$.
- Step 3: Propagate the k-th signal forward through the network and compute the output values for all i neurons at every layer (l) using $o_i^l(k) = f(\sum_{p=0}^{n_{l-1}} w_{ip}^l o_p^{l-1})$.
- Step 4: Compute the total error value $E = E(k) + E$ and the error signal $\delta_i^{(L)}$ using formulae $\delta_i^{(L)} = [t_i - o_i^{(L)}][f'(tot)_i^{(L)}]$.

Required Steps for BPL (cont.)

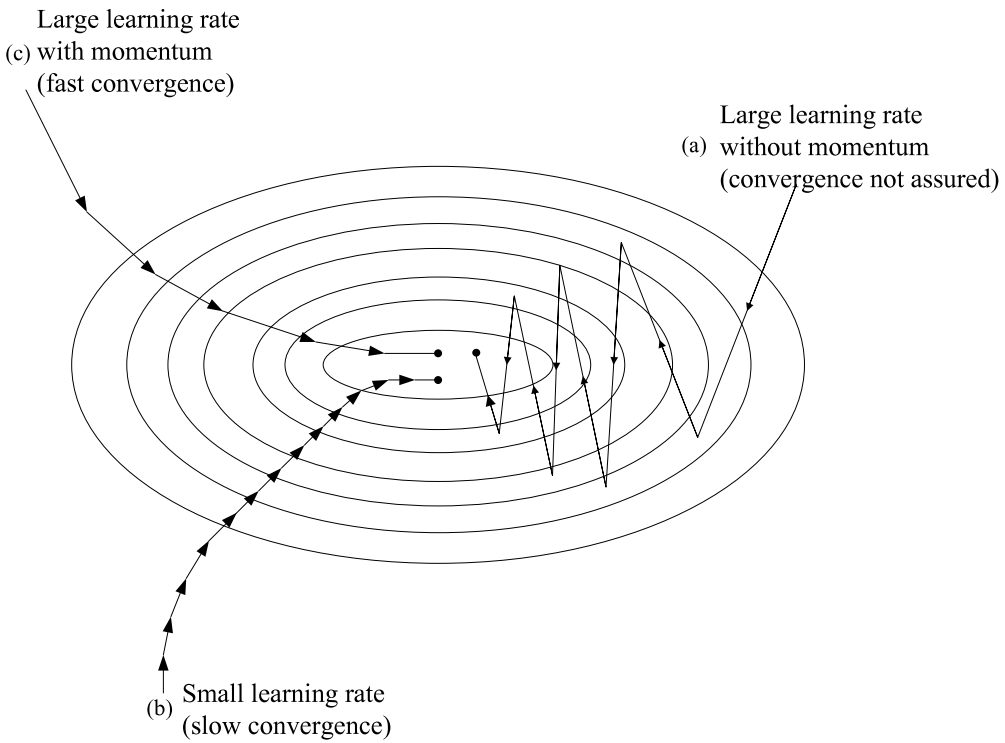
- Step 5: Update the weights according to $\Delta w_{ij}^{(l)} = -\eta \delta_i^{(l)} o_j^{(l-1)}$, for $l = L, \cdots, 1$ using $\delta_i^{(L)} = [t_i - o_i^{(L)}][f'(tot)_i^{(L)}]$ and proceeding backward using $\delta_i^{(l)} = o_i^l(1 - o_i^l) \sum_{p=1}^{n_l} \delta_p^{l+1} w_{pi}^{l+1}$ for $l < L$.
- Step 6: Repeat the process starting from step 2 using another exemplar. Once all exemplars have been used, we then reach what is known as one epoch training.
- Step 7: Check if the cumulative error E in the output layer has become less than a predetermined value. If so we say the network has been trained. If not, repeat the whole process for one more epoch.

Momentum

- The gradient descent requires by nature infinitesimal differentiation steps.
- For small values of the learning parameter η, this leads most often to a very slow convergence rate of the algorithm.
- Larger learning parameters have been known to lead to unwanted oscillations in the weight space.
- To avoid these issues, the concept of momentum has been introduced.

Momentum (cont.)

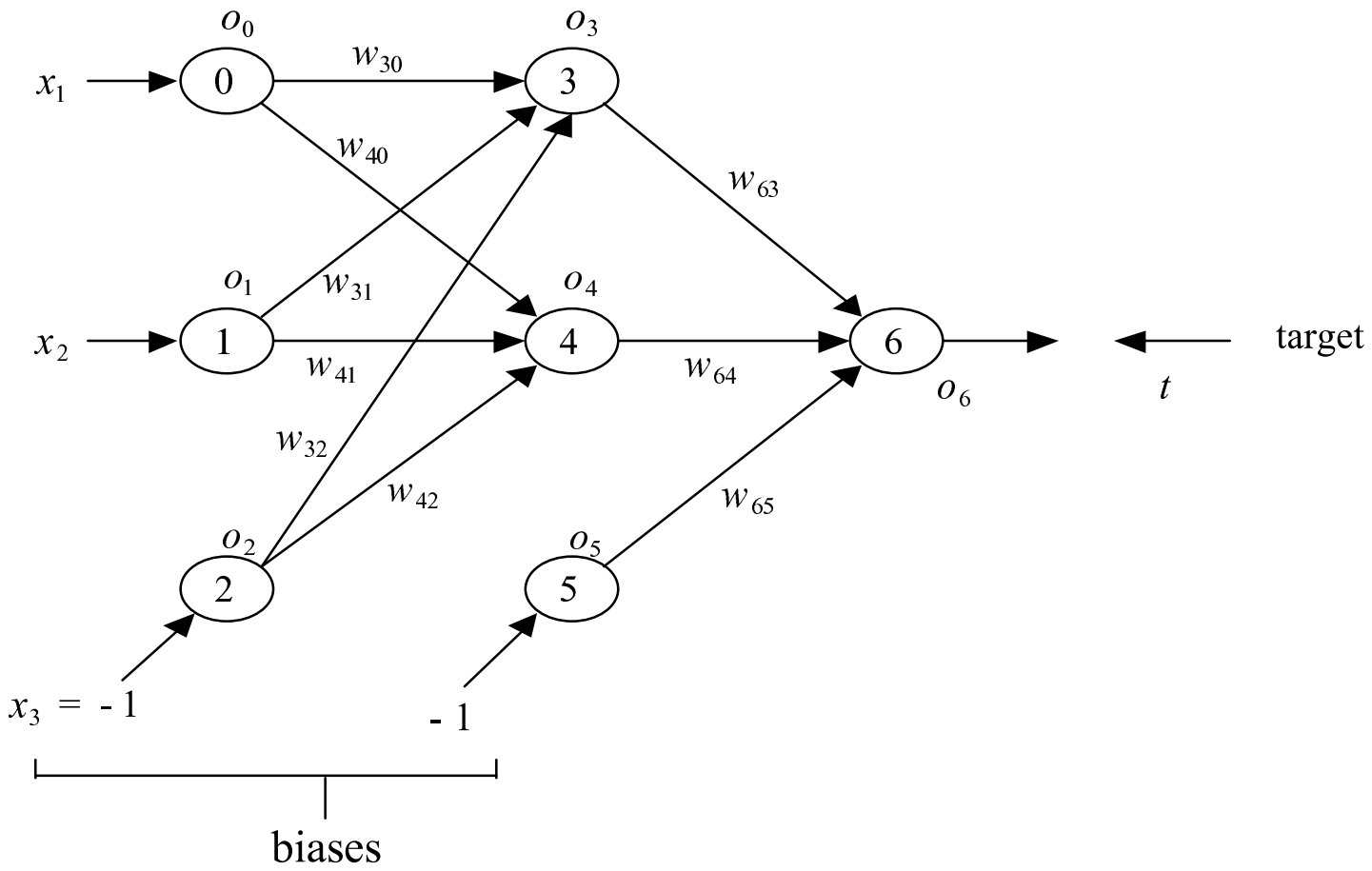
The modified weight update formulae including momentum term given as: $\Delta w^{(l)}(t+1) = -\eta \frac{\partial E_c(t)}{\partial w^l} + \gamma \Delta w^l(t)$.



Example 1

- To illustrate this powerful algorithm, we apply it for the training of the following network shown in the next page.
- x : training patterns, and t : output data
  $x^{(1)} = (0.3, 0.4),\ \ t(1) = 0.88$
  $x^{(2)} = (0.1, 0.6),\ \ t(2) = 0.82$
  $x^{(3)} = (0.9, 0.4),\ \ t(3) = 0.57$
- Biases: -1
- Sigmoid activation function: $f(tot) = \frac{1}{1+e^{-\lambda tot}}$, using $\lambda = 1$, then $f'(tot) = f(tot)(1 - f(tot))$.

Example 1: Structure of the Network



Example 1: Training Loop (1)

- Step (1) Initialization
  - Initialize the weights to 0.2, set learning rate to $\eta = 0.2$; set maximum tolerable error to $E_{max} = 0.01$ (i.e. 1% error), set $E = 0$ and $k = 1$.
- Step (2) - Apply input pattern
  - Apply the 1^{st} input pattern to the input layer. $x^{(1)} = (0.3, 0.4),\ t(1) = 0.88$, then,

    $o_0 = x_1 = 0.3;\ o_1 = x_2 = 0.4;\ o_2 = x_3 = -1;$

Example 1: Training Loop (1)

- Step (3) - Forward propagation
  - Propagate the signal forward through the network

$$o_3 = f(w_{30}o_0 + w_{31}o_1 + w_{32}o_2) = 0.485$$

$$o_4 = f(w_{40}o_0 + w_{41}o_1 + w_{42}o_2) = 0.485$$

$$o_5 = -1$$

$$o_6 = f(w_{63}o_3 + w_{64}o_4 + w_{65}o_5) = 0.4985$$

Example 1: Training Loop (1)

- Step (4) - Output error measure
  - Compute the error value E

$$E = \frac{1}{2}(t - o_6)^2 + E = 0.0728$$

  - Compute the error signal δ_6 of the output layer

$$\begin{aligned}\delta_6 &= f'(tot_6)(t - o_6) \\ &= o_6(1 - o_6)(t - o_6) \\ &= 0.0945\end{aligned}$$

Example 1: Training Loop (1)

Step (5) - Error back-propagation

- Third layer weight updates:

$$\Delta w_{63} = \eta \delta_6 o_3 = 0.0093 \qquad w_{63}^{new} = w_{63}^{old} + \Delta w_{63} = 0.2093$$

$$\Delta w_{64} = \eta \delta_6 o_4 = 0.0093 \qquad w_{64}^{new} = w_{64}^{old} + \Delta w_{64} = 0.2093$$

$$\Delta w_{65} = \eta \delta_6 o_5 = 0.0191 \qquad w_{65}^{new} = w_{65}^{old} + \Delta w_{65} = 0.1809$$

- Second layer error signals:

$$\delta_3 = f_3'(tot_3) \sum_{i=6}^{6} w_{i3}\delta_i = o_3(1 - o_3) w_{63}\delta_6 = 0.0048$$

$$\delta_4 = f_4'(tot_4) \sum_{i=6}^{6} w_{i4}\delta_i = o_4(1 - o_4) w_{64}\delta_6 = 0.0048$$

Example 1: Training Loop (1)

Step (5) - Error back-propagation (cont.)

- Second layer weight updates:

$$\Delta w_{30} = \eta\delta_3 o_0 = 0.00028586 \quad w_{30}^{new} = w_{30}^{old} + \Delta w_{30} = 0.2003$$

$$\Delta w_{31} = \eta\delta_3 o_1 = 0.00038115 \quad w_{31}^{new} = w_{31}^{old} + \Delta w_{31} = 0.2004$$

$$\Delta w_{32} = \eta\delta_3 o_2 = -0.00095288 \quad w_{32}^{new} = w_{32}^{old} + \Delta w_{32} = 0.199$$

$$\Delta w_{40} = \eta\delta_4 o_0 = 0.00028586 \quad w_{40}^{new} = w_{40}^{old} + \Delta w_{40} = 0.2003$$

$$\Delta w_{41} = \eta\delta_4 o_1 = 0.00038115 \quad w_{41}^{new} = w_{41}^{old} + \Delta w_{41} = 0.2004$$

$$\Delta w_{42} = \eta\delta_4 o_2 = -0.00095288 \quad w_{42}^{new} = w_{42}^{old} + \Delta w_{42} = 0.199$$

Example 1: Training Loop (2)

- Step (2) - Apply the 2^{nd} input pattern
  $x^{(2)} = (0.1, 0.6),\ t(2) = 0.82$, then,
  $o_0 = 0.1;\ o_1 = 0.6;\ o_2 = -1;$
- Step (3) - Forward propagation
  $o_3 = f(w_{30}o_0 + w_{31}o_1 + w_{32}o_2) = 0.4853$
  $o_4 = f(w_{40}o_0 + w_{41}o_1 + w_{42}o_2) = 0.4853$
  $o_5 = -1$
  $o_6 = f(w_{63}o_3 + w_{64}o_4 + w_{65}o_5) = 0.5055$
- Step (4) - Output error measure
  $E = \frac{1}{2}(t - o_6)^2 + E = 0.1222$
  $= o_6(1 - o_6)(t - o_6) = 0.0786$

Training Loop - Loop (2)

Step (5) - Error back-propagation

- Third layer weight updates:

$$\Delta w_{63} = \eta \delta_6 o_3 = 0.0076 \qquad w_{63}^{new} = w_{63}^{old} + \Delta w_{63} = 0.2169$$

$$\Delta w_{64} = \eta \delta_6 o_4 = 0.0076 \qquad w_{64}^{new} = w_{64}^{old} + \Delta w_{64} = 0.2169$$

$$\Delta w_{65} = \eta \delta_6 o_5 = 0.0157 \qquad w_{65}^{new} = w_{65}^{old} + \Delta w_{65} = 0.1652$$

- Second layer error signals:

$$\delta_3 = f_3'(tot_3) \sum_{i=6}^{6} w_{i3}\delta_i = o_3(1 - o_3) w_{63}\delta_6 = 0.0041$$

$$\delta_4 = f_4'(tot_4) \sum_{i=6}^{6} w_{i4}\delta_i = o_4(1 - o_4) w_{64}\delta_6 = 0.0041$$

Example 1: Training Loop (2)

Step (5) - Error back-propagation (cont.)

