

Artificial Intelligence

AI 2002

Lecture 18

Ms. Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

Perceptron Training Rule

- The ***perceptron training rule***, which revises the weight w_i associated with input x_i according to the rule:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Where:

- t is target value
- o is perceptron output
- η is small constant (e.g., 0.1) called *learning rate*

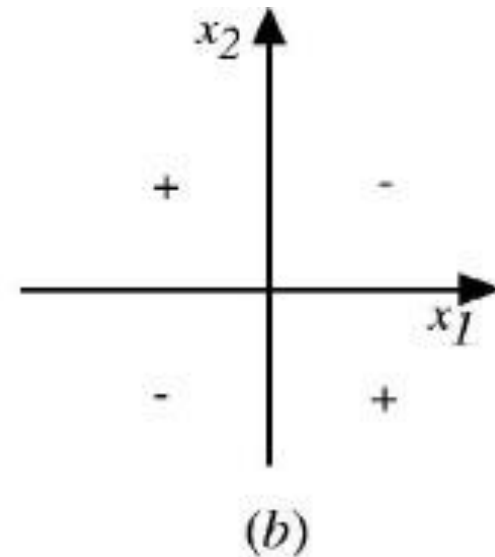
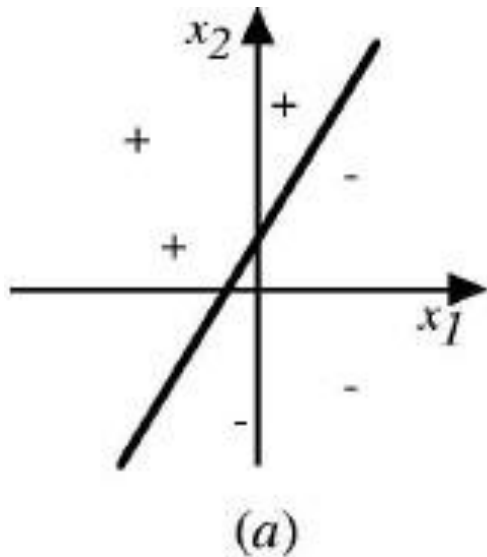
Perceptron Training Rule

- The **perceptron rule** finds a successful weight vector when the training examples are **linearly separable**,
- It fails to converge if the examples are **not linearly separable**.
- The solution is ... **Delta Rule** also known as **(Widrow- Hoff Rule)**

Delta Rule

- use *gradient descent* to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

Perceptron



The **decision surface** represented by a **two-input perceptron x_1 and x_2** .
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. (b) A set of training examples that is not linearly separable.

Delta Rule

Delta Rule

- In perceptron training rule we employ *thresholded perceptron*

$$o(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

- The delta training rule is the task of training an *unthresholded perceptron*;
 - a *linear unit* for which the output o is given by

$$o(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$$

- A linear unit corresponds to the *first stage* of a perceptron, **without** the threshold.

Delta Rule

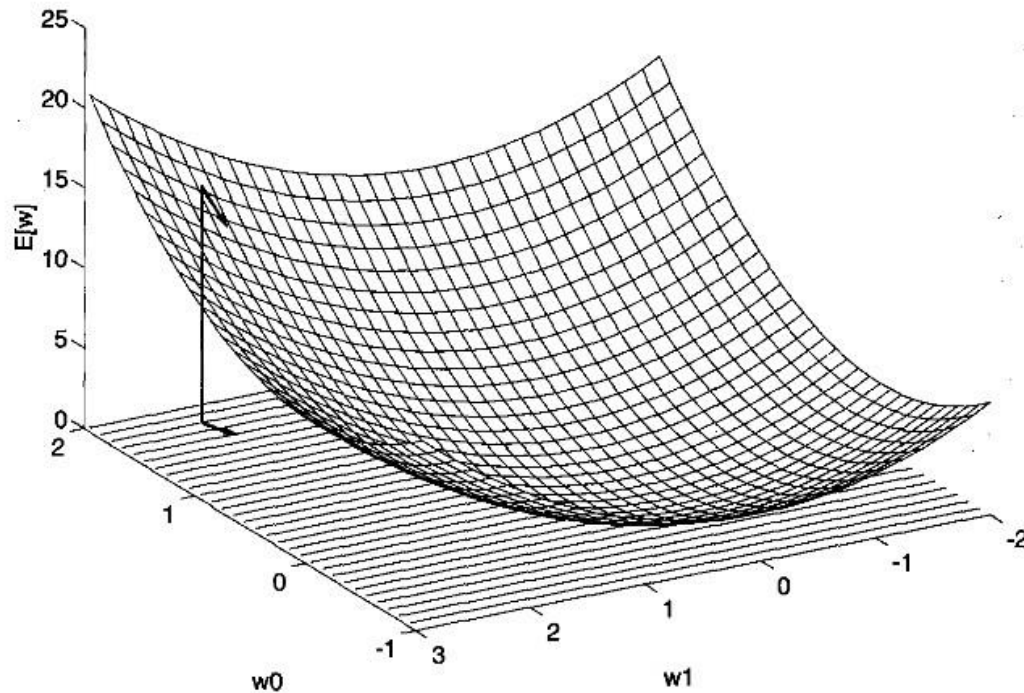
- In order to derive *a weight learning rule for linear units*,
 - Specify a measure for the *training error* of a hypothesis (weight vector), relative to the training examples.

