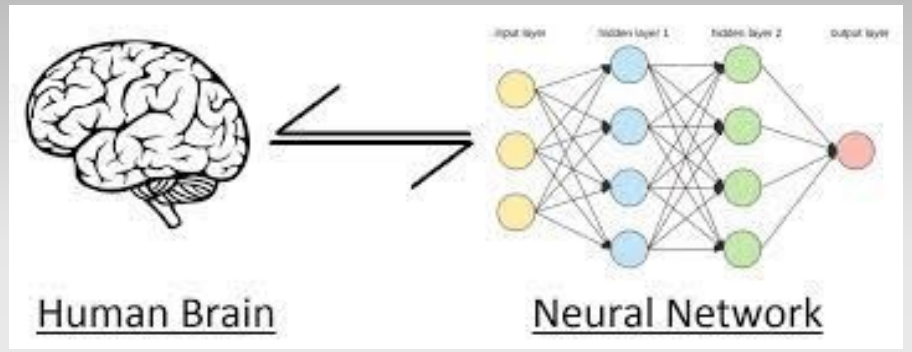
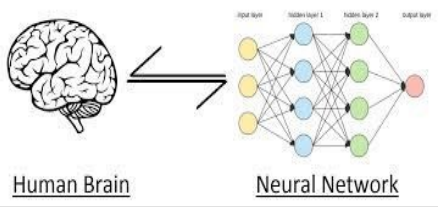


Artificial Neural Network and Deep Learning
Lecture 3

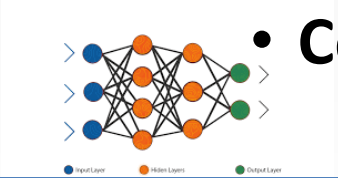
Resenblatt's Perceptron

Least Mean Square Algorithm for Single Layer Network



Agenda

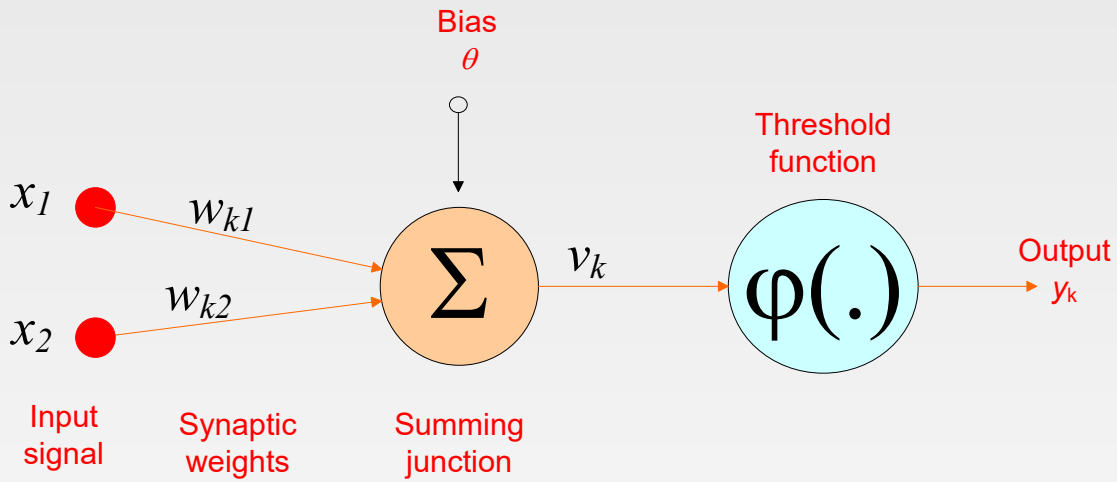
- Single-layer feedforward networks and Classification Problems (McCulloch-Pitts neuron model)
  - Decision Surface
  - Rosenblatt's Perceptron Learning Rule (The Perceptron Convergence Theorem)
  - Multiple-output-Neurons Perceptron
- **Adaline (Adaptive Linear Neuron) Networks**
- **Derivation of the LMS algorithm**
- **Compare ADALINE with Perceptron**



going back to

The McCulloch-Pitts neuron model (1943)

Threshold Logic Units (TLU)



$$y_k = \varphi(v_k) = \varphi(w_{k1}x_1 + w_{k2}x_2 + \theta) = \begin{cases} 1 & if v_k \geq 0 \\ 0 & if v_k < 0 \end{cases}$$

TLU try to solve Simple Classification Problem:

The goal of pattern classification is to assign a physical object or event to one of a set of a pre-specified classes (or categories)

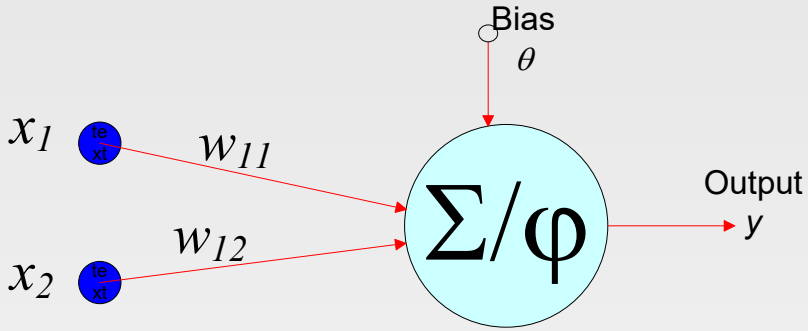
Examples:

- Boolean functions
- Pixel Patterns

The AND problem

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

$Let \quad w_{11} = w_{12} = 1$



$$with \quad x_1 = 1 \quad and \quad x_2 = 1 \quad , \quad y = \varphi(1\times 1 + 1\times 1 + \theta) = \varphi(2+\theta) = 1 \quad if\,(2+\theta \geq 0)$$

$$with \quad x_1 = 1 \quad and \quad x_2 = 0 \quad , \quad y = \varphi(1\times 1 + 1\times 0 + \theta) = \varphi(1+\theta) = 0 \quad if\,(1+\theta < 0)$$

$$with \quad x_1 = 0 \quad and \quad x_2 = 1 \quad , \quad y = \varphi(1\times 0 + 1\times 1 + \theta) = \varphi(1+\theta) = 0 \quad if\,(1+\theta < 0)$$

$$with \quad x_1 = 0 \quad and \quad x_2 = 0 \quad , \quad y = \varphi(1\times 0 + 0\times 1 + \theta) = \varphi(\theta) = 0 \quad if\,(\theta < 0)$$

$$2+\theta \geq 0$$
$$1+\theta < 0, \quad \theta < 0$$

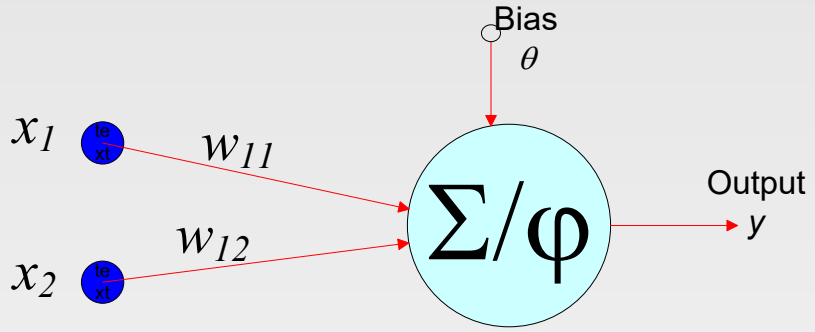
$$\theta\ ?$$

$$\theta = -1.5$$

The OR problem

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

Let $w_{11} = w_{12} = 1$



$$with \quad x_1 = 1 \quad and \quad x_2 = 1 \quad , \quad y = \varphi(1\times 1 + 1\times 1 + \theta) = \varphi(2+\theta) = 1 \quad if\,(2+\theta \geq 0)$$

$$with \quad x_1 = 1 \quad and \quad x_2 = 0 \quad , \quad y = \varphi(1\times 1 + 1\times 0 + \theta) = \varphi(1+\theta) = 1 \quad if\,(1+\theta \geq 0)$$

$$with \quad x_1 = 0 \quad and \quad x_2 = 1 \quad , \quad y = \varphi(1\times 0 + 1\times 1 + \theta) = \varphi(1+\theta) = 1 \quad if\,(1+\theta \geq 0)$$

$$with \quad x_1 = 0 \quad and \quad x_2 = 0 \quad , \quad y = \varphi(1\times 0 + 0\times 1 + \theta) = \varphi(\theta) = 0 \quad if\,(\theta < 0)$$

$$2+\theta \geq 0, \quad 1+\theta \geq 0$$
$$\theta < 0$$

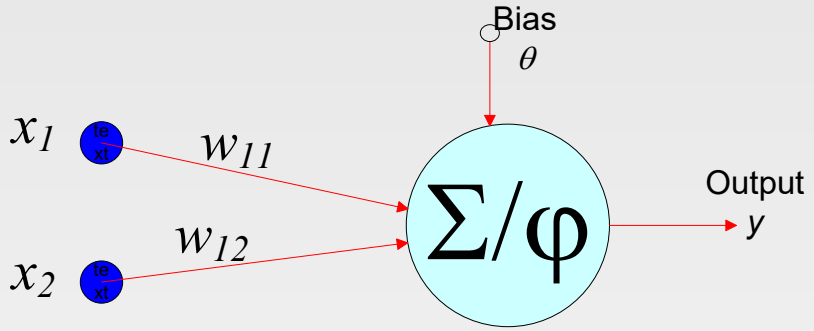
$$\theta\ ?$$

$$\theta = -0.5$$

The XOR problem

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

$$Let \quad w_{11} = w_{12} = 1$$



$$with \quad x_1 = 1 \quad and \quad x_2 = 1 \quad ,the \quad output \quad y = \varphi(1\times 1 + 1\times 1 + \theta) = \varphi(2+\theta)$$

$$with \quad x_1 = 1 \quad and \quad x_2 = 0 \quad ,the \quad output \quad y = \varphi(1\times 1 + 1\times 0 + \theta) = \varphi(1+\theta)$$

$$with \quad x_1 = 0 \quad and \quad x_2 = 1 \quad ,the \quad output \quad y = \varphi(1\times 0 + 1\times 1 + \theta) = \varphi(1+\theta)$$

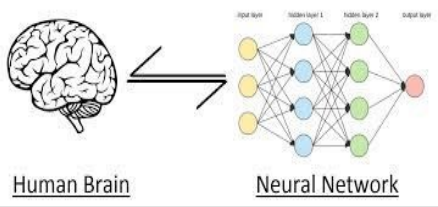
$$with \quad x_1 = 0 \quad and \quad x_2 = 0 \quad ,the \quad output \quad y = \varphi(1\times 0 + 0\times 1 + \theta) = \varphi(\theta)$$

What is the suitable value of θ that make:

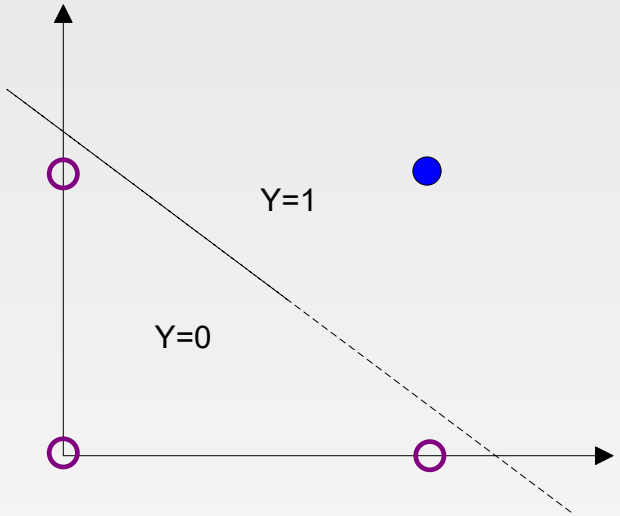
1- $2+\theta < 0$ and $\theta < 0$ to get on an output **y=0**

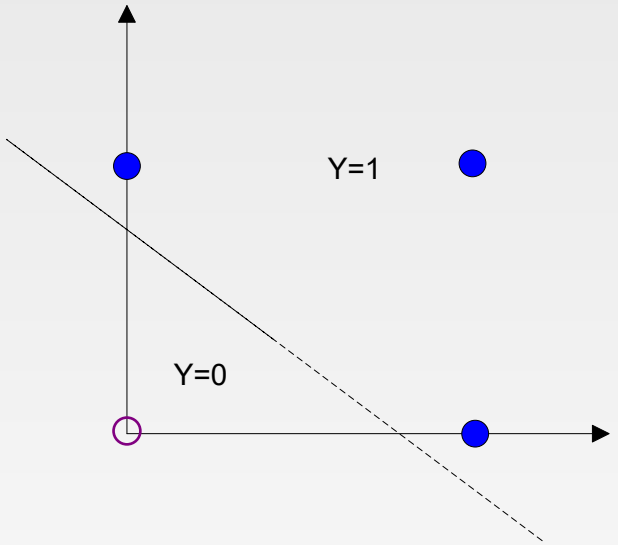
2- $1+\theta > 0$ to get on an output **y=1**

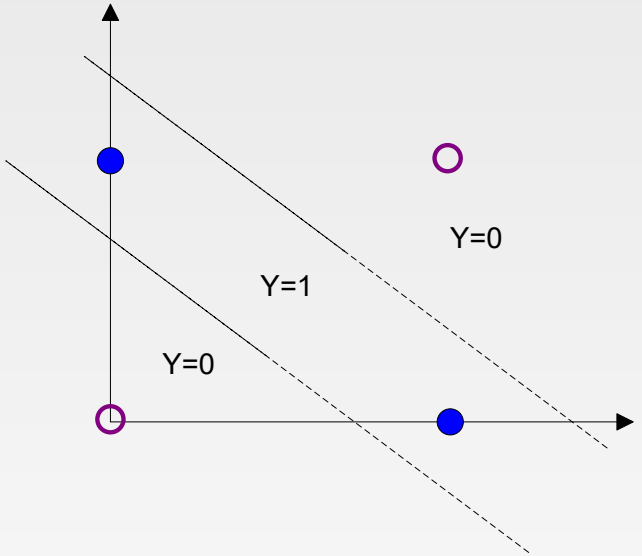
No suitable value for θ can solve the XOR problem.