- Second layer weight updates:

$$\Delta w_{30} = \eta\delta_3 o_0 = 0.000082169 \quad w_{30}^{new} = w_{30}^{old} + \Delta w_{30} = 0.2004$$

$$\Delta w_{31} = \eta\delta_3 o_1 = 0.00049302 \quad w_{31}^{new} = w_{31}^{old} + \Delta w_{31} = 0.2009$$

$$\Delta w_{32} = \eta\delta_3 o_2 = -0.00082169 \quad w_{32}^{new} = w_{32}^{old} + \Delta w_{32} = 0.1982$$

$$\Delta w_{40} = \eta\delta_4 o_0 = 0.000082169 \quad w_{40}^{new} = w_{40}^{old} + \Delta w_{40} = 0.2004$$

$$\Delta w_{41} = \eta\delta_4 o_1 = 0.00049302 \quad w_{41}^{new} = w_{41}^{old} + \Delta w_{41} = 0.2009$$

$$\Delta w_{42} = \eta\delta_4 o_2 = -0.00082169 \quad w_{42}^{new} = w_{42}^{old} + \Delta w_{42} = 0.1982$$

Example 1: Training Loop (3)

- Step (2) - Apply the 2^{nd} input pattern
  $x^{(3)} = (0.9, 0.4),\ t(3) = 0.57$, then,
  $o_0 = 0.9;\ o_1 = 0.4;\ o_2 = -1;$
- Step (3) - Forward propagation

$$o_3 = f(w_{30}o_0 + w_{31}o_1 + w_{32}o_2) = 0.5156$$

$$o_4 = f(w_{40}o_0 + w_{41}o_1 + w_{42}o_2) = 0.5156$$

$$o_5 = -1$$

$$o_6 = f(w_{63}o_3 + w_{64}o_4 + w_{65}o_5) = 0.5146$$

- Step (4) - Output error measure

$$E = \frac{1}{2}(t - o_6)^2 + E = 0.1237$$
$$= o_6(1 - o_6)(t - o_6) = 0.0138$$

Example 1: Training Loop (3)

Step (5) - Error back-propagation

- Third layer weight updates:

$$\Delta w_{63} = \eta \delta_6 o_3 = 0.0014 \qquad w_{63}^{new} = w_{63}^{old} + \Delta w_{63} = 0.2183$$

$$\Delta w_{64} = \eta \delta_6 o_4 = 0.0014 \qquad w_{64}^{new} = w_{64}^{old} + \Delta w_{64} = 0.2183$$

$$\Delta w_{65} = \eta \delta_6 o_5 = -0.0028 \qquad w_{65}^{new} = w_{65}^{old} + \Delta w_{65} = 0.1624$$

- Second layer error signals:

$$\delta_3 = f_3'(tot_3) \sum_{i=6}^{6} w_{i3}\delta_i = o_3(1 - o_3) w_{63}\delta_6 = 0.00074948$$

$$\delta_4 = f_4'(tot_4) \sum_{i=6}^{6} w_{i4}\delta_i = o_4(1 - o_4) w_{64}\delta_6 = 0.00074948$$

Example 1: Training Loop (3)

Step (5) - Error back-propagation (cont.)

- Second layer weight updates:

$$\Delta w_{30} = \eta\delta_3 o_0 = 0.00013491 \quad w_{30}^{new} = w_{30}^{old} + \Delta w_{30} = 0.2005$$

$$\Delta w_{31} = \eta\delta_3 o_1 = 0.000059958 \quad w_{31}^{new} = w_{31}^{old} + \Delta w_{31} = 0.2009$$

$$\Delta w_{32} = \eta\delta_3 o_2 = -0.0001499 \quad w_{32}^{new} = w_{32}^{old} + \Delta w_{32} = 0.1981$$

$$\Delta w_{40} = \eta\delta_4 o_0 = 0.00013491 \quad w_{40}^{new} = w_{40}^{old} + \Delta w_{40} = 0.2005$$

$$\Delta w_{41} = \eta\delta_4 o_1 = 0.000059958 \quad w_{41}^{new} = w_{41}^{old} + \Delta w_{41} = 0.2009$$

$$\Delta w_{42} = \eta\delta_4 o_2 = -0.0001499 \quad w_{42}^{new} = w_{42}^{old} + \Delta w_{42} = 0.1981$$

Example 1: Final Decision

- Step (6) - One epoch looping

  The training patterns have been cycled one epoch.

- Step (7) - Total error checking

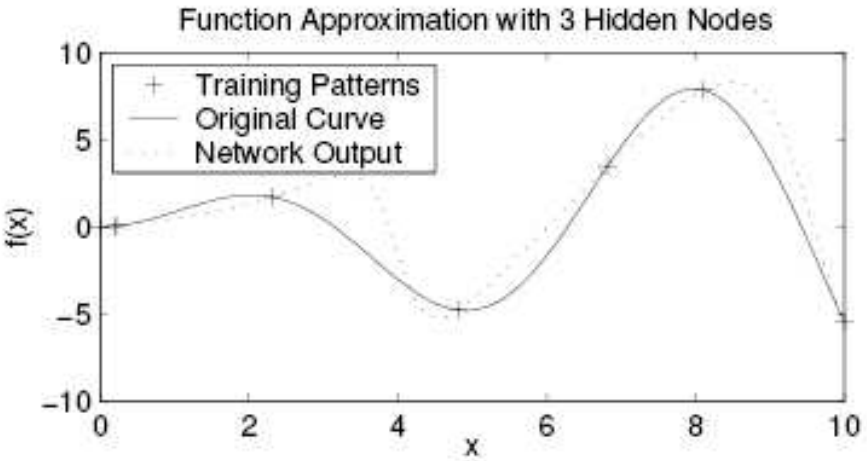
  $E = 0.1237$ and $E_{max} = 0.01$, which means that we have to continue with the next epoch by cycling the training data again.

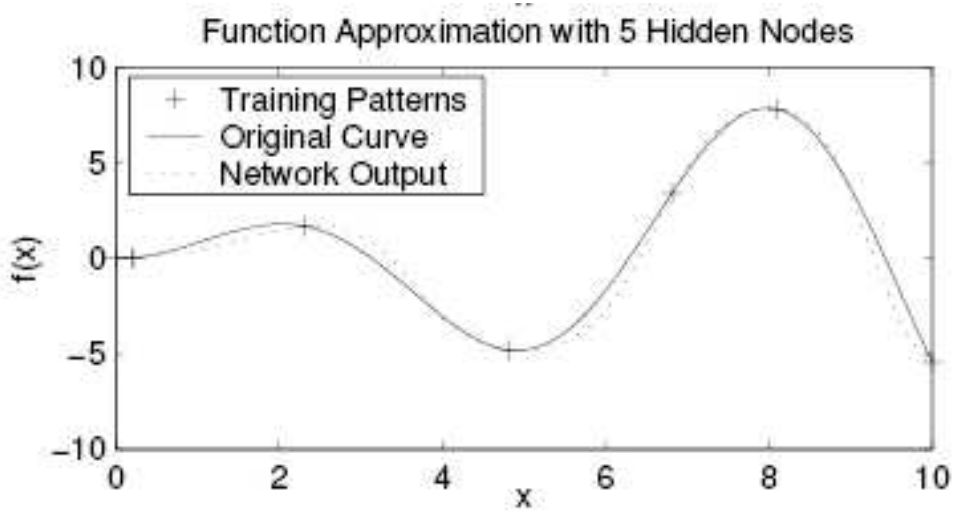
Example 2

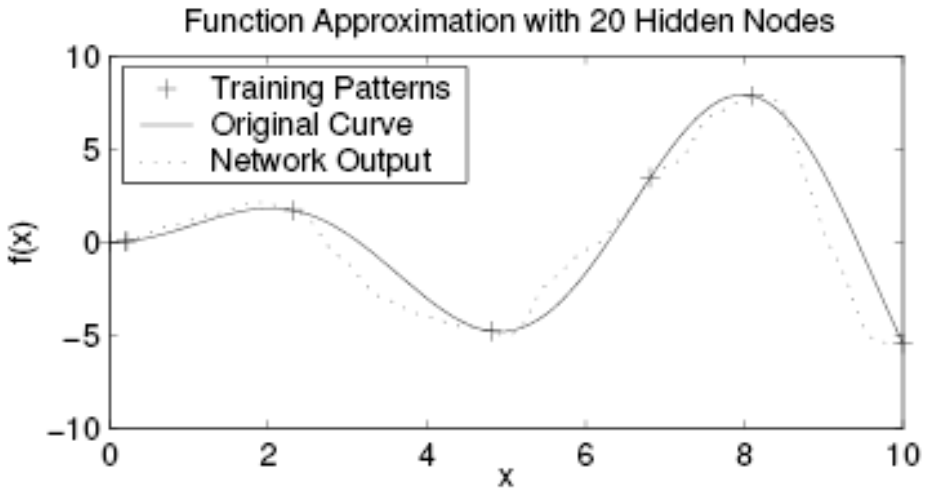
Effect of Hidden Nodes on Function Approximation

- Consider this function $f(x) = x\sin(x)$
- Six input/output samples were selected from the range $[0, 10]$ of the variable x
- The first run was made for a network with 3 hidden nodes
- Another run was made for a network with 5 and 20 nodes, respectively.

Example 2: Different Hidden Nodes







Example 2: Remarks

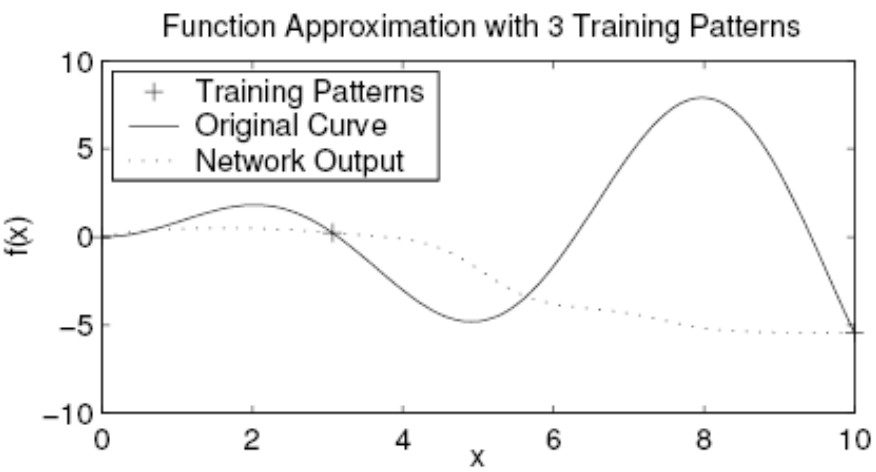
- A higher number of nodes is not always better. It may overtrain the network.
- This happens when the network starts to memorize the patterns instead of interpolating between them.
- A smaller number of nodes was not able to approximate faithfully the function given the nonlinearities induced by the network was not enough to interpolate well in between the samples.
- It seems here that this network (with five nodes) was able to interpolate quite well the nonlinear behavior of the curve.

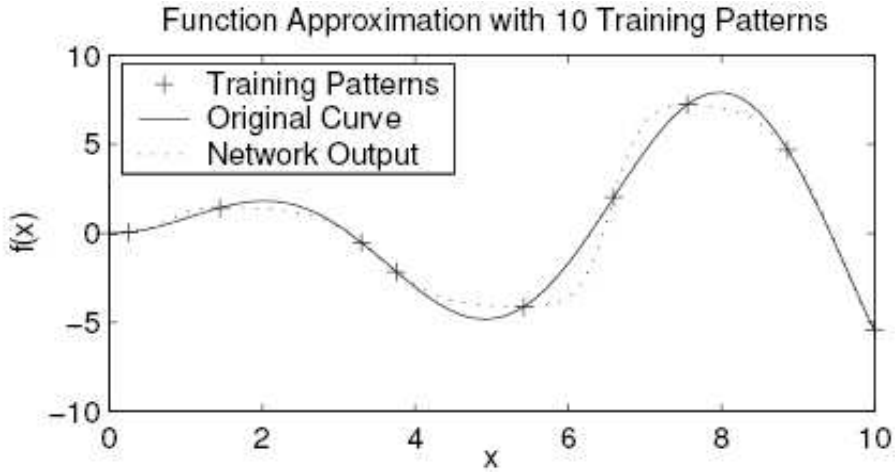
Example 3

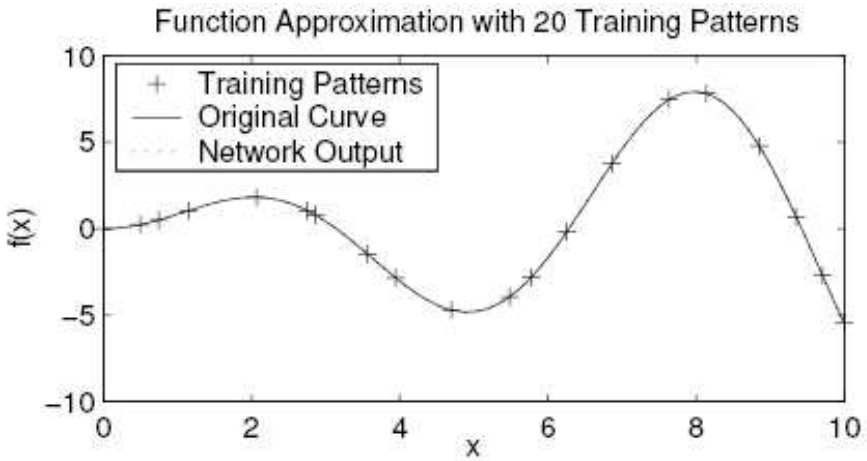
Effect of Training Patterns on Function Approximation

- Consider this function $f(x) = x\sin(x)$
- Assume a network with a fixed number of nodes (taken as five here), but with a variable number of training patterns
- The first run was made for a network with 3 three samples
- Another run was made for a network with 10 and 20 samples, respectively.

Example 3: Different Samples







Example 3: Remarks

- The first run with three samples was not able to provide a good mach with the original curve.
- This can be explained by the fact that the three patterns, in the case of a nonlinear function such as this, are not able to reproduce the relatively high nonlinearities of the function.
- A higher number of training points provided better results.
- The best result was obtained for the case of 20 training patterns. This is due to the fact that a network with five hidden nodes interpolates extremely well in between close training patterns.

Applications of MLP

- Multilayer perceptrons are currently among the most used connectionist models.
- This stems from the relative ease for training and implementing, either in hardware or software forms.

Applications

- Signal processing
- Pattern recognition
- Financial market prediction
- Weather forecasting
- Signal compression

Applications of MLP

- Multilayer perceptrons are currently among the most used connectionist models.
- This stems from the relative ease for training and implementing, either in hardware or software forms.

Applications

- Signal processing
- Pattern recognition
- Financial market prediction
- Weather forecasting
- Signal compression

Limitations of MLP

- Among the well-known problems that may hinder the generalization or approximation capabilities of MLP is the one related to the convergence behavior of the connection weights during the learning stage.
- In fact, the gradient descent based algorithm used to update the network weights may never converge to the global minima.
- This is particularly true in the case of highly nonlinear behavior of the system being approximated by the network.

Limitations of MLP

- Many remedies have been proposed to tackle this issue either by retraining the network a number of times or by using optimization techniques such as those based on:
  - Genetic algorithms,
  - Simulated annealing.

