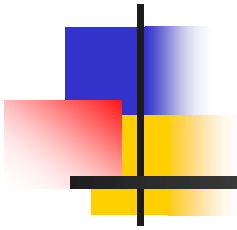
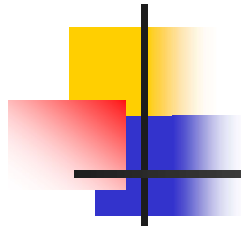


# Neural Networks



Some slides were adapted/taken from various sources, including Prof. Andrew Ng's Coursera Lectures, Stanford University, Prof. Kilian Q. Weinberger's lectures on Machine Learning, Cornell University, Prof. Sudeshna Sarkar's Lecture on Machine Learning, IIT Kharagpur, Prof. Bing Liu's lecture, University of Illinois at Chicago (UIC), CS231n: Convolutional Neural Networks for Visual Recognition lectures, Stanford University and many more. We thankfully acknowledge them. Students are requested to use this material for their study only and **NOT** to distribute it.

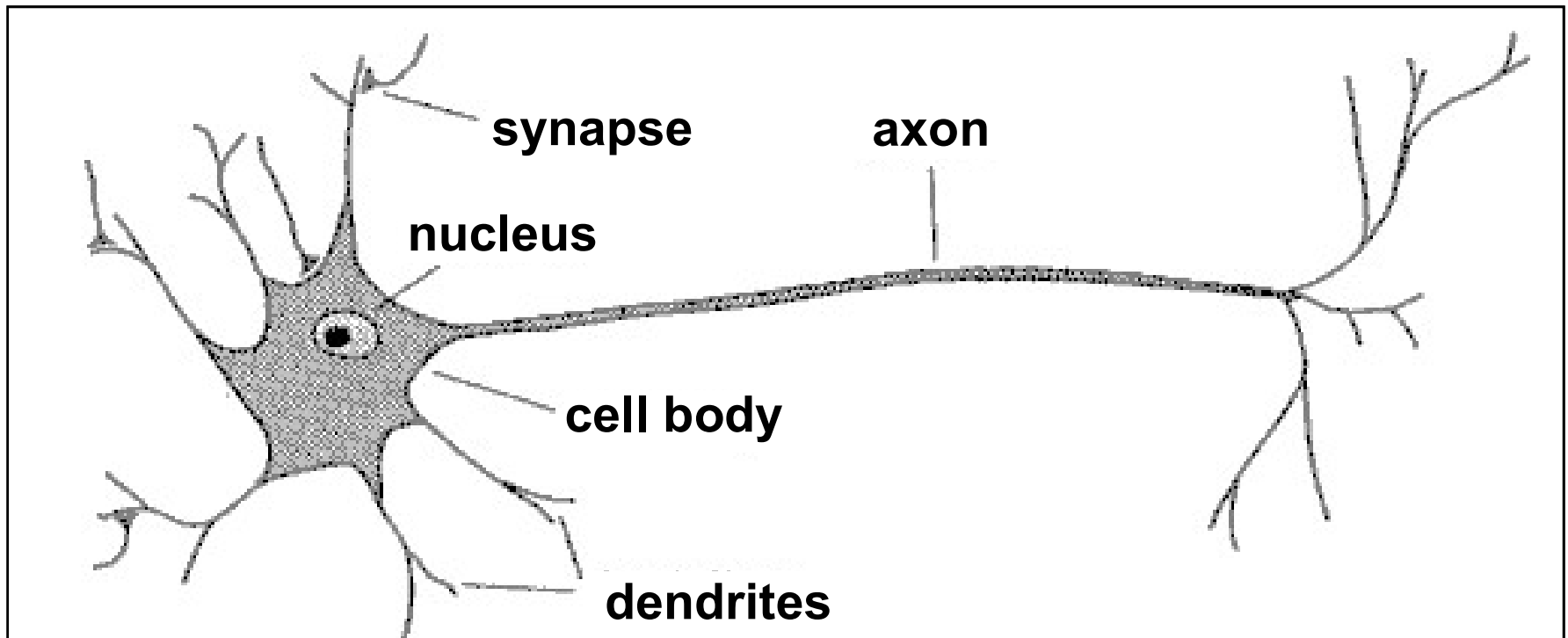


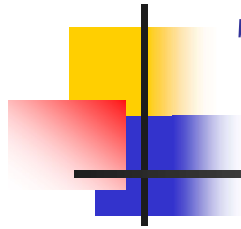
# Outline

---

- The Brain
- Perceptrons
- Gradient descent
- Multi-layer networks
- Backpropagation

# The Structure of Neurons





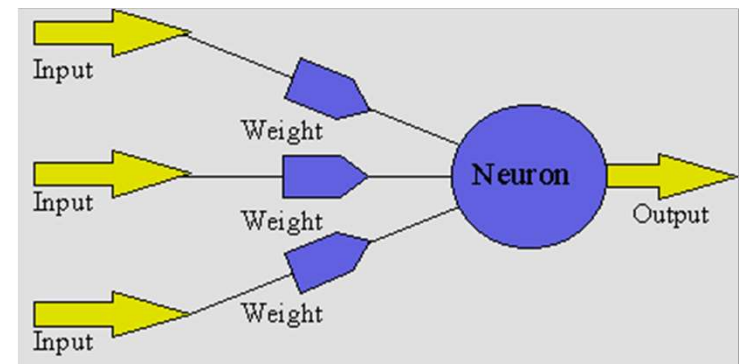
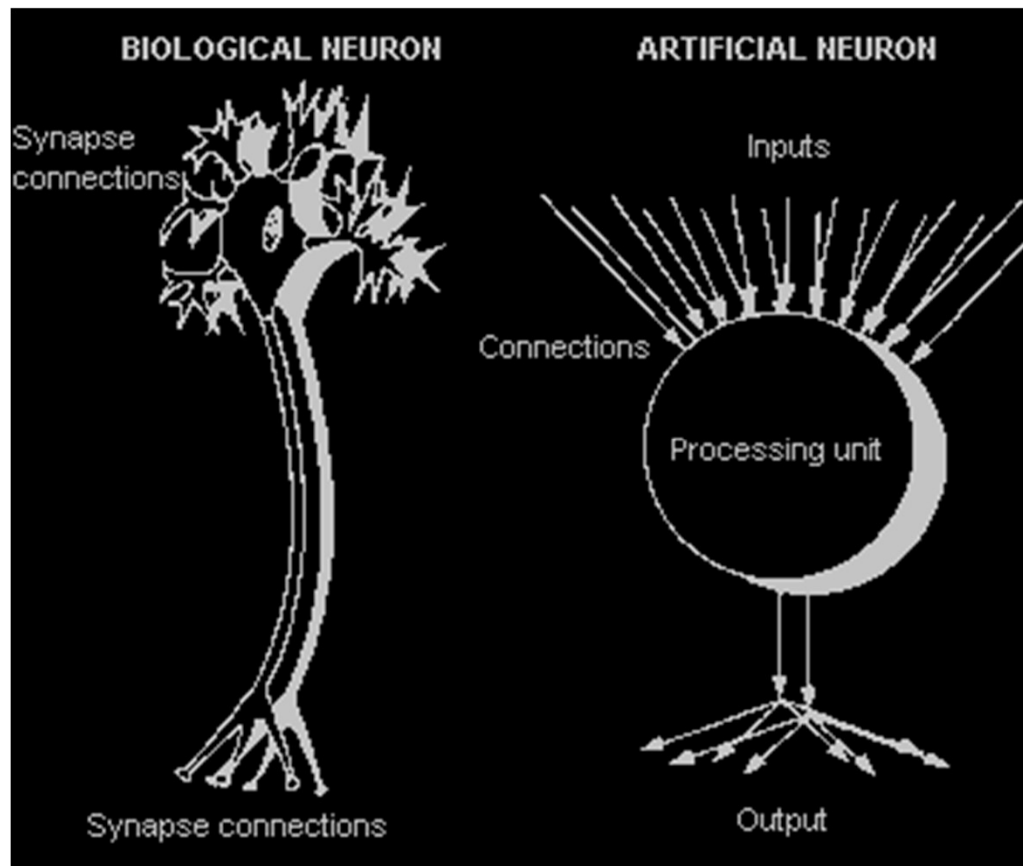
# The Structure of Neurons

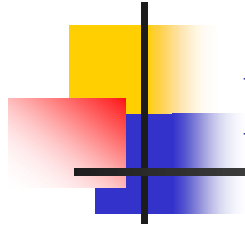
---

A neuron has a cell body, a branching **i**input structure (the dendr**I**te) and a branching **o**output structure (the ax**O**n)

- Axons connect to dendrites via synapses.
- Electro-chemical signals are propagated from the dendritic input, through the cell body, and down the axon to other neurons

# Properties of Artificial Neural Nets (ANNs)





## Properties of Artificial Neural Nets (ANNs)

---

- Many simple neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed processing
- Learning by tuning the connection weights



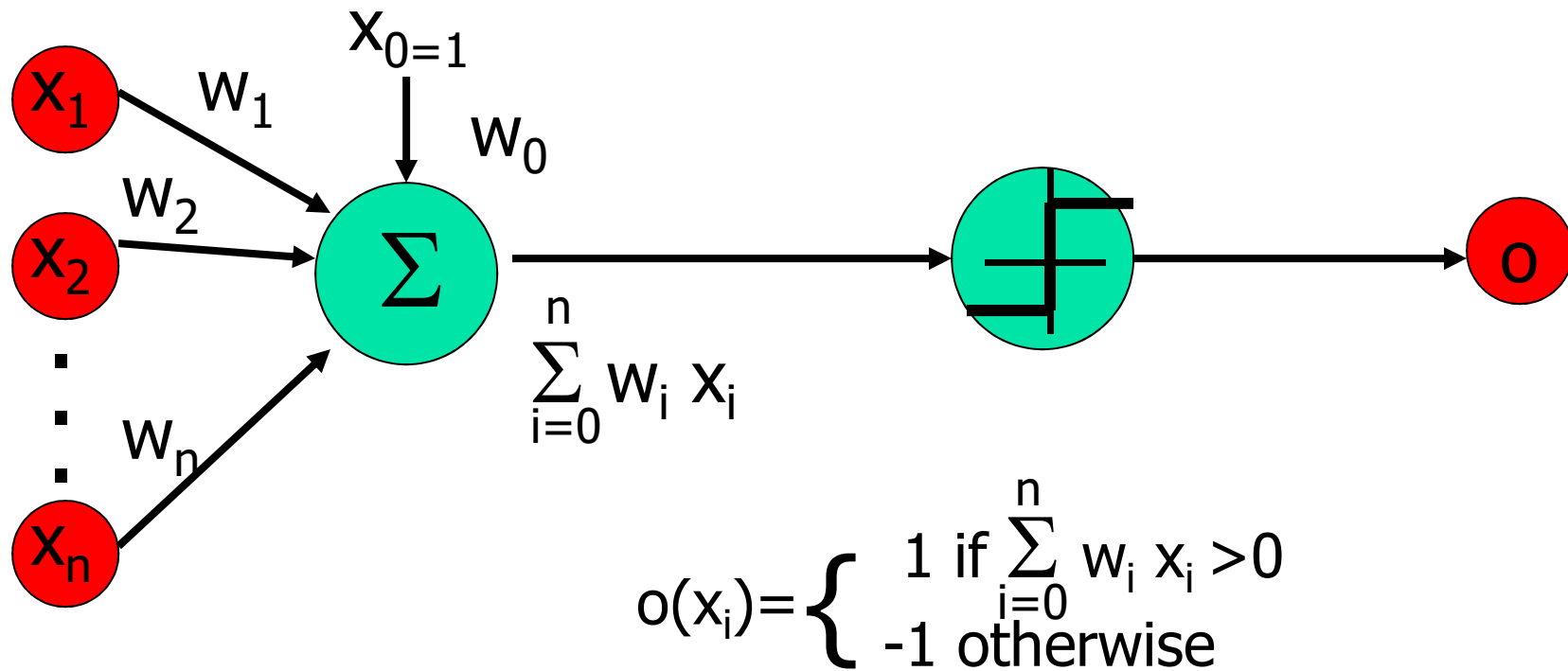
# Appropriate Problem Domains for Neural Network Learning

---

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Form of target function is unknown
- Humans do not need to interpret the results (black box model)

# Perceptron

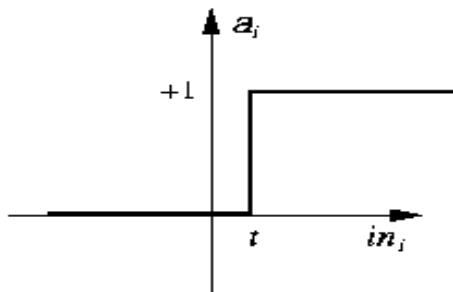
- Linear threshold unit (LTU)



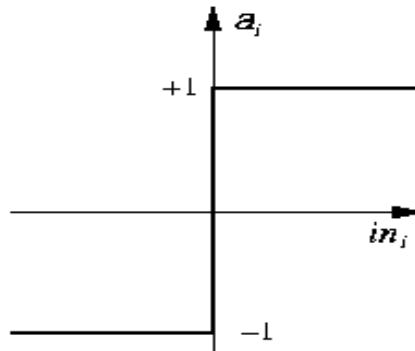


# Activation functions

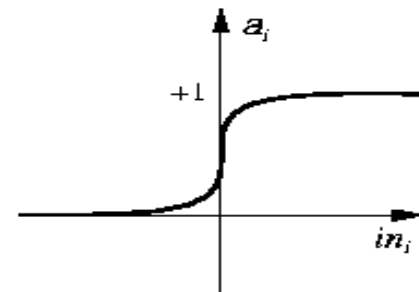
- Transforms neuron's input into output.
- Features of activation functions:
  - A squashing effect is required
    - Prevents accelerating growth of activation levels through the network.