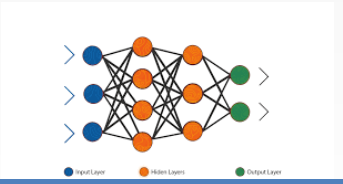


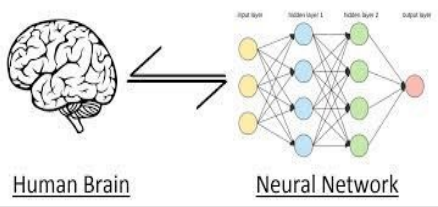
The **graph for** AND, OR, and XOR Problems





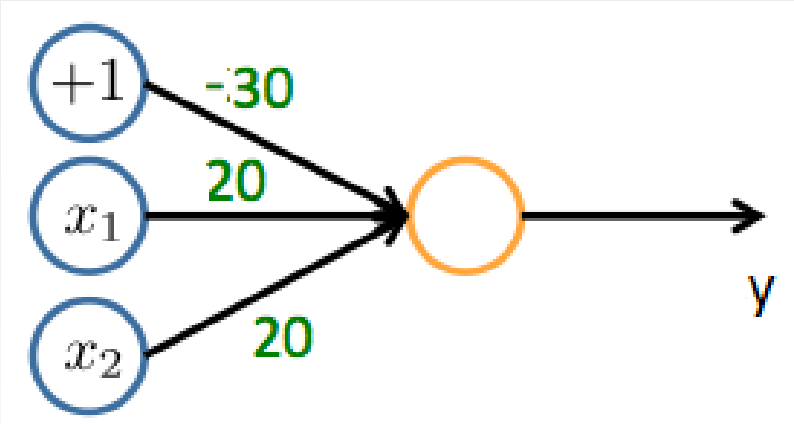






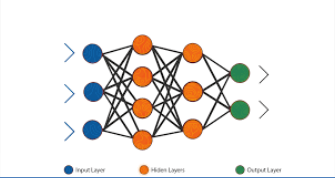
Example

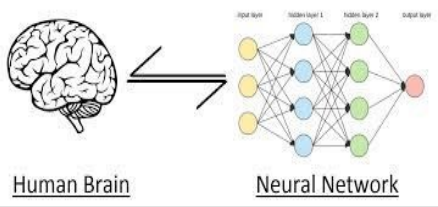
Consider the following neural network which takes two binary-valued inputs x_1, x_2 $\epsilon\{0, 1\}$ and output y. Which one the logical functions does it (approximate) compute?



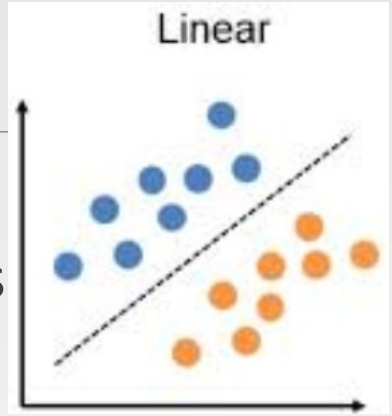
x_1	x_2	y
1	1	
1	0	
0	1	
0	0	

$$y = \varphi(-30 + 20x_1 + 20x_2)$$





Decision Surface or Boundary



The Decision Boundary means to get a **break** between two classes.

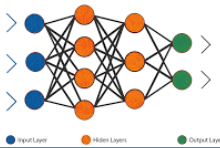
In case of the binary classification, the break will separate two classes where there are any example of class 1 existed in class 2 region.

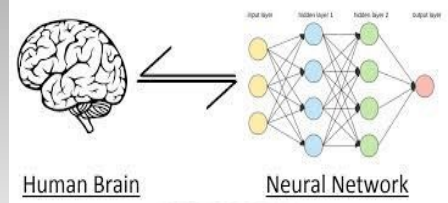
- For m input variables $x_1, x_2, \ldots, x_m$. The decision surface is the surface at which the output of the node is precisely equal to the threshold (bias), and defined by

$$\sum_{i=1}^{m} w_i x_i + \theta = 0$$

In this case, the decision surface is called **hyperplane.**

- On one side of this surface, the output of decision unit, y, will be 0, and on the other side it will be 1.

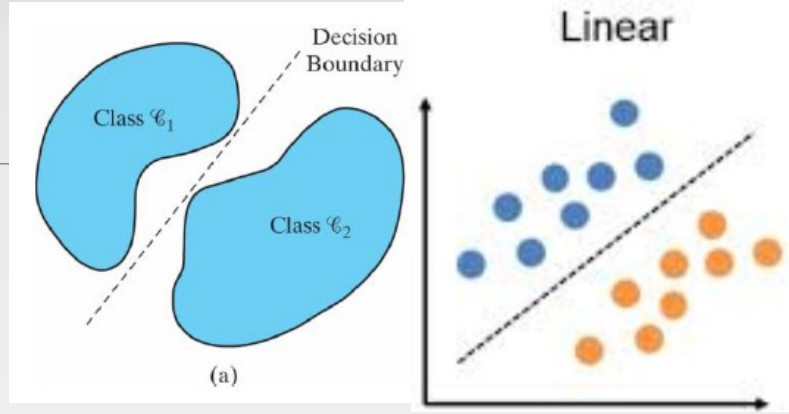




Linear separable problem

A function $f:\{0, 1\}^n \rightarrow \{0, 1\}$ is **linearly separable** if the space of input vectors yielding 1 can be separated from those yielding 0 by a **linear surface** (**hyperplane**) in n dimensions.

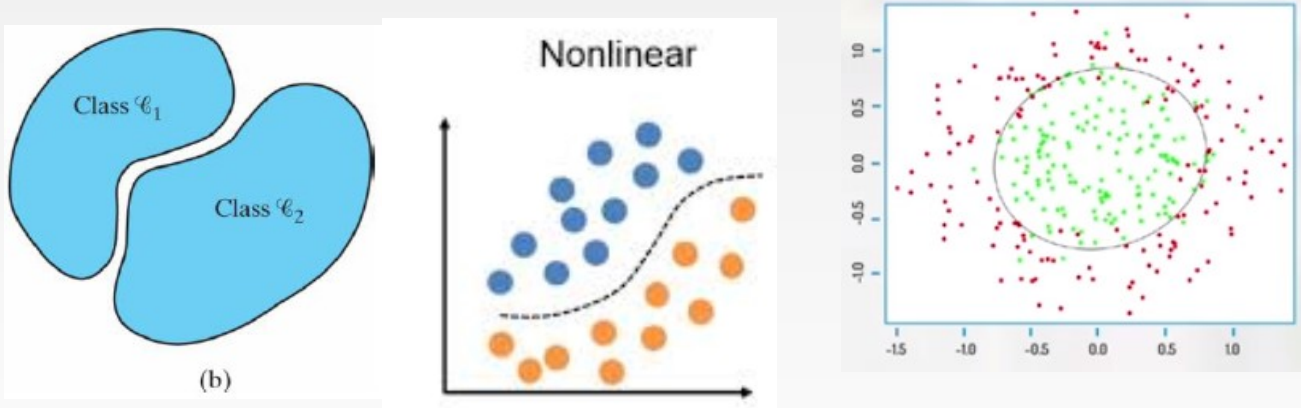
Examples: in case of two-dimensional, a hyperpane is a straight line.



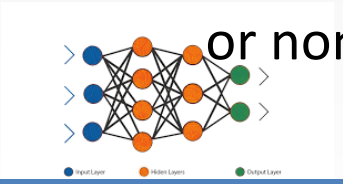
linearly separable

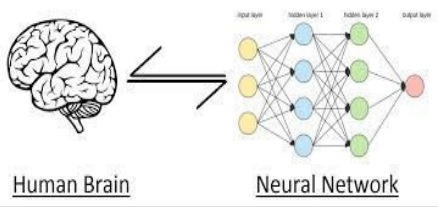
Non-linearly separable problem

- The decision boundary or the classifier in the machine learning language is not always linear.
- We could have linear decision boundary or non-linear decision boundary.



Non-linearly separable

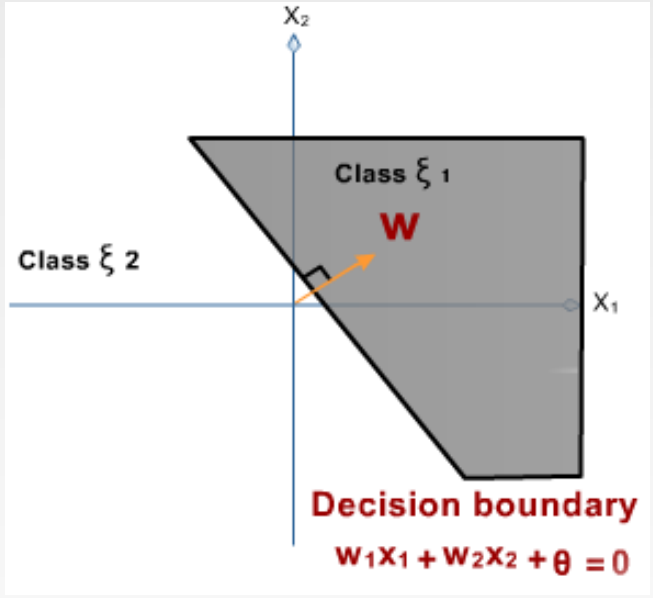




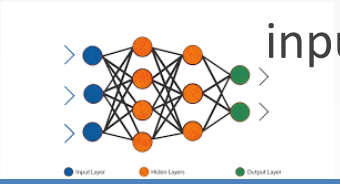
Decision Surfaces in 2-D

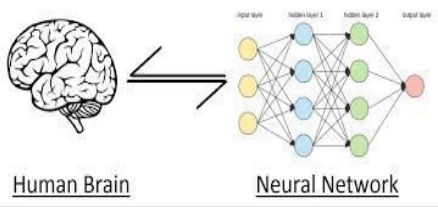
- Two classification regions are separated by the decision boundary line L:

$$w_1 x_1 + w_2 x_2 + \theta = 0$$



- This line is shifted away from the origin according to the bias θ.
- If the **bias is 0**, then the decision surface passes through the **origin**.
- This line is perpendicular to the weight vector W $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$
- Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin.





Decision Surfaces in 2-D, cont.

In 2-dim, the surface is:

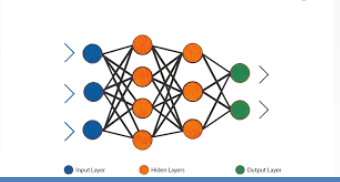
$$w_1 x_1 + w_2 x_2 + \theta = 0$$

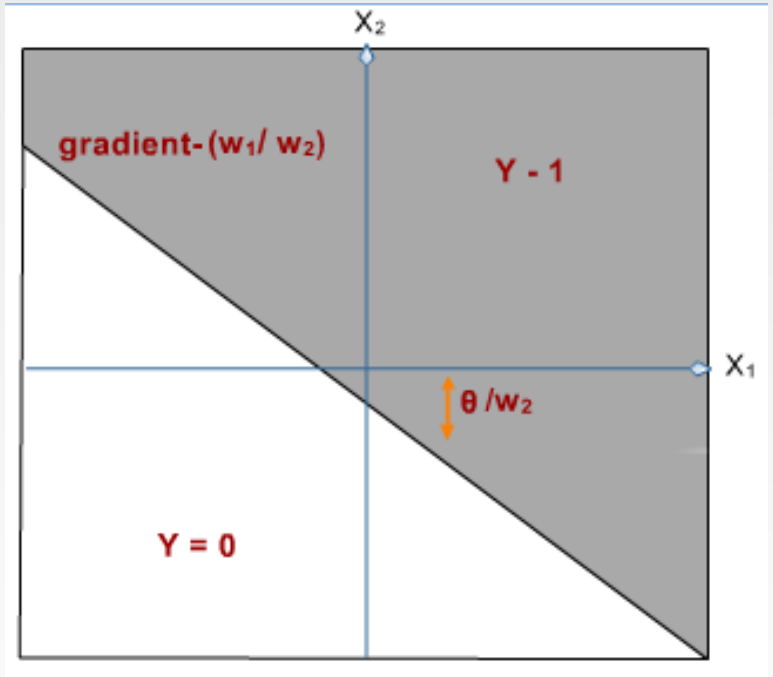
Which we can write

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{\theta}{w_2}$$

Which is the equation of a line of gradient $-\frac{w_1}{w_2}$

With intercept $-\frac{\theta}{w_2}$





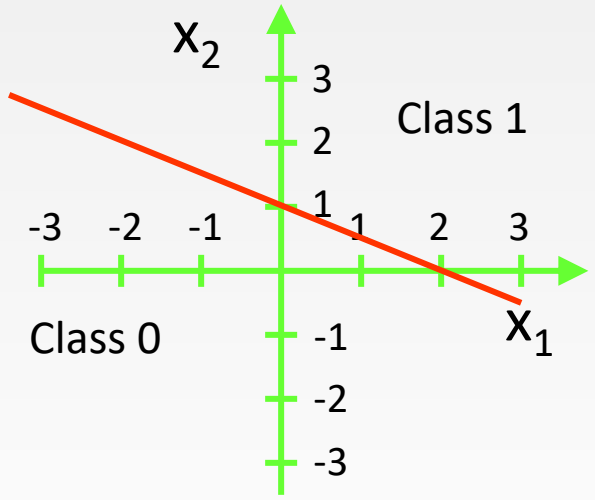
Examples for **Decision Surfaces in 2-D :**

Depending on the values of the weights, the decision boundary line will separate the possible inputs into two categories.