MLP NN: Case Study

Function Estimation (Regression)

MLP NN: Case Study

- Use a feedforward backpropagation neural network that contains a single hidden layer.
- Each of hidden nodes has an activation function of the logistic form.
- Investigate the outcome of the neural network for the following mapping.

$$f(x) = exp(-x^2), \quad x \in [0\ 2]$$

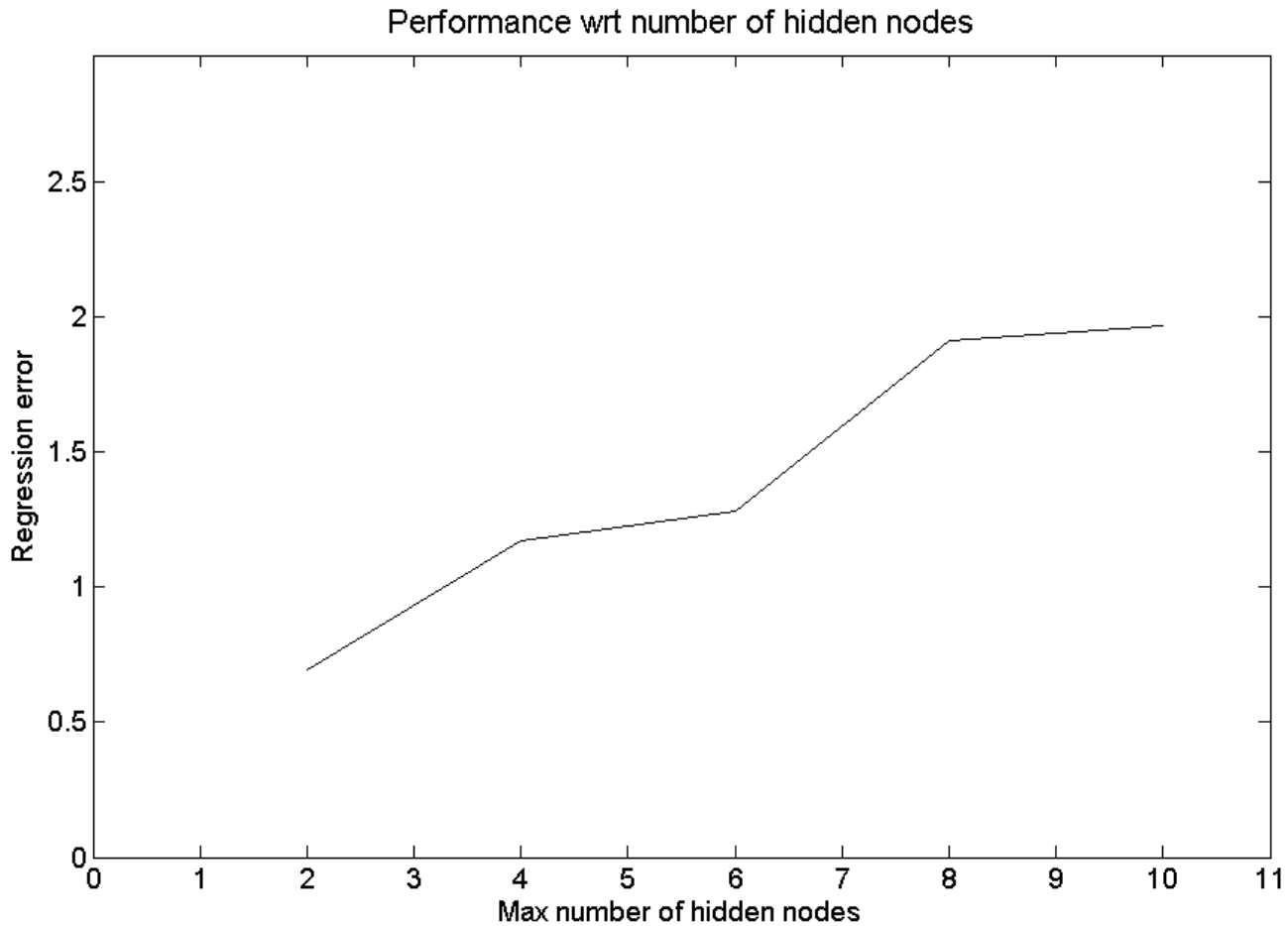
- Experiment with different number of training samples and hidden layer nodes

MLP NN: Case Study

Experiment 1: Vary Number of Hidden Nodes

- Uniformly pick six sample points from [0 2], use half of them for training and the rest for testing
- Evaluate regression performance increasing the number of hidden nodes
- Use sum of regression error (i.e. $\sum_{i \in \text{test samples}} (Output(i) - True\_output(i))$) as performance measure
- Repeat each test 20 times and compute average results, compensating for potential local minima

MLP NN: Case Study

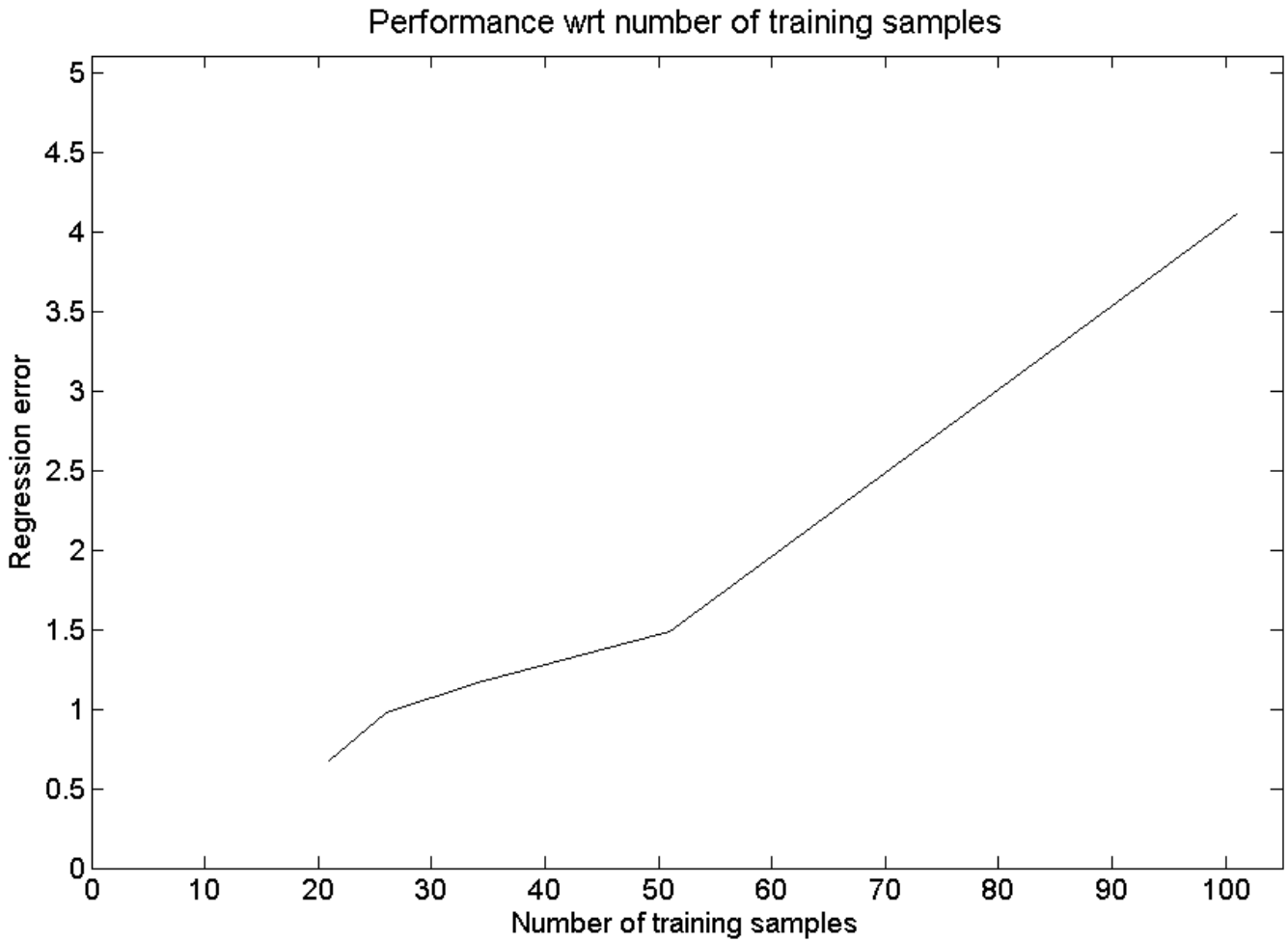


MLP NN: Case Study

Experiment 2: Vary Number of Training Samples

- Construct neural network using three hidden nodes
- Uniformly pick sample points from [0 2], increasing their number for each test
- Use half of sample data points for training and the rest for testing
- Use the same performance measure as experiment 1, i.e. sum of regression error
- Repeat each test 50 times and compute average results

MLP NN: Case Study

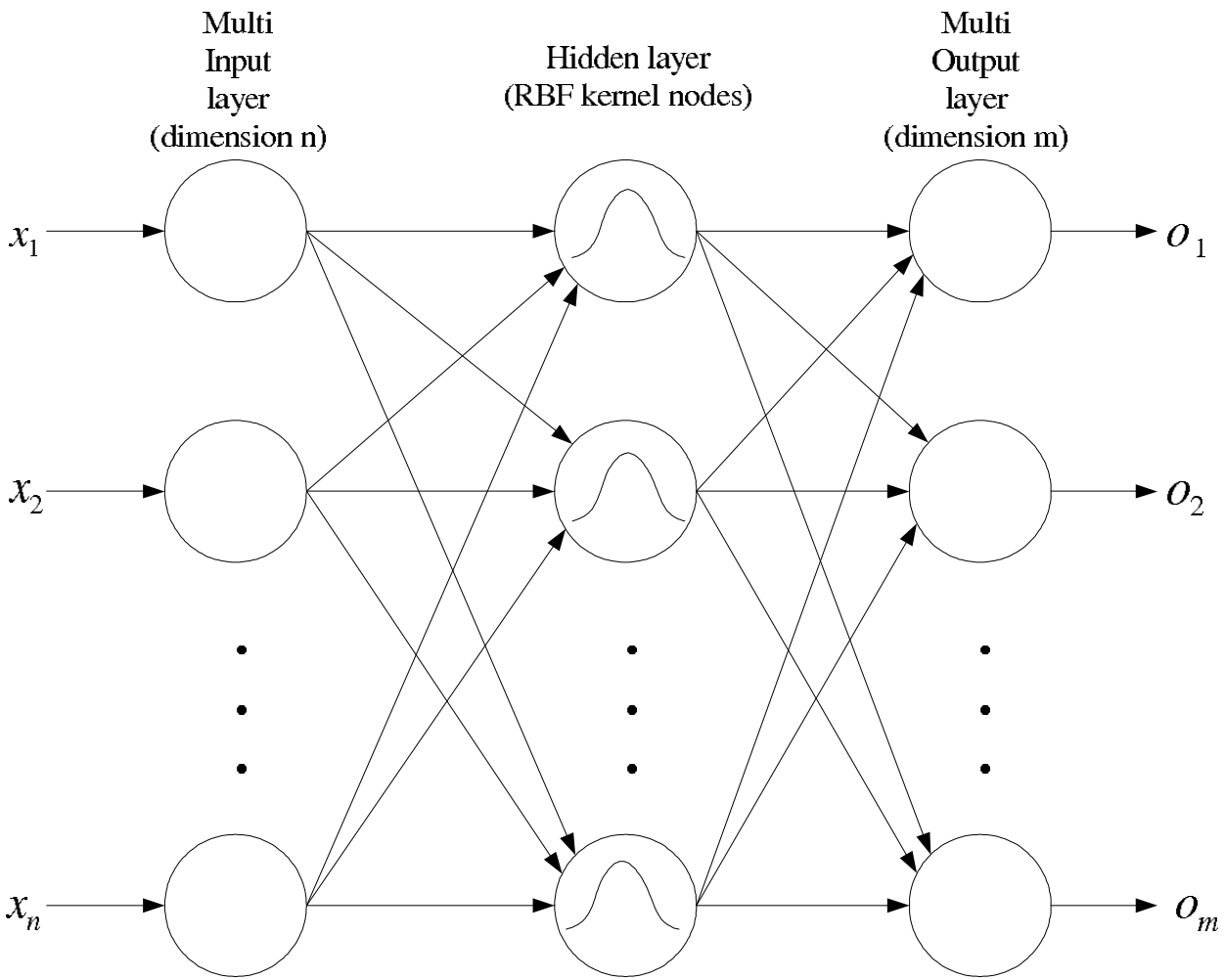


Radial Basis Function Network

Topology

- Radial basis function network (RBFN) represent a special category of the **feedforward** neural networks architecture.
- Early researchers have developed this connectionist model for **mapping nonlinear behavior of static processes** and for **function approximation purposes**.
- The basic RBFN structure consists of **an input layer**, **a single hidden layer** with **radial activation function** and **an output layer**.

Topology: Graphical Representation



Topology (cont.)

- The network structure uses **nonlinear transformations** in its hidden layer (typical transfer functions for hidden functions are Gaussian curves).
- However, it uses **linear transformations** between the hidden and output layers.
- The rationale behind this is that input spaces, cast nonlinearly into high-dimensional domains, are more likely to be linearly separable than those cast into low-dimensional ones.

Topology (cont.)

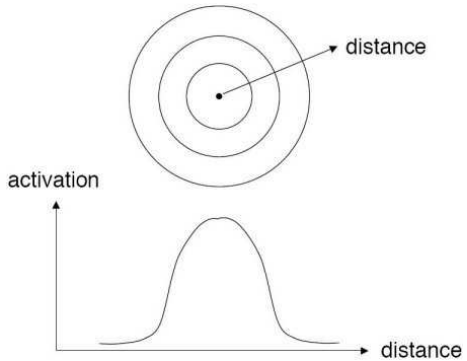
- Unlike most FF neural networks, the connection weights between the input layer and the neuron units of the hidden layer for an RBFN are all equal to **unity**.
- The nonlinear transformations at the hidden layer level have the main characteristics of being symmetrical.
- They also attain their maximum at the function center, and generate positive values that are rapidly decreasing with the distance from the center.

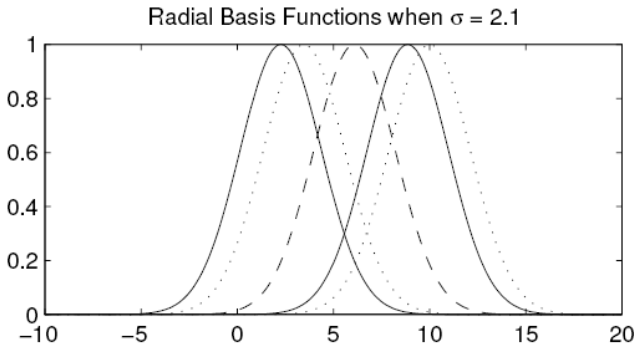
Topology (cont.)

- As such they produce radially activation signals that are bounded and localized.

Parameters of Each activation Function

- The center
- The width





Topology (cont.)

- For an optimal performance of the network, the hidden layer nodes should span the training data input space.
- Too sparse or too overlapping functions may cause the degradation of the network performance.

Radial Function or Kernel Function

- In general the form taken by an RBF function is given as:

$$g_i(x) = r_i\left(\frac{\| x - v_i \|}{\sigma_i}\right)$$

  - where x is the input vector,
  - v_i is the vector denoting the center of the radial function g_i,
  - σ_i is width parameter.

Famous Radial Functions

- The Gaussian kernel function is the most widely used form of RBF given by:

$$g_i(x) = exp(\frac{- \parallel x - v_i \parallel^2}{2\sigma_i^2})$$

- The logistic function has also been used as a possible RBF candidate:

$$g_i(x) = \frac{1}{1 + exp(\frac{\|x - v_i\|^2}{\sigma_i^2})}$$

Output of an RBF Network

- A typical output of an RBF network having n units in the hidden layer and r output units is given by:

$$o_j(x) = \sum_{i=1}^{n} w_{ij} g_i(x), \; j = 1, \cdots, r.$$

  - where w_{ij} is the connection weight between the i-th receptive field unit and the j-th output,
  - g_i is the i-th receptive field unit (radial function).