(a) Step function



(b) Sign function



(c) Sigmoid function



# Standard activation functions

---

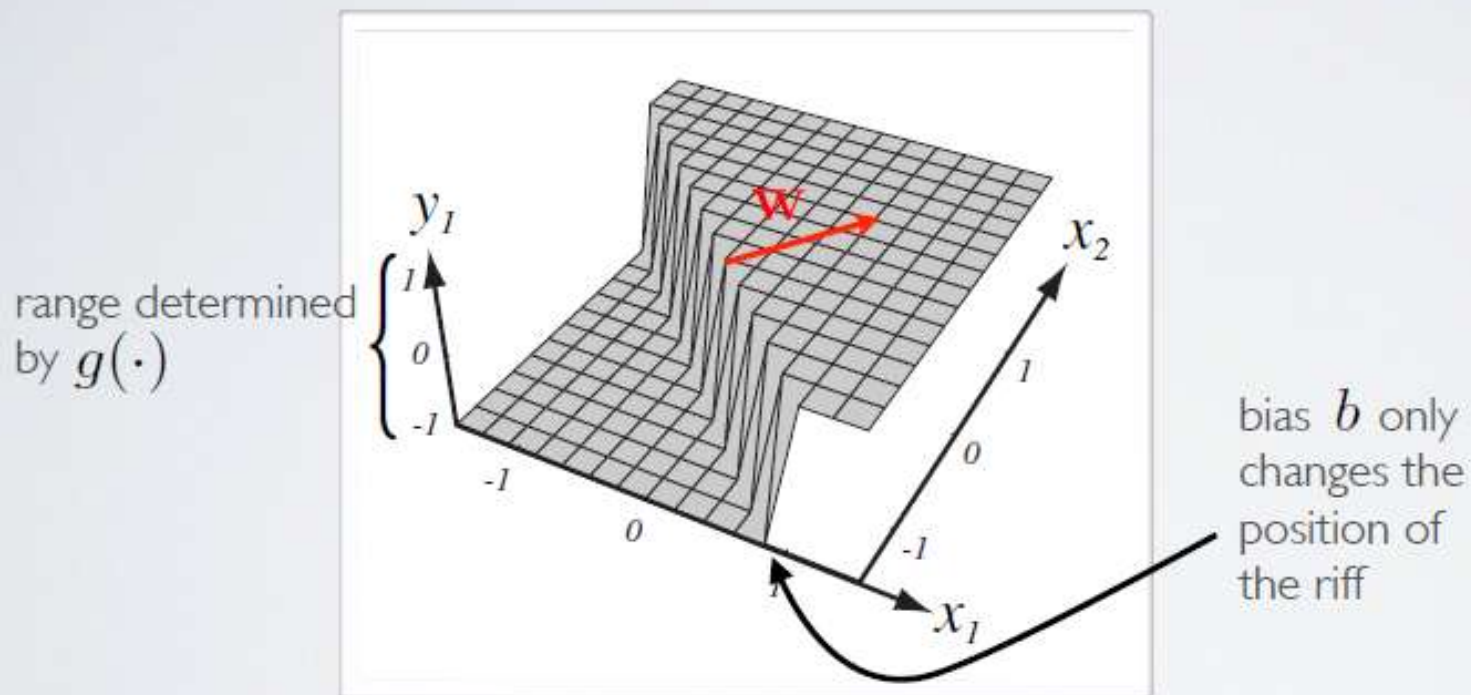
- The hard-limiting threshold function
  - Corresponds to the biological paradigm
    - either fires or not
- Sigmoid functions ('S'-shaped curves)
  - The logistic function
  - The hyperbolic tangent (symmetrical)
  - Both functions have a simple differential
  - Only the shape is important



$$\phi(x) = \frac{1}{1 + e^{-ax}}$$

# ARTIFICIAL NEURON

**Topics:** connection weights, bias, activation function



(from Pascal Vincent's slides)



# Perceptron Learning Rule

---

$$w_i = w_i + \Delta w_i$$

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

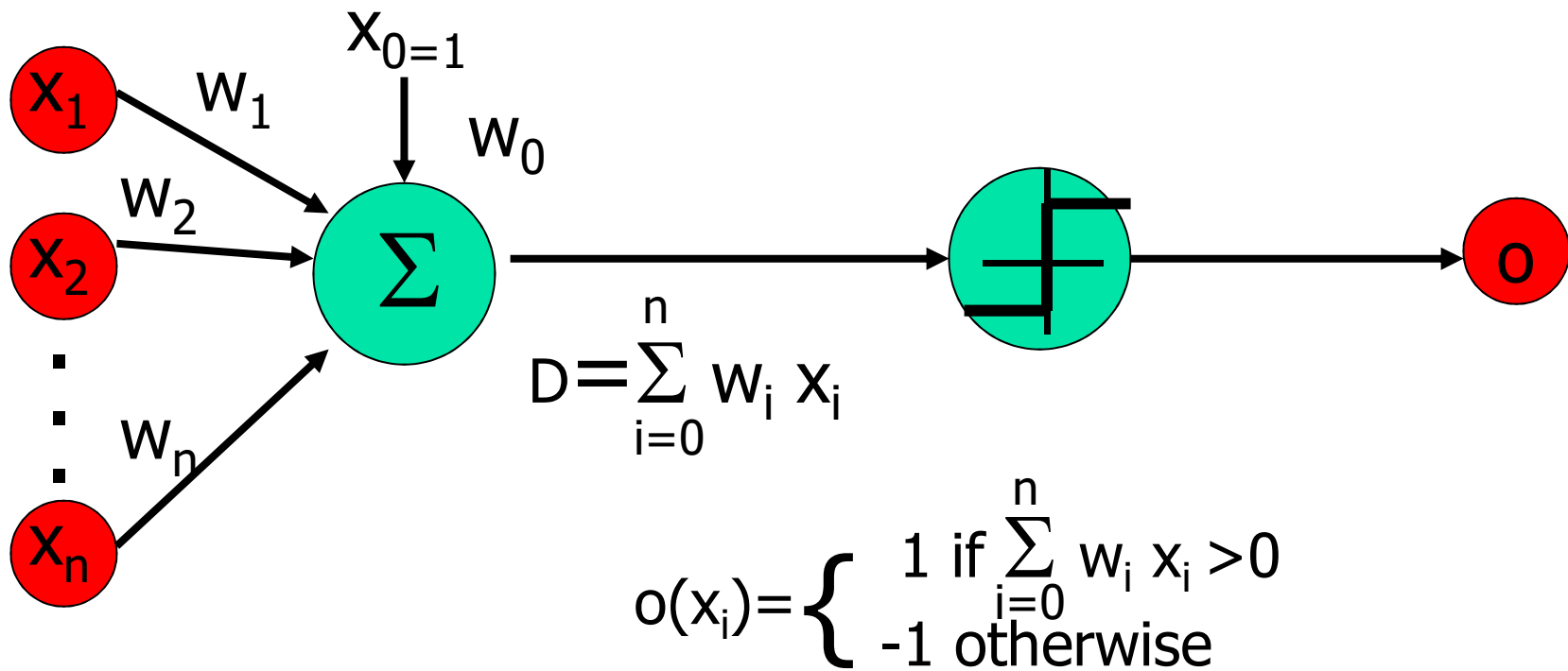
$t=c(x)$  is the target value

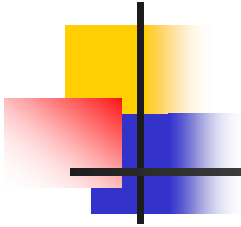
$o$  is the perceptron output

$\eta$  Is a small constant (e.g. 0.1) called *learning rate*

- If the output is correct ( $t=o$ ) the weights  $w_i$  are not changed
- If the output is incorrect ( $t \neq o$ ) the weights  $w_i$  are changed such that the output of the perceptron for the new weights is *closer* to  $t$ .
- The algorithm converges to the correct classification
  - if the training data is linearly separable
  - and  $\eta$  is sufficiently small

# Perceptron Learning Rule





D=predicted output, d = actual (target) output

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

Type equation here.

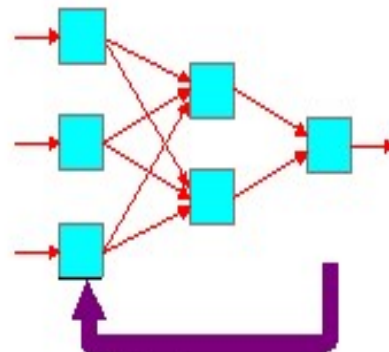


# Supervised Learning

This is a  
linear discriminant  
fn.

if the classes are linearly separable,  
using SLP, we can get a  
linear decision boundary. Every neuron of P  
actually gives an  
eqn of a st. line.

Input Data	Example Outputs



Training process

Results

$$y = \sum w_i x_i + b$$

$$y = mx + c$$

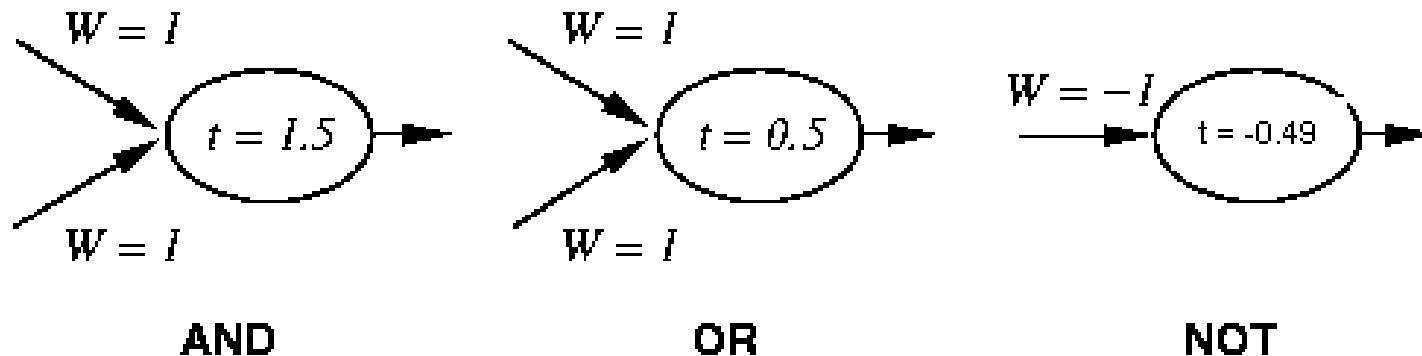
st. line eqn.