Let $w_1 = 1$, $w_2 = 2$, $\theta = -2$

The decision boundary:

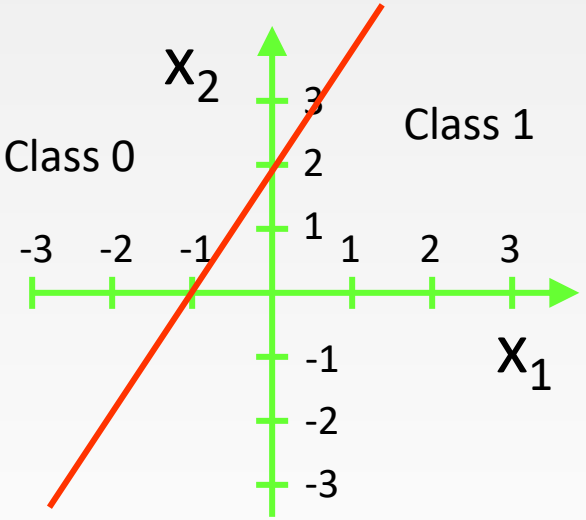
$X_1+2X_2-2=0$



Let $w_1 = -2$, $w_2 = 1$, $\theta = -2$

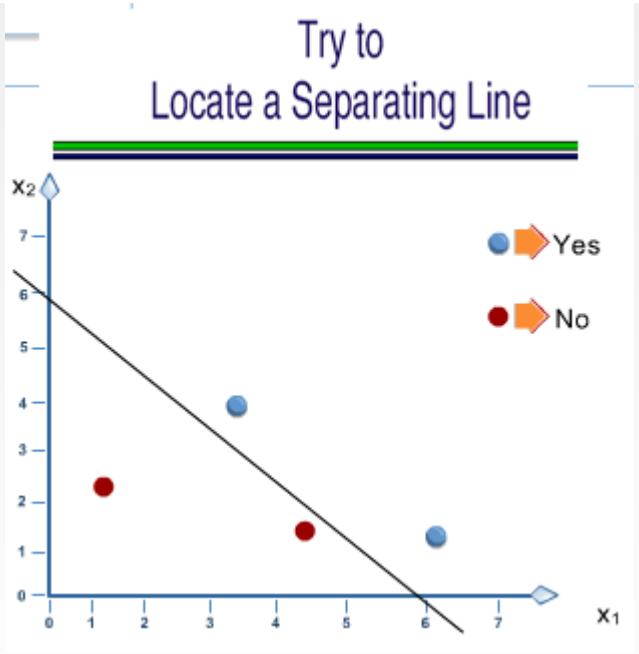
The decision boundary:

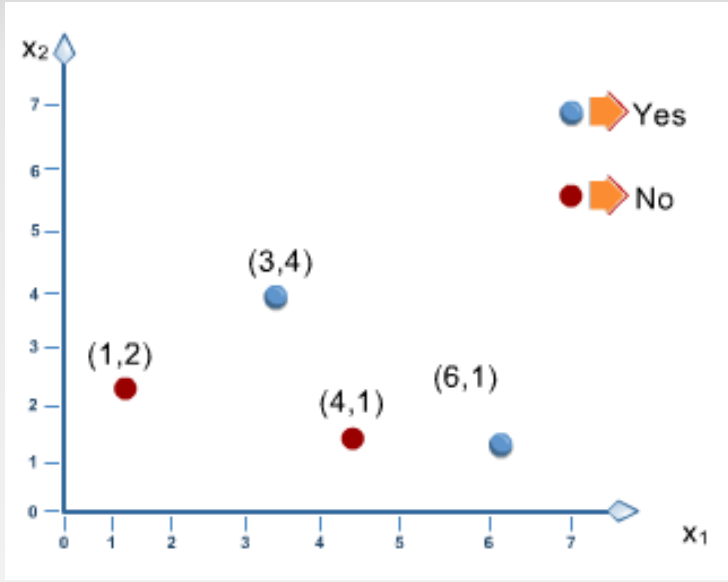
$-2X_1+X_2-2=0$



Example

- Suppose the signals are real-valued, and the following vectors are classified as shown:
  - (3, 4) yes (in the set)
  - (6, 1) yes
  - (4, 1) no (not in the set)
  - (1, 2) no



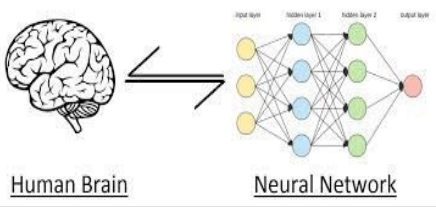


Slope= (6-0)/(0-6)= -1

Intersection at x_2-axis=6

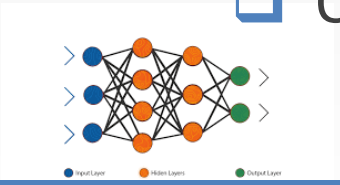
then the hyperplane (straight line) will be $x_2=-x_1+6$

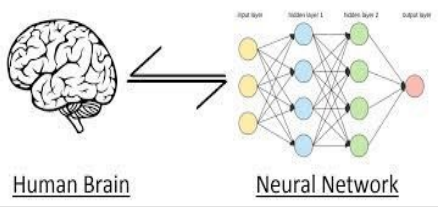
Which is: $x_1+x_2-6=0$



Linear Separability for n-dimensional

- So by varying the weights and the threshold, we can realize **any linear separation** of the input space into a region that yields output 1, and another region that yields output 0.
- As we have seen, a **two-dimensional** input space can be divided by any straight line.
- A **three-dimensional** input space can be divided by any two-dimensional plane.
- In general, an **n-dimensional** input space can be divided by an (n-1)-dimensional plane or hyperplane.
- Of course, for n > 3 this is hard to visualize.

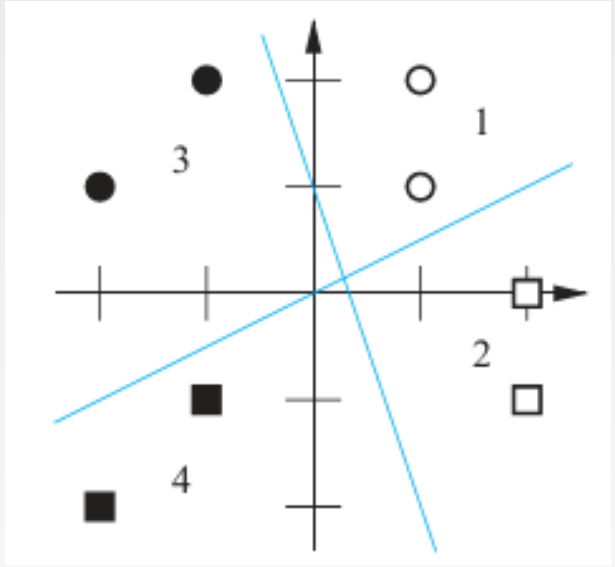


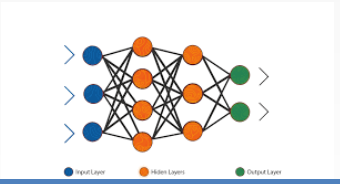


The Perceptron

by Frank Rosenblatt (1958, 1962)

- It is the *simplest form of a neural network.*
- It is a single-layer network whose *synaptic* weights apple to a*djustable.*
- It is *limited to perform pattern classification.*
  - The classes must be *linearly Separable; patterns lie on opposite sides of a hyperplanes.*





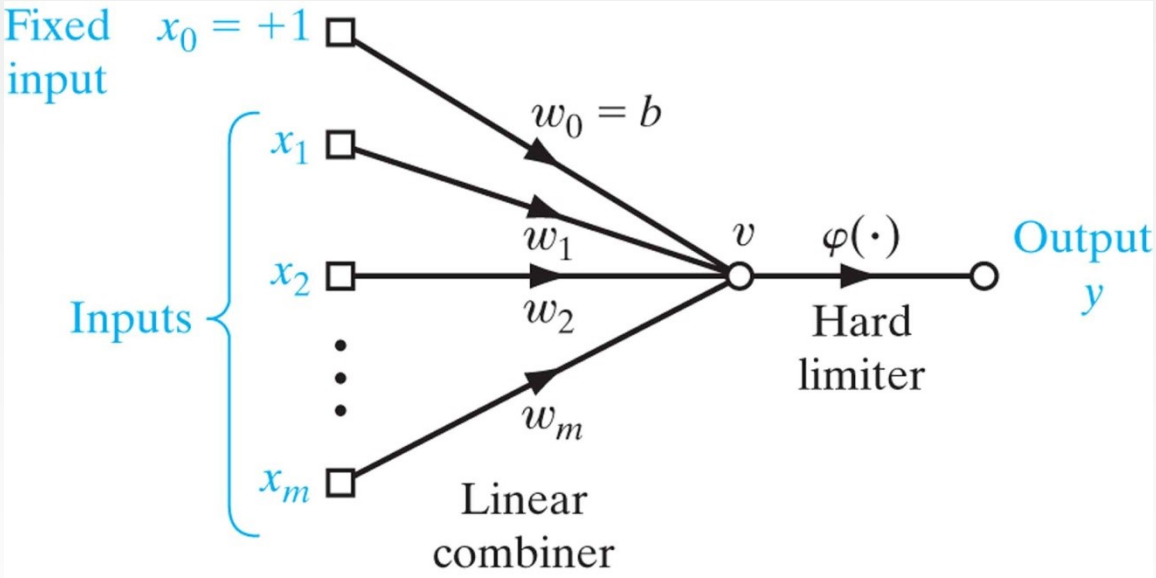
The Perceptron Architecture

- A perceptron nonlinear neuron, which uses the hard-limit transfer function (***signum activation function)***, returns a +1 or -1.
- **Weights are adapted using an error-correction rule,** The training technique used is called *the perceptron learning rule.*

$$v = \sum_{j=1}^{m} w_j x_j + b$$

$$= \sum_{j=0}^{m} w_j x_j = w^T x$$

$$y_k = \varphi(v) = \text{sgn}(v) = \begin{cases} +1 & v \geq 0 \\ -1 & v < 0 \end{cases}$$



The Perceptron Convergence Algorithm

The learning algorithms find a weight vector w such that:

$$\mathbf{w}^T\mathbf{x} > 0 \quad \text{for every input vect or } \mathbf{x} \in C1$$

$$\mathbf{w}^T\mathbf{x} \leq 0 \quad \text{for every input vect or } \mathbf{x} \in C2$$

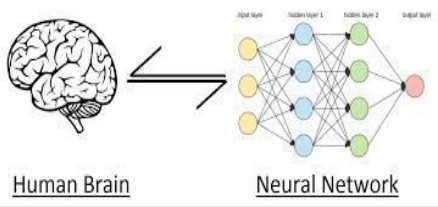
then the training problem is then *to find a weight vector* W *such that the previous two inequalities are satisfied*.

- The learning algorithms find a weight vector w such that:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta w \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) \leq 0 \quad \text{and } x(n) \in C1$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta w \quad \text{if } \mathbf{w}^T(n)\mathbf{x}(n) > 0 \quad \text{and } x(n) \in C2$$

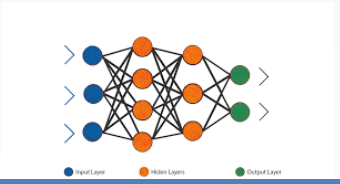
The Perceptron Convergence Theorem



Rosenblatt (1958,1962)

Rosenblatt designed a learning law for **weighted adaptation** in the McCulloch-Pitts model.

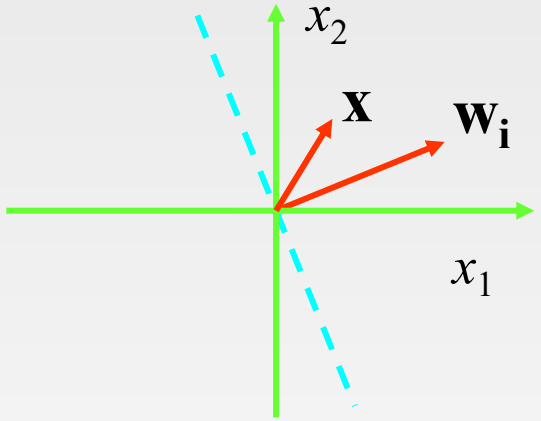
If a problem is linearly separable, then
The perceptron will *converges after some n iterations*
i.e.
The perceptron will learn in a finite number of steps.



Drive the error-correction learning rule - how to adjusting the Weight Vector?

The **net input signal** is the sum of all inputs after passing the synapses:

$$net = \sum_{j=0}^{m} w_j x_j$$



- This can be viewed as computing the **inner product** of the vectors $\mathbf{w_i}$ and **X**:

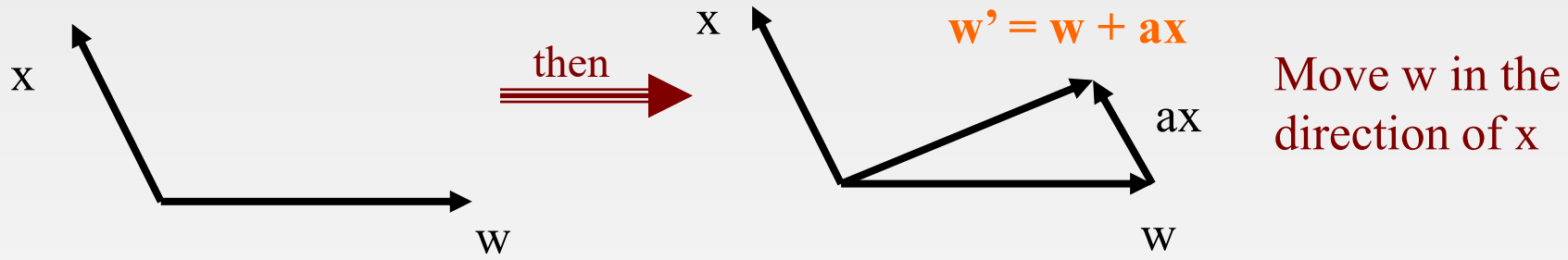
$$net = \|w_j\| . \|x_j\| . \cos(\alpha)$$

where α is the **angle** between the two vectors.

Drive the error-correction learning rule , cont.

Case 1. if Error=t-y=0, then make a change Δw=0.

