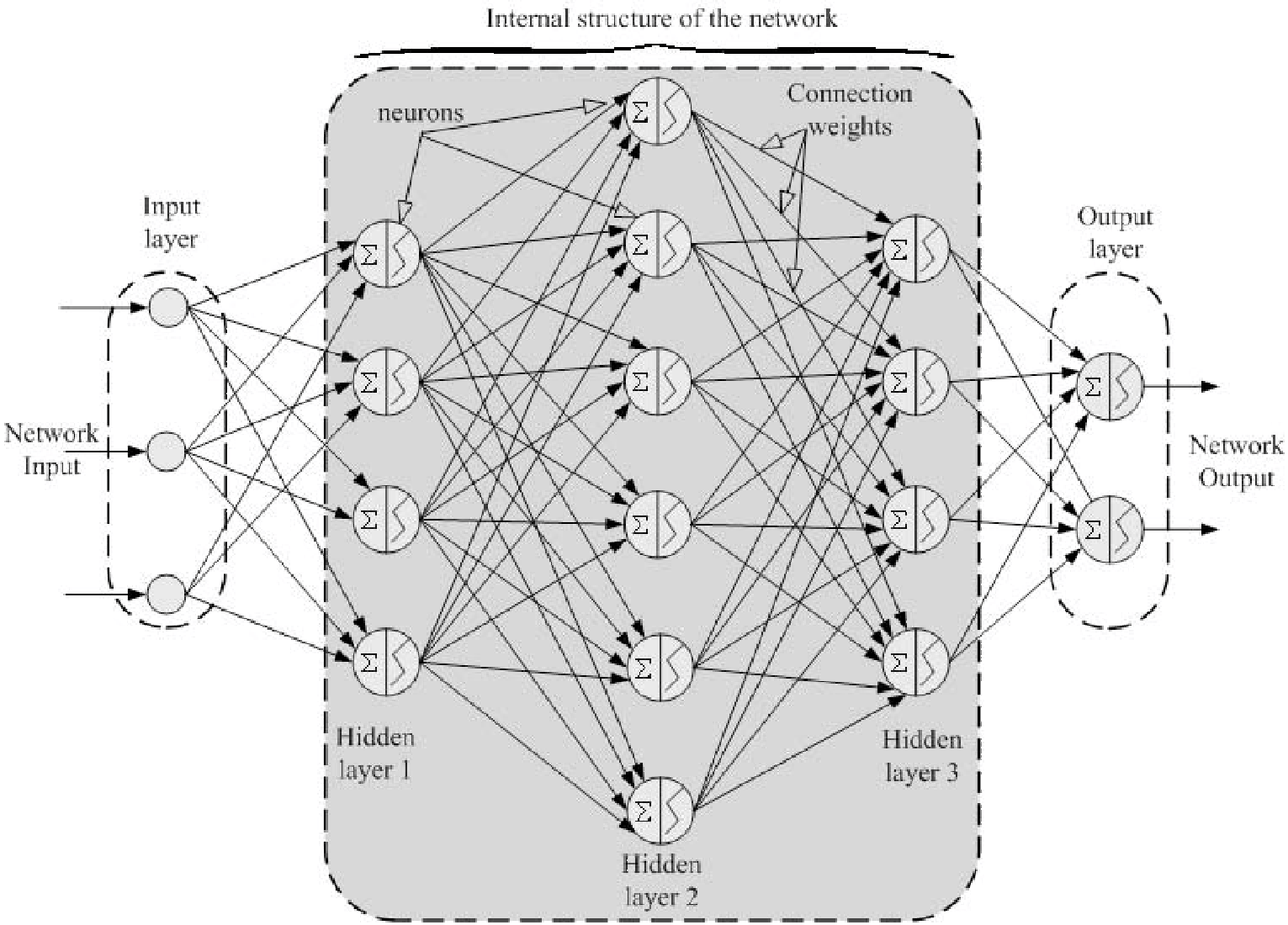
Fundamentals of Artificial Neural Networks