Learning Algorithm

Two-Stage Learning Strategy

- At first, an unsupervised clustering algorithm is used to extract the parameters of the radial basis functions, namely the width and the centers.
- This is followed by the computation of the weights of the connections between the output nodes and the kernel functions using a supervised least mean square algorithm.

Learning Algorithm: Hybrid Approach

- The standard technique used to train an RBF network is the **hybrid approach**.

Hybrid Approach

- Step 1: Train the RBF layer to get the adaptation of centers and scaling parameters using the **unsupervised training**.
- Step 2: Adapt the weights of the output layer using **supervised training algorithm**.

Learning Algorithm: Step 1

- To determine the centers for the RBF networks, typically **unsupervised** training procedures of **clustering** are used:
  - K-means method,
  - "Maximum likelihood estimate" technique,
  - Self-organizing map method.
- This step is very important in the training of RBFN, as the accurate knowledge of v_i and σ_i has a major impact on the performance of the network.

Learning Algorithm: Step 2

- Once the centers and the widths of radial basis functions are obtained, the next stage of the training begins.
- To update the weights between the hidden layer and the output layer, the supervised learning based techniques such as are used:
  - Least-squares method,
  - Gradient method.
- Because the weights exist only between the hidden layer and the output layer, it is easy to compute the weight matrix for the RBFN.

Learning Algorithm: Step 2 (cont.)

- In the case where the RBFN is used for interpolation purposes, we can use the **inverse** or **pseudo-inverse method** to calculate the **weight matrix**.
- If we use Gaussian kernel as the radial basis functions and there are n input data, we have:

$$G = [\{g_{ij}\}],$$

where

$$g_{ij} = exp(\frac{- \parallel x_i - v_j \parallel^2}{2\sigma_j^2}), \ \ i, j = 1, \cdots, n$$

Learning Algorithm: Step 2 (cont.)

- Now we have:

$$D = GW$$

where D is the desired output of the training data.

- If G^{-1} exists, we get:

$$W = G^{-1}D$$

- In practice however, G may be ill-conditioned (close to singularity) or may even be a non-square matrix (if the number of radial basis functions is less than the number of training data) then W is expressed as:

$$W = G^{+}D$$

Learning Algorithm: Step 2 (cont.)

- We had:

$$W = G^{+} D,$$

  - where G^{+} denotes the pseudo-inverse matrix of G, which can be defined as

$$G^{+} = (G^{T} G)^{-1} G^{T}$$

- Once the weight matrix has been obtained, all elements of the RBFN are now determined and the network could operate on the task it has been designed for.

Learning Algorithm: Step 2 (cont.)

- We had:

$$W = G^{+} D,$$

  - where G^{+} denotes the pseudo-inverse matrix of G, which can be defined as

$$G^{+} = (G^{T} G)^{-1} G^{T}$$

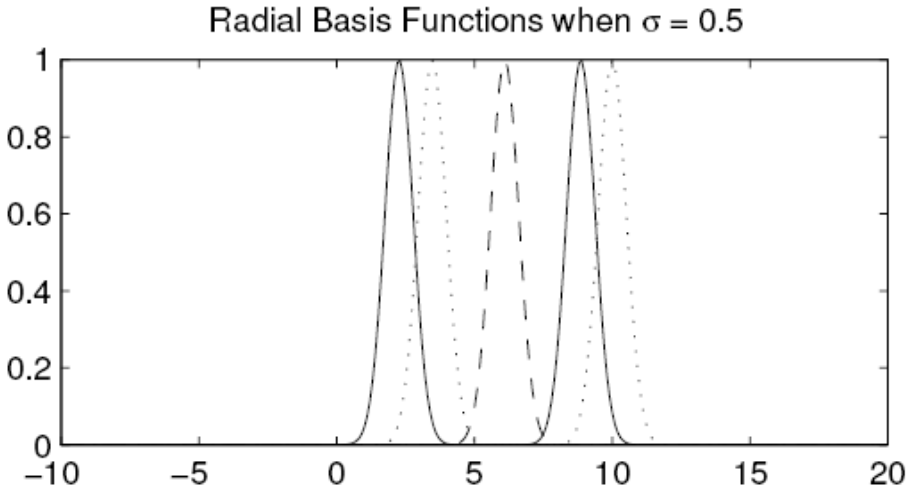
- Once the weight matrix has been obtained, all elements of the RBFN are now determined and the network could operate on the task it has been designed for.

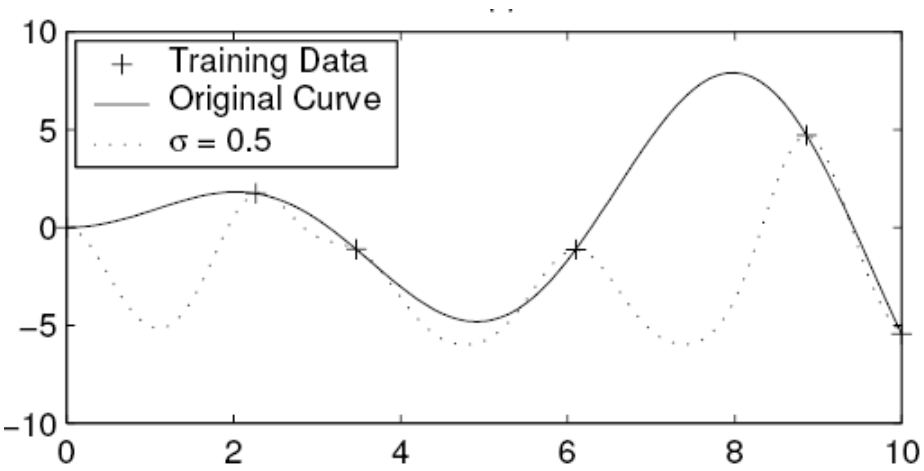
Example

Approximation of Function $f(x)$ Using an RBFN

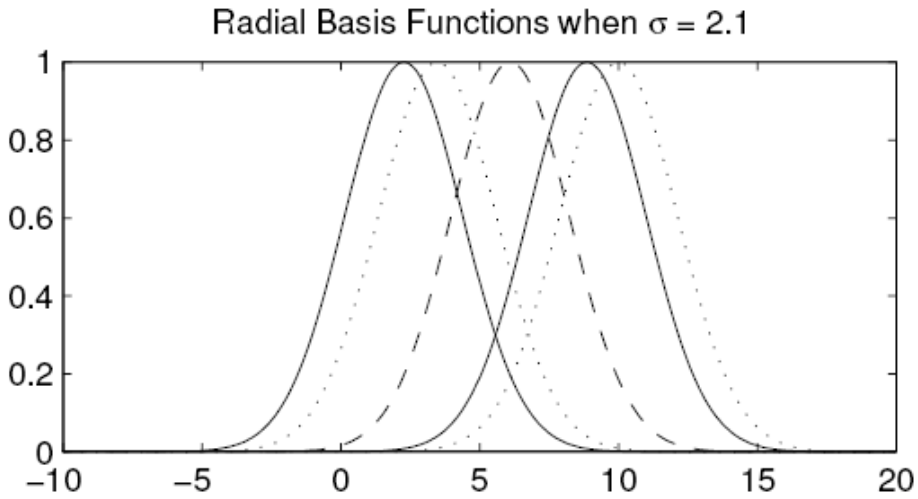
- We use here the same function as the one used in the MLP section, $f(x) = x\sin(x)$.
- The RBF network is composed here of five radial functions.
- Each radial function has its center at a training input data.
- Three width parameters are used here: 0.5, 2.1, and 8.5.
- The results of simulation show that the width of the function plays a major importance.

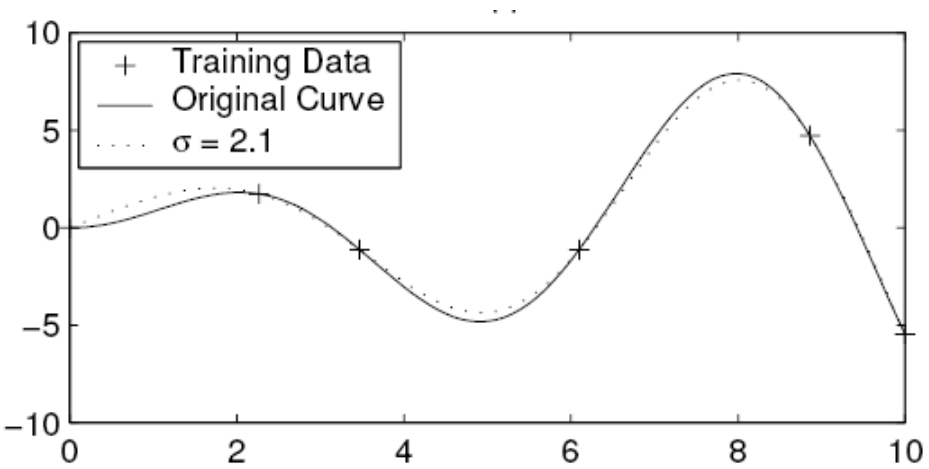
Example: Function Approximation with Gaussian Kernels ($\sigma = 0.5$)



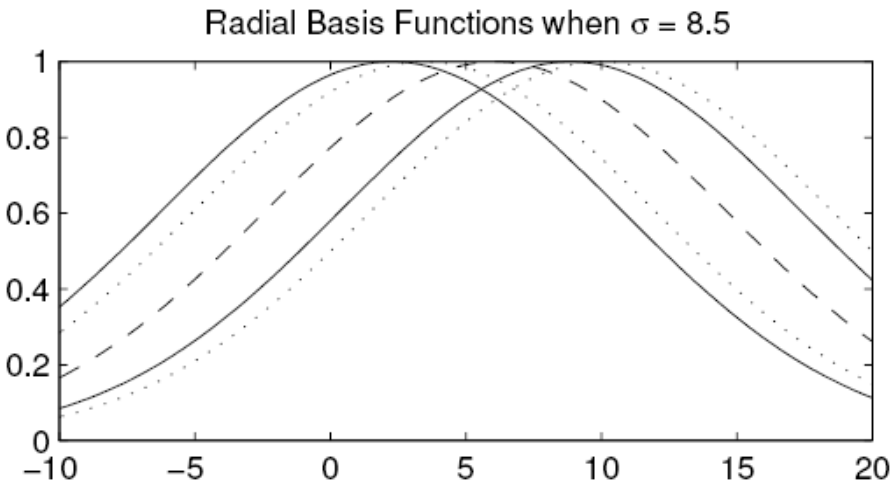


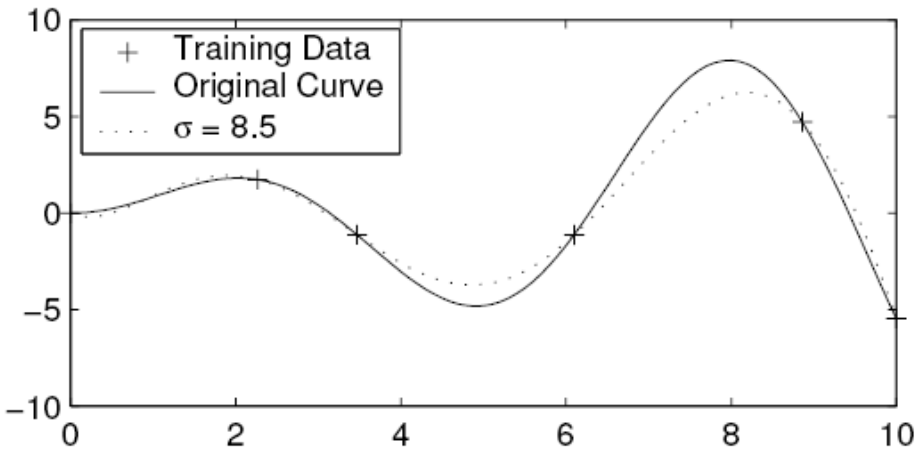
Example: Function Approximation with Gaussian Kernels ($\sigma = 2.1$)



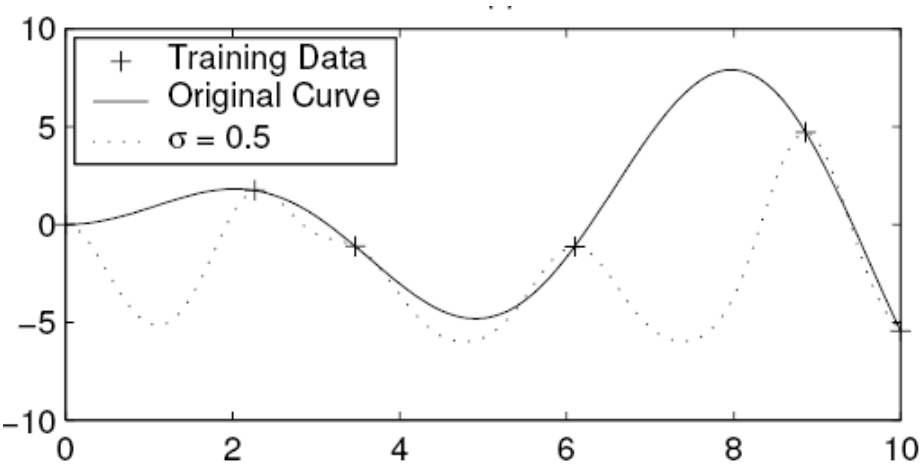


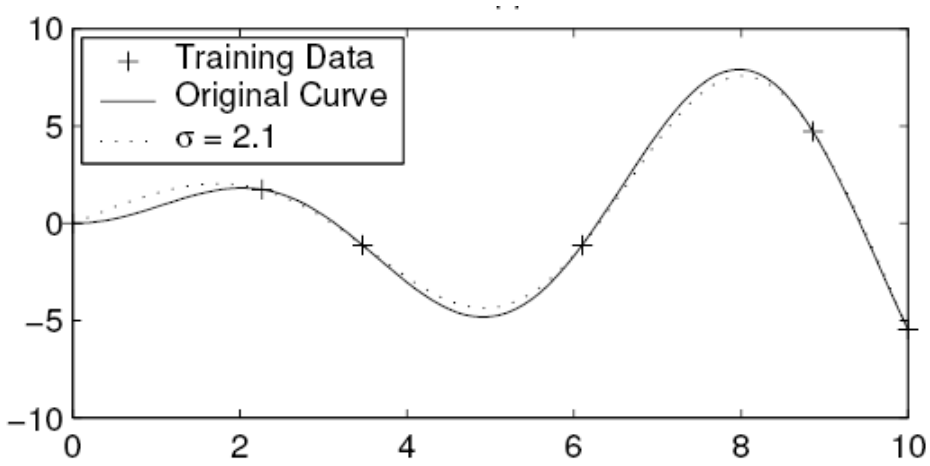
Example: Function Approximation with Gaussian Kernels ($\sigma = 8.5$)

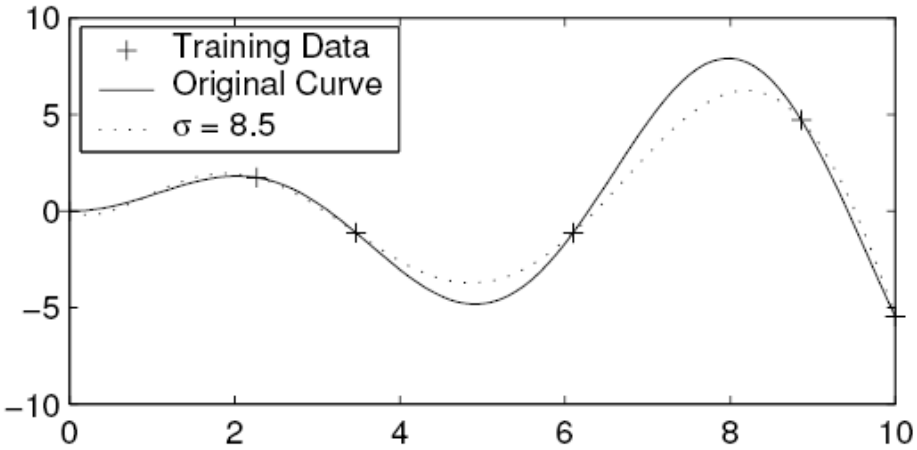




Example: Comparison







Example: Remarks

- A smaller width value 0.5 doesn't seem to provide for a good interpolation of the function in between sample data.
- A width value 2.1 provides a better result and the approximation by RBF is close to the original curve.
  - This particular width value seems to provide the network with the adequate interpolation property.
- A larger width value 8.5 seems to be inadequate for this particular case, given that a lot of information is being lost when the ranges of the radial functions are further away from the original range of the function.