Case 2. if Target t=1, Output y=-1, and the Error=t-y=2

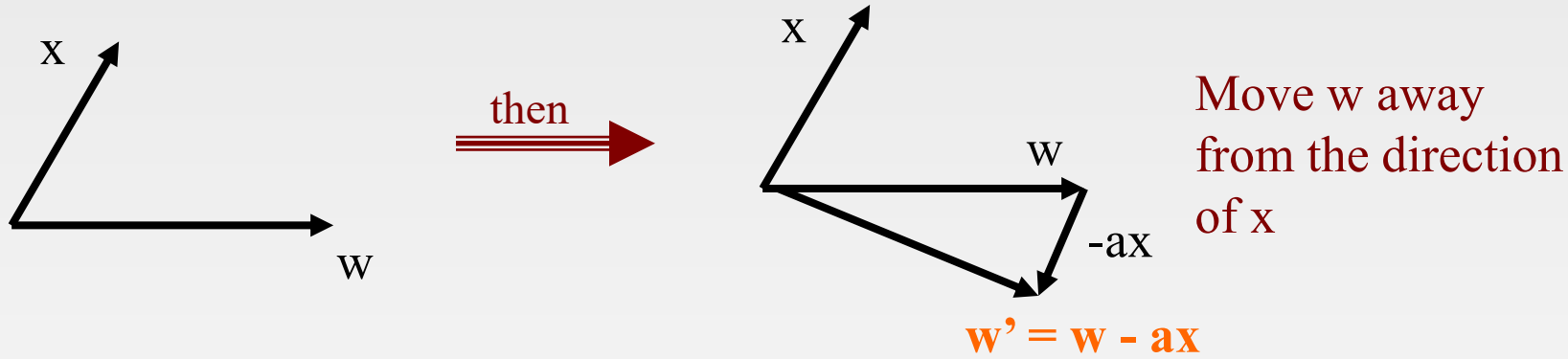


Move w in the direction of x

- The weight vector results in an incorrectly classifies the vector X.
- **The input vector αX is added to the weight vector w.**
- **This makes** the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

Drive the error-correction learning rule , cont.

Case 3. if Target t=-1, Output y=1, and the Error=t-y=-2



The weight vector results in an incorrectly classifies the vector X.
The input vector αX is <u>subtracted</u> from the weight vector w. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector will be classified as a -1 in the future.

Drive the error-correction learning rule , cont.

The perceptron learning rule can be written more succinctly in terms of the error **e = t – y and the change to be made to the weight vector Δw:**

CASE 1. If e = 0, then make a change Δw equal to 0.

CASE 2. If e = 2, then make a change Δw proportional to x.

CASE 3. If e = –2, then make a change Δw proportional to –x.

We can see that the sign of **X** is the same as the sign on the error, *e.* Thus, all three cases can then be written with a single expression:

$$\Delta w = \eta\,(t-y)\,x$$

$$W(n+1) = w(n) + \Delta w(n) = w(n) + \eta\,(t-y)\,x$$

With w(0)= random values. Where n denotes the time-step in applying the algorithm.

The parameter η is called the *learning rate*. It determines the magnitude of weight updates Δw .

Learning Rate η

- η governs the rate at which the training rule converges toward the correct solution.
- Typically $0 < \eta <= 1$.
- Too small an η produces slow convergence.
- Too large of an η can cause oscillations in the process.

Perceptron Convergence Algorithm

1- Start with a randomly chosen weight vector w(0)

2- Repeat

3- **for** each training vector pair ($\mathbf{X}_i$,t_i)

4- evaluate the output y_i when $\mathbf{X}_i$ is the input

5- **if** $y_i \neq t_i$ then

form a new weight vector $\mathbf{w}_{i+1}$ according to

$$w_{i+1} = w_i + \Delta w_i = w_i + \eta\,(t-y)\,X_i$$

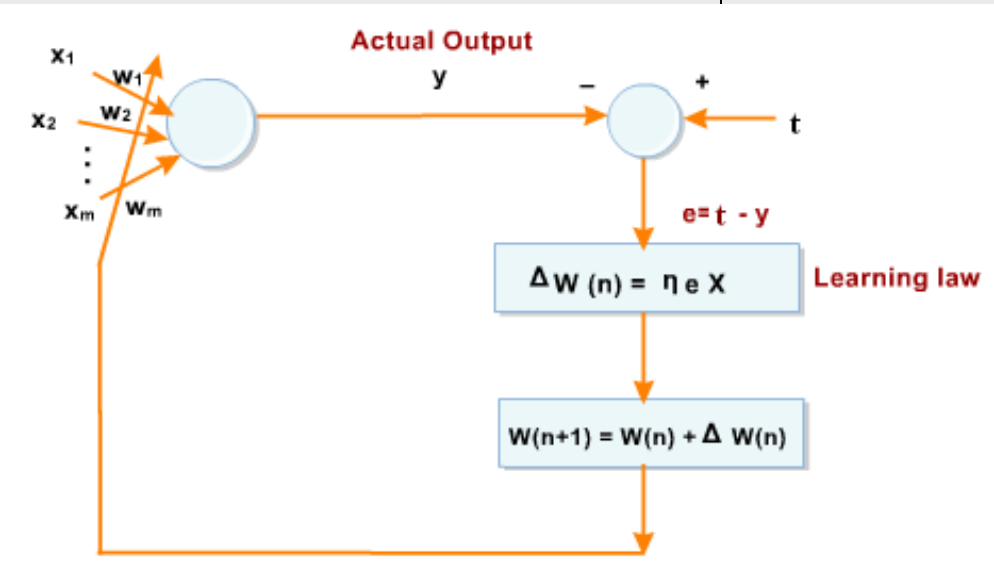
else

do nothing

end if

end for

5- Until the stopping criterion (convergence) is reached



convergence : epoch doesn't produce an error

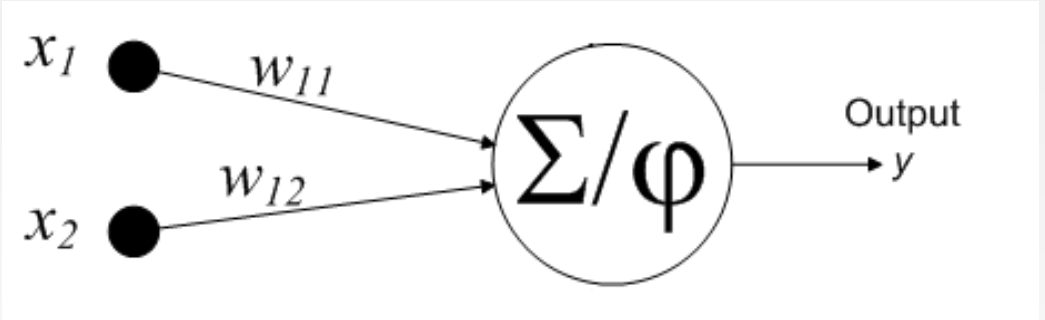
(i.e all training vector classified correctly (y=t) for the same w(n))

(i.e. until error is zero for all training patterns).

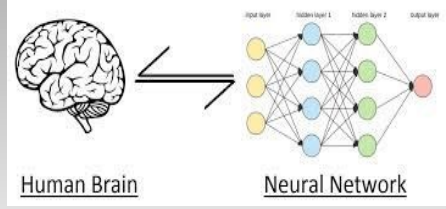
(i.e. no weights change in step 5 for all training patterns)

Example

We will begin with a simple test problem to show how the perceptron learning rule should work. To simplify our problem, we will begin with a network without a bias and have two-inputs and one output. The network will then have just two parameters, w_{11} and w_{12}, to adjustment as shown in the following figure.



By removing the bias we are left with a network whose decision boundary must pass through the origin.

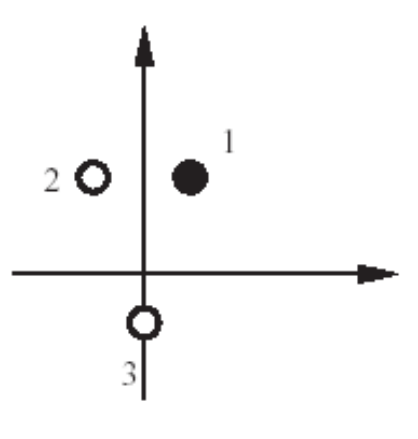


Example 1

The input/target pairs for our test problem are

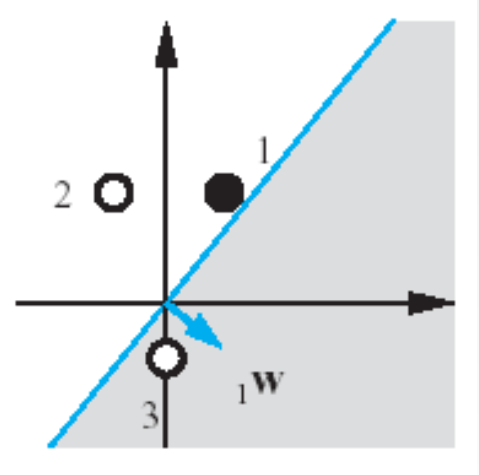
$$X_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}; X_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}; X_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

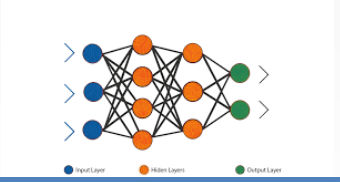
$$d_1 = 1; d_2 = -1; d_3 = -1$$



which are to be used in the training together with the following weights

$$w = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix}; \eta = 0.5$$



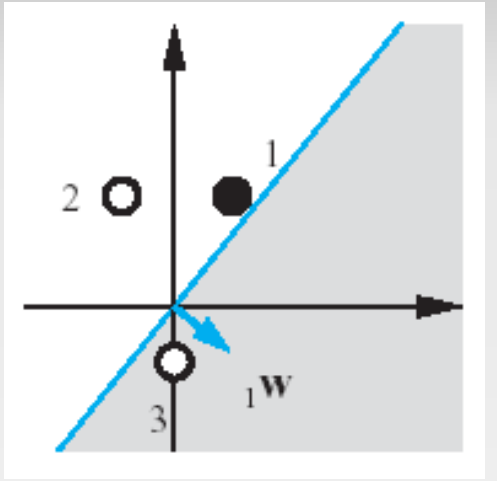


Show how the learning proceeds?

Iteration 1. Input is X_1 and the desired output is d_1:

$$net_1 = w_1^T X_1 = [1.0 \quad -0.8]\begin{bmatrix}1\\2\end{bmatrix} = -0.6$$

$$y = \text{sgn}(net_1) = -1$$



since $d_1=1$, the error signal is

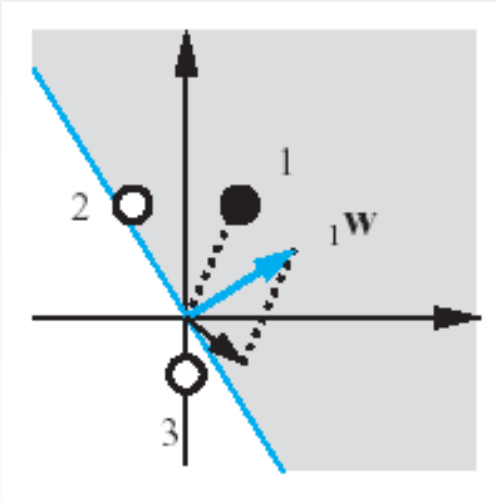
$e = 1-(-1) = 2$

Then the Correction (update weights) in this step is necessary

The updated weight vector is

$$w_2 = w_1 + 0.5*(1-(-1)) * X_1$$

$$w_2 = \begin{bmatrix}1.0\\-0.8\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}2.0\\1.2\end{bmatrix}$$



Iteration 2. Input is X_2 and the desired output is d_2:

$$net_2 = w_2^T X_2 = [2.0 \quad 1.2]\begin{bmatrix} -1 \\ 2 \end{bmatrix} = 0.4$$

$$y = \text{sgn}(net_2) = 1$$

since d_2=-1 , the error signal is

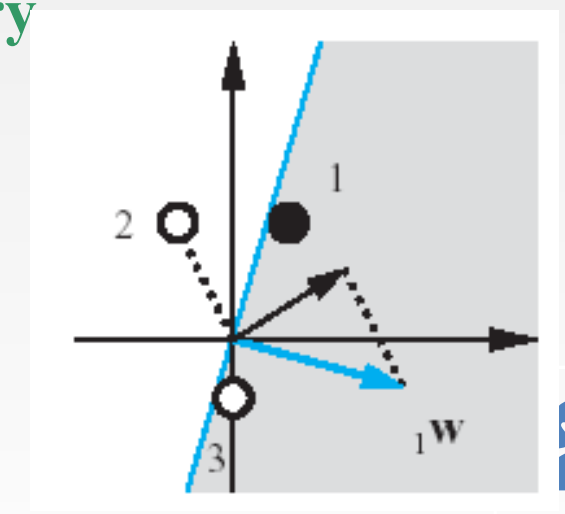
e = -1-1 = -2

Then the Correction (update weights) in this step is necessary

The updated weight vector is

$$w_3 = w_2 + 0.5*(-1 \quad -1) * X_2$$

$$w_3 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix}$$



Iteration 3. Input is X_3 and the desired output is d_3:

$$net_3 = w_3^T X_3 = [3.0 \quad -0.8]\begin{bmatrix} 0 \\ -1 \end{bmatrix} = 0.8$$

$$y = \text{sgn}(net_3) = 1$$

since d_3=-1 , the error signal is

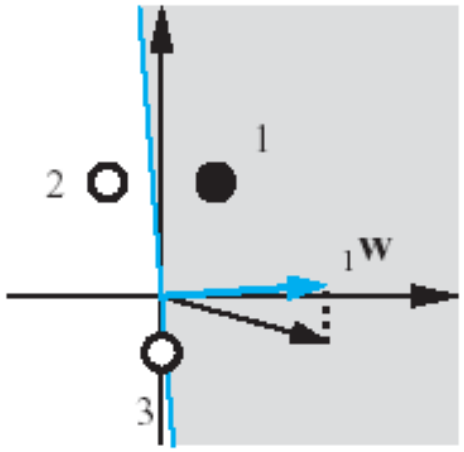
e = -1-1 = -2

Then the Correction (update weights) in this step is necessary

The updated weight vector is

$$w_4 = w_3 + 0.5 * (-1 \quad -1) * X_3$$

$$w_4 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}$$



check

weights change for iteration 1, 2, and 3.