Outline

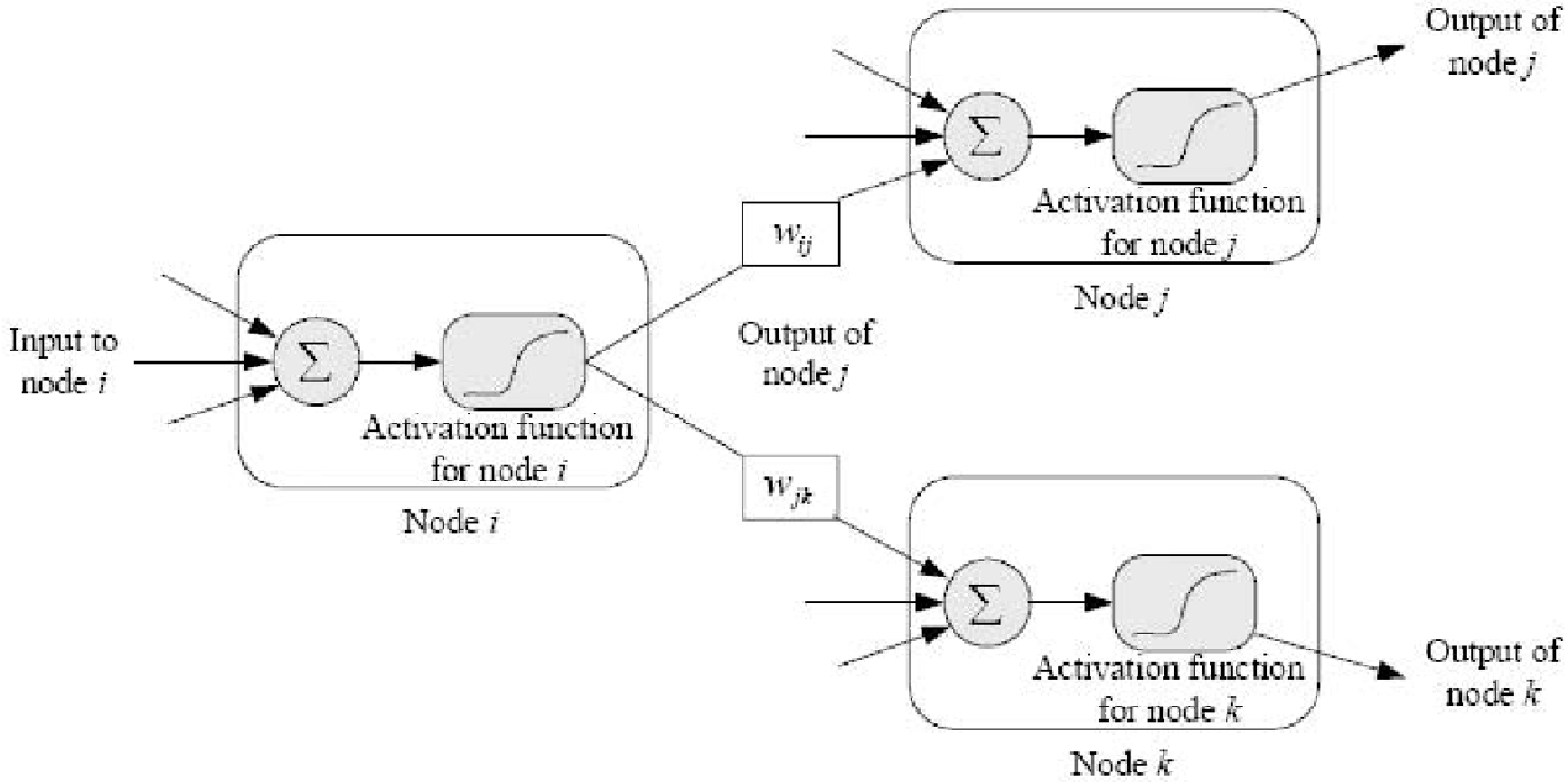
Introduction

- 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

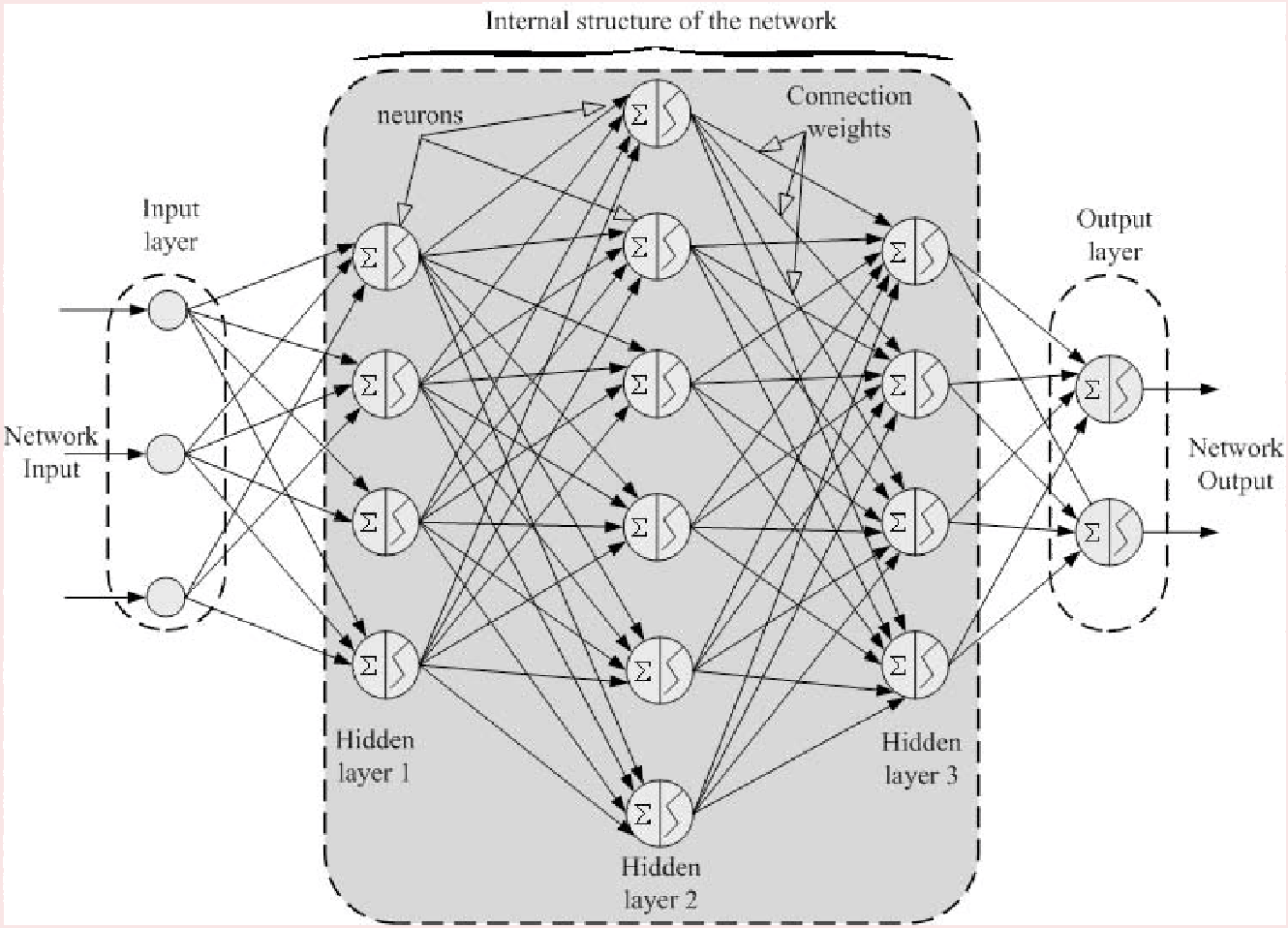
Binary
Continuous

Learning Paradigms

Supervised
Unsupervised
Hybrid

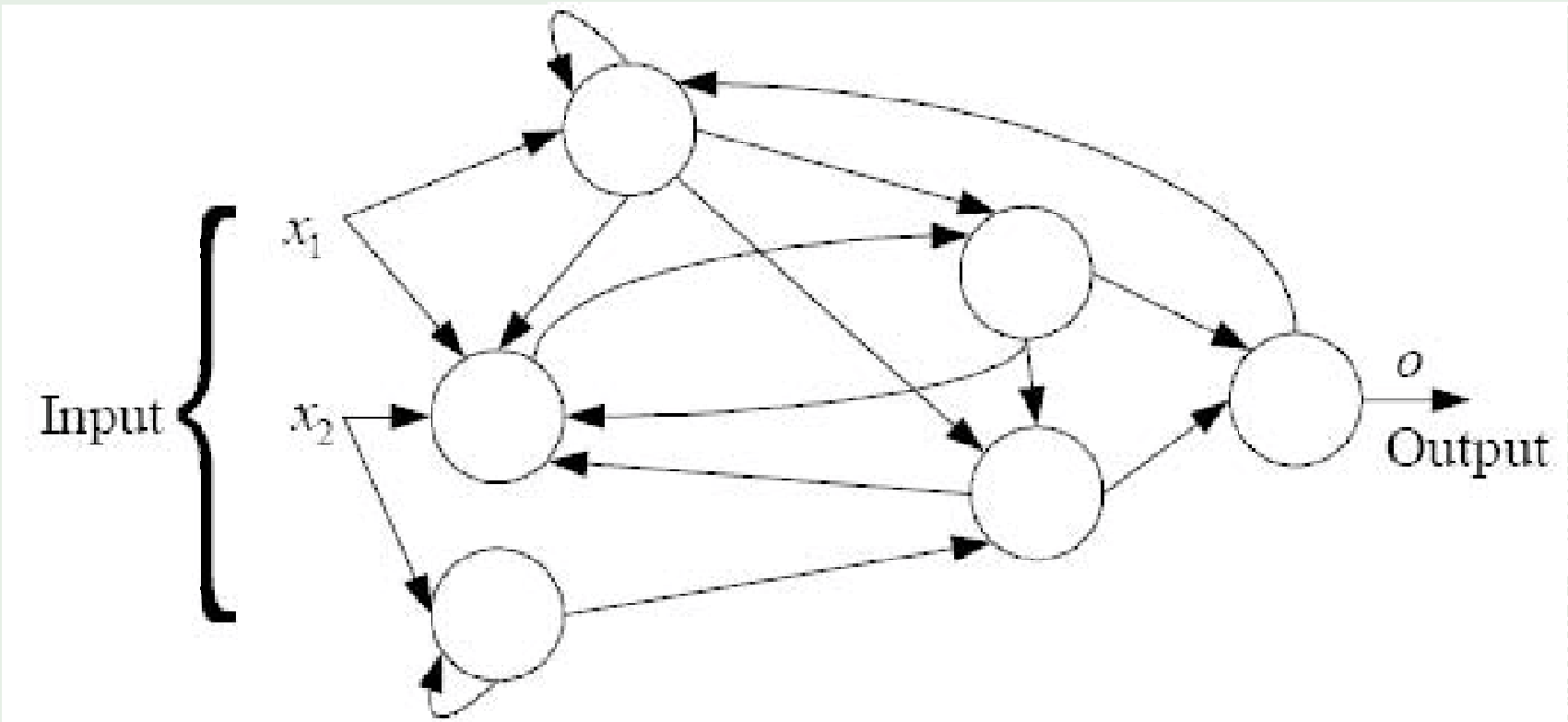
Neural Network Topologies

Feedforward Flow of Information



Neural Network Topologies (cont.)