Decision boundary  
is a st. line

Sepal length	Sepal width	Petal length	Petal width	Class
5.1	3.5	1.4	0.2	0
4.9	3.0	1.4	0.2	2
4.7	3.2	1.3	0.2	0
4.6	3.1	1.5	0.2	1



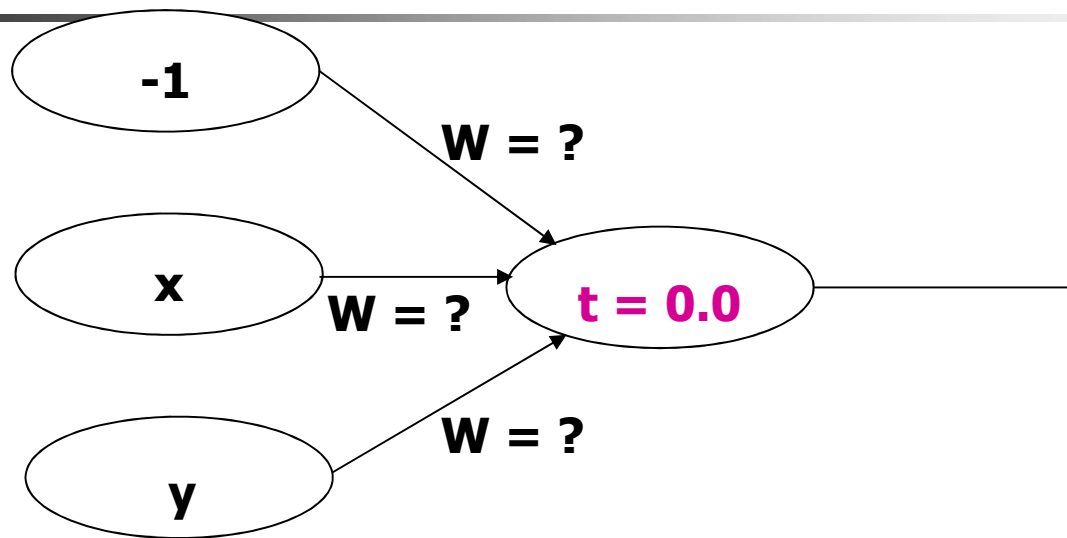
# Perceptron Training



$$\text{Output} = \begin{cases} 1 & \text{if } \sum_{i=0} w_i x_i > t \\ 0 & \text{otherwise} \end{cases}$$

- Linear threshold is used.
- $W$  - weight value
- $t$  - threshold value

# Training Perceptrons

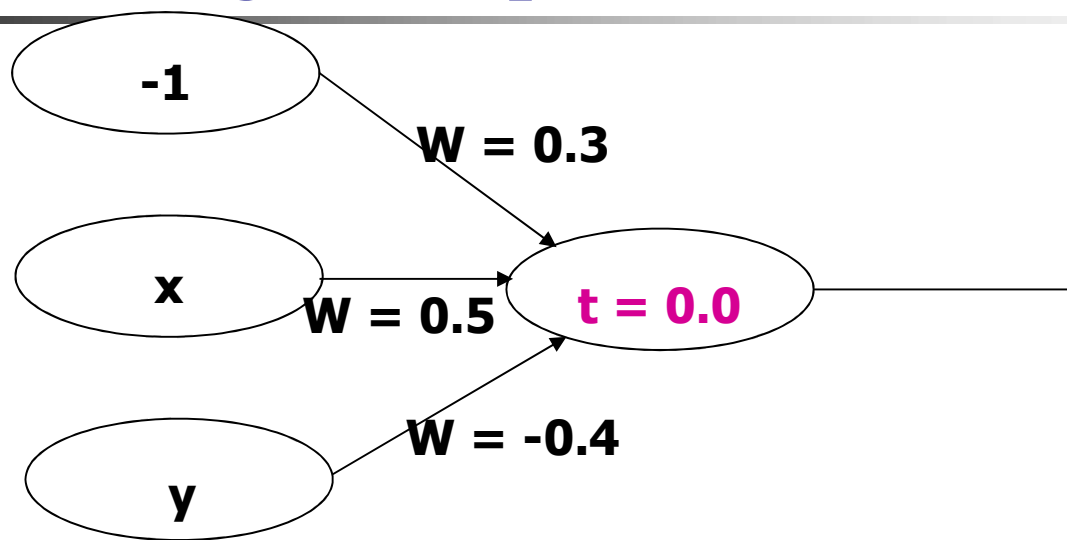


**For AND**

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

- What are the weight values?
- Initialize with random weight values

# Training Perceptrons



**For AND**

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

$I_1$	$I_2$	$I_3$	Summation	Output
-1	0	0	$(-1 \cdot 0.3) + (0 \cdot 0.5) + (0 \cdot -0.4) = -0.3$	0
-1	0	1	$(-1 \cdot 0.3) + (0 \cdot 0.5) + (1 \cdot -0.4) = -0.7$	0
-1	1	0	$(-1 \cdot 0.3) + (1 \cdot 0.5) + (0 \cdot -0.4) = 0.2$	1
-1	1	1	$(-1 \cdot 0.3) + (1 \cdot 0.5) + (1 \cdot -0.4) = -0.2$	0

# Simple network

## For AND

A B Output

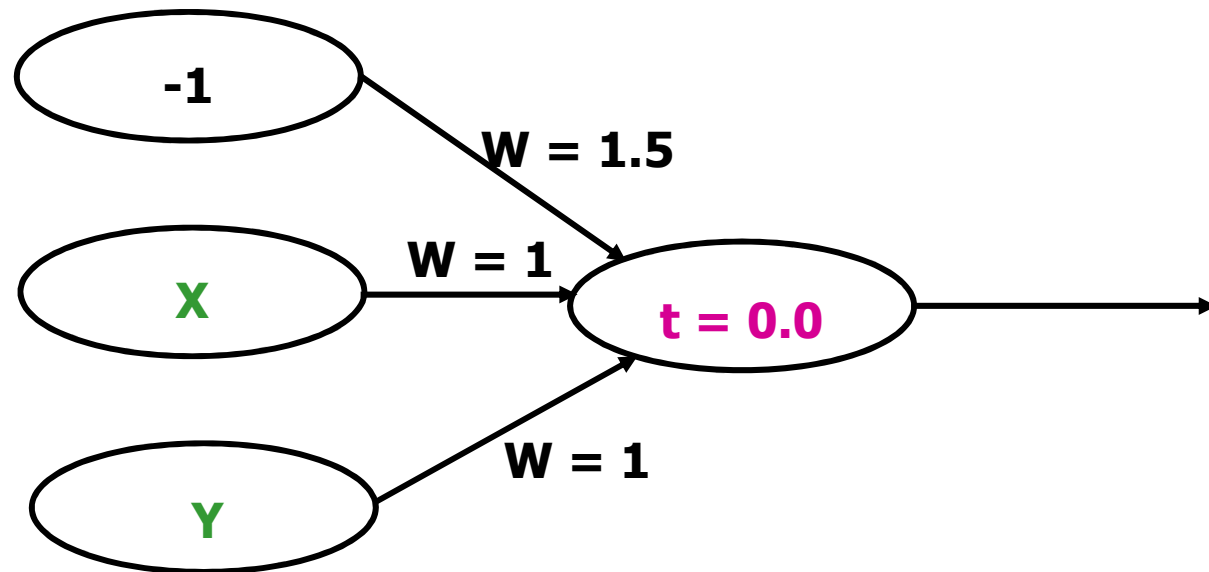
0 0 0

0 1 0

1 0 0

1 1 1

$$\text{output} = \begin{cases} 1 & \text{if } \sum_{i=0} w_i x_i > t \\ 0 & \text{otherwise} \end{cases}$$







# Learning algorithm

---

Epoch : Presentation of the entire training set to the neural network.

In the case of the AND function an epoch consists of four sets of inputs being presented to the network (i.e.  $[0,0]$ ,  $[0,1]$ ,  $[1,0]$ ,  $[1,1]$ )

Error: The error value is the amount by which the value output by the network differs from the target value. For example, if we required the network to output 0 and it output a 1, then  $\text{Error} = -1$



# Learning algorithm

---

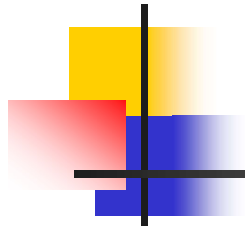
**Target Value, T** : When we are training a network we not only present it with the input but also with a value that we require the network to produce. For example, if we present the network with [1,1] for the AND function the training value will be 1

**Output , O** : The output value from the neuron

**I<sub>i</sub>** : Inputs being presented to the neuron

**W<sub>i</sub>** : Weight from input neuron (I<sub>j</sub>) to the output neuron

**LR** : The learning rate. This dictates how quickly the network converges. It is set by a matter of experimentation. It is typically 0.1

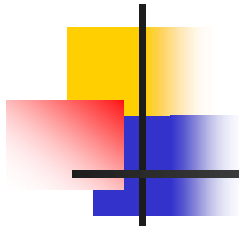


# Decision boundaries

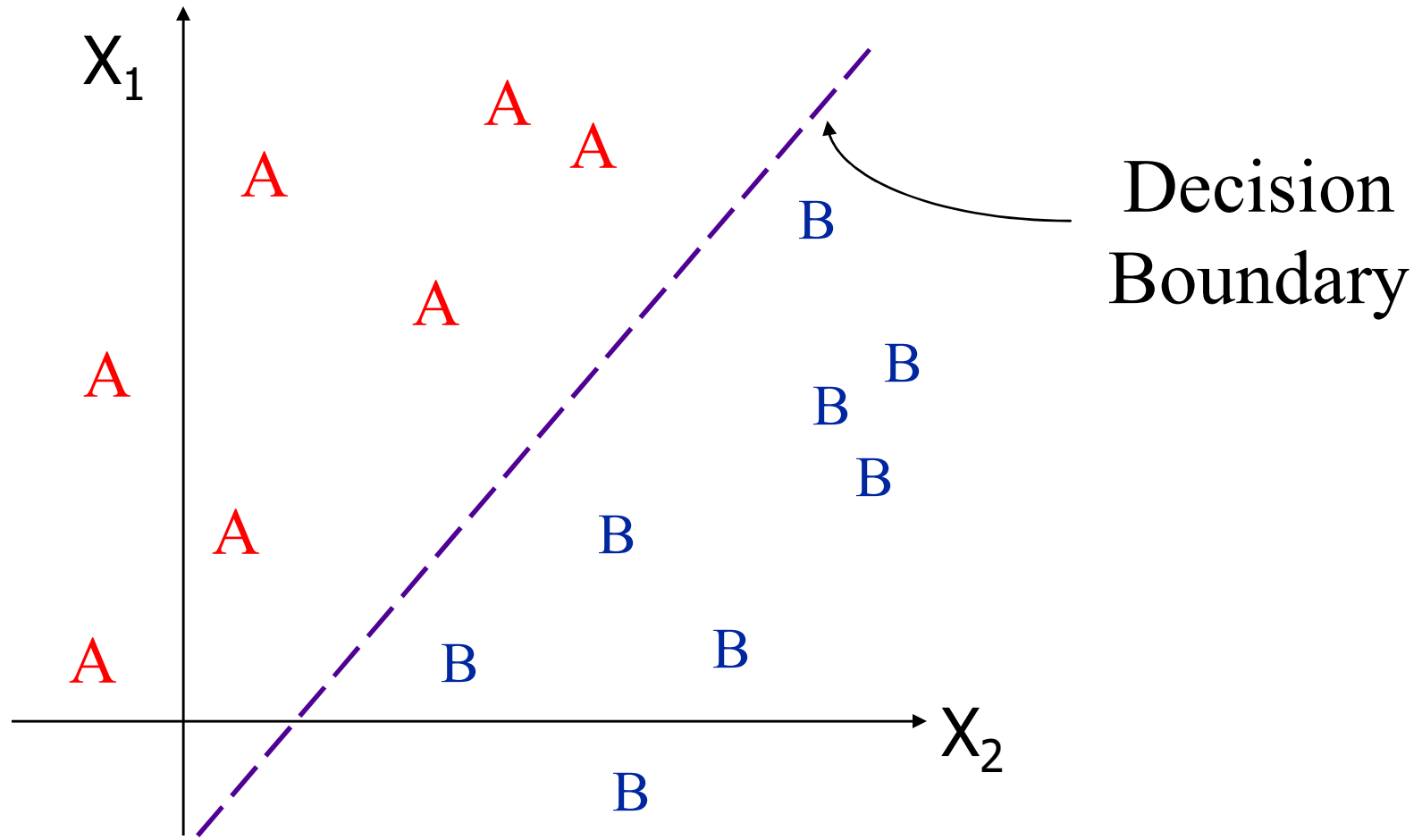
---

- In simple cases, divide feature space by drawing a hyperplane across it.
- Known as a **decision boundary**.
- **Discriminant function**: returns different values on opposite sides. (straight line)
- Problems which can be thus classified are **linearly separable**.