No stop

Iteration 4:

$$net_1 = w_4^T X_1 = [3.0 \quad 0.2]\begin{bmatrix}1\\2\end{bmatrix} = 3.4$$

$$y_1 = \text{sgn}(net_1) = 1 = d_1$$

✓

Iteration 5:

$$net_2 = w_4^T X_2 = [3.0 \quad 0.2]\begin{bmatrix}-1\\2\end{bmatrix} = -2.6$$

$$y_2 = \text{sgn}(net_2) = -1 = d_2$$

✓

Iteration 6:

$$net_3 = w_4^T X_3 = [3.0 \quad 0.2]\begin{bmatrix}0\\-1\end{bmatrix} = -0.2$$

$$y = \text{sgn}(net_3) = -1 = d_3$$

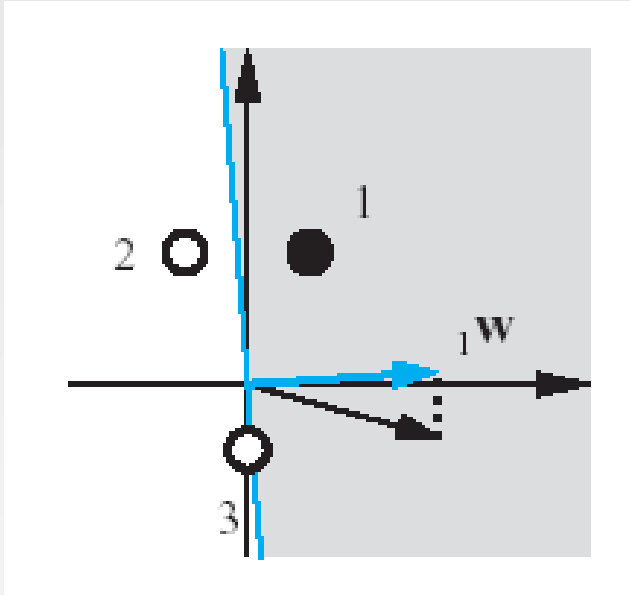
✓

---

check No weights change for iteration 4, 5, and 6

Stopping criteria is satisfied

The diagram to the left shows that the perceptron has finally learned to classify the three vectors properly.
If we present any of the input vectors to the neuron, it will output the correct class for that input vector.



Your turn (Homework 1)

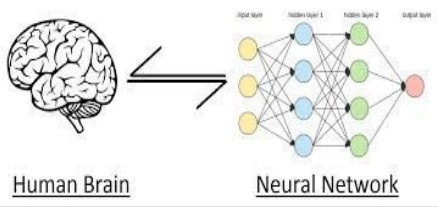
Consider the following set of training vectors x_1, x_2, and x_3,

$$X_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}; X_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}; X_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$$

With learning rate equals 0.1, and the desired responses and weights initialized as:

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

Show how the learning proceeds?



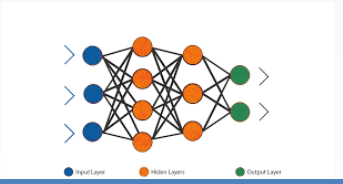
Example 2

Consider the AND problem

$$x_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}; x_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}; x_3 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}; x_4 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

With bias b =0.5 and the desired responses and weights initialized as:

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}; w = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix}; \eta = 0.1$$



Show how the learning proceeds?

Consider:

$$X_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}; X_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}; X_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}; X_4 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \quad w = \begin{bmatrix} 0.5 \\ 0.3 \\ 0.7 \end{bmatrix}$$

Iteration 1. Input X_1 ***present*** and its desired output d_1:

$$net_1 = w_0^T X_1 = [0.5\ 0.3\ 0.7]\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 1.5$$

$$y = \text{sgn}(net_1) = 1$$

since d_1=*1* , the error signal is

$$e_1 = 1-1 = 0$$

Correction (update the weights) in this step is not necessary

$$w_1^T = w_0^T$$

Iteration 2. Input X_2 ***present*** and its desired output d_2:

$$net_2 = w_1^T X_2 = [0.5\ 0.3\ 0.7]\begin{bmatrix}1\\1\\-1\end{bmatrix} = 0.1$$

$$y = \mathrm{sgn}(net_2) = 1$$

since d_2=-1 , the error signal is

$$e_2 = -1-1 = -2$$

Correction in this step is necessary

The updated weight vector is $w_2 = w_1 + 0.1(-1\ -1) * X_2$

$$w_2 = \begin{bmatrix}0.5\\0.3\\0.7\end{bmatrix} + \begin{bmatrix}-0.2\\-0.2\\0.2\end{bmatrix} = \begin{bmatrix}0.3\\0.1\\0.9\end{bmatrix}$$

$$w_2^T = [0.3\ \ 0.1\ \ 0.9]$$

Iteration 3. Input X_3 present and its desired output d_3:

$$net_3 = w_2^T X_3 = [0.3\ 0.1\ 0.9]\begin{bmatrix}1\\-1\\1\end{bmatrix} = 1.1$$

$$y = \mathrm{sgn}(net_3) = 1$$

since d_3=-1 , the error signal is

$$e_3 = -1-1 = -2$$

Correction in this step is necessary

The updated weight vector is

$$w_3 = w_2 + 0.1*(-1\ -1)\ *\ X_3$$

$$w_3 = \begin{bmatrix}0.3\\0.1\\0.9\end{bmatrix} + \begin{bmatrix}-0.2\\0.2\\-0.2\end{bmatrix} = \begin{bmatrix}0.1\\0.3\\0.7\end{bmatrix}$$

$$w_3^T = [0.1\ \ 0.3\ \ 0.7]$$

Iteration 4. Input is X_4 present and its desired output d_4:

$$net_4 = w_3^T X_4 = [0.1\ 0.3\ \ 0.7]\begin{bmatrix}1\\-1\\-1\end{bmatrix} = -0.9$$

$$y = \mathrm{sgn}(net_4) = -1$$

since d_4=-1 , the error signal is

$$e_4 = -1+1 = 0$$

Correction in this step is not necessary

$$w_4^T = w_3^T$$

check weights change for iteration 2 and 3.

Stopping criteria is not satisfied, continuous with epoch 2

Iteration 5

$$net_5 = w_4^T X_1 = [0.1\ 0.3\ 0.7]\begin{bmatrix}1\\1\\1\end{bmatrix} = 1.1$$

$$y = \text{sgn}(net_5) = 1 = d_1$$

$$w_5^T = w_4^T \quad \checkmark$$

Iteration 6

$$net_6 = w_5^T X_2 = [0.1\ 0.3\ 0.7]\begin{bmatrix}1\\1\\-1\end{bmatrix} = -0.3$$

$$y = \text{sgn}(net_6) = -1 = d_2$$

$$w_6^T = w_5^T \quad \checkmark$$

Iteration 7

X

$$net_7 = w_6^T X_3 = [0.1\ 0.3\ 0.7]\begin{bmatrix}1\\-1\\1\end{bmatrix} = 0.5$$

$$y = \text{sgn}(net_7) = 1 \neq d_3$$

Iteration 7.

Correction in this step is necessary

$$w_7 = w_6 + 0.1*(-1\ -1)\ *\ X_3$$

$$w_7 = \begin{bmatrix} 0.1 \\ 0.3 \\ 0.7 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.2 \\ -0.2 \end{bmatrix} = \begin{bmatrix} -0.1 \\ 0.5 \\ 0.5 \end{bmatrix}$$

$$w_7^T = \begin{bmatrix} -0.1 & 0.5 & 0.5 \end{bmatrix}$$

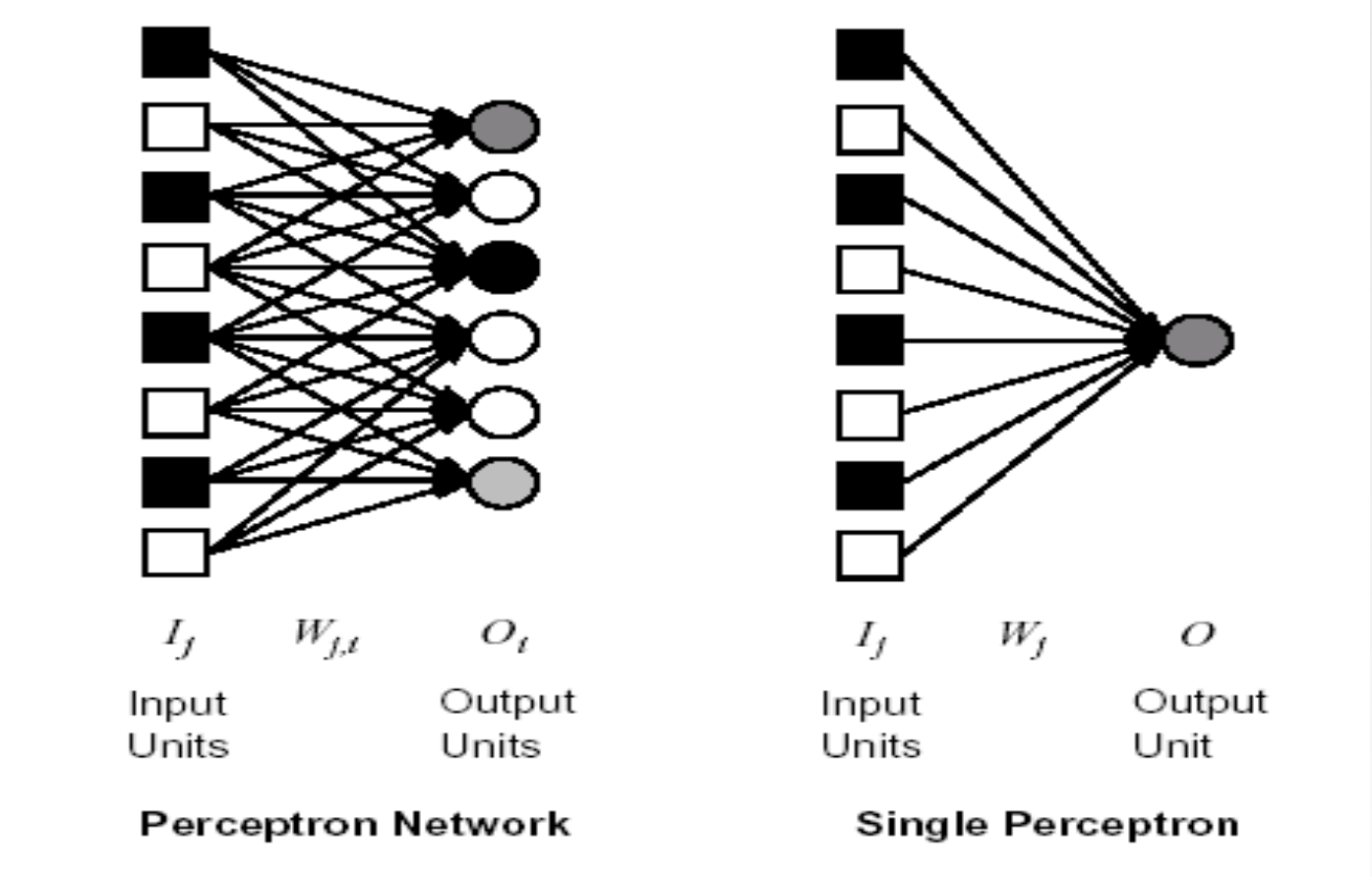
Iteration 8. Now all the patterns will give the correct response. Try it.

We can draw the solution found using the line:

$$0.5x_1 + 0.5x_2 - 0.1 = 0$$

Perceptrons Architecture

There are many kinds of perceptrons:

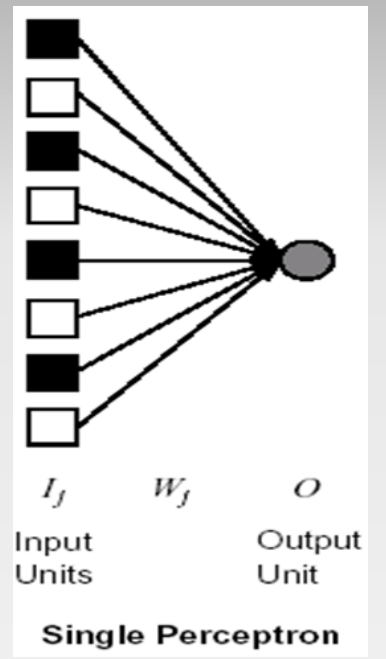


Multiple-output-neurons

single-output-neuron

Single-output-neuron

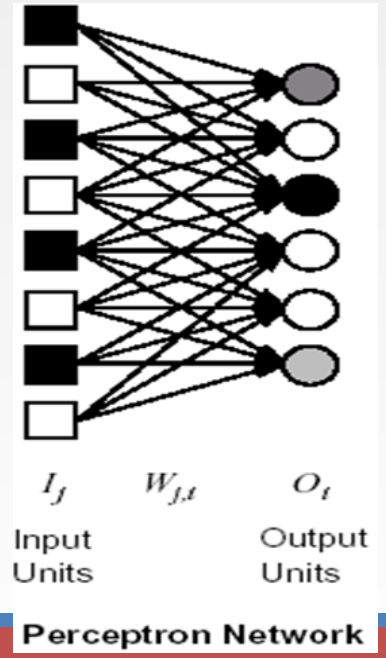
A single-output-neuron can classify input vectors into two categories, since its output can be either -1 or 1.



Multiple-output Neuron Perceptron

A multiple-output-neuron perceptron can classify inputs into many categories:

- Each category is represented by a different output vector.
- Since each element of the output vector can be either -1or 1.
- There are a total of 2^S possible categories, where S is the number of output's neurons. *For Example: 2 output neurons, there are 4 classes: each class can has a code: 1 1, 1 -1, -1 1, -1 -1.*
- ***Or using* "1-of-M" or "one-vs-all" output coding.**



“1-of-M” or “one-vs-all” output coding

Training with the “1-of-M” coding is

Set the target output to 1 for the correct class, and set all of the other target outputs to 0 or -1.