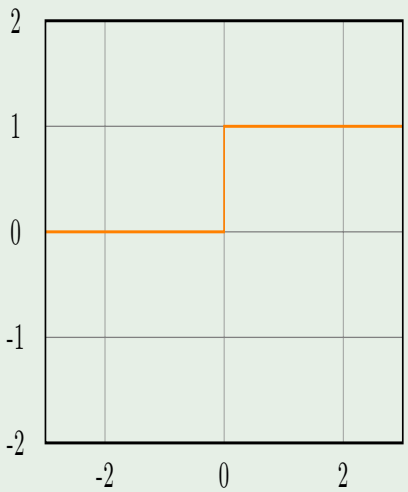
Recurrent Flow of Information



Binary Activation Functions

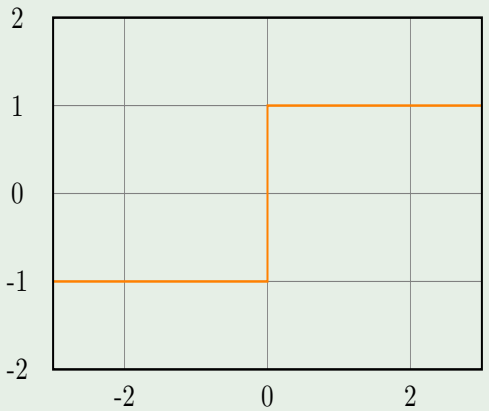
Step Function

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



Signum Function

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

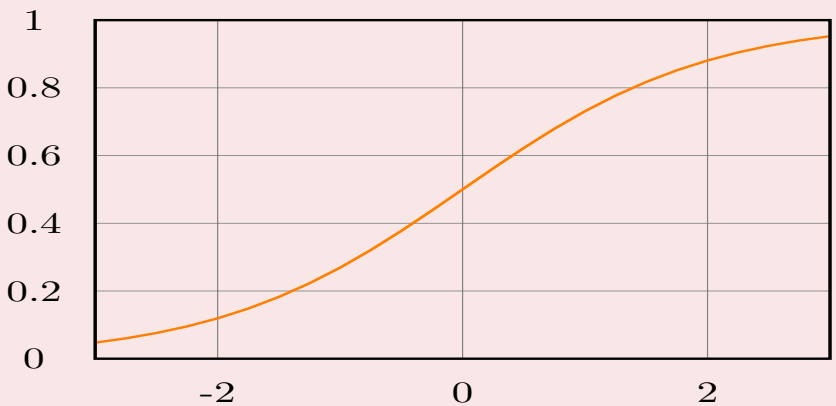


Differentiable Activation Functions

Differentiable functions

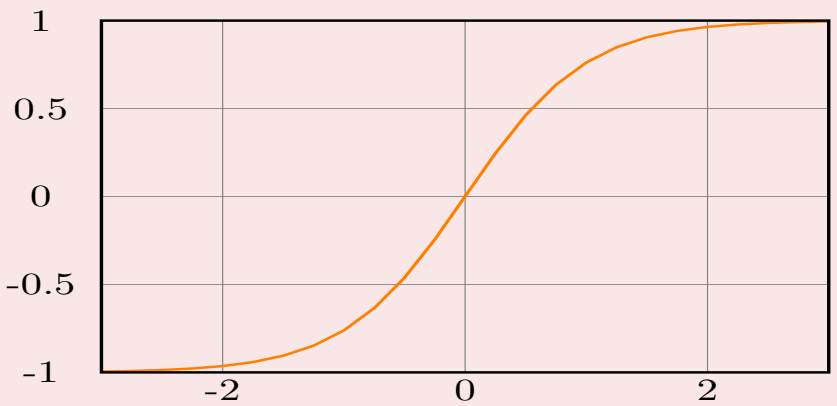
Sigmoid function

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



Hyperbolic tangent

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

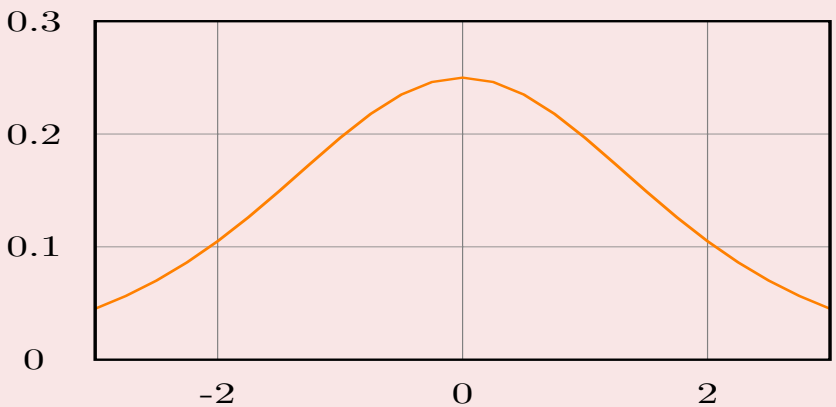


Differentiable Activation Functions (cont.)