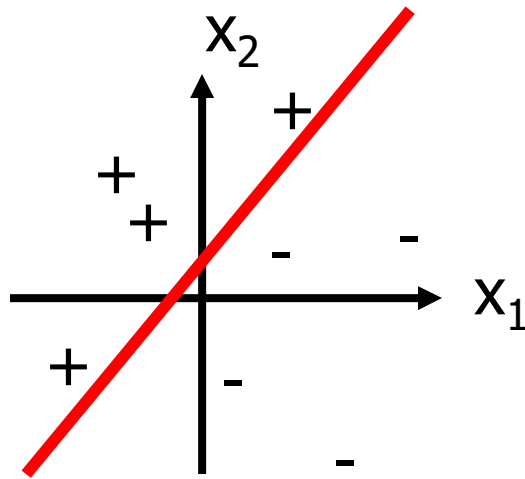


# Linear Separability

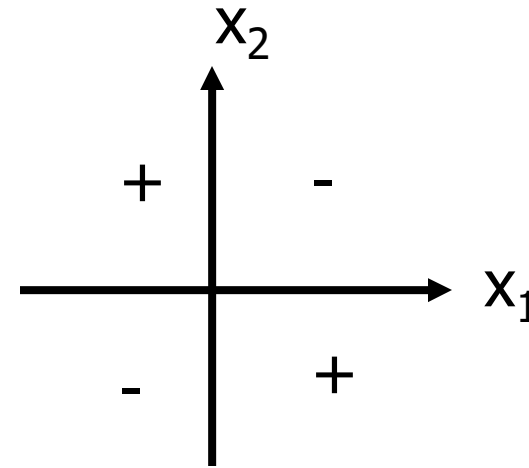




# Decision Surface of a Perceptron



Linearly separable



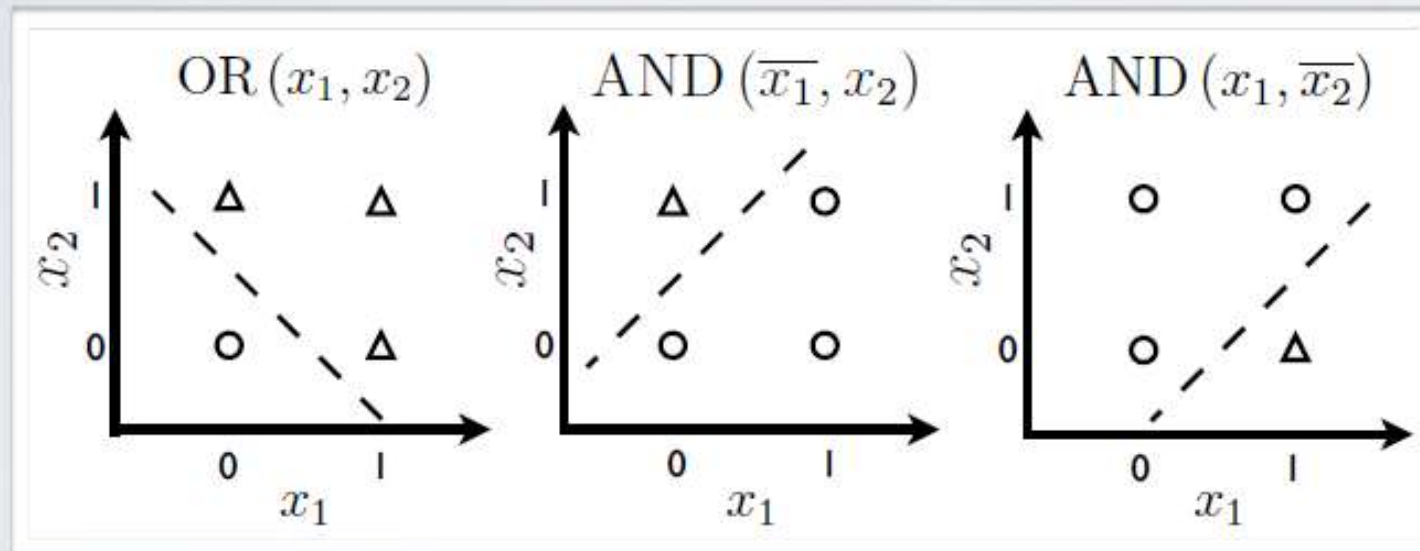
Non-Linearly separable

- Perceptron is able to represent some useful functions
- $\text{AND}(x_1, x_2)$  choose weights  $w_0 = -1.5$ ,  $w_1 = 1$ ,  $w_2 = 1$
- But functions that are not linearly separable (e.g. XOR) are not representable

# ARTIFICIAL NEURON

**Topics:** capacity of single neuron

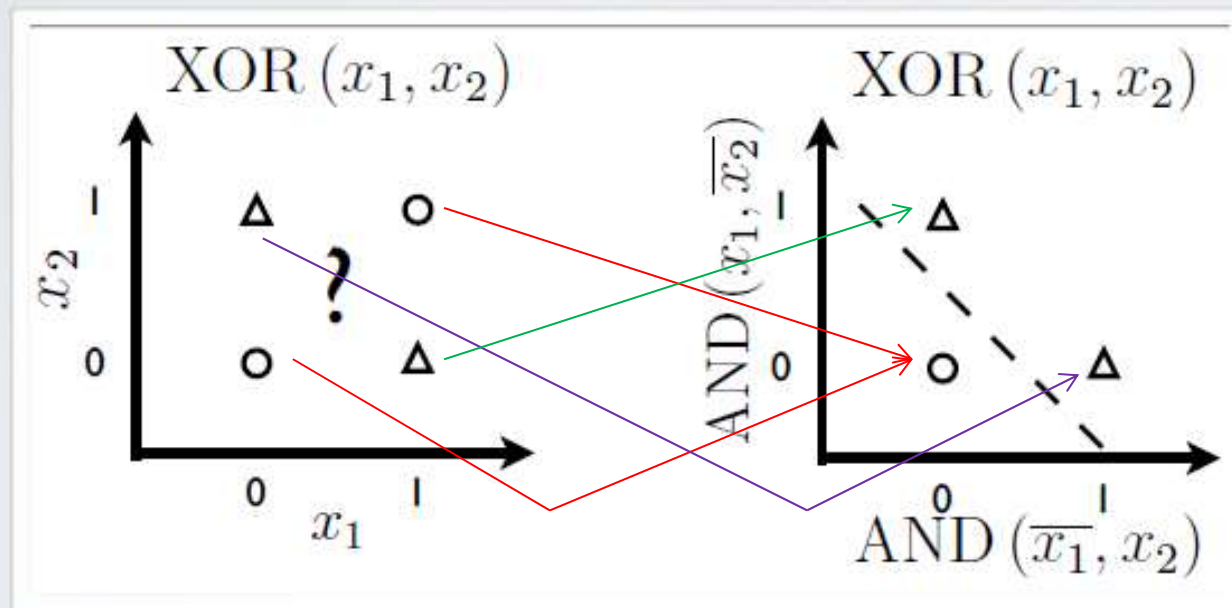
- Can solve linearly separable problems



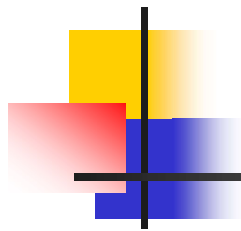
# ARTIFICIAL NEURON

**Topics:** capacity of single neuron

- Can't solve non linearly separable problems...



- ... unless the input is transformed in a better representation

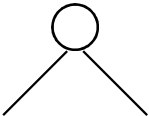
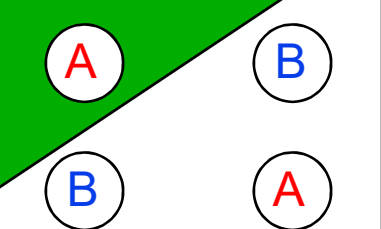
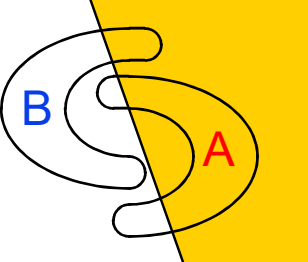
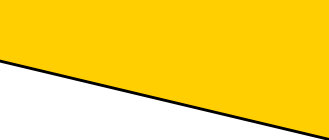
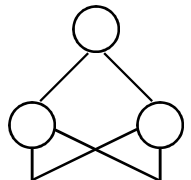
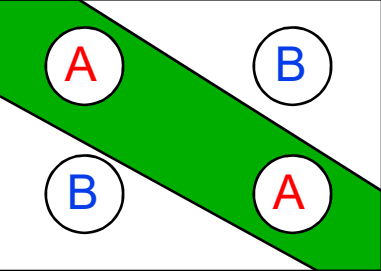
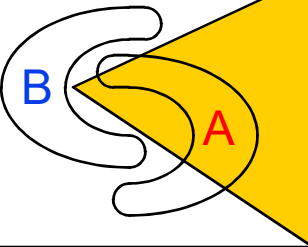
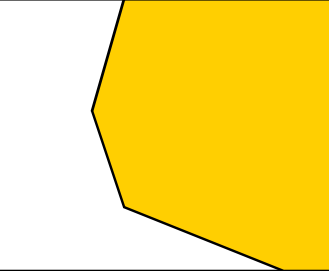
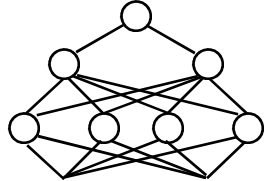
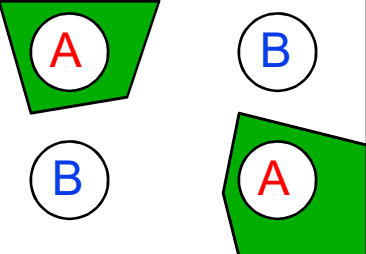
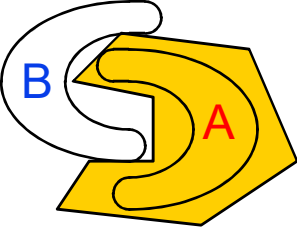
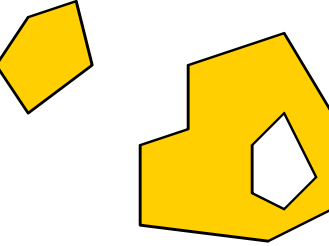


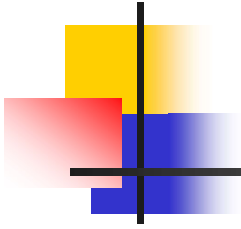
# Hyperplane partitions

---

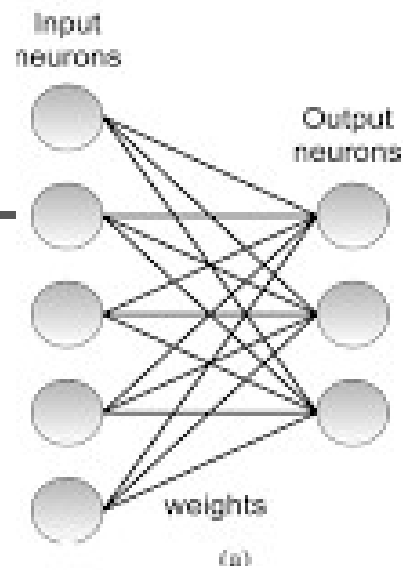
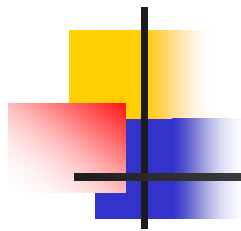
- An extra layer models a convex hull
  - “An area with no dents in it”
  - Perceptron models, but can’t learn
  - Sigmoid function learning of convex hulls
  - Two layers add convex hulls together
  - Sufficient to classify anything “sane”.
- In theory, further layers add nothing
- In practice, extra layers may be better