Advantages/Disadvantages

- Unsupervised learning stage of an RBFN is not an easy task.
- RBF trains faster than a MLP.
- Another advantage that is claimed is that the hidden layer is easier to interpret than the hidden layer in an MLP.
- Although the RBF is quick to train, when training is finished and it is being used it is slower than a MLP, so where speed is a factor a MLP may be more appropriate.

Applications

- Known to have **universal approximation capabilities**, **good local structures** and **efficient training algorithms**, RBFN have been often used for nonlinear mapping of complex processes and for solving a wide range of **classification problems**.
- They have been used as well for control systems, audio and video signals processing, and pattern recognition.

Applications (cont.)

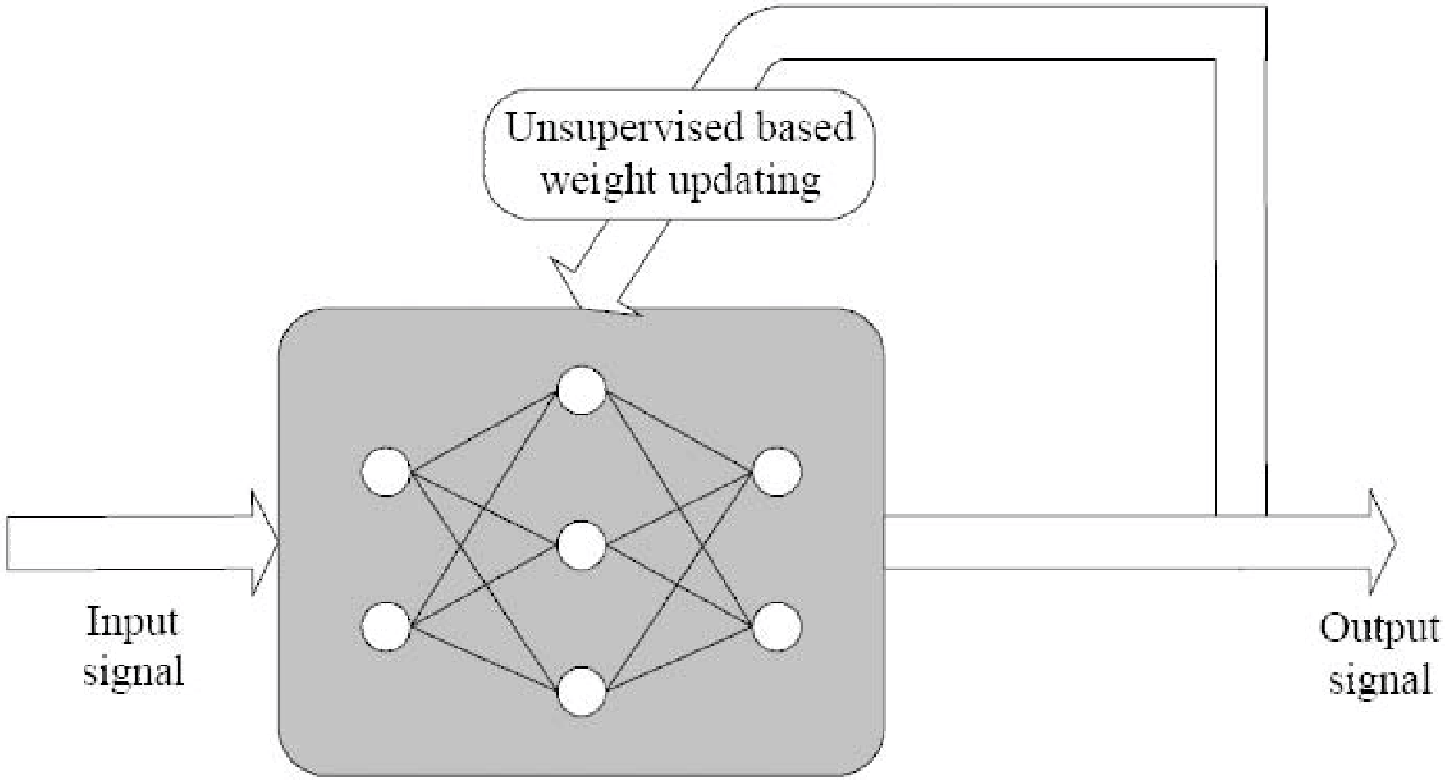
- They have also been recently used for **chaotic time series prediction**, with particular application to weather and power load forecasting.
- Generally, RBF networks have an undesirably high number of hidden nodes, but the dimension of the space can be reduced by careful planning of the network.

Kohonen's Self-Organizing Network

Topology

- The Kohonen's Self-Organizing Network (KSON) belongs to the class of **unsupervised learning networks**.
- This means that the network, unlike other forms of supervised learning based networks updates its weighting parameters without the need for a performance feedback from a **teacher** or a **network trainer**.

Unsupervised Learning



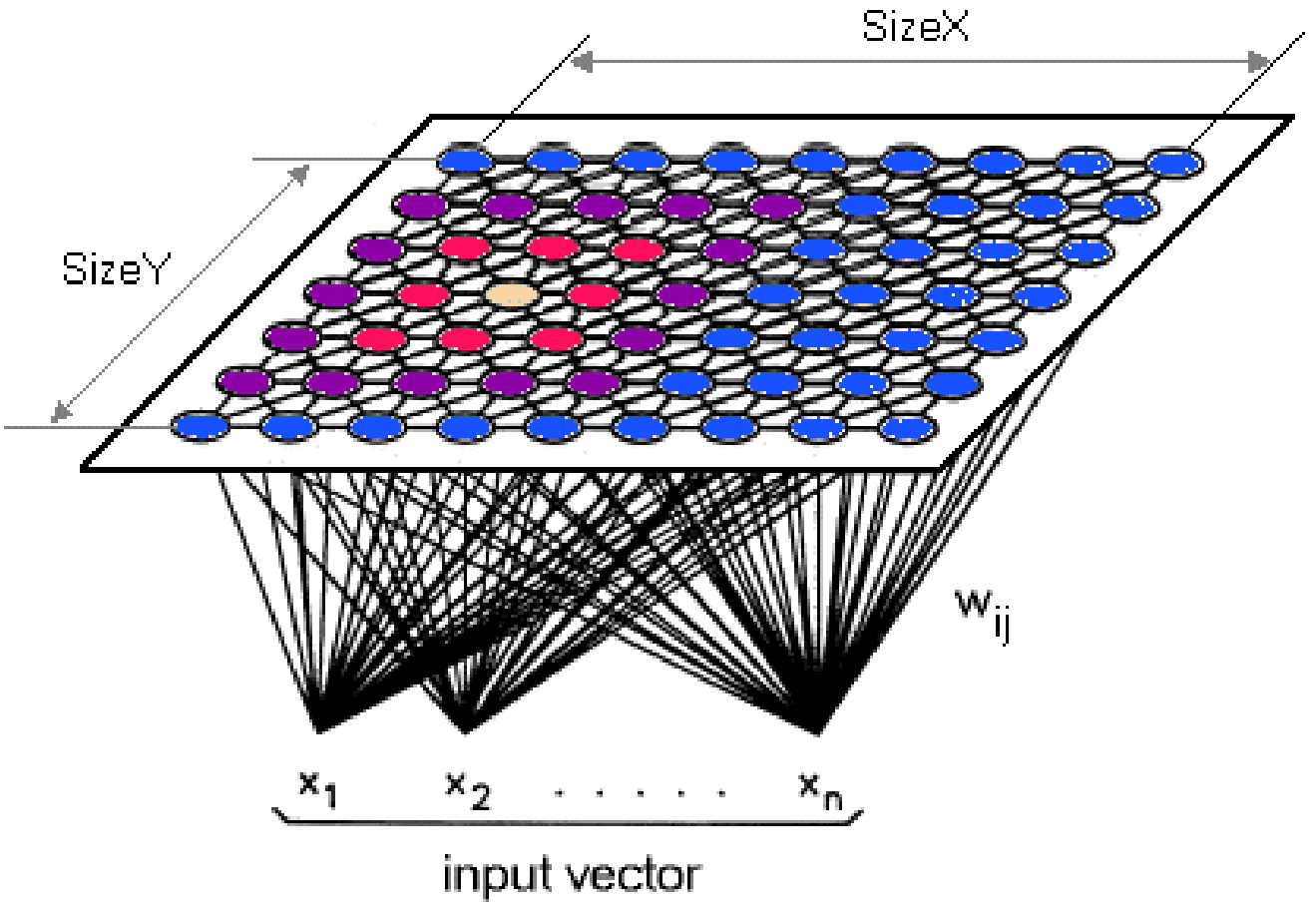
Topology (cont.)

- One major feature of this network is that the nodes distribute themselves across the input space to recognize groups of similar input vectors.
- However, the output nodes compete among themselves to be fired one at a time in response to a particular input vector.
- This process is known as **competitive learning**.

Topology (cont.)

- Two input vectors with similar pattern characteristics excite two physically close layer nodes.
- In other words, the nodes of the KSON can recognize groups of similar input vectors.
- This generates a topographic mapping of the input vectors to the output layer, which depends primarily on the pattern of the input vectors and results in dimensionality reduction of the input space.

A Schematic Representation of a Typical KSOM



Learning

- The learning here permits the clustering of input data into a smaller set of elements having similar characteristics (features).
- It is based on the competitive learning technique also known as the winner take all strategy.
- Presume that the input pattern is given by the vector x.
- Assume w_{ij} is the weight vector connecting the input elements to an output node with coordinate provided by indices i and j.

Learning

- N_c is defined as the neighborhood around the winning output candidate.
- Its size decreases at every iteration of the algorithm until convergence occurs.

Steps of Learning Algorithm

- Step 1: Initialize **all weights** to small random values. Set a value for the initial **learning rate** α and a value for the **neighborhood** N_c.
- Step 2: Choose an input pattern x from the input data set.
- Step 3: Select the winning unit c (the index of the best matching output unit) such that the performance index I given by the Euclidian distance from x to w_{ij} is minimized:

$$I = \|x - w_c\| = min_{ij}\|x - w_{ij}\|$$

Steps of Learning Algorithm (cont.)

- Step 4: Update the weights according to the global network updating phase from iteration k to iteration $k+1$ as:

$$w_{ij}(k+1) = \begin{cases} w_{ij}(k) + \alpha(k)[x - w_{ij}(k)] & \text{if} \quad (i,j) \in N_c(k), \\ w_{ij}(k) & \text{otherwise.} \end{cases}$$

  - where $\alpha(k)$ is the adaptive learning rate (strictly positive value smaller than the unity),
  - $N_c(k)$ the neighborhood of the unit c at iteration k.

Steps of Learning Algorithm (cont.)

- Step 5: The learning rate and the neighborhood are decreased at every iteration according to an appropriate scheme.
  - For instance, Kohonen suggested a shrinking function in the form of $\alpha(k) = \alpha(0)(1 - k/T)$, with T being the total number of training cycles and $\alpha(0)$ the starting learning rate bounded by one.
  - As for the neighborhood, several researchers suggested an initial region with the size of half of the output grid and shrinks according to an exponentially decaying behavior.
- Step 6: The learning scheme continues until enough number of iterations has been reached or until each output reaches a threshold of sensitivity to a portion of the input space.

Steps of Learning Algorithm (cont.)

- Step 5: The learning rate and the neighborhood are decreased at every iteration according to an appropriate scheme.
  - For instance, Kohonen suggested a shrinking function in the form of $\alpha(k) = \alpha(0)(1 - k/T)$, with T being the total number of training cycles and $\alpha(0)$ the starting learning rate bounded by one.
  - As for the neighborhood, several researchers suggested an initial region with the size of half of the output grid and shrinks according to an exponentially decaying behavior.
- Step 6: The learning scheme continues until enough number of iterations has been reached or until each output reaches a threshold of sensitivity to a portion of the input space.

Steps of Learning Algorithm (cont.)

- Step 5: The learning rate and the neighborhood are decreased at every iteration according to an appropriate scheme.
  - For instance, Kohonen suggested a shrinking function in the form of $\alpha(k) = \alpha(0)(1 - k/T)$, with T being the total number of training cycles and $\alpha(0)$ the starting learning rate bounded by one.
  - As for the neighborhood, several researchers suggested an initial region with the size of half of the output grid and shrinks according to an exponentially decaying behavior.
- Step 6: The learning scheme continues until enough number of iterations has been reached or until each output reaches a threshold of sensitivity to a portion of the input space.

Steps of Learning Algorithm (cont.)

- Step 5: The learning rate and the neighborhood are decreased at every iteration according to an appropriate scheme.
  - For instance, Kohonen suggested a shrinking function in the form of $\alpha(k) = \alpha(0)(1 - k/T)$, with T being the total number of training cycles and $\alpha(0)$ the starting learning rate bounded by one.
  - As for the neighborhood, several researchers suggested an initial region with the size of half of the output grid and shrinks according to an exponentially decaying behavior.
- Step 6: The learning scheme continues until enough number of iterations has been reached or until each output reaches a threshold of sensitivity to a portion of the input space.