Differentiable functions

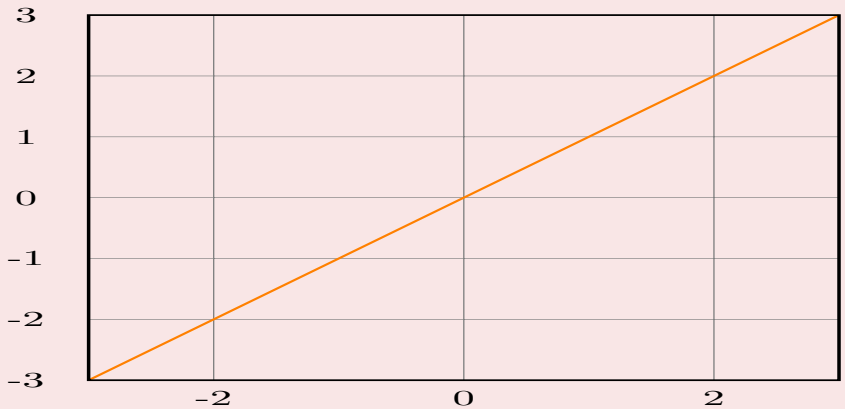
Sigmoid derivative

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



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

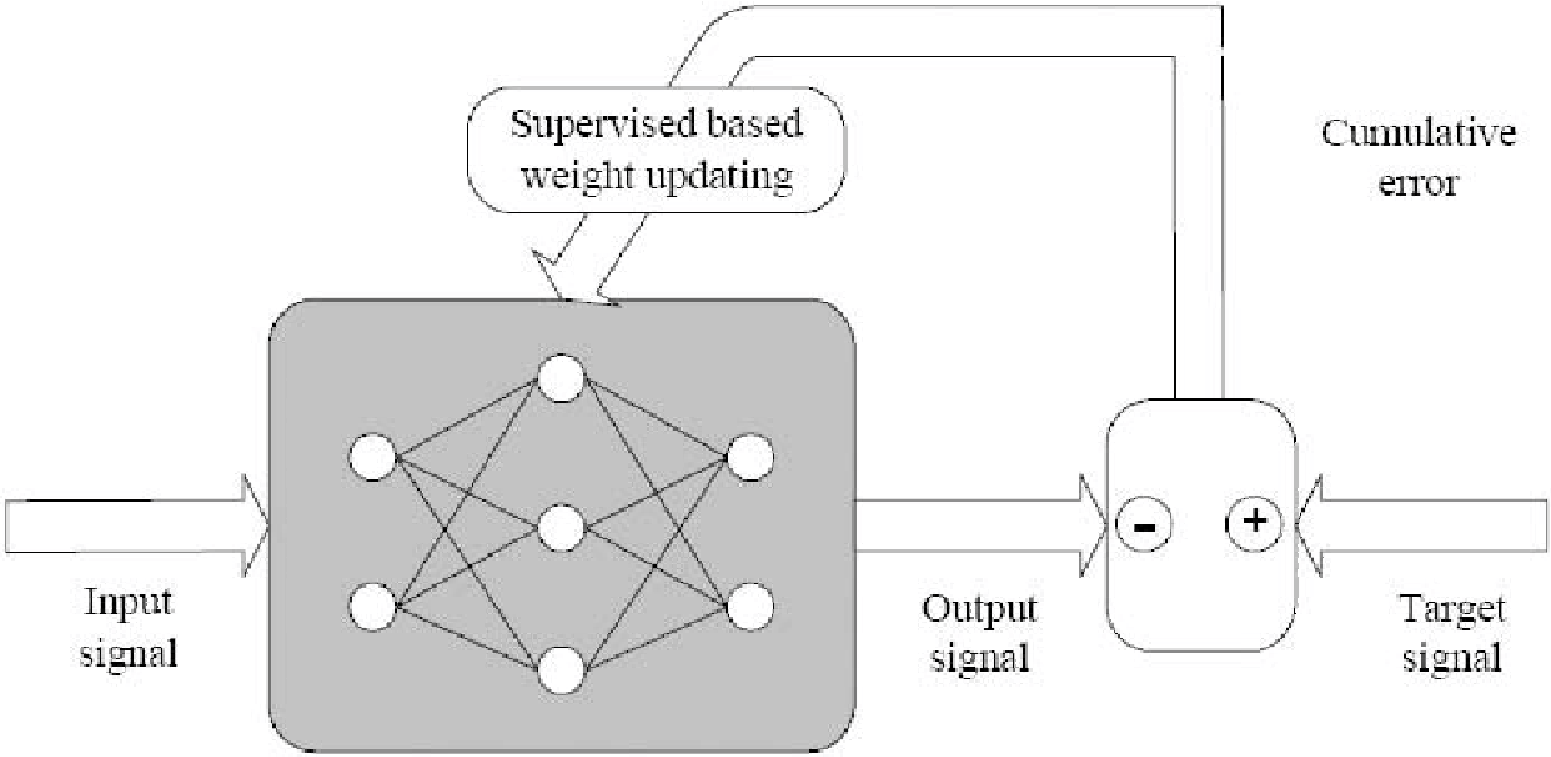
- Competitive learning networks
- Kohonen self-organizing networks
- ART (adaptive resonant theory)

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.)