Example 1: if we want to recognition 5 faces, so we have 5 classes:

Class 1	1 0 0 0 0
Class 2	0 1 0 0 0
Class 3	0 0 1 0 0
Class 4	0 0 0 1 0
Class 5	0 0 0 0 1

OR

Class 1	1 -1 -1 -1 -1
Class 2	-1 1 -1 -1 -1
Class 3	-1 -1 1 -1 -1
Class 4	-1 -1 -1 1 -1
Class 5	-1 -1 -1 -1 1

Multiple Output units: One-vs-all


Pedestrian


Car


Motorcycle


Truck

Want $y \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $y \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $y \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.

when pedestrian when car when motorcycle

Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$

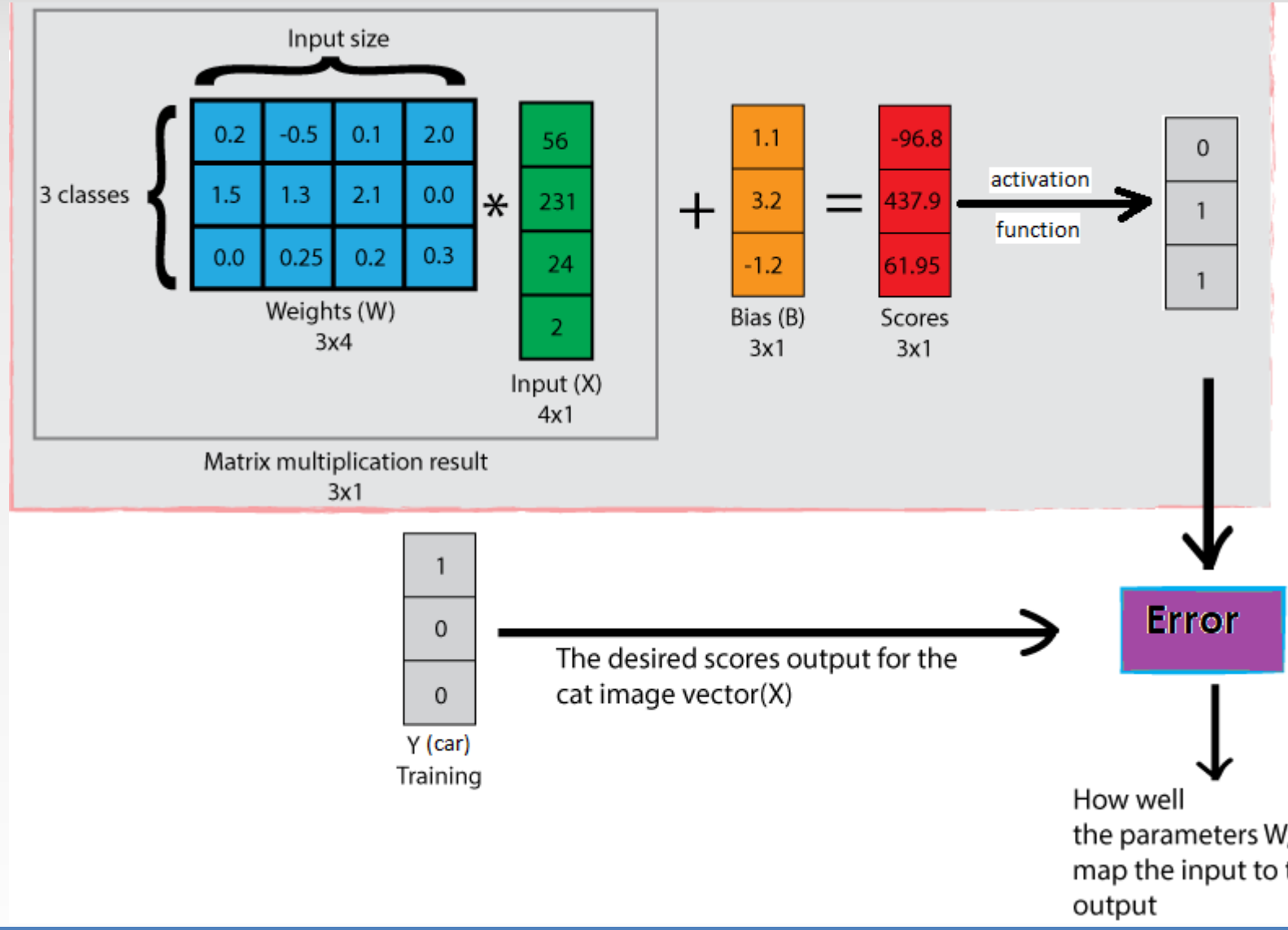
$y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

Example

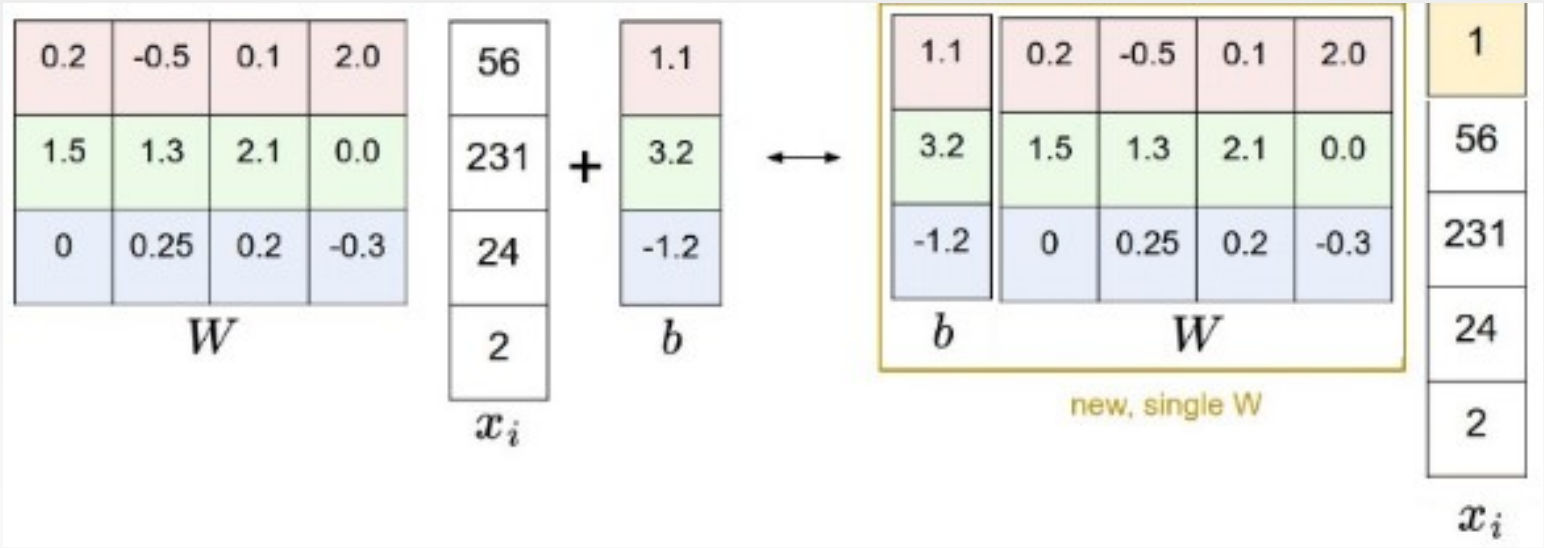
We need to classify images as (Car, Deer, airplane)

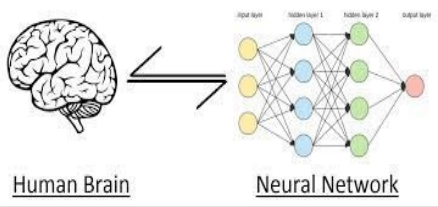
We transform the matrix of numbers (image) on a 1d vector.

Let image size 2 x 2 pixels, the images become an array 4x1.



Consider the bias as part of the weight matrix, the advantage of this approach is that we can solve the linear classification with a single matrix multiplication



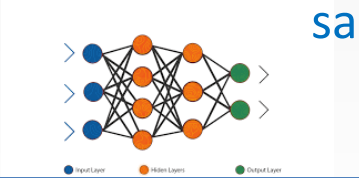


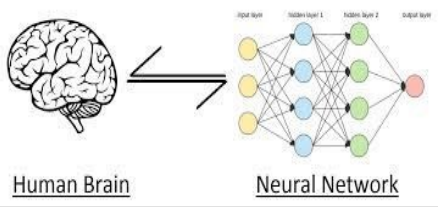
Limit of Perceptron Learning rule

If there is no separating hyperplane, the perceptron will never classify the samples 100% correctly.

But there is nothing from trying. So we need to add something to stop the training, like:

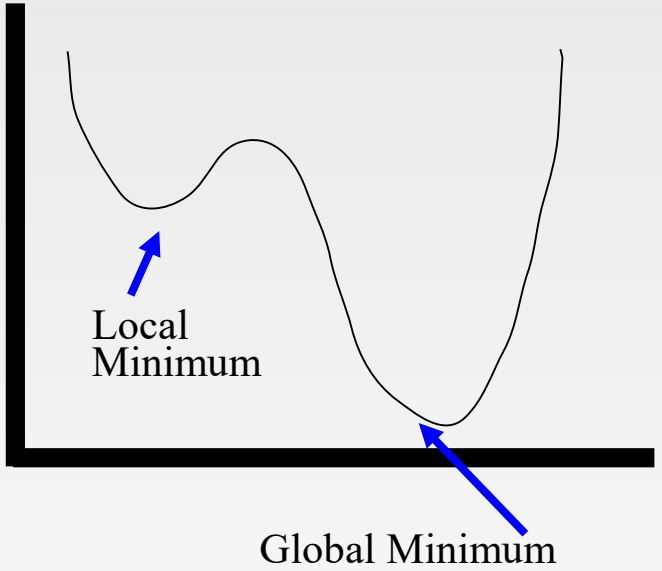
- Put a **limit** on the **number of iterations**, so that the algorithm will terminate even if the sample set is not linearly separable.
- Include an **error bound**. The algorithm can stop as soon as the portion of misclassified samples is less than this bound. **This ideal is developed in the Adaline training algorithm.**





Error Correcting Learning

The objective of this learning is to start from an arbitrary point error and then move toward a global minimum error, in a step-by-step fashion.

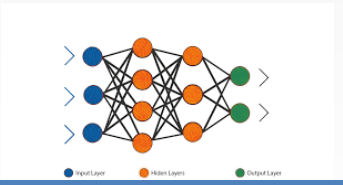


- The arbitrary point error determined by the initial values assigned to the synaptic weights.
- It is closed-loop feedback learning.

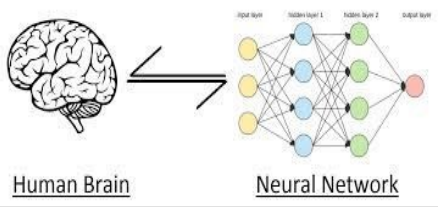
- Examples of error-correction learning:

  The **least-mean-square (LMS) algorithm** (Windrow and Hoff), also called ***delta rule.***

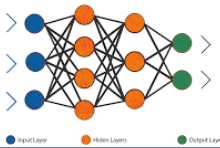
  and its generalization known as the ***back-propagation*** **(BP) algorithm.**

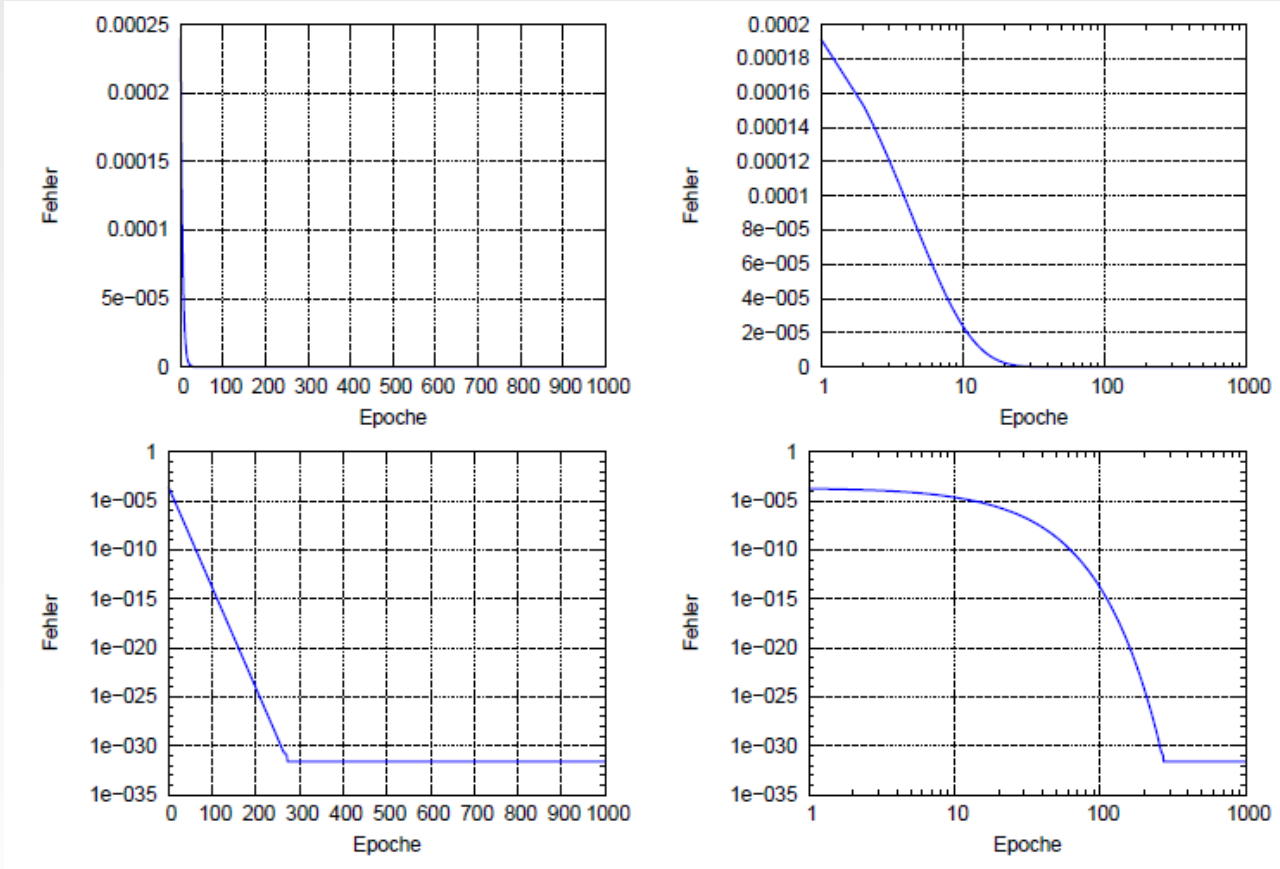


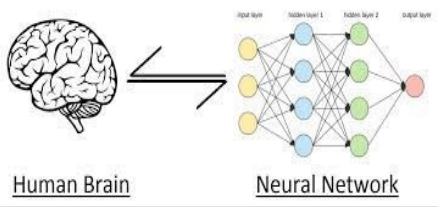
learning curve



- The learning curve indicates the progress of the error. The motivation to create a learning curve is that such a curve can indicate whether the network is progressing or not.
- Let
  - Initial state: a random set of weights
  - Goal state: a set of weights that minimizes the error on the training set.
  - Evaluation function (performance index, cost function, or lost function): an error function.
  - Operators: how to move from one state to the other; defined by the learning algorithm.