Example

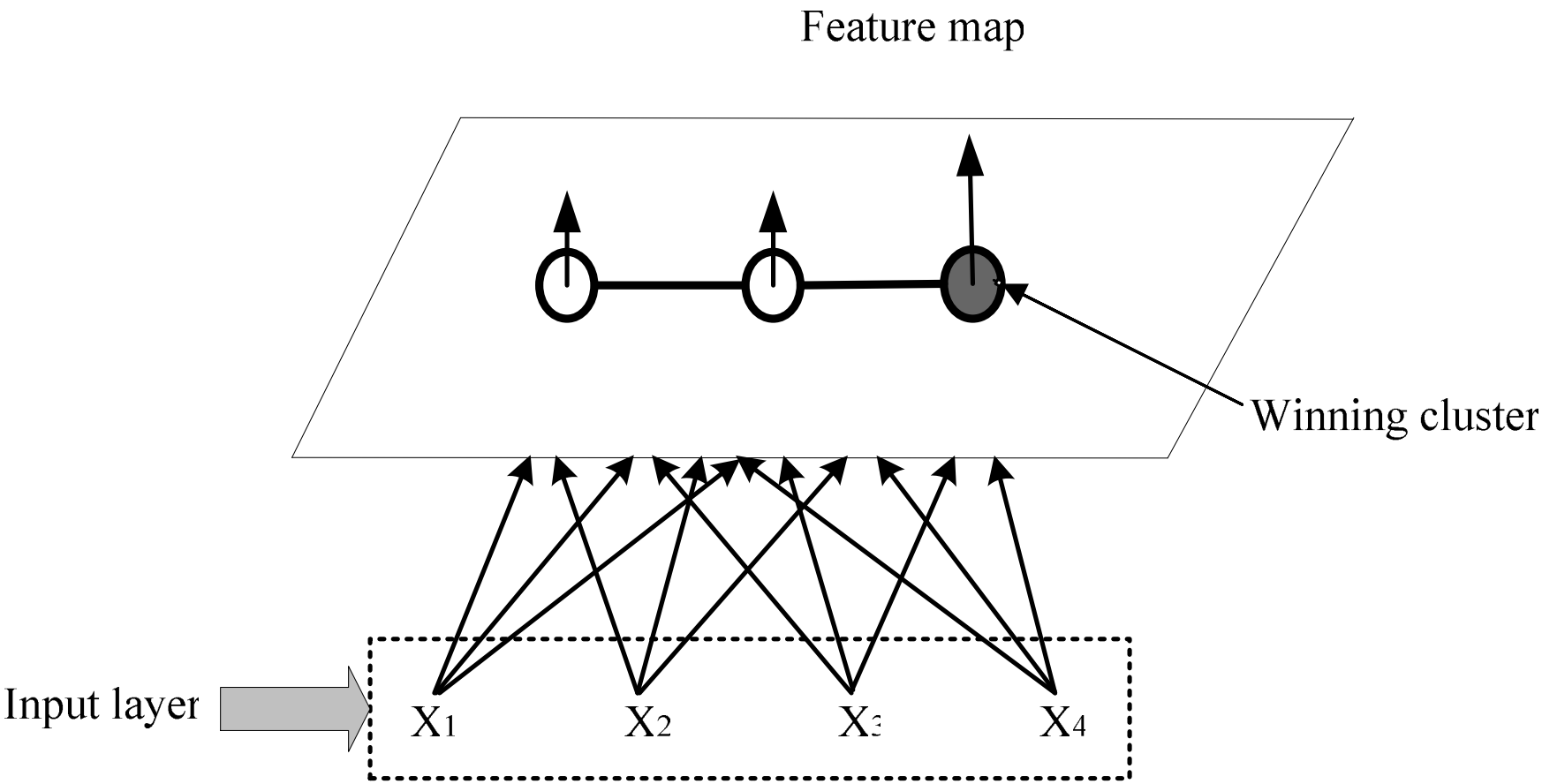
- A Kohonen self-organizing map is used to cluster four vectors given by:
  - $(1, 1, 1, 0)$,
  - $(0, 0, 0, 1)$,
  - $(1, 1, 0, 0)$,
  - $(0, 0, 1, 1)$.
- The maximum numbers of clusters to be formed is $m = 3$.

Example

- Suppose the learning rate (geometric decreasing) is given by:
  - $\alpha(0) = 0.3$,
  - $\alpha(t+1) = 0.2\alpha(t)$.

With only three clusters available and the weights of only one cluster are updated at each step (i.e., $N_c = 0$), find the weight matrix. Use one single epoch of training.

Example: Structure of the Network



Example: Step 1

- The initial weight matrix is:

$$W = \begin{bmatrix} 0.2 & 0.4 & 0.1 \\ 0.3 & 0.2 & 0.2 \\ 0.5 & 0.3 & 0.5 \\ 0.1 & 0.1 & 0.1 \end{bmatrix}$$

- Initial radius: $N_c = 0$
- Initial learning rate: $\alpha(0) = 0.3$

Example: Repeat Steps 2-3 for Pattern 1

- Step 2: For the first input vector $(1, 1, 1, 0)$, do steps 3 - 5.
- Step 3:
$$I(1) = (1-0.2)^2 + (1-0.3)^2 + (1-0.5)^2 + (0-0.1)^2 = \mathbf{1.39}$$
$$I(2) = (1-0.4)^2 + (1-0.2)^2 + (1-0.3)^2 + (0-0.1)^2 = 1.5$$
$$I(3) = (1-0.1)^2 + (1-0.2)^2 + (1-0.5)^2 + (0-0.1)^2 = 1.71$$
- The input vector is closest to output node 1. Thus node 1 is the winner. The weights for node 1 should be updated.

Example: Repeat Step 4 for Pattern 1

- Step 4: weights on the winning unit are updated:

$$
\begin{aligned}
w^{new}(1) &= w^{old}(1) + \alpha(x - w^{old}(1)) \\
&= (0.2,\ 0.3,\ 0.5,\ 0.1) + 0.3(0.8,\ 0.7,\ 0.5,\ 0.9) \\
&= (0.44,\ 0.51,\ 0.65,\ 0.37)
\end{aligned}
$$

$$
W = \begin{bmatrix} 0.44 & 0.4 & 0.1 \\ 0.51 & 0.2 & 0.2 \\ 0.65 & 0.3 & 0.5 \\ 0.37 & 0.1 & 0.1 \end{bmatrix}
$$

Example: Repeat Steps 2-3 for Pattern 2

- Step 2: For the second input vector $(0, 0, 0, 1)$, do steps 3 - 5.
- Step 3:

$$
\begin{aligned}
I(1) &= (0 - 0.44)^2 + (0 - 0.51)^2 + (0 - 0.65)^2 + (1 - 0.37)^2 \\
&= 1.2731 \\
I(2) &= (0 - 0.4)^2 + (0 - 0.2)^2 + (0 - 0.3)^2 + (1 - 0.1)^2 = \mathbf{1.1} \\
I(3) &= (0 - 0.1)^2 + (0 - 0.2)^2 + (0 - 0.5)^2 + (1 - 0.1)^2 = 1.11
\end{aligned}
$$

- The input vector is closest to output node 2. Thus node 2 is the winner. The weights for node 2 should be updated.

Example: Repeat Step 4 for Pattern 2

- Step 4: weights on the winning unit are updated:

$$
\begin{aligned}
w^{new}(2) &= w^{old}(2) + \alpha(x - w^{old}(2)) \\
&= (0.4,\ 0.2,\ 0.3,\ 0.1) + 0.3(-0.4,\ -0.2,\ -0.3,\ 0.9) \\
&= (0.28,\ 0.14,\ 0.21,\ 0.37)
\end{aligned}
$$

$$
W = \begin{bmatrix} 0.44 & 0.28 & 0.1 \\ 0.51 & 0.14 & 0.2 \\ 0.65 & 0.21 & 0.5 \\ 0.37 & 0.37 & 0.1 \end{bmatrix}
$$

Example: Repeat Steps 2-3 for Pattern 3

- Step 2: For the second input vector $(1, 1, 0, 0)$, do steps 3 - 5:
- Step 3:

$$
\begin{aligned}
I(1) &= (1 - 0.44)^2 + (1 - 0.51)^2 + (0 - 0.65)^2 + (0 - 0.37)^2 \\
&= \mathbf{1.1131} \\
I(2) &= (1 - 0.28)^2 + (1 - 0.14)^2 + (0 - 0.21)^2 + (0 - 0.37)^2 \\
&= 1.439 \\
I(3) &= (1 - 0.1)^2 + (1 - 0.2)^2 + (0 - 0.5)^2 + (0 - 0.1)^2 = 1.71
\end{aligned}
$$

- The input vector is closest to output node 1. Thus node 1 is the winner. The weights for node 1 should be updated.

Example: Repeat Step 4 for Pattern 3

- Step 4: weights on the winning unit are updated:

$$
\begin{aligned}
w^{new}(1) &= w^{old}(1) + \alpha(x - w^{old}(1)) \\
&= (0.44, 0.51, 0.65, 0.37) + 0.3(0.56, 0.49, -0.65, -0.37) \\
&= (0.608,\ 0.657,\ 0.455,\ 0.259)
\end{aligned}
$$

$$
W = \begin{bmatrix} 0.608 & 0.28 & 0.1 \\ 0.657 & 0.14 & 0.2 \\ 0.455 & 0.21 & 0.5 \\ 0.259 & 0.37 & 0.1 \end{bmatrix}
$$

Example: Repeat Steps 2-3 for Pattern 4

- Step 2: For the second input vector $(0, 0, 1, 1)$, do steps 3 - 5:
- Step 3:

$$
\begin{aligned}
I(1) &= (0 - 0.608)^2 + (0 - 0.657)^2 + (1 - 0.455)^2 + (1 - 0.259)^2 \\
&= 1.647419 \\
I(2) &= (0 - 0.28)^2 + (0 - 0.14)^2 + (1 - 0.21)^2 + (1 - 0.37)^2 \\
&= 1.119 \\
I(3) &= (0 - 0.1)^2 + (0 - 0.2)^2 + (1 - 0.5)^2 + (1 - 0.1)^2 = \mathbf{1.11}
\end{aligned}
$$

- The input vector is closest to output node 3. Thus node 3 is the winner. The weights for node 3 should be updated.

Example: Repeat Step 4 for Pattern 4

- Step 4: weights on the winning unit are updated:

$$
\begin{aligned}
w^{new}(3) &= w^{old}(3) + \alpha(x - w^{old}(3)) \\
&= (0.1,\ 0.2,\ 0.5,\ 0.1) + 0.3(-0.1,\ -0.2,\ 0.5,\ 0.9) \\
&= (0.07,\ 0.14,\ 0.65,\ 0.37)
\end{aligned}
$$

$$
W = \begin{bmatrix} 0.608 & 0.28 & 0.07 \\ 0.657 & 0.14 & 0.14 \\ 0.455 & 0.21 & 0.65 \\ 0.259 & 0.37 & 0.37 \end{bmatrix}
$$

Example: Step 5

- Epoch 1 is complete.
- Reduce the learning rate:
  $\alpha(t+1) = 0.2\alpha(t) = 0.2(0.3) = 0.06$
- Repeat from the start for new epochs until Δw_j becomes steady for all input patterns or the error is within a tolerable range.

Applications

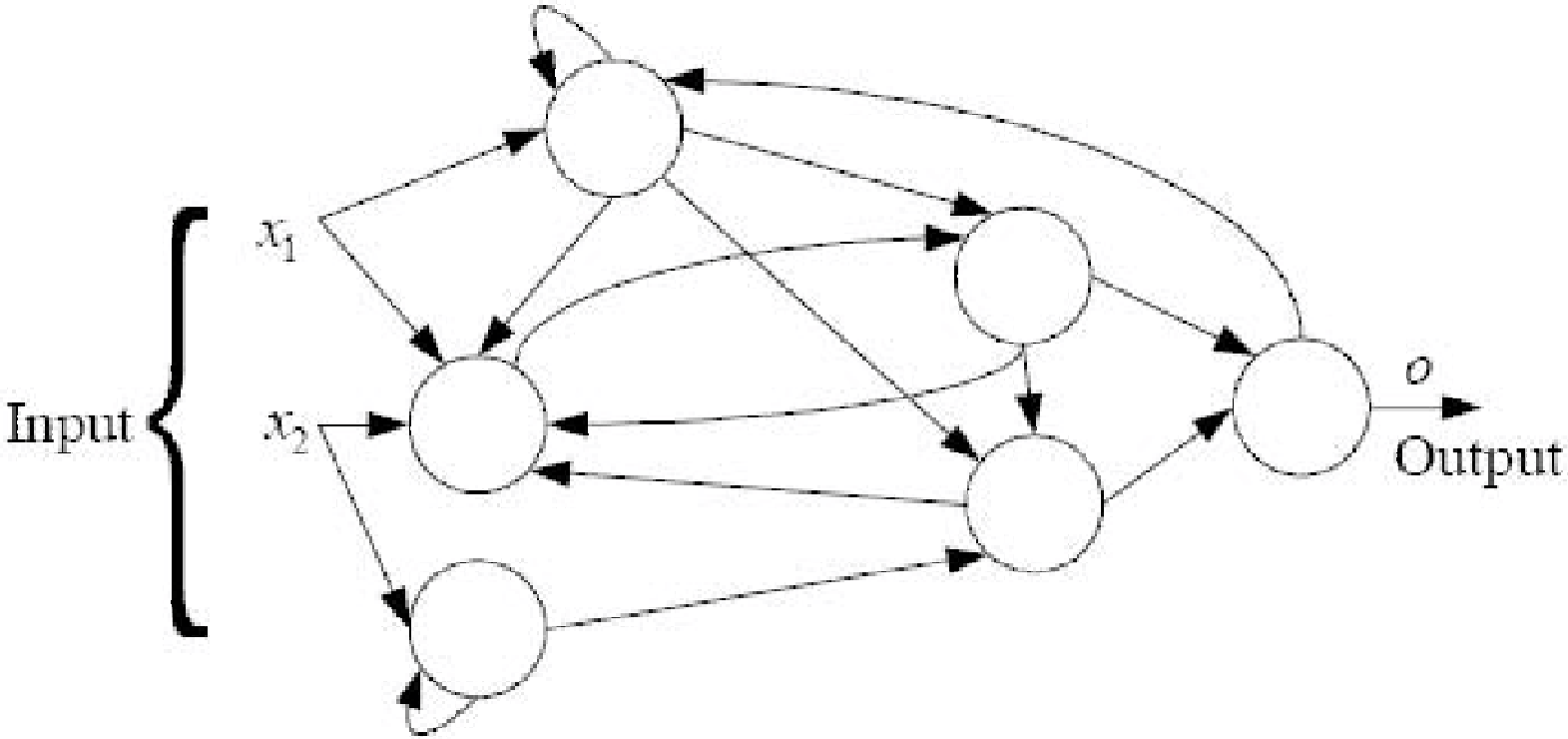
- A Variety of KSONs could be applied to different applications using the different parameters of the network, which are:
  - Neighborhood size,
  - Shape (circular, square, diamond),
  - Learning rate decaying behavior, and
  - Dimensionality of the neuron array (1-D, 2-D or n-D).

Applications (cont.)

- Given their self-organizing capabilities based on the competitive learning rule, KSONs have been used extensively for clustering applications such as
  - Speech recognition,
  - Vector coding,
  - Robotics applications, and
  - Texture segmentation.

Hopfield Network

Recurrent Topology



Origin

- A very special and interesting case of the recurrent topology.
- It is the pioneering work of Hopfield in the early 1980's that led the way for the designing of neural networks with feedback paths and dynamics.
- The work of Hopfield is seen by many as the starting point for the implementation of associative (content addressable) memory by using a special structure of recurrent neural networks.