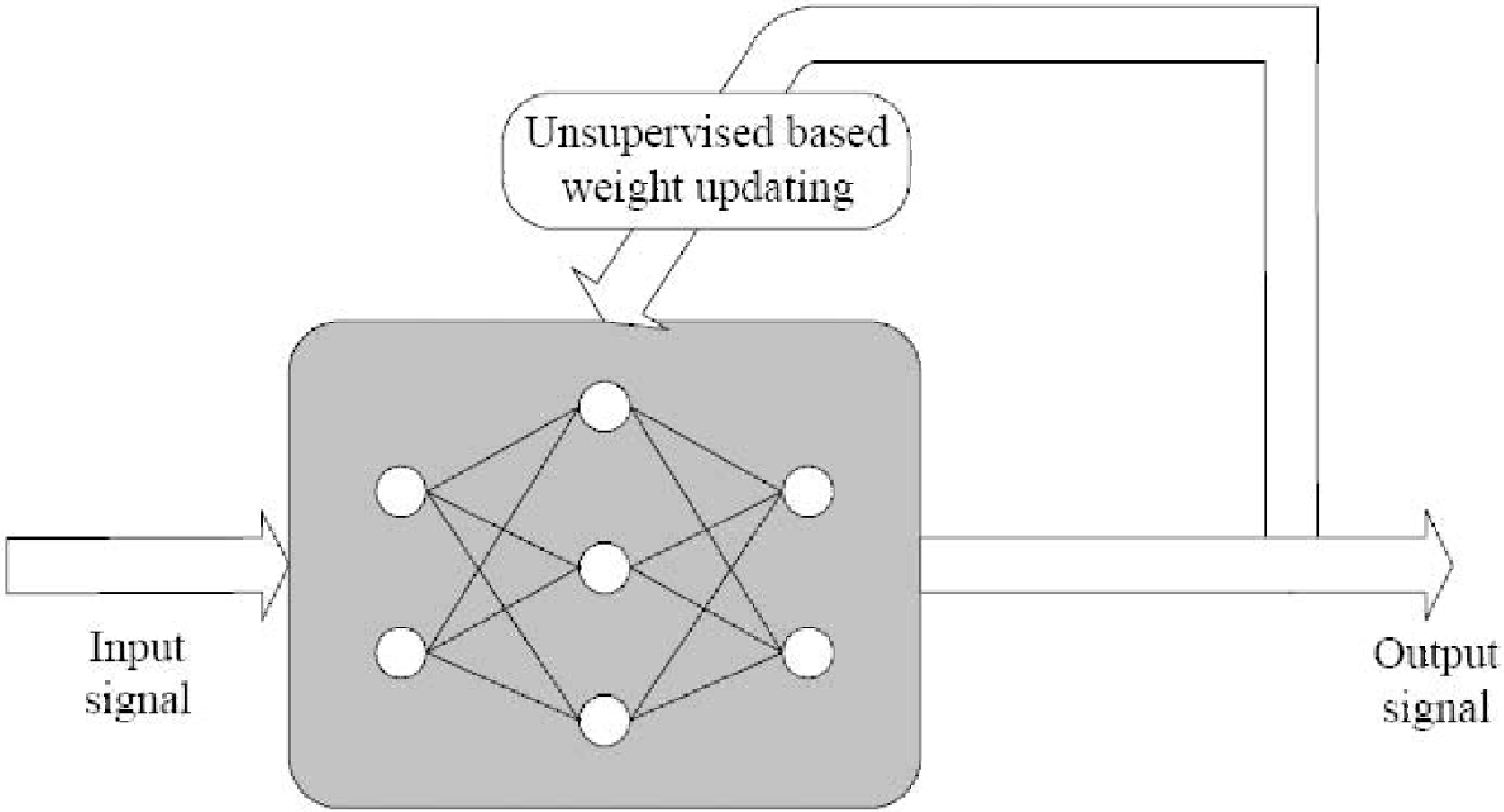
Training Algorithm

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

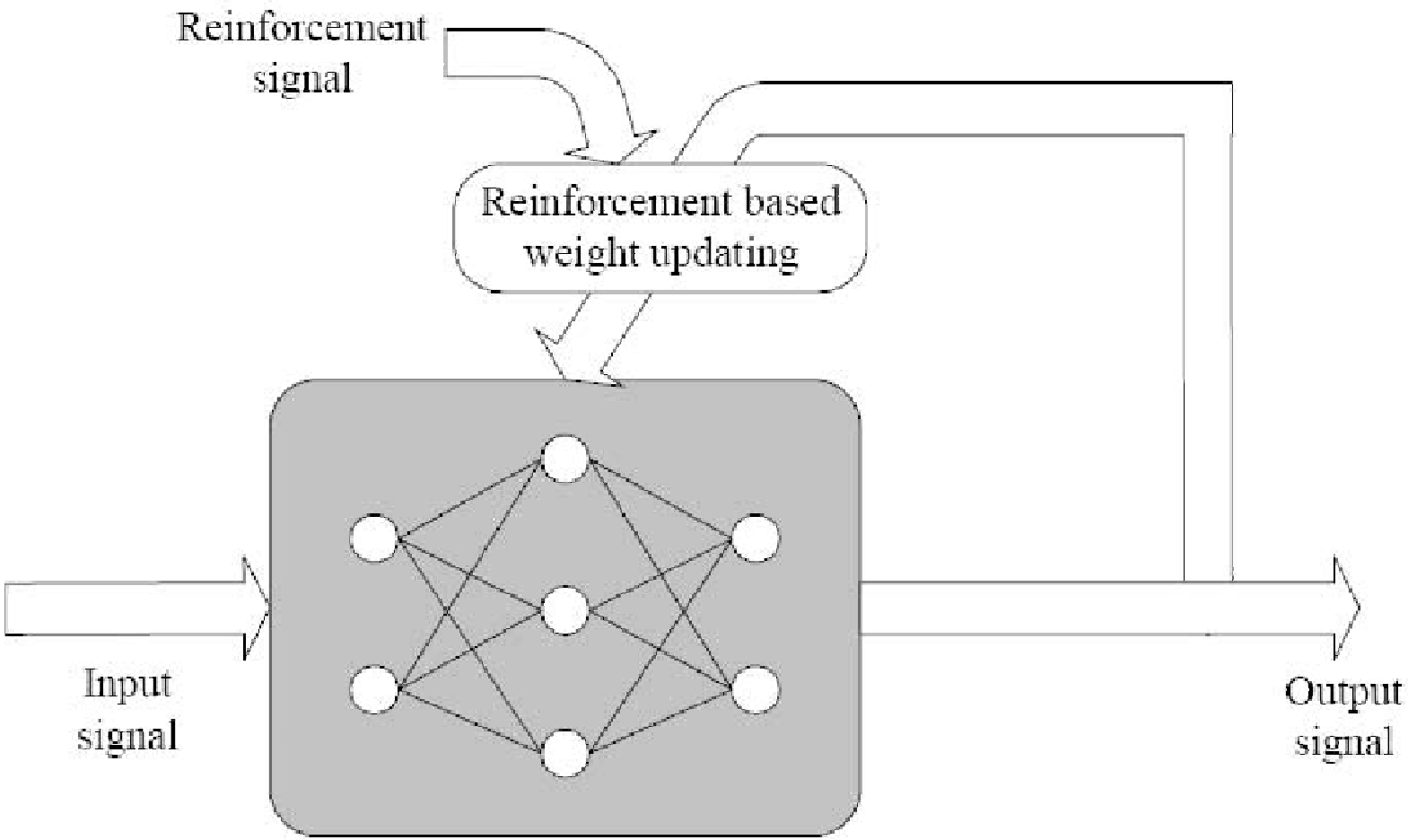
Rationale

- 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)

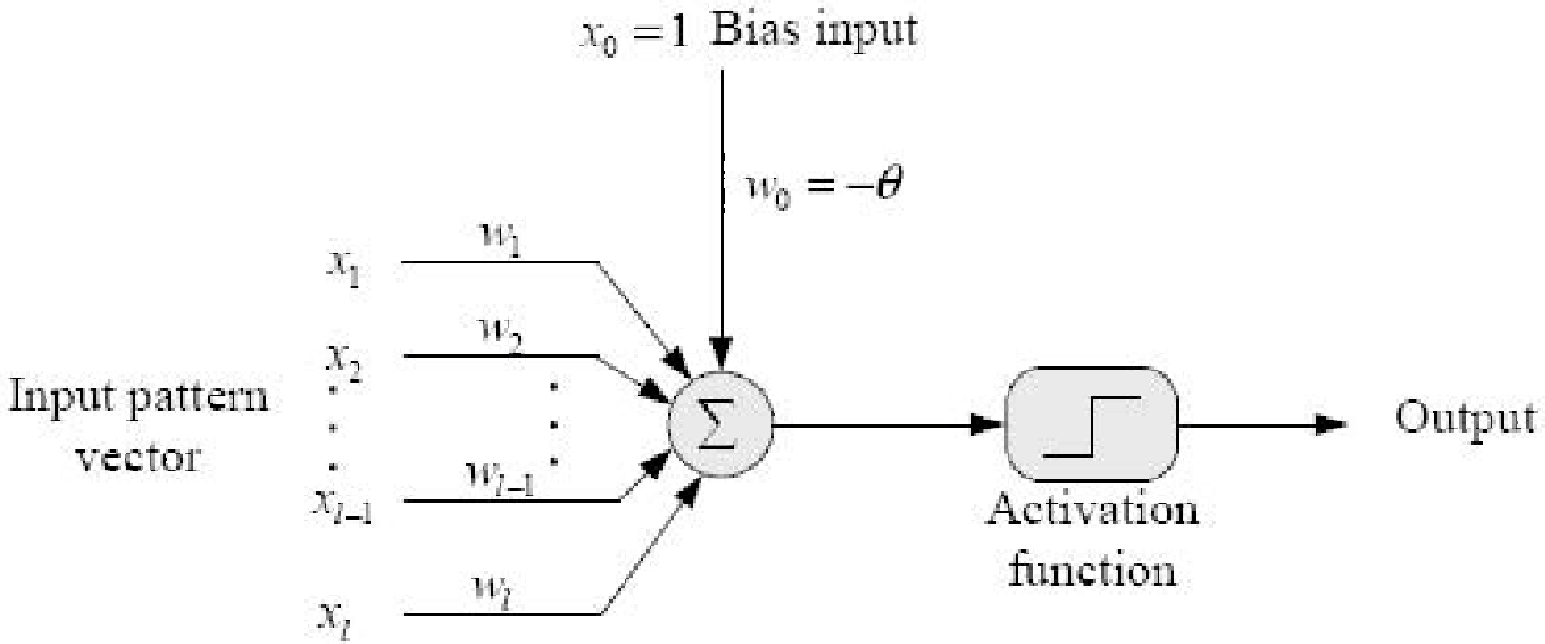
Mid 1980's : BPL II and Multi Layer Perceptron (by Rumelhart and Hinton)