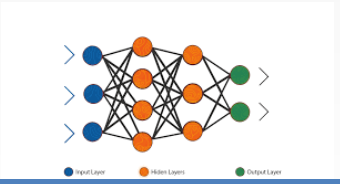


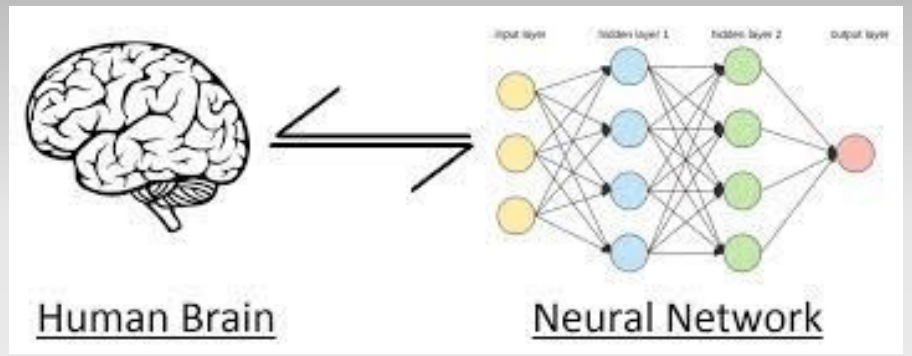




Loss functions

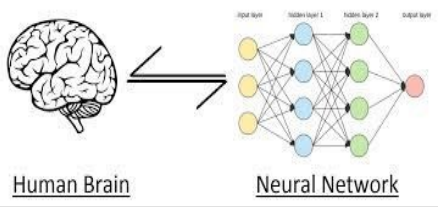
- The purpose of the loss function is to evaluate the models result by comparing it to the ground truth.
- That is to compare the input to its targets.
- The task of the optimization algorithms is to reduce the loss and so loss functions are essential to improve the network.





Adaline (Adaptive Linear Neuron) Networks

Adaline (Adaptive Linear Neuron) Networks



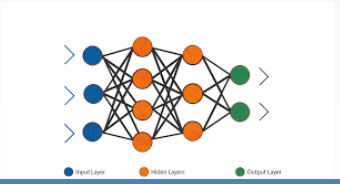
1960 - **Bernard Widrow** and his student **Marcian Hoff** introduced the ADALINE Networks and its learning rule which they called the Least mean square (LMS) algorithm (or Widrow-Hoff algorithm or delta rule)

The Widrow-Hoff algorithm

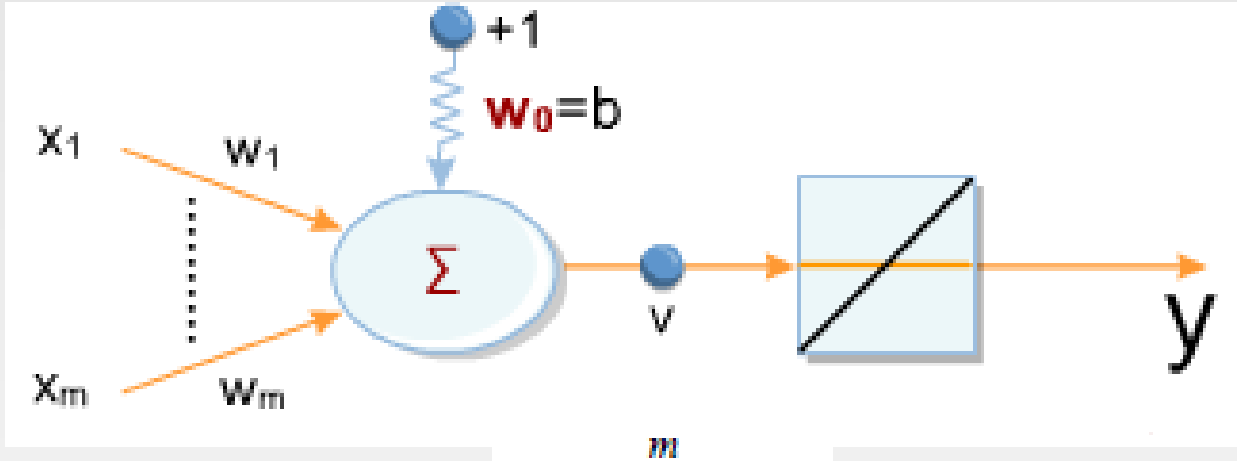
- can only train by single-Layer network.

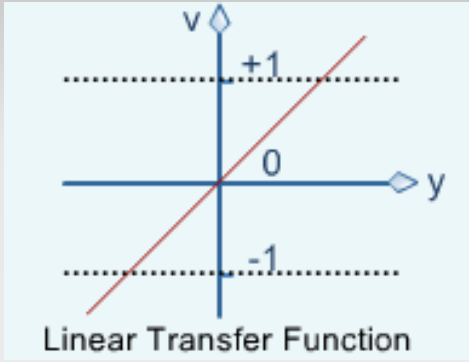
Both the Perceptron and Adaline can only solve linearly separable problems

- (i.e., the input patterns can be separated by a linear plane into groups, like AND and OR problems).



Adaline Architecture

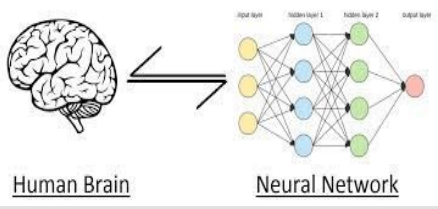




$$v = \sum_{i=0}^{m} x_i w_i$$

$$y = f(v) = v$$

- Given:
  - $x_k(n)$: an input value for a neuron k at iteration n,
  - $d_k(n)$: the desired response or the target response for neuron k.
- Let:
  - $y_k(n)$: the actual response of neuron k.



The Cost function (Objective function, loss function, Empirical Risk)

The **loss** of our network measures the cost incurred from incorrect outputs.

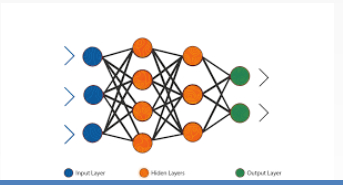
Error: difference between the target and actual network output.

- error signal for neuron k at iteration n: $e_k(n) = d_k(n) - y_k(n)$

Mean square error can be used to measure the total loss over our entire dataset (batch mode):

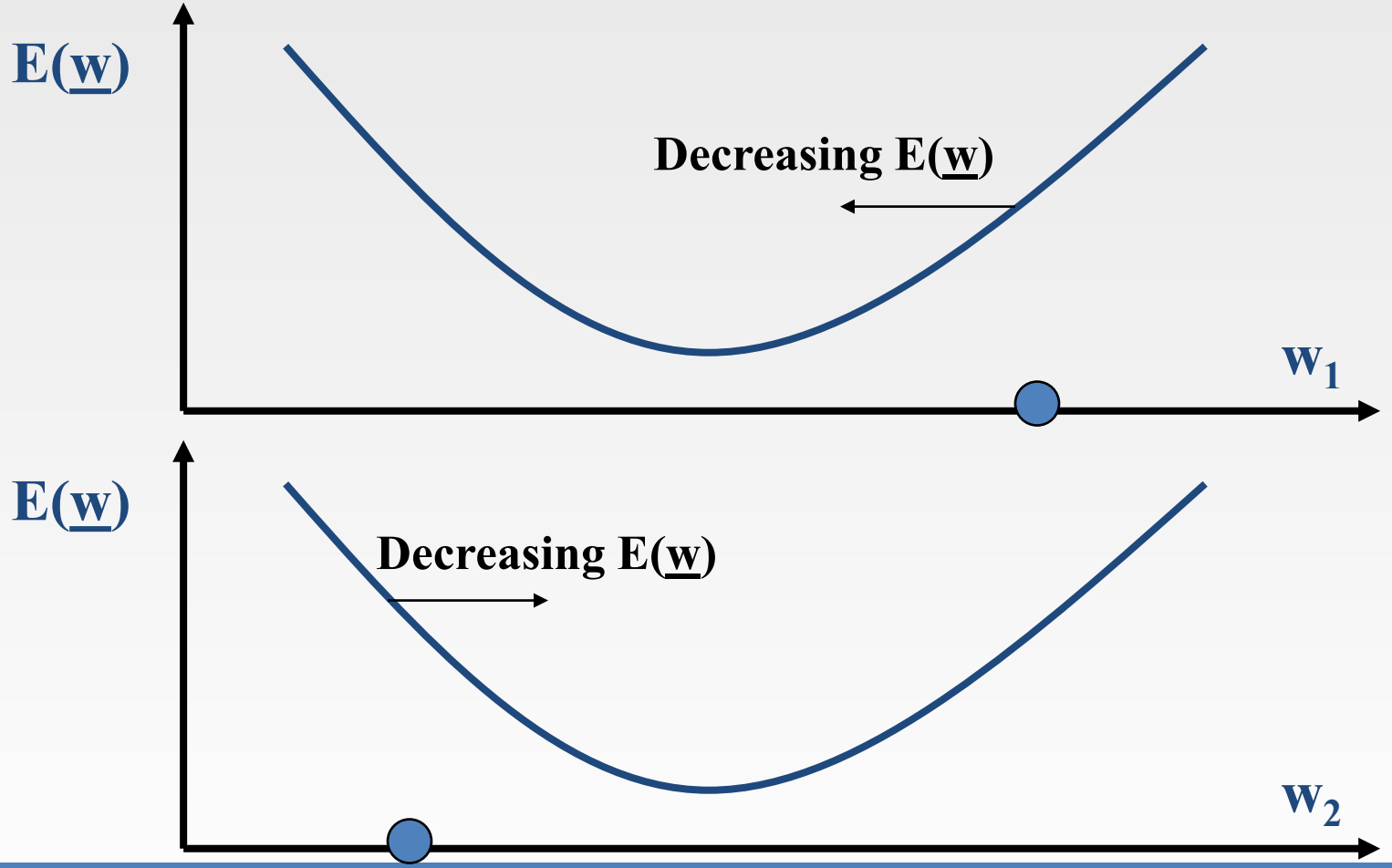
- batch mode means take the *mean squared error* over all ***m training patterns:***

$$E(n) = \frac{1}{m}\sum_{p=1}^{m}\frac{1}{2}(d^p - y^p)^2$$



Error Landscape in Weight Space

- Total error signal is a function of the weights
  - Ideally, we would like to find the global minimum (i.e. the optimal solution)

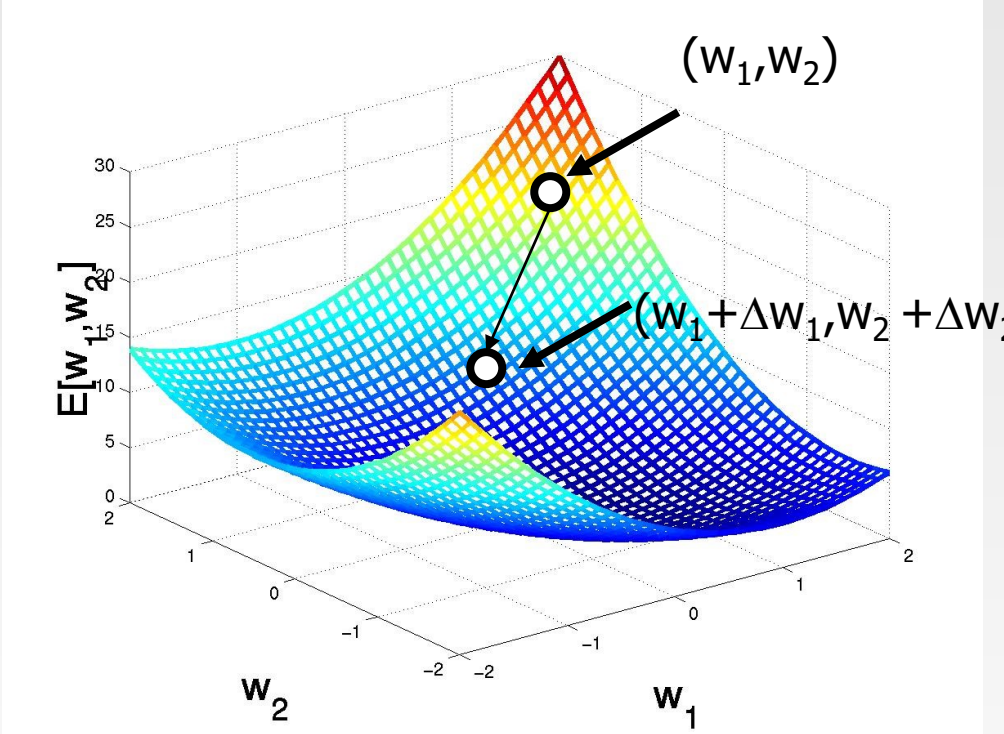


Error Landscape in Weight Space, cont.

- In the error space of the linear networks (ADALINE's), we need to an optimisation algorithm that finds the weights (parameters or coefficients of a function) where the function has only one minimum called the global minimum.
- Takes steps downhill, moves down as fast as possible

i.e. moves in the direction that makes the largest reduction in error

- how is this direction called?

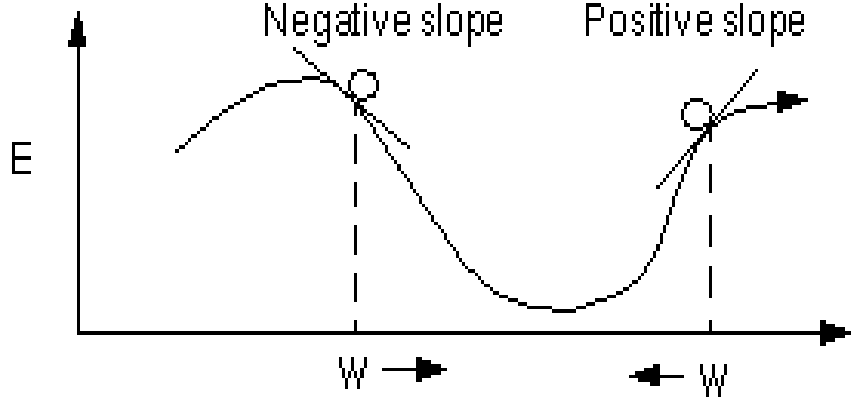


Steepest Descent

- The direction of the steepest descent is called gradient and can be computed.
- Any function
  - increases most rapidly when the direction of the movement is in the direction of the gradient
  - decreases most rapidly when the direction of movement is in the direction of the negative of the gradient
- Change the weights so that we move a short distance in the direction of the **greatest rate of decrease** of the error, i.e., in the direction of –ve gradient.