# Different Non-Linearly Separable Problems

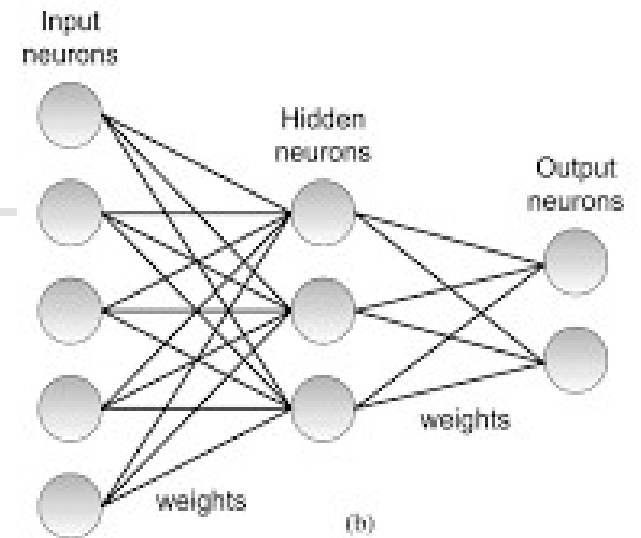
Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
Single-Layer 	Half Plane Bounded By Hyperplane			
Two-Layer 	Convex Open Or Closed Regions			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			



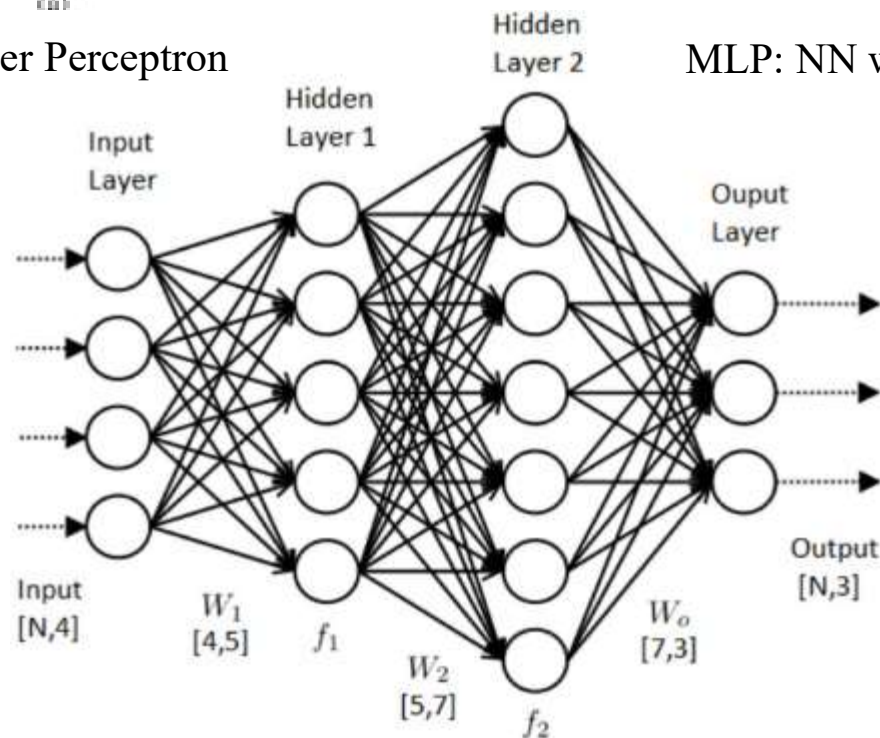
# Multi Layer Perceptron



SLP: Single Layer Perceptron

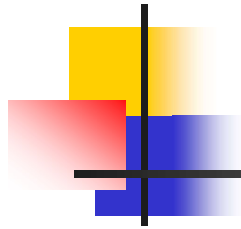


MLP: NN with single hidden layer



MLP: NN with two hidden layer

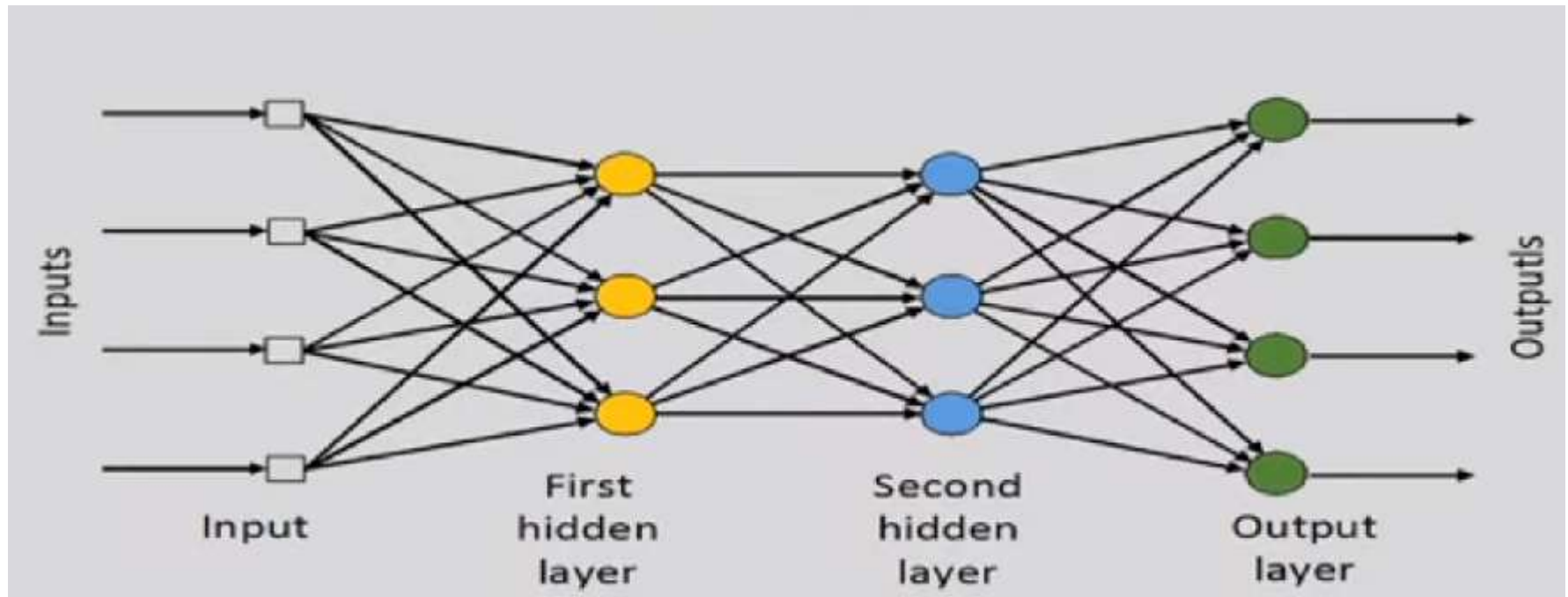
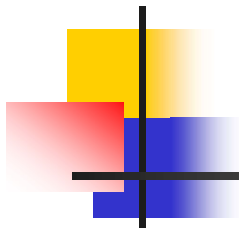


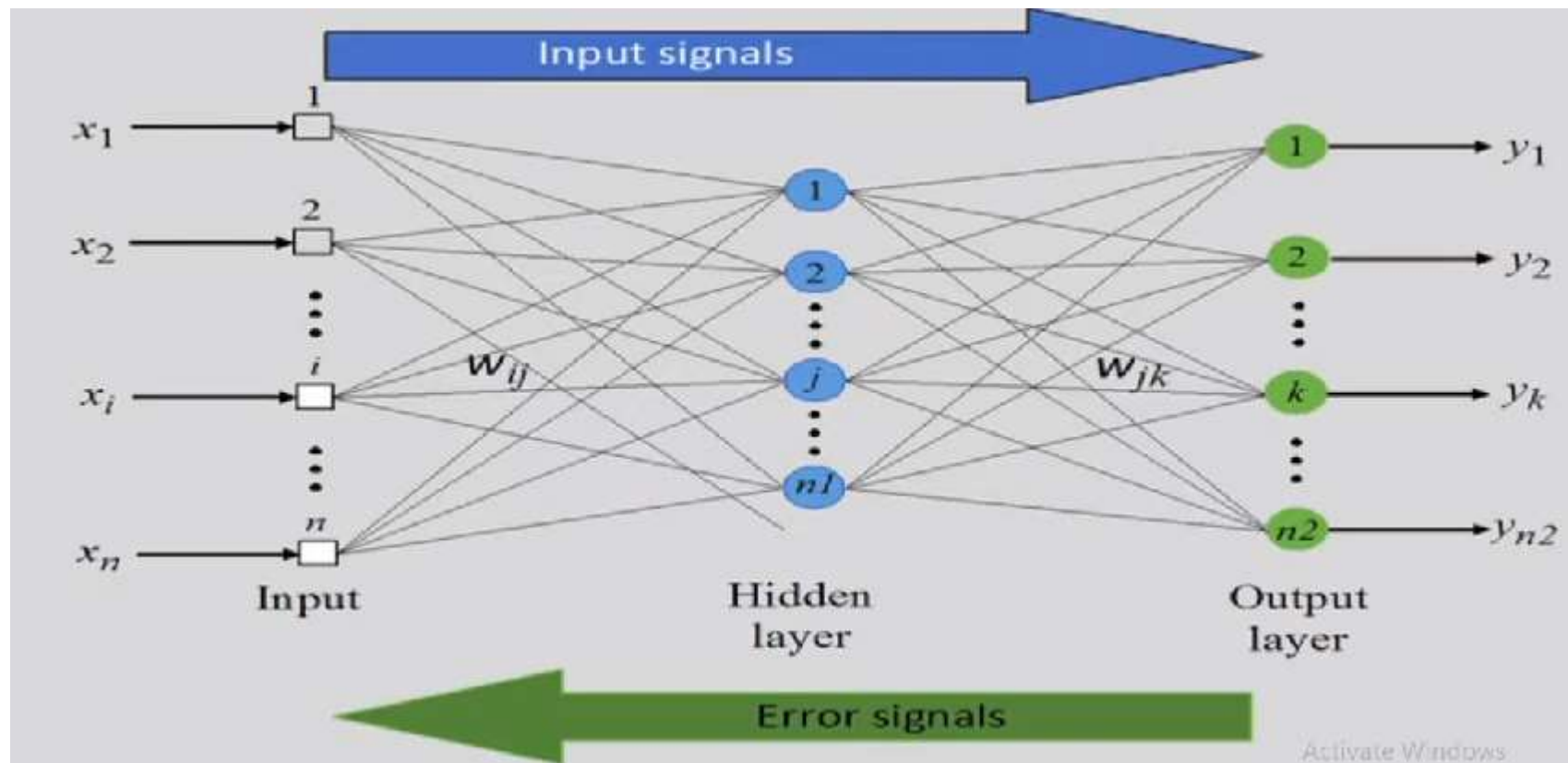
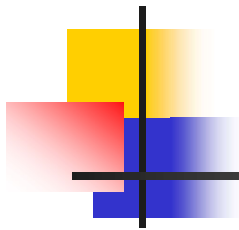


# Representation of the Neural Network

---

- Single layer nets have limited representation power (linear separability problem). Multi layer nets of (or nets with non linear hidden unit) may overcome linear inseparability problem.
- Every Boolean function can be realized by a network with single hidden layer.
- Every bounded continuous function can be approximated with arbitrary small error, by network with one hidden layer.
- Any function can be approximated to arbitrary accuracy by a neural network having two hidden layer.