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

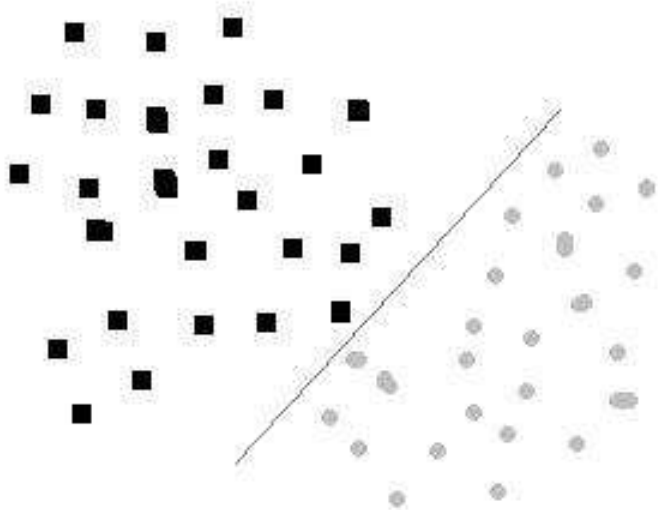
Overview

- 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.

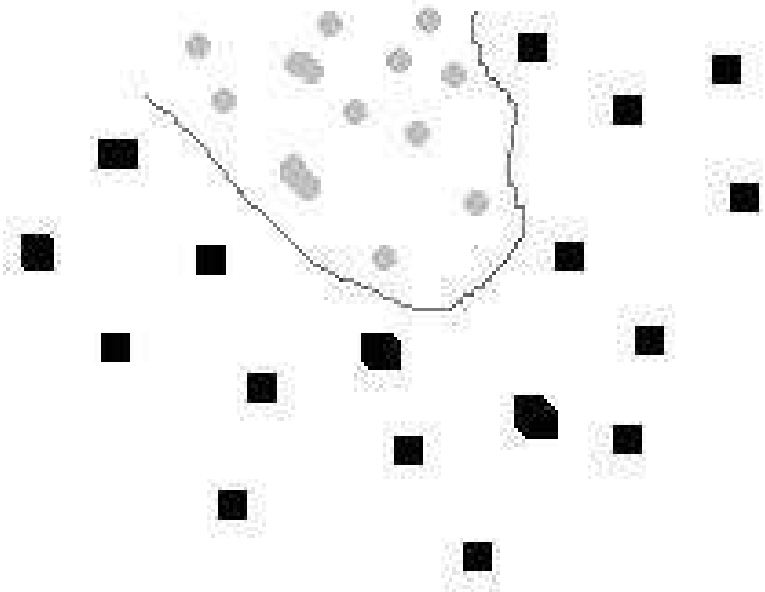
Definition: Linear Separation

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

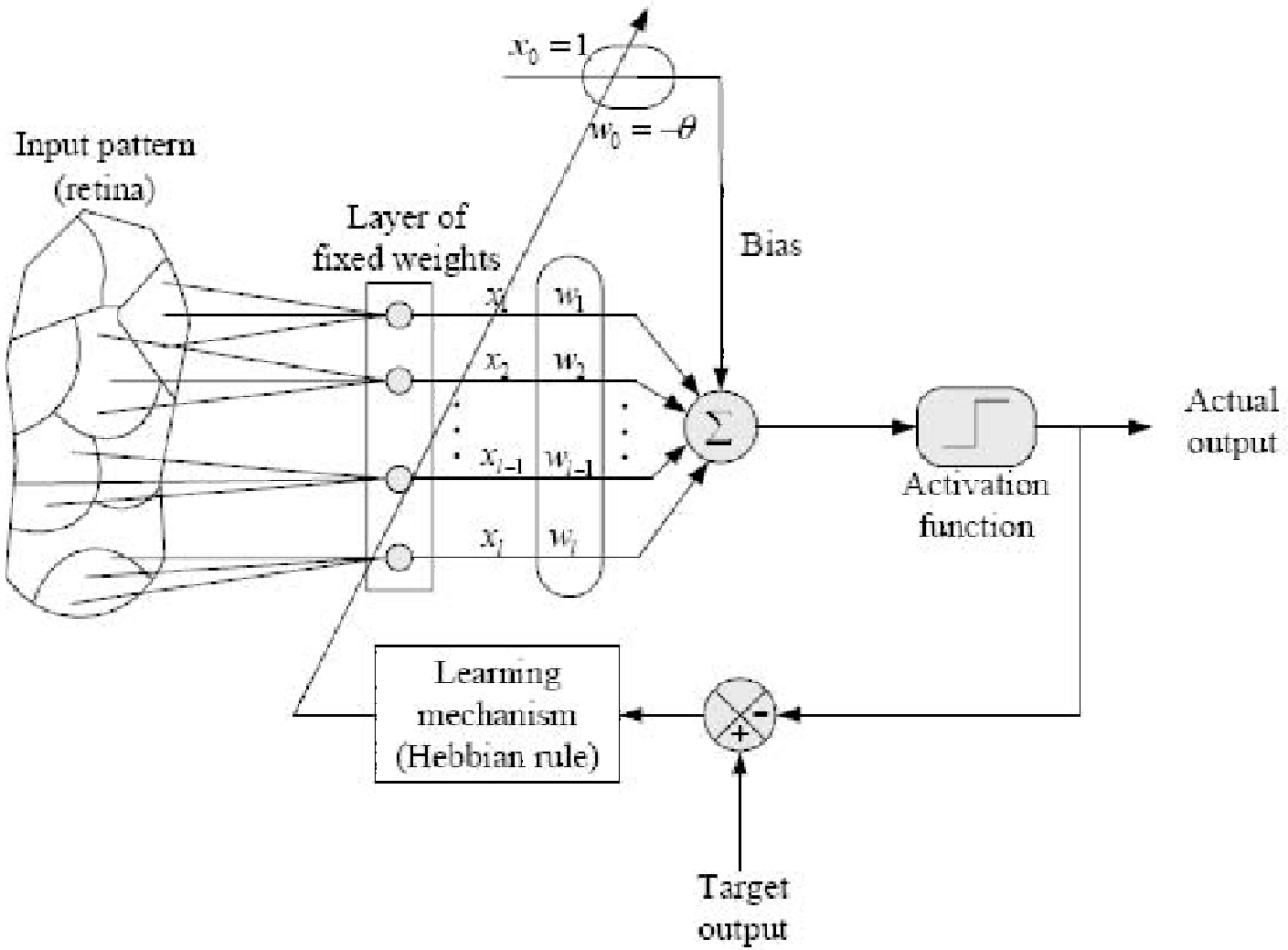


Perceptron

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

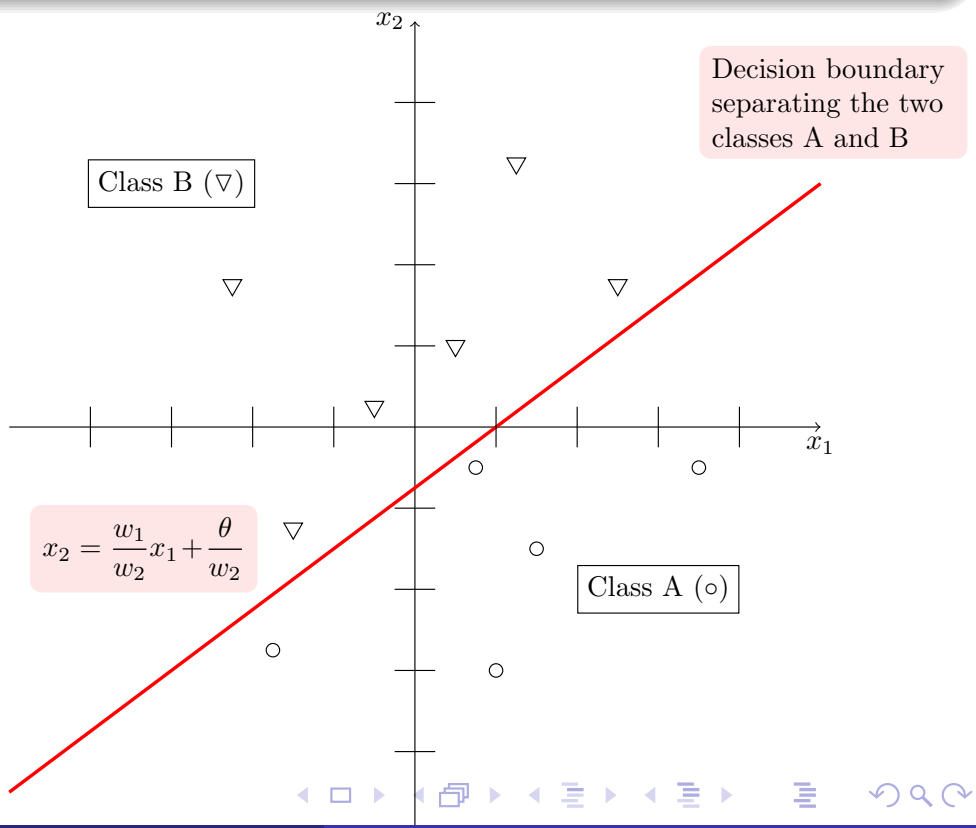