Associative Memory Concept

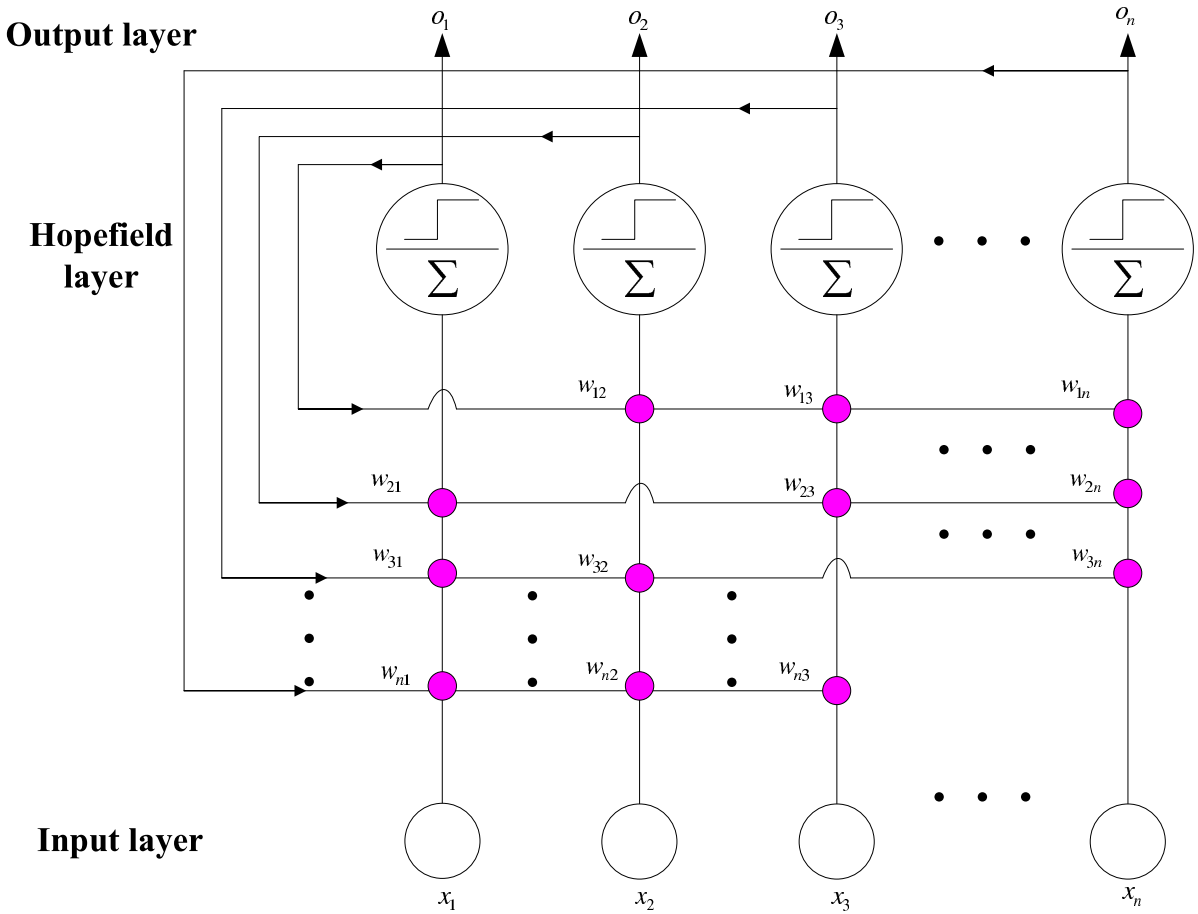
- The associative memory concept is able to recognize newly presented (noisy or incomplete) patterns using an already stored 'complete' version of that pattern.
- We say that the new pattern is 'attracted' to the stable pattern already stored in the network memories.
- This could be stated as having the network represented by an energy function that keeps decreasing until the system has reached stable status.

General Structure of the Hopfield Network

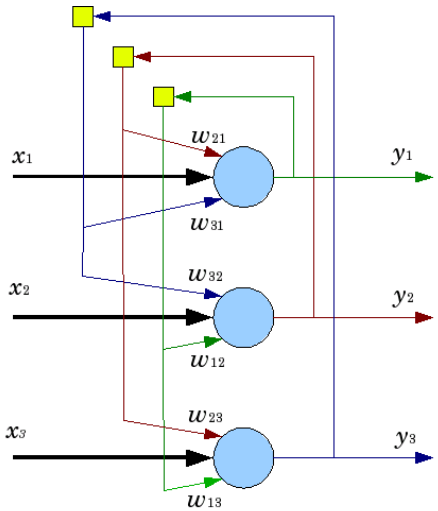
The structure of Hopfield network is made up of a number of processing units configured in one single layer (besides the input and the output layers) with symmetrical synaptic connections; i.e.,

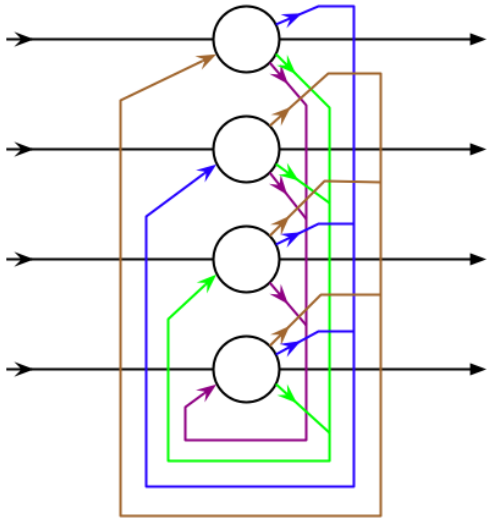
$$w_{ij} = w_{ji}$$

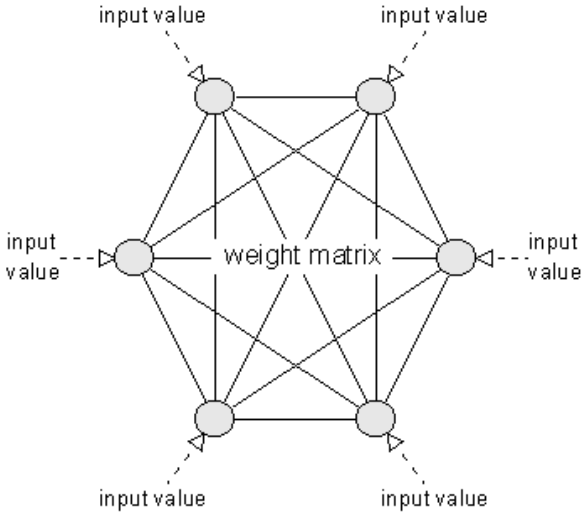
General Structure of the Hopfield Network (cont.)



Hopfield Network: Alternative Representations







Network Formulation

- In the original work of Hopfield, the output of each unit can take a binary value (either 0 or 1) or a bipolar value (either -1 or 1).
- This value is fed back to all the input units of the network except to the one corresponding to that output.
- Let us suppose here that the state of the network with dimension n (n neurons) takes bipolar values.

Network Formulation: Activation Function

- The activation rule for each neuron is provided by the following:

$$o_i = sign(\sum_{j=1}^{n} w_{ij} o_j - \theta_i) = \begin{cases} 1 & \text{if } \sum_{i \neq j} w_{ij} o_j > \theta_i \\ -1 & \text{if } \sum_{i \neq j} w_{ij} o_j < \theta_i \end{cases}$$

  - o_i: the output of the current processing unit (Hopfield neuron)
  - θ_i: threshold value

Network Formulation: Energy Function

- An energy function for the network

$$E = -1/2 \sum \sum_{i \neq j} w_{ij} o_i o_j + \sum o_i \theta_i$$

- E is so defined as to decrease monotonically with variation of the output states until a minimum is attained.

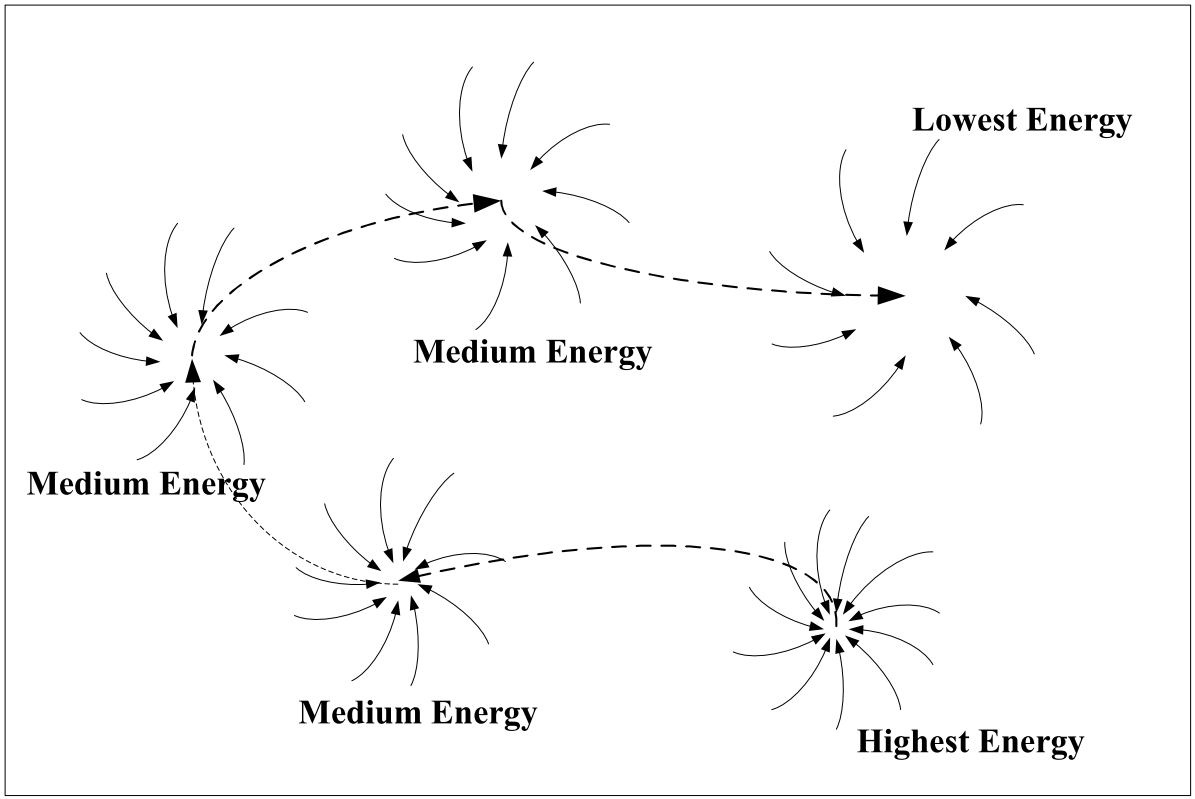
Network Formulation: Energy Function (cont.)

- This could be readily noticed from the expression relating the variation of E with respect to the output states variation.

$$\Delta E = -\Delta o_i \left(\sum_{i \neq j} w_{ij} o_j - \theta_i \right)$$

  - This expression shows that the energy function E of the network continues to decrease until it settles by reaching a local minimum.

Transition of Patterns from High Energy Levels to Lower Energy Levels



Hebbian Learning

- The learning algorithm for the Hopfield network is based on the so called Hebbian learning rule.
- This is one of the earliest procedures designed for carrying out supervised learning.
- It is based on the idea that when two units are simultaneously activated, their interconnection weight increase becomes proportional to the product of their two activities.

Hebbian Learning (cont.)

- The Hebbian learning rule also known as the outer product rule of storage, as applied to a set of q presented patterns $p_k (k = 1, ..., q)$ each with dimension n (n denotes the number of neuron units in the Hopfield network), is expressed as:

$$w_{ij} = \begin{cases} \frac{1}{n} \sum_{k=1}^{q} p_{kj} p_{ki} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

- The weight matrix $W = \{w_{ij}\}$ could also be expressed in terms of the outer product of the vector p_k as:

$$W = \{w_{ij}\} = \frac{1}{n} \sum_{k=1}^{q} p_k p_k^T - \frac{q}{n} \mathrm{I}$$

Learning Algorithm

- *Step 1 (storage)*: The first stage is to store the patterns through establishing the connection weights. Each of the q fundamental memories presented is a vector of bipolar elements (+1 or -1).
- *Step 2 (initialization)*: The second stage is initialization and consists in presenting to the network an unknown pattern u with same dimension as the fundamental patterns.

  Every component of the network outputs at the initial iteration cycle is set as

$$o(0) = u$$

Learning Algorithm

- *Step 1 (storage)*: The first stage is to store the patterns through establishing the connection weights. Each of the q fundamental memories presented is a vector of bipolar elements (+1 or -1).
- *Step 2 (initialization)*: The second stage is initialization and consists in presenting to the network an unknown pattern u with same dimension as the fundamental patterns.

  Every component of the network outputs at the initial iteration cycle is set as

$$o(0) = u$$

Learning Algorithm (cont.)

- *Step 3 (retrieval 1)*: Each one of the component o_i of the output vector o is updated from cycle l to cycle $l+1$ by:

$$o_i(l+1) = sgn(\sum_{j=1}^{n} w_{ij} o_j(l))$$

  - This process is known as asynchronous updating.
  - The process continues until no more changes are made and convergence occurs.

- *Step 4 (retrieval 2)*: Continue the process for other presented unknown patterns by starting again from step 2.

Learning Algorithm (cont.)

- *Step 3 (retrieval 1)*: Each one of the component o_i of the output vector o is updated from cycle l to cycle $l+1$ by:

$$o_i(l+1) = sgn(\sum_{j=1}^{n} w_{ij} o_j(l))$$

  - This process is known as asynchronous updating.
  - The process continues until no more changes are made and convergence occurs.

- *Step 4 (retrieval 2)*: Continue the process for other presented unknown patterns by starting again from step 2.

Learning Algorithm (cont.)

- *Step 3 (retrieval 1)*: Each one of the component o_i of the output vector o is updated from cycle l to cycle $l+1$ by:

$$o_i(l+1) = sgn(\sum_{j=1}^{n} w_{ij} o_j(l))$$

  - This process is known as asynchronous updating.
  - The process continues until no more changes are made and convergence occurs.

- *Step 4 (retrieval 2)*: Continue the process for other presented unknown patterns by starting again from step 2.

Example

Problem Statement

- We need to store a **fundamental pattern (memory)** given by the vector $O = [1, 1, 1, -1]^T$ in a four node binary Hopefield network.
- Presume that the threshold parameters are all equal to zero.

Establishing Connection Weights

- Weight matrix expression discarding $1/4$ and having $q = 1$

$$W = \frac{1}{n}\sum_{k=1}^{q} p_k p_k^T - \frac{q}{n}\mathrm{I} = p_1 p_1^T - \mathrm{I}$$

- Therefore:

$$W = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & -1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

Network' States and Their Code

Total number of states: There are $2^n = 2^4 = 16$ different states.