Slope of E positive
=> decrease W
Slope of E negative
=> increase W

$$\Delta w = - \eta * \partial E / \partial w$$



Gradient Descent Rule

- It consists of computing the **gradient** of the error function, then taking a **small step** in the **direction of negative gradient**, which hopefully corresponds to decrease function value, then repeating for the new value of the dependent variable.
- i.e. Gradient Descent is an iterative process that finds the minima of a function. This is an optimisation algorithm that finds the weights (parameters or coefficients of a function) where the function has a minimum value. Although this function does not always guarantee to find a global minimum and can get stuck at a local minimum.
- In order to do that, we calculate the partial derivative of the error with respect to each weight.
- The change in the weight proportional to the derivative of the error with respect to each weight, and additional proportional constant (learning rate) is tied to adjust the weights.

$$\Delta w = - \eta * \partial E / \partial w$$

Gradient Descent

Algorithm

1. Initialize weights randomly
2. Loop until convergence:
3. Compute gradient, $\frac{\partial \mathbf{E}(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial \mathbf{E}(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side.

LMS Algorithm - Derivation

Given

- $x_k(n)$: an input value for a neuron k at iteration n,
- $d_k(n)$: the desired response or the target response for neuron k.

Let:

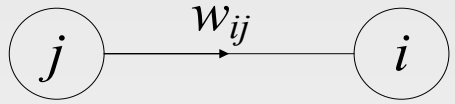
- $y_k(n)$: the actual response of neuron k.
- $e_k(n)$: error signal = $d_k(n) - y_k(n)$

Let's first work it out for the case of if we have **only one training example**, so that we can neglect the sum in the definition of E.

$$E(n) = \frac{1}{2} e_k^2(n)$$

LMS Algorithm – Derivation, cont.

The derivative of the error with respect to each weight $\frac{\partial E}{\partial w_{ij}}$ can be written as:

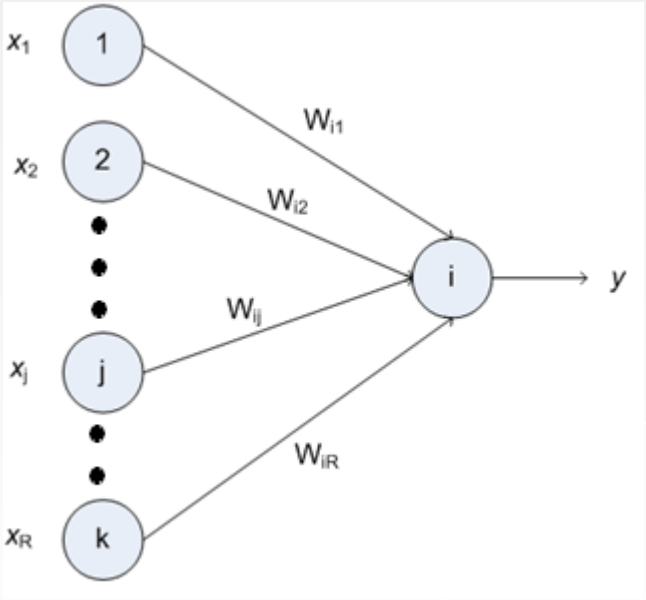


$$\nabla E(w_{ij}) = \frac{\partial E}{\partial w_{ij}} = \frac{\partial \frac{1}{2} e_i^2}{\partial w_{ij}} = \frac{\partial \frac{1}{2}(d_i - y_i)^2}{\partial w_{ij}}$$

- Next we use the chain rule to split this into two derivatives:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial \frac{1}{2}(d_i - y_i)^2}{\partial y_i} \frac{\partial y_i}{\partial w_{ij}} \qquad \frac{\partial E}{\partial w_{ij}} = \left(\frac{1}{2} * 2(d_i - y_i) * -1\right) * \frac{\partial f\left(\sum_{j=1}^{R} w_{ij} x_j\right)}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = -(d_i - y_i) * x_j * f'\left(\sum_{j=1}^{R} w_{ij} x_j\right)$$



LMS Algorithm – Derivation, cont.

$$\nabla E(w_{ij}) = \frac{\partial E}{\partial w_{ij}} = -(d_i - y_i) * x_j * f'(net_i)$$

$$\Delta w_{ij} = -\eta \nabla E(w_{ij}) = \eta (d_i - y_i) f'(net_i) x_j$$ This is called the Delta Learning rule.

- The widrow-Hoff learning rule is a special case of Delta learning rule. Since the Adaline's transfer function is **linear function:**

$$f(net_i) = net_i \qquad and \qquad f'(net_i) = 1$$

- then

$$\nabla E(w_{ij}) = \frac{\partial E}{\partial w_{ij}} = -(d_i - y_i) * x_j$$

- The widrow-Hoff learning rule is: $$\Delta w_{ij} = -\eta \nabla E(w_{ij}) = \eta (d_i - y_i) x_j$$

Variants of Gradient descent

We'd derived the LMS rule for when there was only a single training example. There are two ways to modify this method for a training set of more than one example.

First way:

$$\Delta w_{ij} = -\eta \nabla E(w_{ij}) = \eta \sum_{P=1}^{m} (d_i^P - y_i^P)\, x_j$$

We can easily verify that the quantity in the summation in the update rule above is just ∂E(θ)/∂Wj (for the original definition of E).

So, this is **simply Gradient descent** on the original cost function E. This method looks at every example in the entire training set on every step, and is called **batch Gradient descent**.

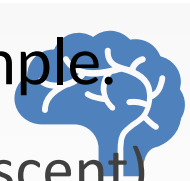
Variants of Gradient descent, cont.

Second way:

Loop {
    for p=1 to m {
        $\Delta w_{ij} = -\eta \nabla E(w_{ij}) = \eta(d_i^P - y_i^P)x_j \quad (for\ every\ j)$
    }
}

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the weights (parameters) according to the gradient of the error with respect to that single training example only.

Stochastic Gradient Descent (SGD) computes the gradient using a single sample.



This algorithm is called **Stochastic Gradient Descent** (also incremental gradient descent).

Adaline Training Algorithm (Stochastic Gradient Descent)

1- Initialize the weights to small random values and select a learning rate, (η)

2- Repeat

3- **for m** training patterns

select input vector **X** , with target output, t,

compute the output: **$v = b + w^T x$, $y = f(v)$**

Compute the output error **e=t-y**

update the bias and weights **$w_i(new) = w_i(old) + \eta\ (t - y)\ x_i$**

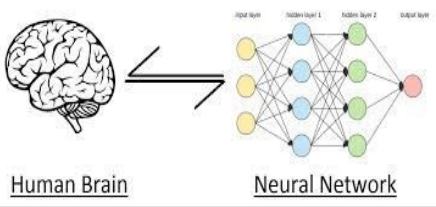
4- end for

5- until the stopping criteria is reached by find the Mean square error across all the training samples

$$mse(n) = \frac{1}{m}\sum_{k=1}^{m}\frac{1}{2}e(n)^2$$

stopping criteria: if the Mean Squared Error across all the training samples **is less than a specified value, stop the training.**

Otherwise , cycle through the training set again (go to step 2)

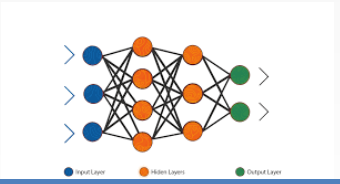


Batch Gradient Descent vs Stochastic Gradient Descent

Whereas batch Gradient descent has to scan through the entire training set before taking a single step—a costly operation if m is large.

Often, stochastic Gradient descent gets “close” to the minimum error much faster than batch Gradient descent.

For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch Gradient descent.



Convergence Phenomenon and Learning rate

The performance of an ADALINE neuron depends heavily on the choice of the learning rate. How to choose it?

- Too big: The system will oscillate and the system will not converge
- Too small: The system will take a long time to converge

Typically, h is selected by trial and error

- typical range: $0.01 < h < 1.0$
- often start at 0.1
- sometimes it is suggested that:
  $0.1/m < h < 1.0/m$
  where m is the number of inputs.

Usually, we take the value of the learning rate to be 0.1, 0.01 or 0.001. It is a hyper-parameter and you need to experiment with its values.

Example

The input/target pairs for our test problem are

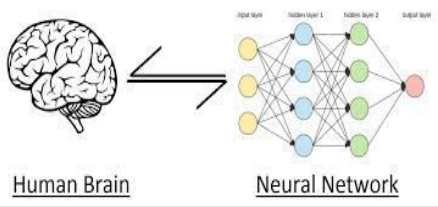
$$\left\{P_1 = \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix}, t_1 = [-1]\right\} \qquad \left\{P_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = [1]\right\}$$

Learning rate: $\eta = 0.4$

Stopping criteria: mse < 0.01

Initial weight: [0 0 0]

Show how the learning proceeds using the LMS algorithm?



Example Iteration One

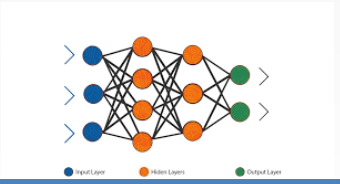
First iteration – p_1

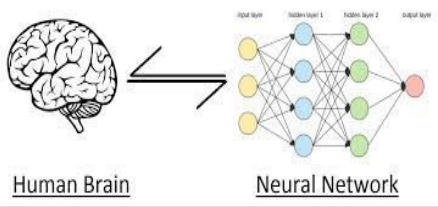
$$y = w(0) * P_1 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = 0$$

e = t – y = -1 – 0 = -1

$$w(1) = w(0) + \eta * e * P_1$$

$$w(1) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.4 \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.4 \\ -0.4 \\ 0.4 \end{bmatrix}$$





Example Iteration Two

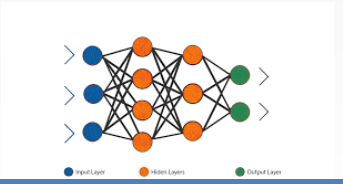
Second iteration – p_2

$$y = w(1) * P_2 = \begin{bmatrix} 0.4 & -0.4 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = -0.4$$

e = t – y = 1 – (-0.4) = 1.4

$$w(2) = w(1) + \eta * e * P_2$$

$$w(2) = \begin{bmatrix} 0.4 \\ -0.4 \\ 0.4 \end{bmatrix} + 0.4(1.4) \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.96 \\ 0.16 \\ -0.16 \end{bmatrix}$$



End of epoch 1, check the stopping criteria

Example – Check Stopping Criteria

For input P_1 $e_1 = t_1 - y_1 = t_1 - w(2)^T * P_1$

$$= -1 - \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -1 - (-0.64) = -0.36$$

For input P_2 $e_2 = t_2 - y_2 = t_2 - w(2)^T * P_2$

$$= 1 - \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = 1 - (1.28) = -0.28$$

$$mse = \frac{1}{2}\left(\frac{(-0.36)^2 + (-0.28)^2}{2}\right) = 0.05185 > 0.01$$

Stopping criteria is not satisfied, continue with epoch 2

Example – Next Epoch (epoch 2)

Third iteration – p_1

$$y = w(2) * P_1 = \begin{bmatrix} 0.96 & 0.16 & -0.16 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = -0.64$$

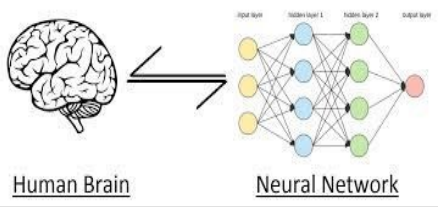
e = t – y = -1 – 0.64 = -0.36

$$w(3) = w(2) + \eta * e * P_1$$

$$w(3) = \begin{bmatrix} 0.96 \\ 0.16 \\ -0.16 \end{bmatrix} + 0.4(-0.36) \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1.104 \\ -0.016 \\ -0.016 \end{bmatrix}$$

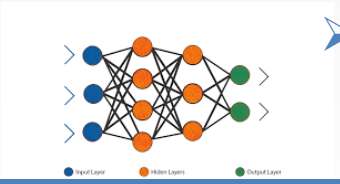
if we continue this procedure, the algorithm converges to:

W(…) = [1 0 0]

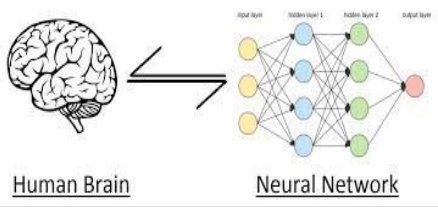


Compare ADALINE with Perceptron

- Both ADALINE and perceptron suffer from the same inherent limitation - can only solve linearly separable problems
- LMS, however, is more powerful than the perceptron's learning rule:
  - if the patterns are not linearly separable, i.e. the perfect solution does not exist, an ADALINE will find the best solution possible by **minimizing the error** (given the learning rate is small enough)
  - Adaline always converges; see what happens with XOR **since it stop when error reaches a specific bound.**
  - While Perceptron shall stop **when error is 0, which never happen, and consequently infinite loop occurred**.
- Perceptron's rule is guaranteed to converge to a solution that correctly categorizes the training patterns but the resulting network can be sensitive to noise patterns as patterns often lie close to the decision boundary.



- Since it learned the training set till reaching ZERO error, no when trying new strange test cases, usually not good accuracy is obtained.

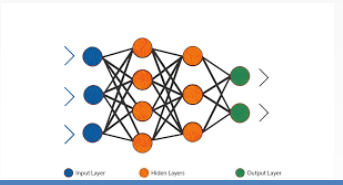


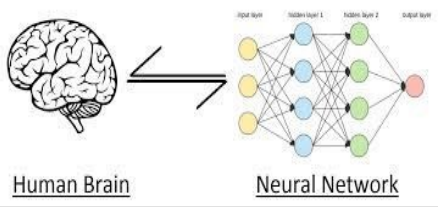
What's Next Lecture

Multilayer Perceptron (MLP) Network

with

Back-Propagation Algorithm





Thanks
for your attention…