Perceptron (cont.)

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

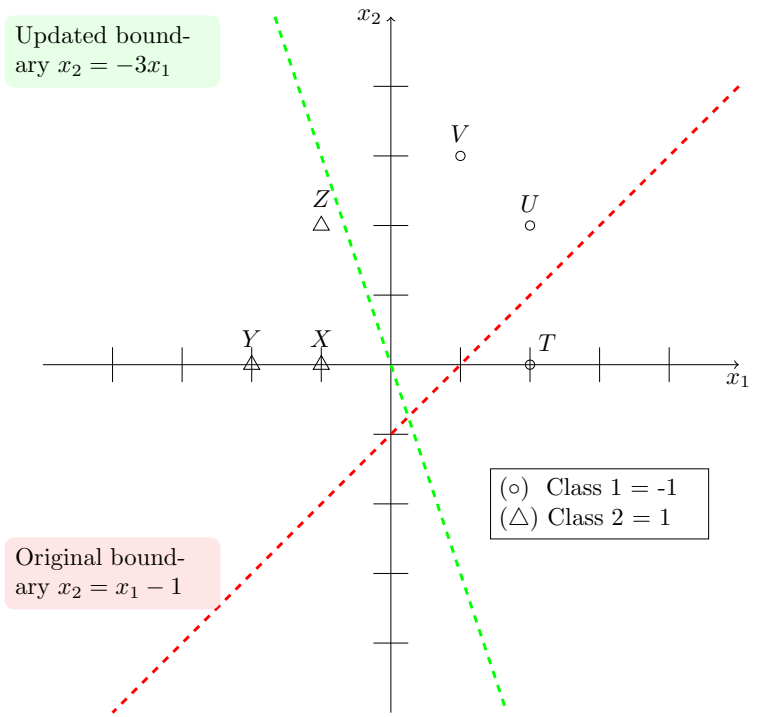
Solution

- 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

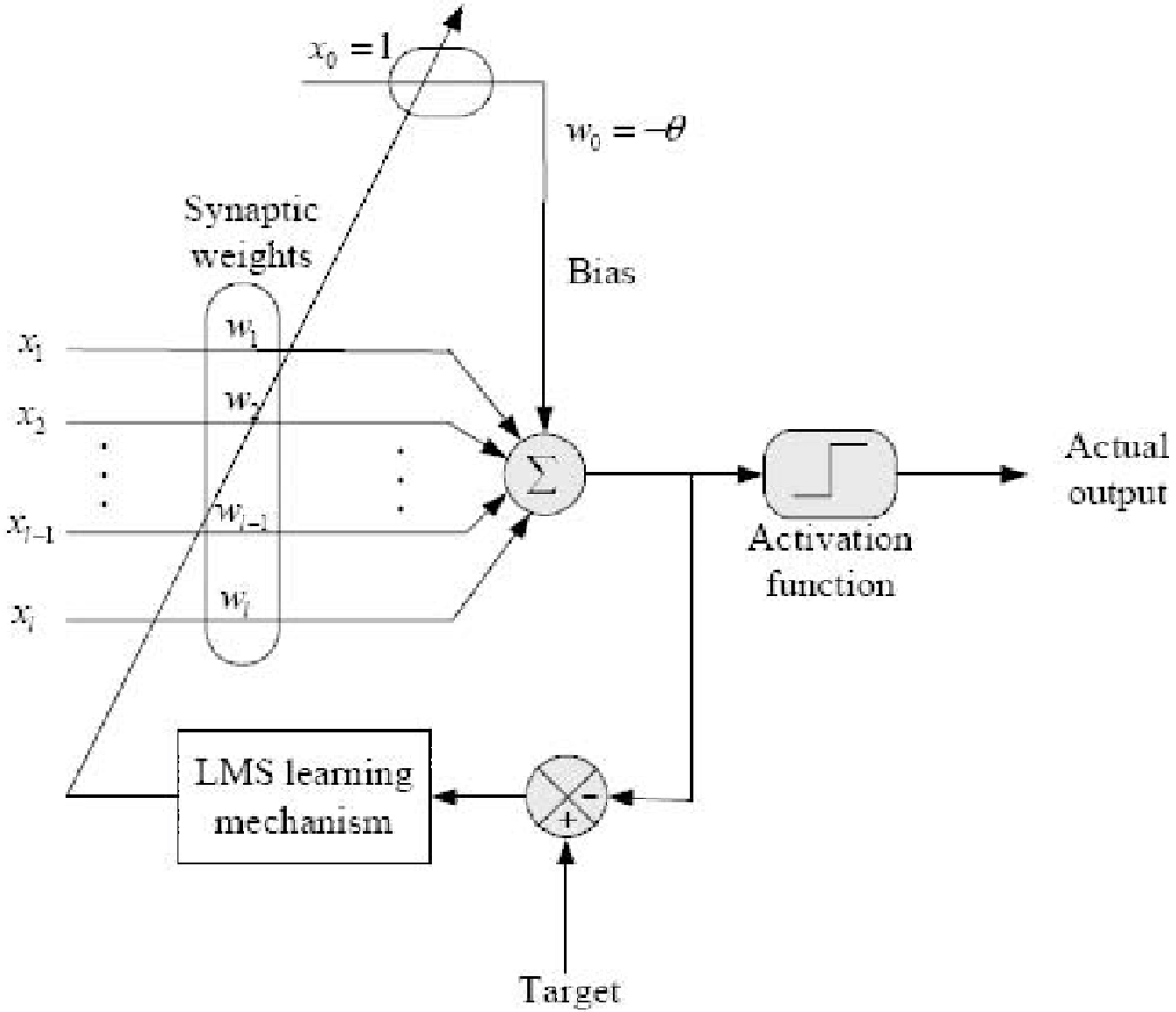
- Simple-layer perceptrons suffer from two major shortcomings:
  1. Cannot separate linearly non-separable patterns.
  2. Lack of generalization: once trained, it cannot adapt its weights to a new set of data.