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

- D is the set of training examples,
- t_d is the target output for training example d ,
- o_d is the output of the linear unit for training example d
- E is characterized as a function of \mathbf{w} , because the linear unit output o depends on this weight vector.

Hypothesis Space

- For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane.



Error of different hypotheses.

Gradient Descent

- Gradient descent search determines *a weight vector that minimizes E* by
 - ❑ Starting with an arbitrary initial weight vector,
 - ❑ Repeatedly modifying it in small steps.
 - ❑ At each step, the ***weight vector is altered in the direction*** that produces the **steepest descent** along the error surface,
 - ❑ This process continues until the **global minimum error** is reached.

Gradient Descent

How can we calculate the direction of steepest descent along the error surface?

- This direction can be found by computing the derivative of E with respect to each component of the vector \mathbf{w} .

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

- Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the \mathbf{w}_i .

Gradient Descent

- The **gradient specifies the direction** that produces the steepest increase in E . The training rule for gradient descent is,

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

where,

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

- The negative sign is present because we want to move the weight vector in the direction that ***decreases*** E .

Gradient Descent

- This training rule can also be written in its **component form**,

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

$$w_i \leftarrow w_i + \Delta w_i$$

where,

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

- The steepest descent is achieved by altering each component w_i in proportion to $\frac{\partial E}{\partial w_i}$

Gradient Descent

- The vector of derivatives $\frac{\partial}{\partial w_i}$ that form the gradient can be obtained by differentiating E from delta rule

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

$$o(\mathbf{X}) = \mathbf{w} \cdot \mathbf{x}$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \end{aligned}$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

AI 2002

Gradient Descent

- We now have an equation that gives $\frac{\partial}{\partial w_i}$ in terms of
 - the linear unit inputs x_{id} ,
 - outputs o_d ,
 - target values t_d associated with the training examples
- The weight update rule for gradient descent becomes,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

Gradient Descent

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - Initialize each Δw_i to zero.
 - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
 - Input the instance \vec{x} to the unit and compute the output o
 - For each linear unit weight w_i , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

- For each linear unit weight w_i , Do

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$

Gradient Descent

- Gradient descent is an important general paradigm for learning.
- It is a *strategy for searching through a large or infinite hypothesis space* that can be applied whenever
 - 1) the hypothesis space contains **continuously parameterized hypotheses** (e.g., the weights in a linear unit),
 - 2) the **error can be differentiated** with respect to these hypothesis parameters

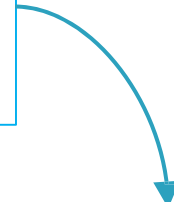
Gradient Descent

- The **key practical difficulties** in applying gradient descent are:
 - a) **converging to a local minimum*** can sometimes be **quite slow** (i.e., it can require many thousands of gradient descent steps),
 - b) **if there are multiple local minima*** in the error surface, then there is no guarantee that the procedure will find the global minimum.

Stochastic Gradient Descent

- The idea behind stochastic gradient descent is to approximate the gradient descent search by **updating weights incrementally**, following the calculation of the error for **each individual example**.

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id}$$


$$\Delta w_i = \eta (t - o) x_i$$

Stochastic Gradient Descent

- One way to view this stochastic gradient descent is to consider a **distinct error function** defined for *each individual training example d as follows*.

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

- where t , and o_d are the target value and the unit output value for training example d .
- Stochastic gradient descent iterates over the training examples d in D ,
- at each iteration altering weights according to the gradient

Stochastic Gradient Descent

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - Initialize each w_i to zero.
 - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
 - Input the instance \vec{x} to the unit and compute the output o
 - For each linear unit weight w_i , Do

$$w_i \leftarrow w_i + \eta(t - o)x_i$$

The Key Difference

- In **stochastic gradient descent**, weights are updated upon examining *each* training example.
- Whereas in **standard gradient descent**, the error is summed over *all* examples before updating weights,
 - ❑ **Standard gradient descent** requires **more computation** per weight update step.
 - ❑ **Standard gradient descent** is often used with a **larger step size** per weight update than stochastic gradient descent.

The Key Difference

- When there are multiple local minima with respect to $E(\mathbf{w})$,
 - ❑ The **stochastic gradient descent** can sometimes *avoid falling into these local minima*,
 - ❑ It is due to the reason that it uses various $\nabla E_d(\vec{\mathbf{w}})$ rather than $\nabla E(\vec{\mathbf{w}})$ to guide its search.

Training Rules

- **Perceptron Training Rule:** guarantee to succeed if
 - training examples are linearly separable
 - Sufficiently small learning rate
- **Delta Rule:**
 - use gradient descent
 - converges only asymptotically toward the minimum error hypothesis,
 - converges regardless of whether the training data are linearly separable.