# Multi Layer Perceptron (MLP)

**Topics:** single hidden layer neural network

- Hidden layer pre-activation:

$$\mathbf{a}(\mathbf{x}) = \mathbf{b}^{(1)} + \mathbf{W}^{(1)}\mathbf{x}$$

$$(a(\mathbf{x})_i = b_i^{(1)} + \sum_j W_{i,j}^{(1)} x_j)$$

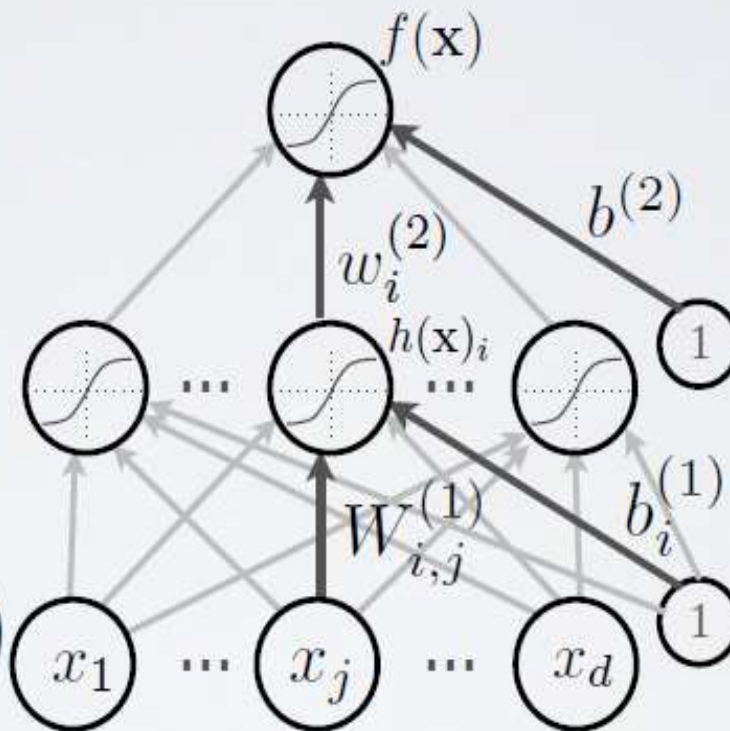
- Hidden layer activation:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{a}(\mathbf{x}))$$

- Output layer activation:

$$f(\mathbf{x}) = o\left(b^{(2)} + \mathbf{w}^{(2)\top} \mathbf{h}^{(1)}\mathbf{x}\right)$$

output activation function





# Multi Layer Perceptron (MLP)

**Topics:** softmax activation function

- For multi-class classification:
  - we need multiple outputs (1 output per class)
  - we would like to estimate the conditional probability  $p(y = c|\mathbf{x})$
- We use the softmax activation function at the output:
$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[ \frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^\top$$
  - strictly positive
  - sums to one
- Predicted class is the one with highest estimated probability

# Multi Layer Perceptron (MLP)

**Topics:** multilayer neural network

- Could have  $L$  hidden layers:

- layer pre-activation for  $k > 0$  ( $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$ )

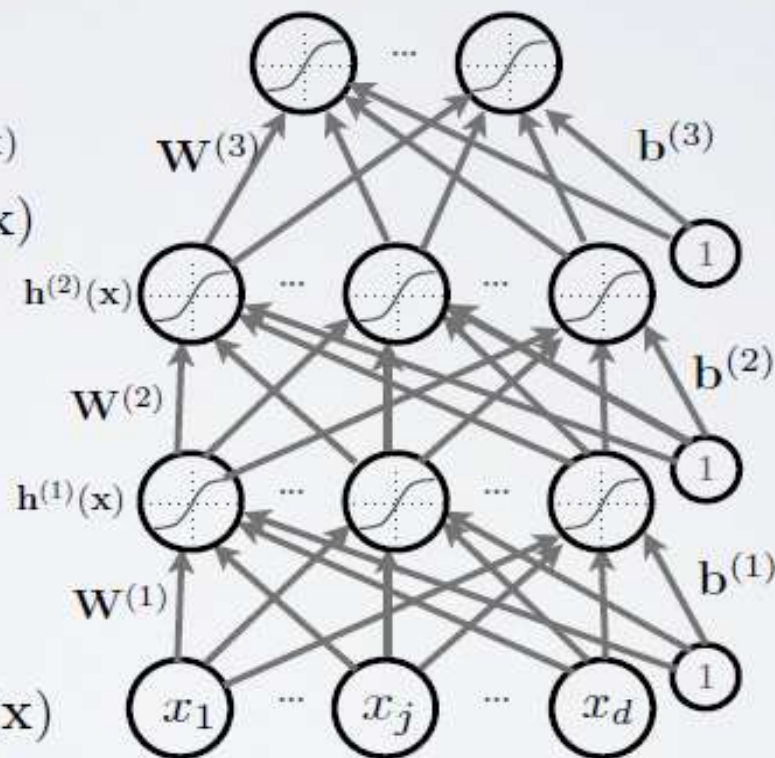
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation ( $k$  from 1 to  $L$ ):

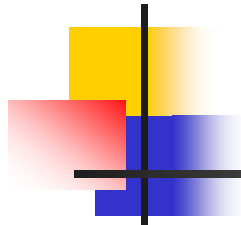
$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- output layer activation ( $k = L + 1$ ):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$







# Types of Layers

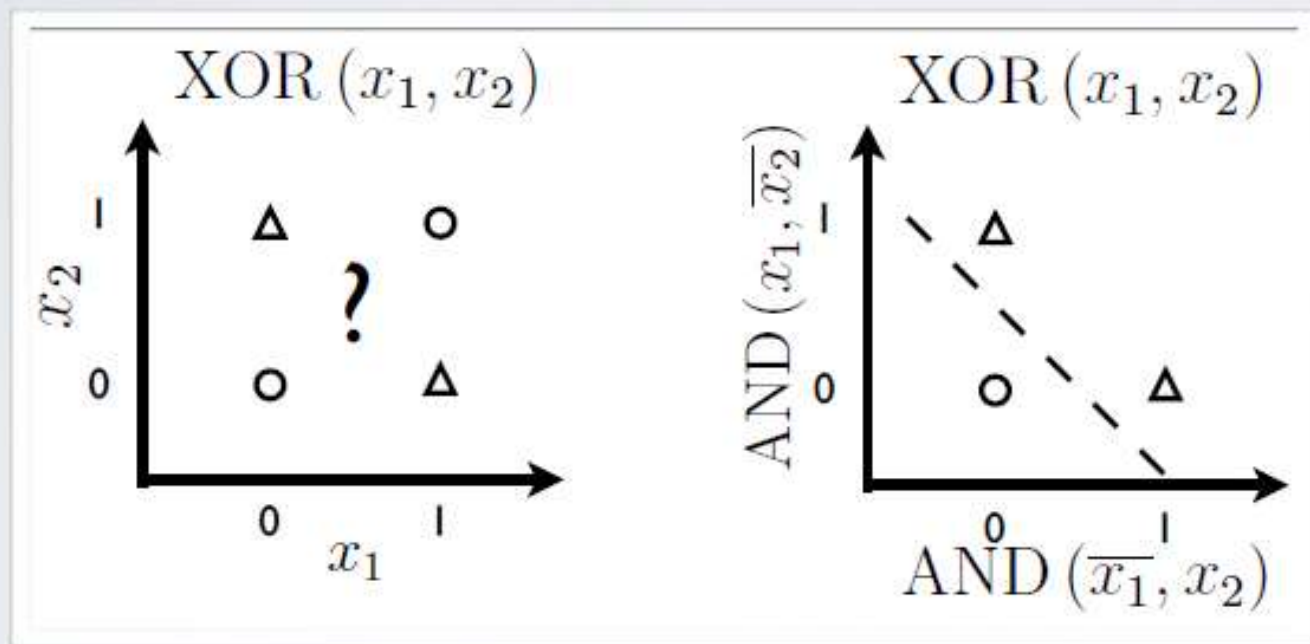
---

- The input layer.
  - Introduces input values into the network.
  - No activation function or other processing.
- The hidden layer(s).
  - Perform classification of features
  - Two hidden layers are sufficient to solve any problem
  - Features imply more layers may be better
- The output layer.
  - Functionally just like the hidden layers
  - Outputs are passed on to the world outside the neural network.

# Capacity of MLP

**Topics:** capacity of single neuron

- Can't solve non linearly separable problems...



- ... unless the input is transformed in a better representation



# Capacity of MLP

**Topics:** multilayer neural network

- Could have  $L$  hidden layers:

- layer pre-activation for  $k > 0$  ( $\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x}$ )

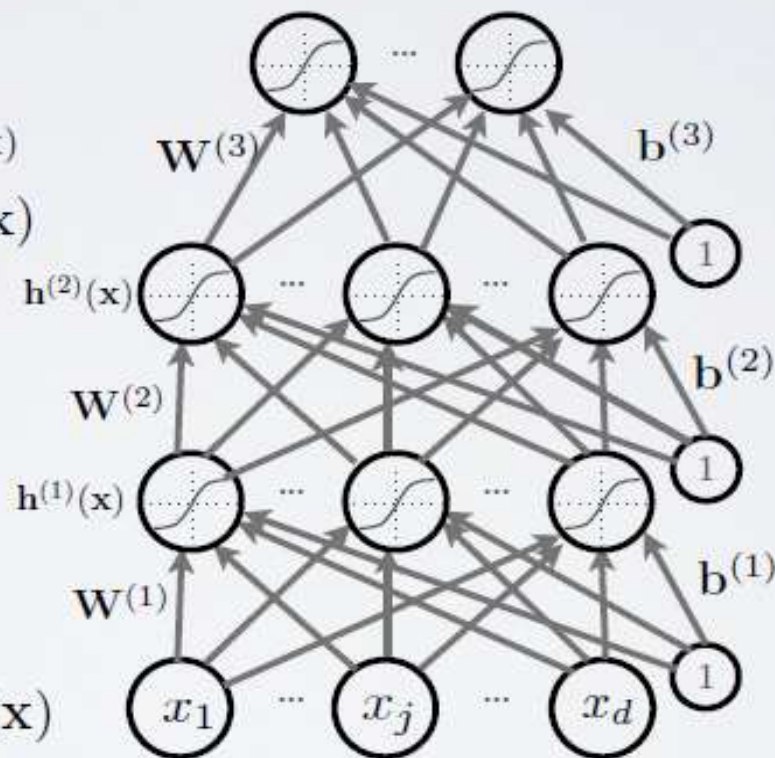
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}(\mathbf{x})$$

- hidden layer activation ( $k$  from 1 to  $L$ ):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

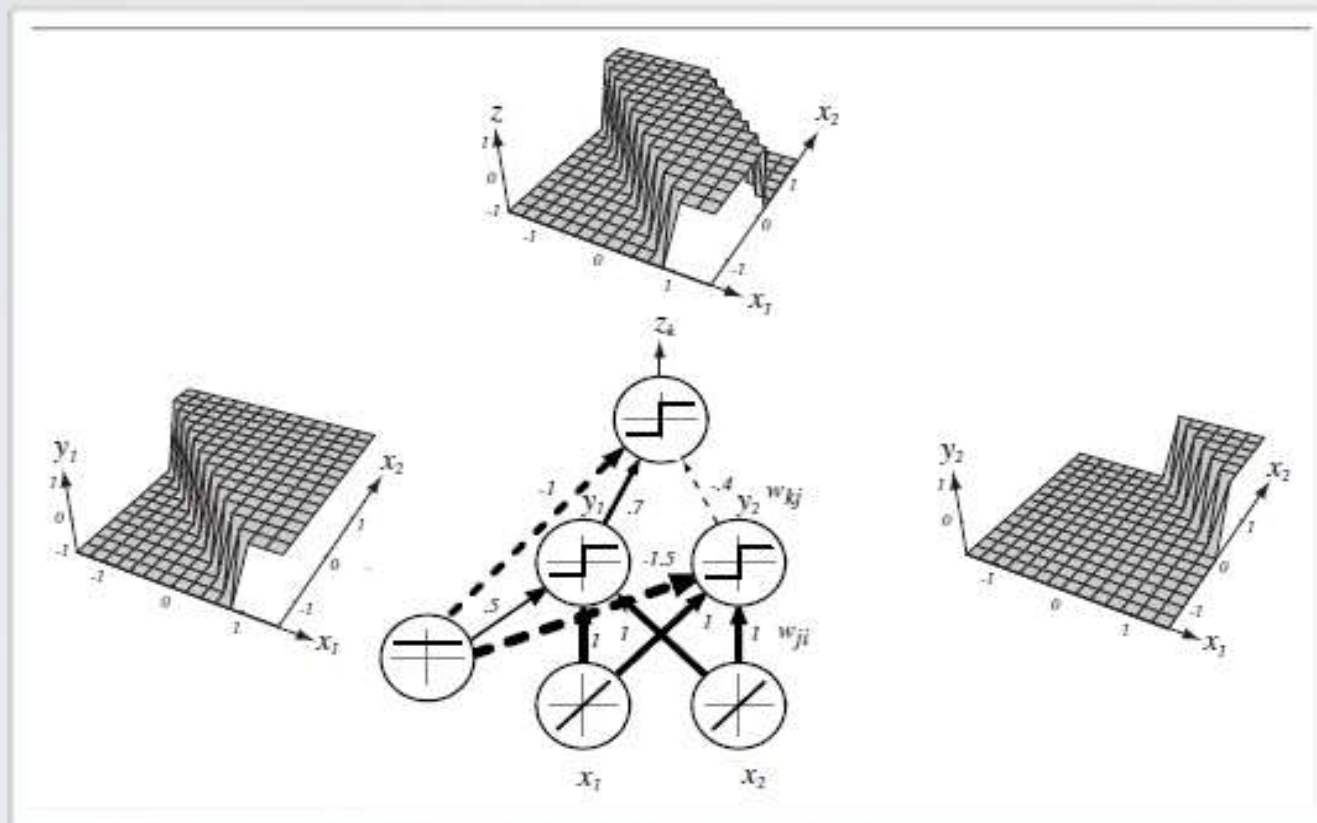
- output layer activation ( $k = L + 1$ ):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