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.

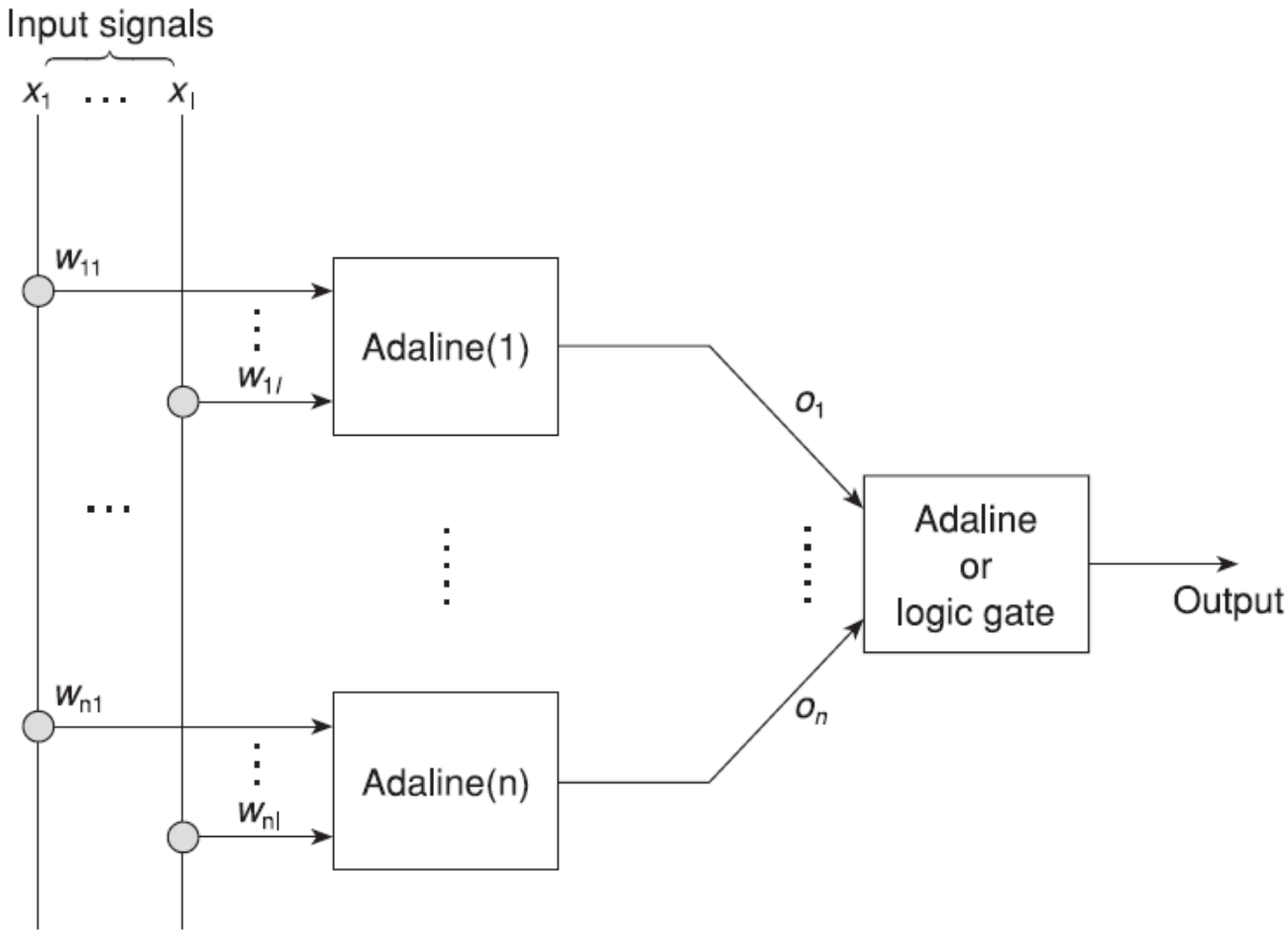
Madaline

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.

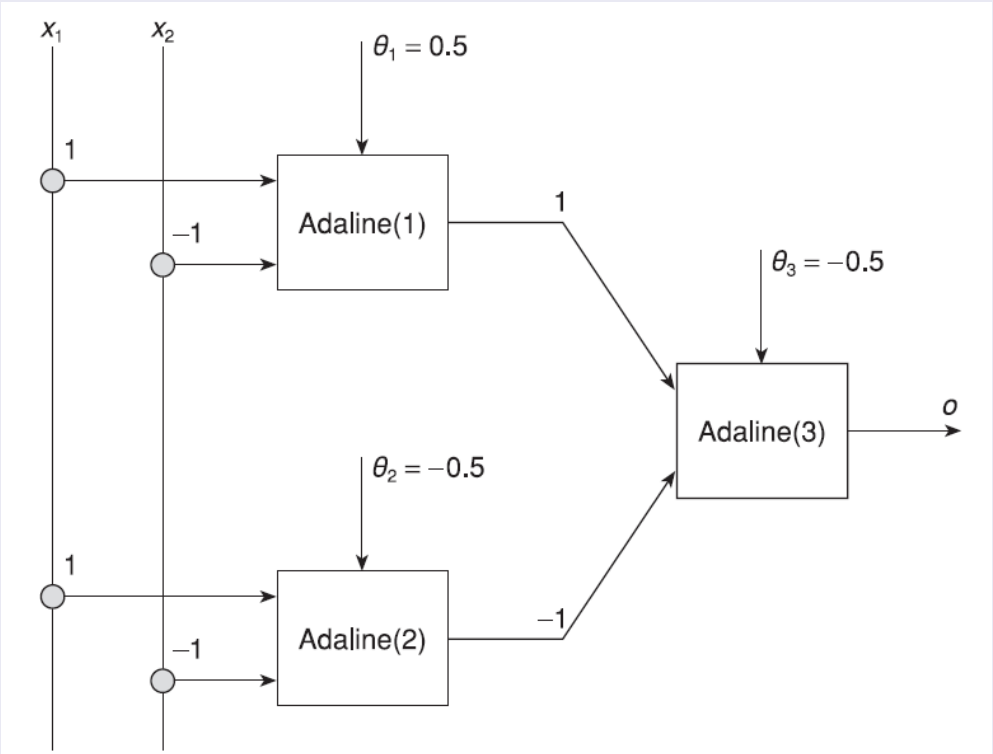
Madaline: Graphical Representation



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: $w = [w_1, w_2, w_3, w_4] = [-0.2, 0.3, 0.5, 0.3]$.