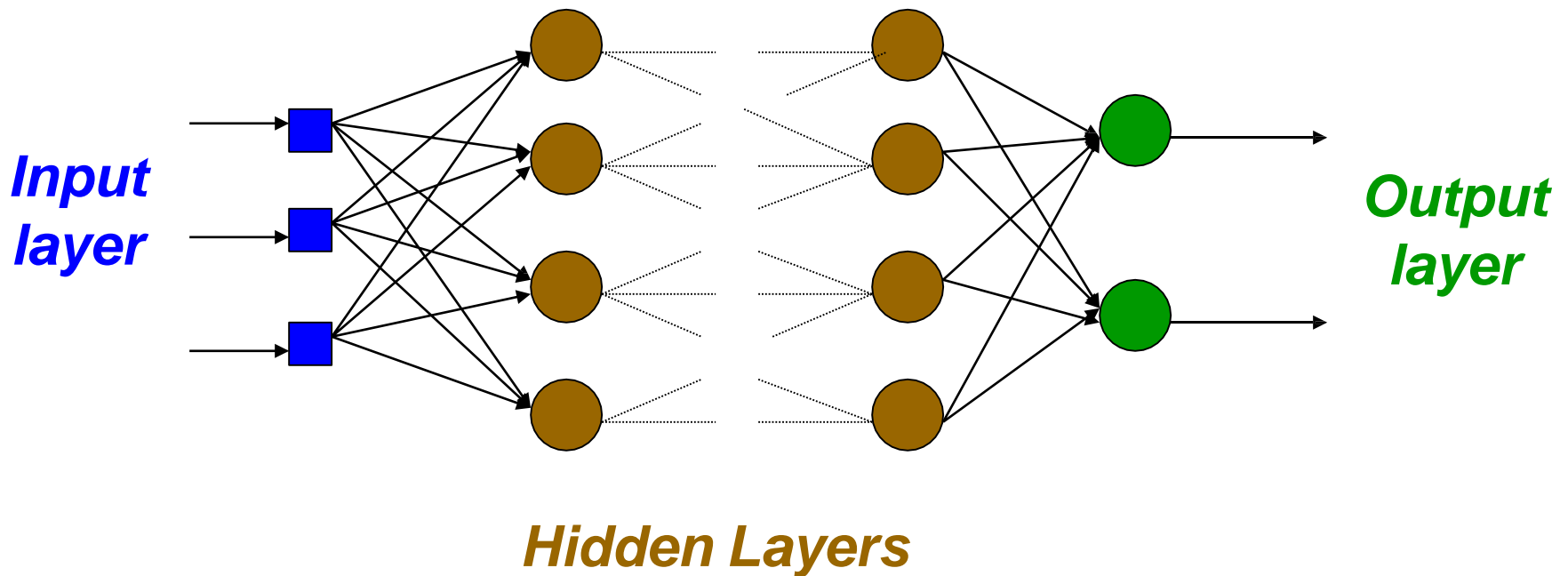
Difference between gradient and stochastic gradient Descent Algorithm

Gradient Descent	Stochastic Gradient Descent
Computes gradient using the whole Training sample	Computes gradient using a single Training sample
Slow and computationally expensive algorithm	Faster and less computationally expensive than Batch GD
Not suggested for huge training samples.	Can be used for large training samples.
Deterministic on nature	Stochastic in nature

Multilayer Networks

Multilayer Perceptron Architecture

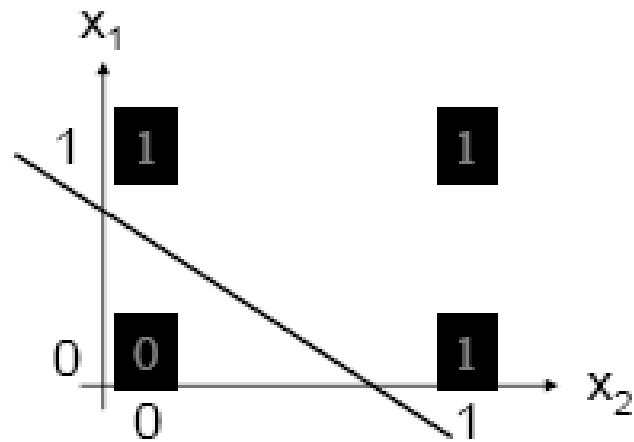
MLP used to describe any general feedforward (no recurrent connections) network



Multilayer Networks

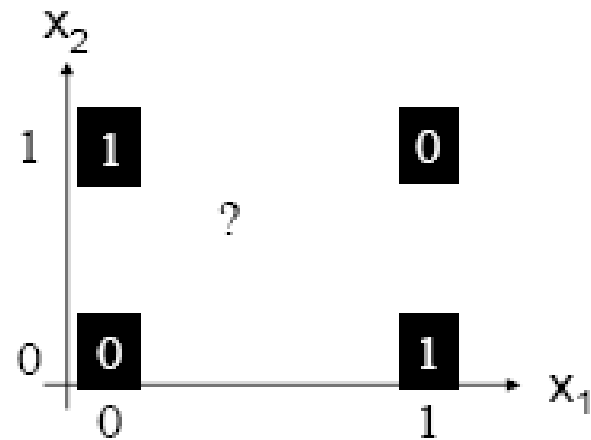
OR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