# Capacity of MLP

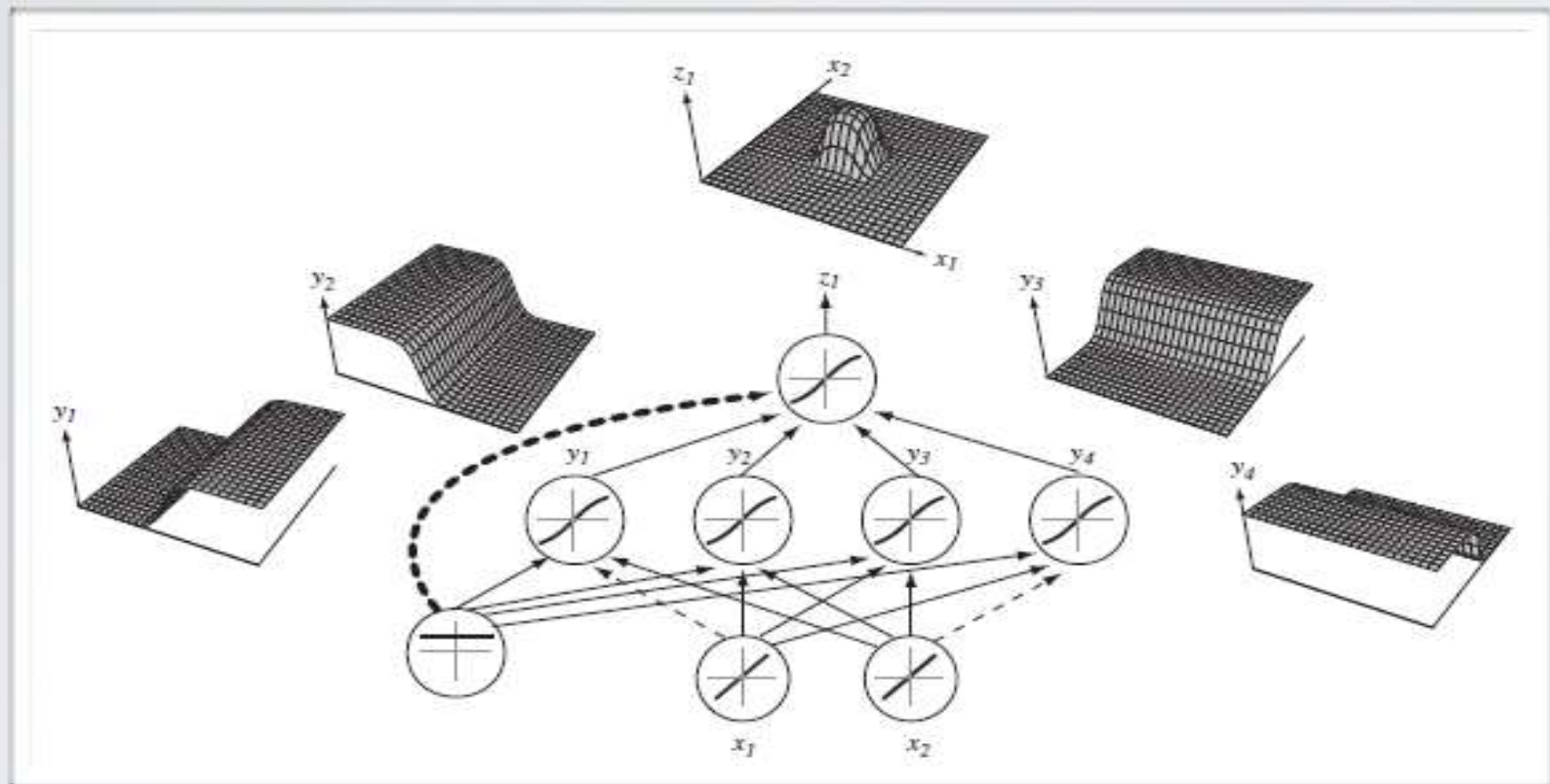
**Topics:** single hidden layer neural network



(from Pascal Vincent's slides)

# Capacity of MLP

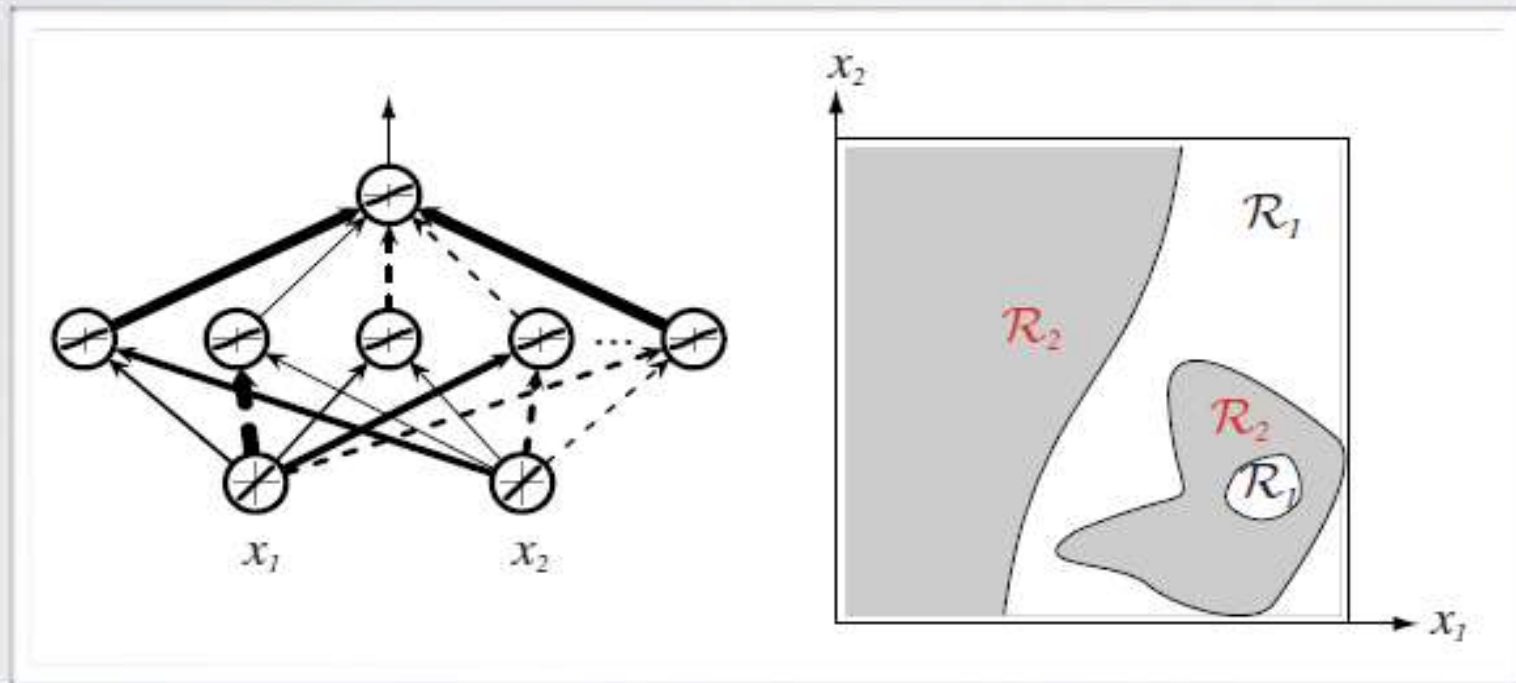
**Topics:** single hidden layer neural network



(from Pascal Vincent's slides)

# Capacity of MLP

**Topics:** single hidden layer neural network



(from Pascal Vincent's slides)

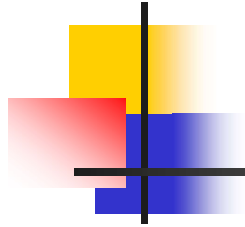


# Capacity of MLP

---

## **Topics:** universal approximation

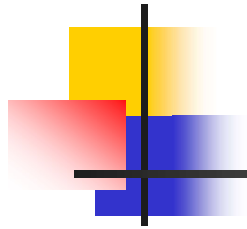
- Universal approximation theorem (Hornik, 1991):
  - “a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units”
- The result applies for sigmoid, tanh and many other hidden layer activation functions
- This is a good result, but it doesn't mean there is a learning algorithm that can find the necessary parameter values!



# Training Algorithms

---

- Adjust neural network weights to map inputs to outputs.
- Use a set of sample patterns where the desired output (given the inputs presented) is known.
- The purpose is to learn to generalize
  - Recognize features which are common to good and bad exemplars

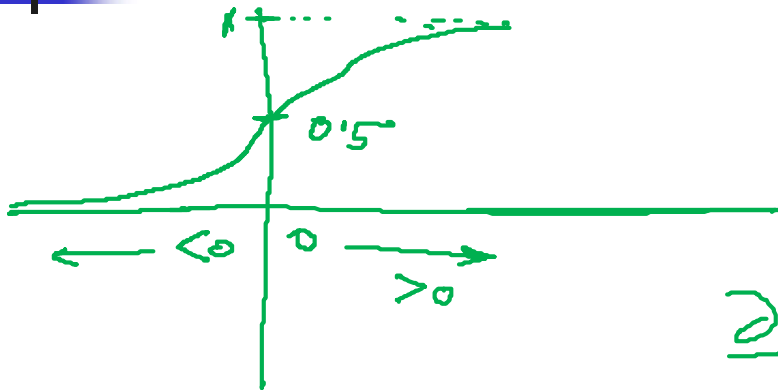


# Back-Propagation

---

- A training procedure which allows multi-layer feedforward Neural Networks to be trained;
- Can theoretically perform “any” input-output mapping;
- Can learn to solve linearly inseparable problems.