State	Code			
A	1	1	1	1
B	1	1	1	-1
C	1	1	-1	-1
D	1	1	-1	1
E	1	-1	-1	1
F	1	-1	-1	-1
G	1	-1	1	-1
H	1	-1	1	1

State	Code			
I	-1	-1	1	1
J	-1	-1	1	-1
K	-1	-1	-1	-1
L	-1	-1	-1	1
M	-1	1	-1	1
N	-1	1	-1	-1
O	-1	1	1	-1
P	-1	1	1	1

Computing Energy Level of State $A = [1, 1, 1, 1]$

- All thresholds are equal to zero: $\theta_i = 0, \quad i = 1, 2, 3, 4\cdot$ Therefore,

$$E = -1/2 \sum_{i=1}^{4} \sum_{j=1}^{4} w_{ij} o_i o_j$$

$$\begin{aligned} E = -1/2 (& w_{11} o_1 o_1 + w_{12} o_1 o_2 + w_{13} o_1 o_3 + w_{14} o_1 o_4 + \\ & w_{21} o_2 o_1 + w_{22} o_2 o_2 + w_{23} o_2 o_3 + w_{24} o_2 o_4 + \\ & w_{31} o_3 o_1 + w_{32} o_3 o_2 + w_{33} o_3 o_3 + w_{34} o_3 o_4 + \\ & w_{41} o_4 o_1 + w_{42} o_4 o_2 + w_{43} o_4 o_3 + w_{44} o_4 o_4) \end{aligned}$$

Computing Energy Level of State A (cont.)

- For state A, we have $A = [o_1, o_2, o_3, o_4] = [1, 1, 1, 1]$. Thus,

$$\begin{aligned} E = -1/2(&0 + (1)(1)(1) + (1)(1)(1) + (-1)(1)(1) + \\ &(1)(1)(1) + 0 + (1)(1)(1) + (-1)(1)(1) + \\ &(1)(1)(1) + (1)(1)(1) + 0 + (-1)(1)(1) + \\ &(-1)(1)(1) + (-1)(1)(1) + (-1)(1)(1) + 0) \end{aligned}$$

$$\begin{aligned} E = -1/2(&0 + 1 + 1 - 1 + \\ &1 + 0 + 1 - 1 + \\ &1 + 1 + 0 - 1 + \\ &-1 - 1 - 1 + 0) \end{aligned}$$

$$E = -1/2(6 - 6) = 0$$

Energy Level of All States

- Similarly, we can compute the energy level of the other states.
- Two potential attractors: the original **fundamental pattern** $[1, 1, 1, -1]^T$ and its **complement** $[-1, -1, -1, 1]^T$.

State	Code				Energy
A	1	1	1	1	0
B	1	1	1	-1	-6
C	1	1	-1	-1	0
D	1	1	-1	1	2
E	1	-1	-1	1	0
F	1	-1	-1	-1	2
G	1	-1	1	-1	0
H	1	-1	1	1	2
I	-1	-1	1	1	0
J	-1	-1	1	-1	2
K	-1	-1	-1	-1	0
L	-1	-1	-1	1	-6
M	-1	1	-1	1	0
N	-1	1	-1	-1	2
O	-1	1	1	-1	0
P	-1	1	1	-1	2

Retrieval Stage

- We update the components of each state asynchronously using equation:

$$o_i = sgn(\sum_{j=1}^{n} w_{ij} o_j - \theta_i)$$

- Updating the state asynchronously means that for every state presented we activate one neuron at a time.
- **All states** change from **high energy** to **low energy levels**.

State Transition for State $J = [-1, -1, 1, -1]^T$

Transition 1 (o_1)

$$
\begin{aligned}
o_1 &= sgn(\sum_{j=1}^{4} w_{ij} o_j - \theta_i) = sgn(w_{12} o_2 + w_{13} o_3 + w_{14} o_4 - 0) \\
&= sgn((1)(-1) + (1)(1) + (-1)(-1)) \\
&= sgn(+1) \\
&= +1
\end{aligned}
$$

- As a result, the first component of the state J changes from -1 to 1. In other words, the state J transits to the state G at the end of first transition.

$$J = [-1, -1, 1, -1]^T \ (2) \rightarrow G = [1, -1, 1, -1]^T \ (0)$$

State Transition for State J (cont.)

Transition 2 (o_2)

$$
\begin{aligned}
o_2 &= sgn(\sum_{j=1}^{4} w_{ij} o_j - \theta_i) = sgn(w_{21} o_1 + w_{23} o_3 + w_{24} o_4) \\
&= sgn((1)(1) + (1)(1) + (-1)(-1)) \\
&= sgn(+3) \\
&= +1
\end{aligned}
$$

- As a result, the second component of the state G changes from -1 to 1. In other words, the state G transits to the state B at the end of first transition.

$$G = [1, -1, 1, -1]^T \ (0) \rightarrow B = [1, 1, 1, -1]^T \ (-6)$$

State Transition for State J (cont.)

Transition 3 (o_3)

As state B is a fundamental pattern, no more transition will occur. Let us see!

$$\begin{aligned} o_3 &= sgn(\sum_{j=1}^{4} w_{ij} o_j - \theta_i) = sgn(w_{31} o_1 + w_{32} o_2 + w_{34} o_4) \\ &= sgn((1)(1) + (1)(1) + (-1)(-1)) \\ &= sgn(+3) \\ &= +1 \end{aligned}$$

- No transition is observed.

$$B = [1, 1, 1, -1]^T \ (-6) \rightarrow B = [1, 1, 1, -1]^T \ (-6)$$

State Transition for State J (cont.)

Transition 4 (o_4)

Again as state B is a fundamental pattern, no more transition will occur. Let us see!

$$
\begin{aligned}
o_4 &= sgn(\sum_{j=1}^{4} w_{ij} o_j - \theta_i) = sgn(w_{41} o_1 + w_{42} o_2 + w_{43} o_3) \\
&= sgn((-1)(1) + (-1)(1) + (-1)(1)) \\
&= sgn(-3) \\
&= -1
\end{aligned}
$$

- No transition is observed.

$$B = [1, 1, 1, -1]^T \ (-6) \rightarrow B = [1, 1, 1, -1]^T \ (-6)$$

Asynchronous State Transition Table

By repeating the same procedure for the other states, asynchronous transition table is easily obtained.

State	Code	Transition 1 (o_1)	Transition 2 (o_2)	Transition 3 (o_3)	Transition 4 (o_4)
A	1 1 1 1	1 1 1 1 (A)	1 1 1 1 (A)	1 1 1 1 (A)	1 1 1 -1 (B)
B	1 1 1 -1	1 1 1 -1 (B)	1 1 1 -1 (B)	1 1 1 -1 (B)	1 1 1 -1 (B)
C	1 1 -1 -1	1 1 -1 -1 (C)	1 1 -1 -1 (C)	1 1 1 -1 (B)	1 1 -1 1 (B)
D	1 1 -1 1	-1 1 -1 1 (M)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)
E	1 -1 -1 1	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)
F	1 -1 -1 -1	-1 -1 -1 -1 (K)	-1 -1 -1 -1 (K)	-1 -1 -1 -1 (K)	-1 -1 -1 1 (L)
G	1 -1 1 -1	1 -1 1 -1 (G)	1 1 1 -1 (B)	1 1 1 -1 (B)	1 1 1 -1 (B)
H	1 -1 1 1	-1 -1 1 1 (I)	-1 -1 1 1 (I)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)
I	-1 -1 1 1	-1 -1 1 1 (I)	-1 -1 1 1 (I)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)
J	-1 -1 1 -1	1 -1 1 -1 (G)	1 1 1 -1 (B)	1 1 1 -1 (B)	1 1 1 -1 (B)
K	-1 -1 -1 -1	-1 -1 -1 -1 (K)	-1 -1 -1 -1 (K)	-1 -1 -1 -1 (K)	-1 -1 -1 1 (L)
L	-1 -1 -1 1	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)
M	-1 1 -1 1	-1 1 -1 1 (M)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)	-1 -1 -1 1 (L)
N	-1 1 -1 -1	1 1 -1 -1 (C)	1 1 -1 -1 (C)	1 1 1 -1 (B)	1 1 1 -1 (B)
O	-1 1 1 -1	1 1 1 -1 (B)	1 1 1 -1 (B)	1 1 1 -1 (B)	1 1 1 -1 (B)
P	-1 1 1 1	1 1 1 1 (A)	1 1 1 1 (A)	1 1 1 1 (A)	1 1 1 -1 (B)

Some Sample Transitions

Fundamental Pattern $B = [1, 1, 1, -1]^T$

- There is no change of the energy level and no transition occurs to any other state.
- It is in its stable state because this state has the lowest energy.

State $A = [1, 1, 1, 1]^T$

- Only the forth element o_4 is updated asynchronously.
- The state transits to $O = [1, 1, 1, -1]^T$, representing the fundamental pattern with the lowest energy value "-6".

Some Sample Transitions (cont.)

Complement of Fundamental Pattern $L = [-1, -1, -1, 1]^T$

- Its energy level is the same as B and hence it is another stable state.
- **Every complement of a fundamental pattern is a fundamental pattern itself**.
- This means that the Hopefield network has the ability to remember the fundamental memory and its complement.

Some Sample Transitions (cont.)

State $D = [1, 1, -1, 1]^T$

It could transit a few times to end up at state C after being updated asynchronously.

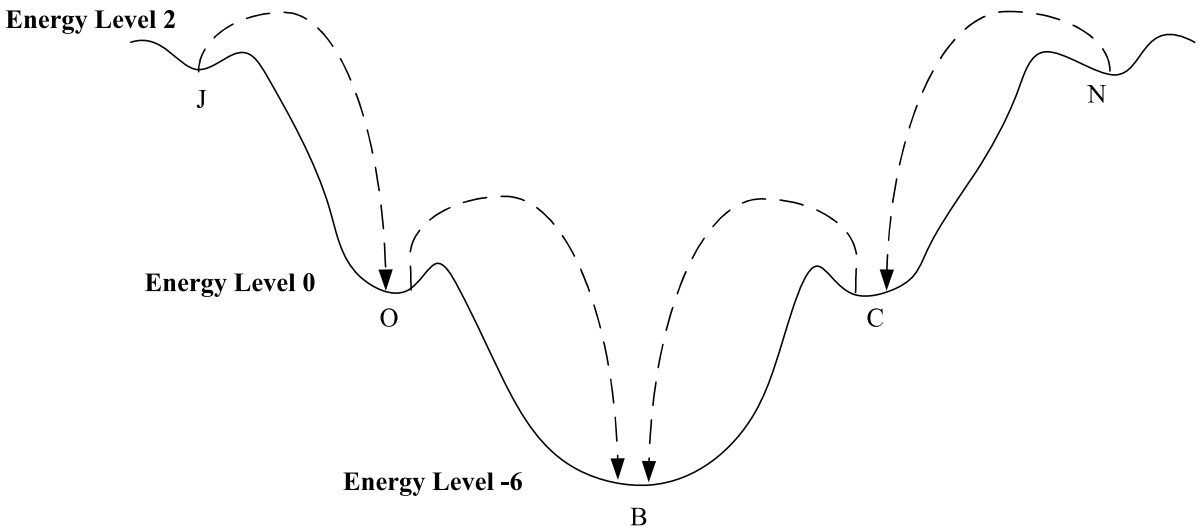
- Update the bit o_1, the state becomes $M = [-1, 1, -1, 1]^T$ with energy 0
- Update the bit o_2, the state becomes $E = [1, -1, -1, 1]^T$ with energy 0
- Update the bit o_3, the state becomes $A = [1, 1, 1, 1]^T$, the state A with energy 0
- Update the bit o_4, the state becomes $C = [1, 1, -1, -1]^T$ with energy 0

Some Sample Transitions (cont.)

State D: Remarks

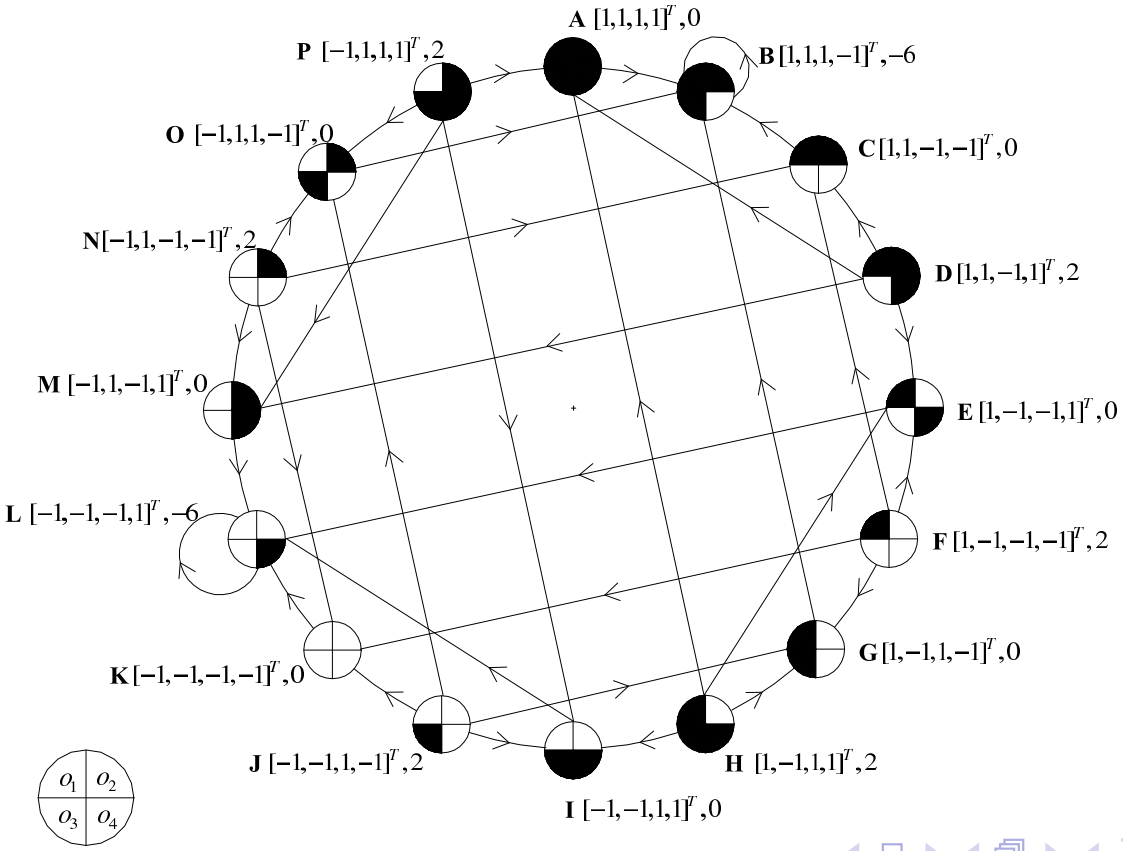
- From the process we know that state D can transit to four different states.
  - This depends on which bit is being updated.
  - If the state D transits to state A or C, it will continue the updating and ultimately transits to the fundamental state B, which has the energy -6, the lowest energy.
  - If the state D transits to state E or M, it will continue the updating and ultimately transits to state L, which also has the lowest energy -6.

Transition of States J and N from High Energy Levels to Low Energy Levels



State Transition Diagram

- Each node is characterized by its vector state and its energy level.



Applications

- Information retrieval and for pattern and speech recognition,
- Optimization problems,
- Combinatorial optimization problems such as the traveling salesman problem.

Limitations

- Limited stable-state storage capacity of the network,
- Hopfield estimated roughly that a network with n processing units should allow for $0.15n$ stable states.
- Many studies have been carried out recently to increase the capacity of the network without increasing much the number of the processing units