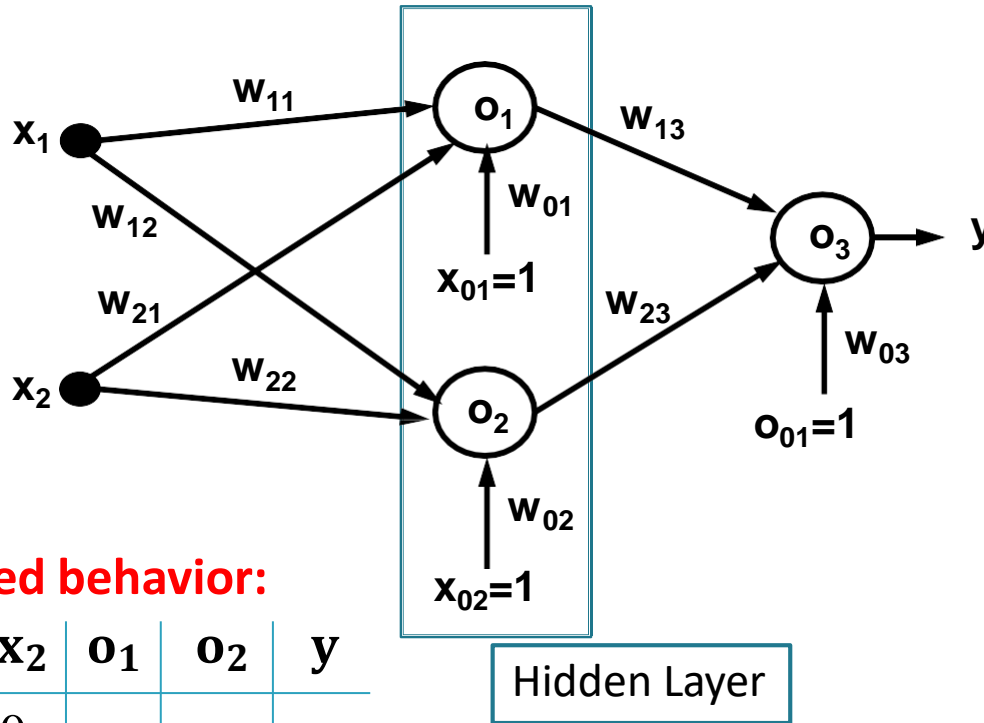


XOR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



Multilayer Networks



2 hidden nodes
1 output

Weights:

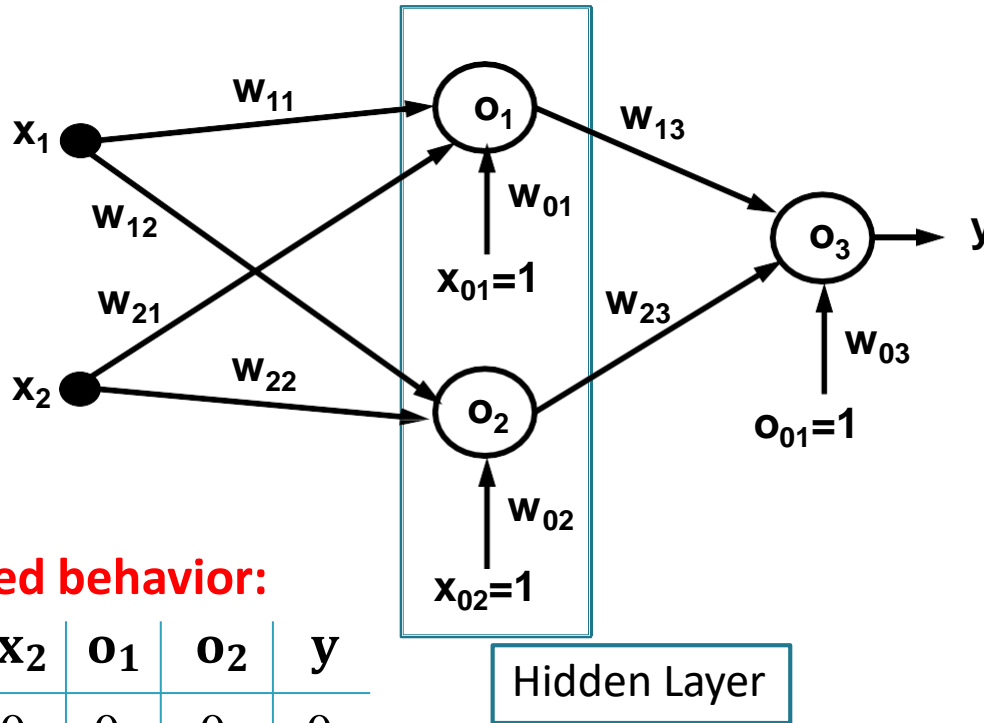
$$\begin{aligned}
 w_{11} &= w_{12} = 1 \\
 w_{21} &= w_{22} = 1 \\
 w_{01} &= -1.5 \\
 w_{02} &= -0.5 \\
 w_{13} &= -1 \\
 w_{23} &= 1 \\
 w_{03} &= -0.5
 \end{aligned}$$

Desired behavior:

x_1	x_2	o_1	o_2	y
0	0			
1	0			
0	1			
1	1			

Piecewise linear classification using an MLP
with threshold (perceptron) units

Multilayer Networks



2 hidden nodes
1 output

Weights:

$$\begin{aligned}
 w_{11} &= w_{12} = 1 \\
 w_{21} &= w_{22} = 1 \\
 w_{01} &= -1.5 \\
 w_{02} &= -0.5 \\
 w_{13} &= -1 \\
 w_{23} &= 1 \\
 w_{03} &= -0.5
 \end{aligned}$$

Desired behavior:

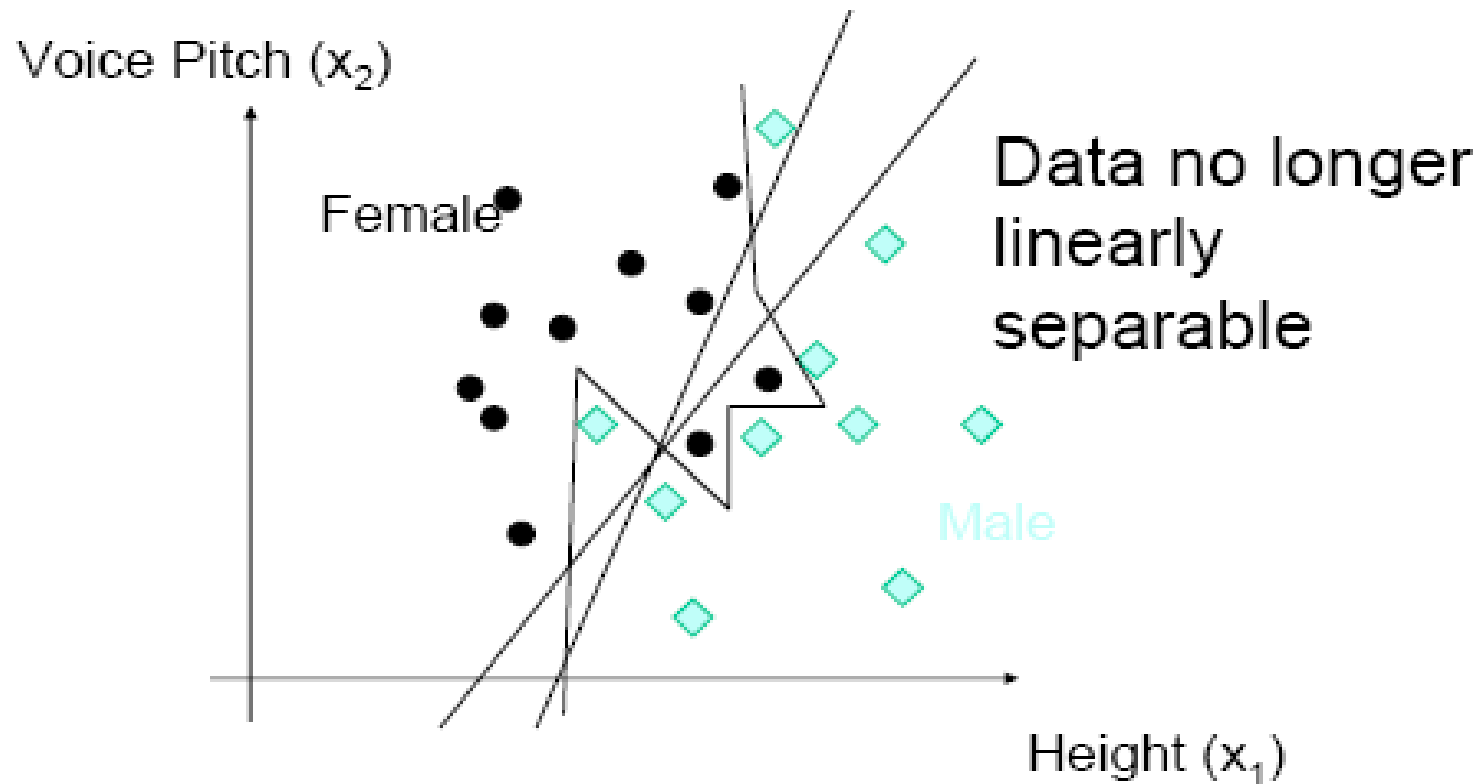
x_1	x_2	o_1	o_2	y
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Piecewise linear classification using an MLP with threshold (perceptron) units

Multilayer Networks

- The single perceptron can only express linear decision surfaces.
- The kind of **multilayer networks** learned by the **back propagation** algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

Multilayer Networks... Example



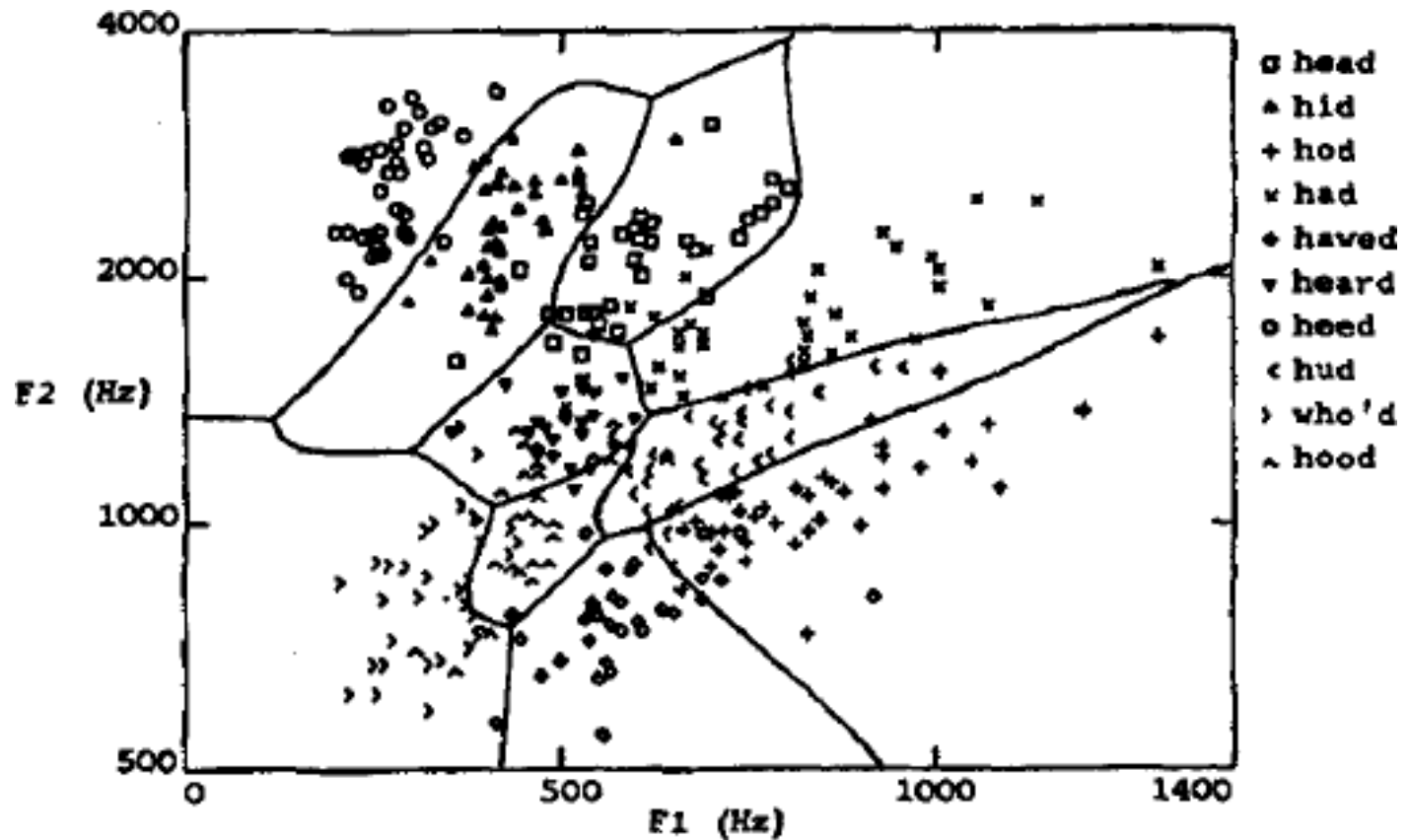
What is a good decision boundary ?

Multilayer Networks... Example

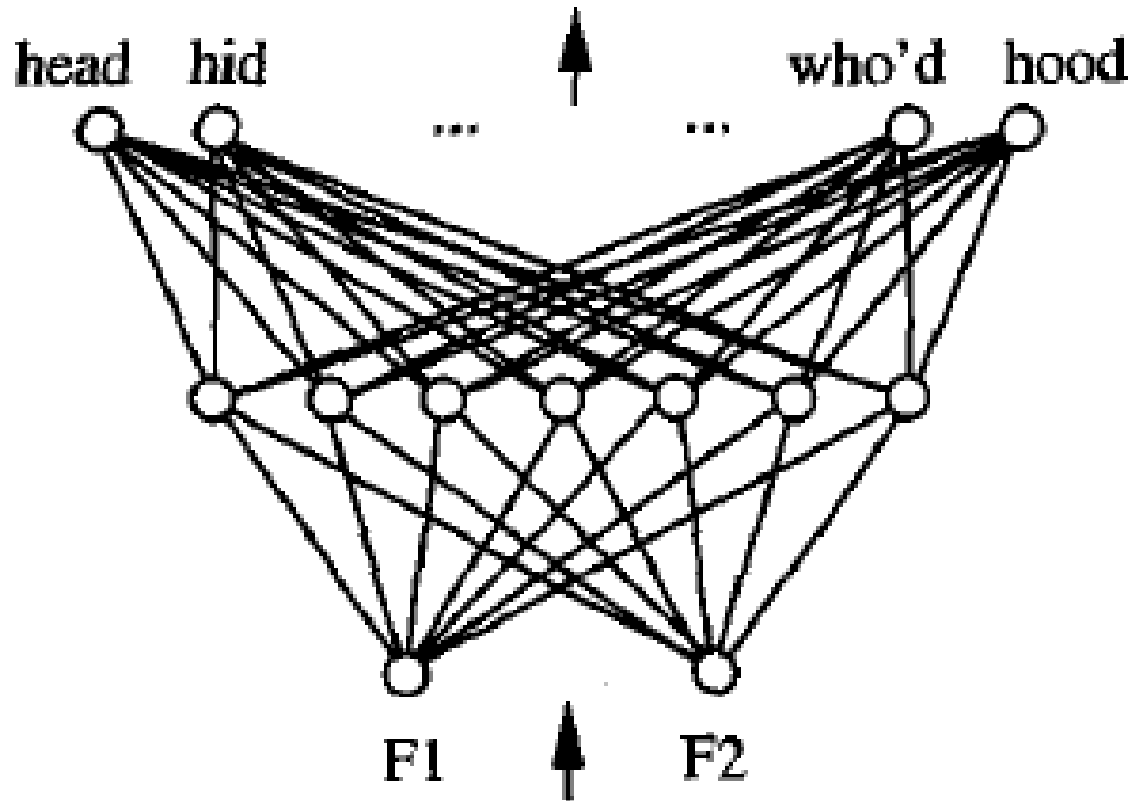
Example:

- The speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.).

Multilayer Networks... Example



Multilayer Networks... Example



Multilayer Networks

- What type of unit shall we use as the basis for constructing multilayer networks?
- Multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions
- The perceptron unit is another possible choice, is it?
 - its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent.

Multilayer Networks

Solution:

- One solution is the **sigmoid unit**:
 - a unit very much like a perceptron, but based on a
 - **smoothed, differentiable threshold function.**
- Like the perceptron, the sigmoid unit
 - first computes a linear combination of its inputs,
 - then applies a threshold to the result.

Multilayer Networks

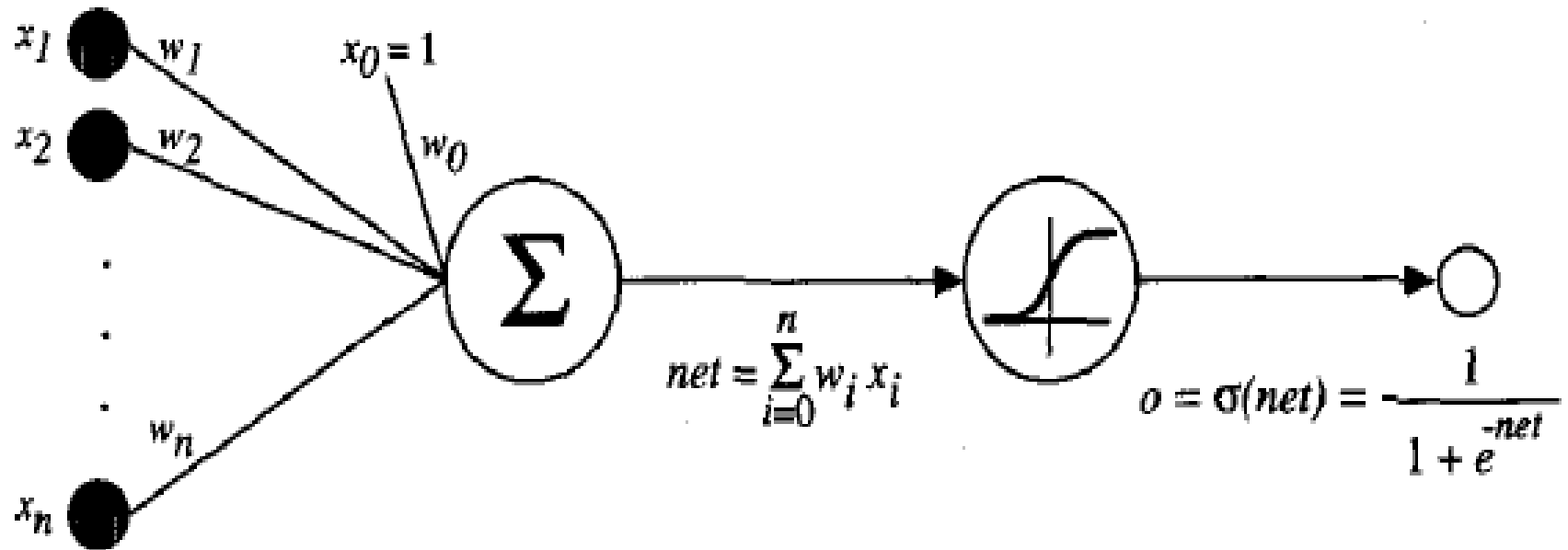
- In case of **sigmoid unit**, however, the **threshold output is a continuous function** of its input.
- More precisely, the sigmoid unit computes its output o as,

$$o = \sigma(\vec{w} \cdot \vec{x})$$

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

- σ is often called the sigmoid function or, alternatively, the logistic function.

Sigmoid Threshold Unit



Sigmoid Function

- Sigmoid function maps a very large input domain to a small range of outputs, it is often referred to as the ***squashing function*** of the unit.
- The sigmoid function has the useful property that **its derivative is easily expressed in terms of its output.**

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$
$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

Sigmoid Function

- The term e^{-y} in the sigmoid function definition is sometimes replaced by $e^{-k \cdot y}$
 - where k is some positive constant that determines the steepness.
- The function ***tanh*** is also sometimes used in place of the sigmoid function.

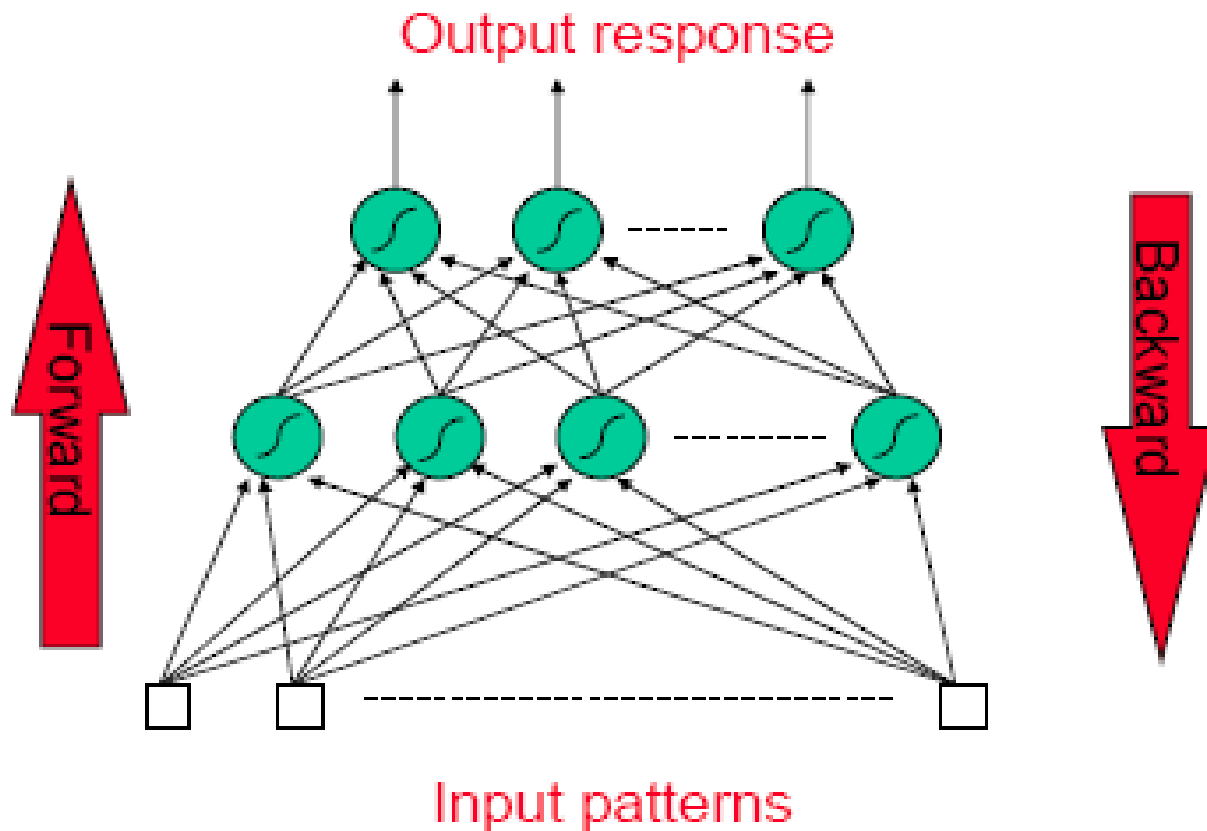
Back Propagation Algorithm

The Back Propagation Algorithm

The Back Propagation algorithm has two phases:

- **Forward pass phase:** computes 'functional signal', feed forward propagation of input pattern signals through network
- **Backward pass phase:** computes 'error signal', *propagates* the error *backwards* through network starting at output units
 - (where the error is the difference between actual and desired output values)

The Back Propagation Algorithm



Conceptually: Forward Activity -
Backward Error

The Back Propagation Algorithm

- The back propagation algorithm learns the weights for a multilayer network,
 - given a network with a fixed set of units and interconnections.
- It **employs gradient descent** to attempt to minimize the squared error between the network output values and the target values for these outputs.
- As we are considering networks with multiple output units, we begin by redefining **E** to sum the errors over all of the network output units.

The Back Propagation Algorithm

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$$

- where **outputs** is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the **kth output unit** and **training example d**.

The Back Propagation Algorithm

BACKPROPAGATION(*training_examples*, η , n_{in} , n_{out} , n_{hidden})

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where \vec{x} is the vector of network input values, and \vec{t} is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji} , and the weight from unit i to unit j is denoted w_{ji} .

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and $.05$).
- Until the termination condition is met, Do

The Back Propagation Algorithm

For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

Propagate the input forward through the network:

1. Input the instance \vec{x} to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k , calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit h , calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh} \delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Reading Material

- **Artificial Intelligence, A Modern Approach**
Stuart J. Russell and Peter Norvig
 - Chapter 18.
- **Machine Learning**
Tom M. Mitchell
 - Chapter 4.