Sigmoid  $f_{\sigma}$

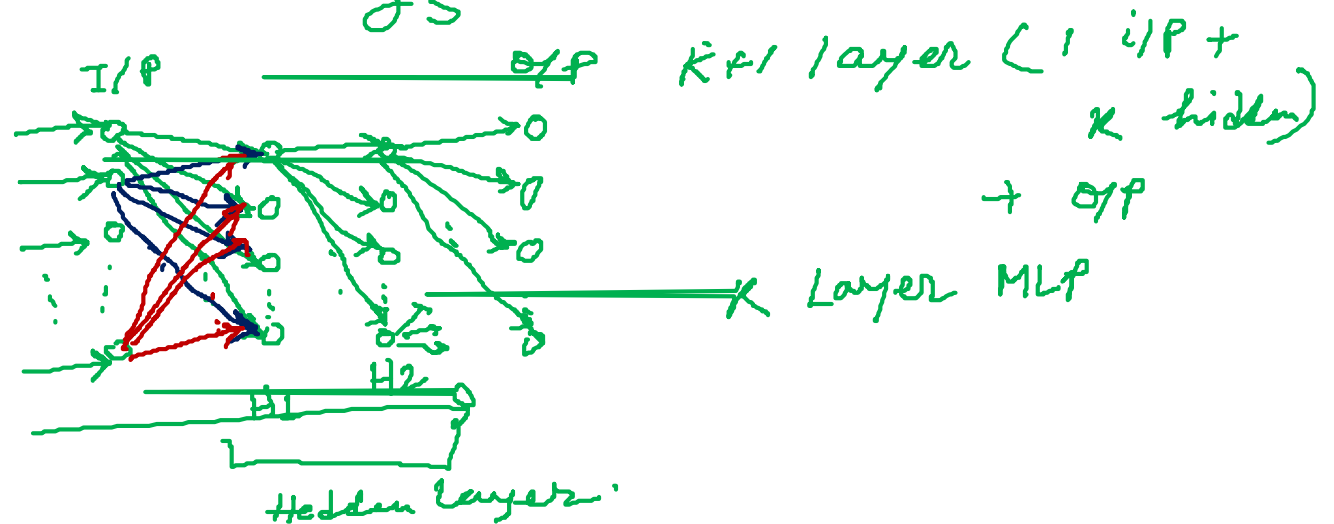


$$R(s) = \frac{1}{1 + e^{-s}}$$

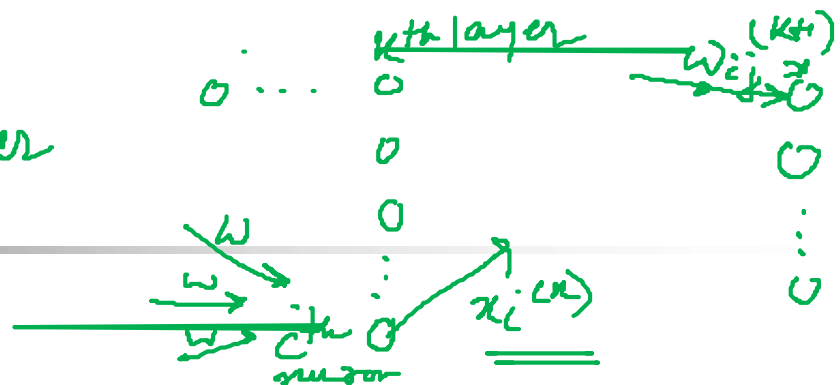
→ Differentiable

$$\frac{\partial R(s)}{\partial s} = R(s) [1 - R(s)]$$

MLP:-





$i^{\text{th}}$  node  $k^{\text{th}}$  layer

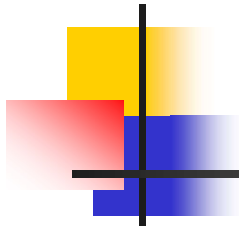
$x_i^{(k)}$  = o/p of  $i^{\text{th}}$  neuron in the  $k^{\text{th}}$  layer

$$x_j^{(k+1)} = \sum_{i=0}^{M_K} w_{ij}^{(k+1)} x_i^{(k)}$$

$M_K = \text{no. of node in the } k^{\text{th}} \text{ hidden layer}$

$$E = \frac{1}{2} \sum_{j=0}^{MK} [x_j^{(K)} - d_j^{(K)}]^2$$

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = \left[ x_j^{(k)} - d_j^{(k)} \right] \frac{\partial \kappa_j^{(k)}}{\partial w_{ij}^{(k)}} \quad (?)$$



$R = \text{Sigmoidal}$

$$\frac{\partial x_j^{(n)}}{\partial w_{ij}^{(k)}}$$

$$= \frac{\partial}{\partial w_{ij}^{(k)}} R \left( \sum_{i=0}^{M_k-1} w_{ij}^{(k)} x_i^{(k-1)} \right)$$

$$= x_j^{(k)} (1 - x_j^{(k)}) x_i^{(k-1)}$$

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = (x_j^{(k)} - d_j^{(k)}) x_j^{(k)} (1 - x_j^{(k)}) x_i^{(k-1)}$$

$$w_{ij}^{(k)}(n+1) = w_{ij}^{(k)}(n) - \eta \frac{(x_j^{(k)} - d_j^{(k)}) x_i^{(k-1)}}{x_j^{(k)} (1 - x_j^{(k)})}$$

$$w_{ij}^{(k)}(t+1) = w_{ij}^{(k)}(t) - \eta \frac{\delta_j^{(k)} x_i^{(k-1)}}{\delta_j^{(k)} = (x_j^{(k)} - d_j^{(k)}) x_j^{(k)} (1 - x_j^{(k)})}$$

Feed forward Back Propagation Algo.

Back Prop. Algo.

1. Initialize  $w_{ij}(k) \leftarrow$  Random values

2. Feed Training Samples

3. Feed forward

for  $k = 0$  to  $K-1$  compute

$$x_j(k+1) = R\left(\sum_{i=0}^{M_k} w_{ij}(k+1) x_{ij}(k)\right)$$

for nodes  $j = 1$  to  $M_{k+1}$

4. Back Propagation

For nodes in the o/p layer

$j = 1$  to  $M_k$  compute

$$\delta_j(k) = x_j(k) (1 - \pi_j(k)) (x_j^{dk} - d_j)$$

For layer  $k-1 \dots 1$  compute

$$\delta_i^{(k)} = x_i^{(k)} (1 - x_i^{(k)}) \sum_{j=1}^{M_{k+1}} \delta_j^{(k+1)} w_{ij}^{(k)}$$

For  $i=1$  to  $M_k$

5. updates the weights

$$w_{ij}^{(k)}(t+1) = w_{ij}^{(k)}(t) - \eta \delta_j^{(k)} x_i^{(k-1)}$$

~~Repeat steps~~ 2 to 5 until convergence



# Activation functions and training

---

- For feed-forward networks:
  - A continuous function can be differentiated allowing gradient-descent.
  - Back-propagation is an example of a gradient-descent technique.
  - Reason for prevalence of sigmoid



# Gradient Descent Learning Rule

---

- Consider linear unit without threshold and continuous output  $o$  (not just  $-1,1$ )
  - $o = w_0 + w_1 x_1 + \dots + w_n x_n$
- Train the  $w_i$ 's such that they minimize the squared error
  - $E[w_1, \dots, w_n] = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$   
where  $D$  is the set of training examples

# Gradient Descent

$$D = \{ \langle (1,1), 1 \rangle, \langle (-1,-1), 1 \rangle, \\ \langle (1,-1), -1 \rangle, \langle (-1,1), -1 \rangle \}$$

Gradient:

$$\nabla E[w] = [\partial E / \partial w_0, \dots, \partial E / \partial w_n]$$

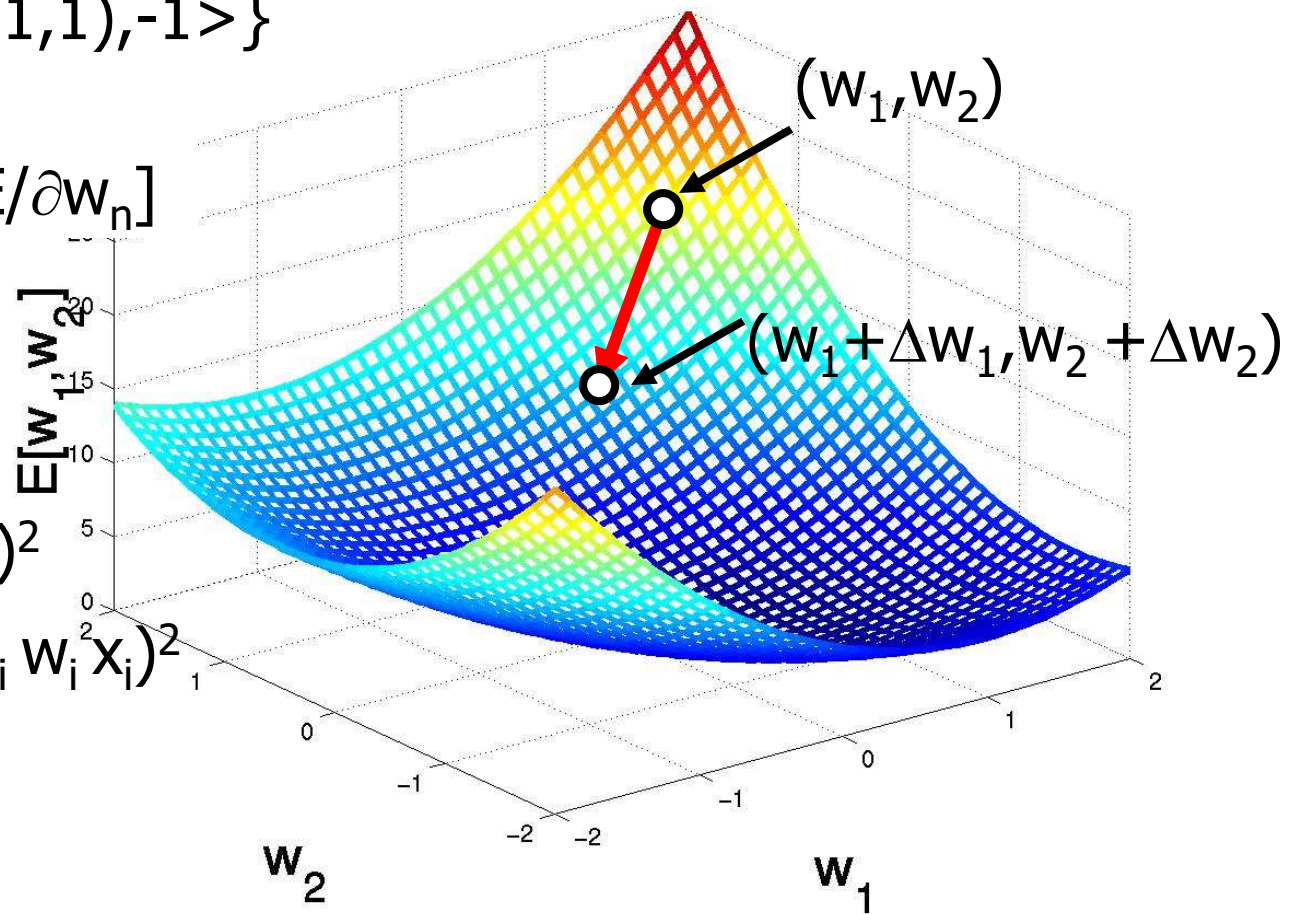
$$\Delta w = -\eta \nabla E[w]$$

$$\Delta w_i = -\eta \partial E / \partial w_i$$

$$= \partial / \partial w_i \frac{1}{2} \sum_d (t_d - o_d)^2$$

$$= \partial / \partial w_i \frac{1}{2} \sum_d (t_d - \sum_i w_i^2 x_i)^2$$

$$= \sum_d (t_d - o_d) (-x_i)$$





# Gradient Descent

---

Gradient-Descent(*training\_examples*,  $\eta$ )

Each training example is a pair of the form  $\langle (x_1, \dots, x_n), t \rangle$  where  $(x_1, \dots, x_n)$  is the vector of input values, and  $t$  is the target output value,  $\eta$  is the learning rate (e.g. 0.1)

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - Initialize each  $\Delta w_i$  to zero
  - For each  $\langle (x_1, \dots, x_n), t \rangle$  in *training\_examples* Do
    - Input the instance  $(x_1, \dots, x_n)$  to the linear unit and compute the output  $o$
    - For each linear unit weight  $w_i$  Do
      - $\Delta w_i = \Delta w_i + \eta (t - o) x_i$
  - For each linear unit weight  $w_i$  Do
    - $w_i = w_i + \Delta w_i$





# Incremental Stochastic Gradient Descent

- Batch mode : gradient descent  
 $w = w - \eta \nabla E_D[w]$  over the entire data  $D$   
 $E_D[w] = 1/2 \sum_d (t_d - o_d)^2$
- Incremental mode: gradient descent  
 $w = w - \eta \nabla E_d[w]$  over individual training examples  $d$   
 $E_d[w] = 1/2 (t_d - o_d)^2$

Incremental Gradient Descent can approximate Batch Gradient Descent arbitrarily closely if  $\eta$  is small enough



# Comparison Perceptron and Gradient Descent Rule

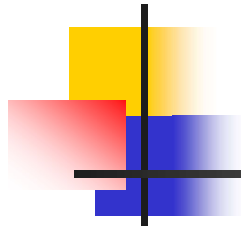
---

Perceptron learning rule guaranteed to succeed if

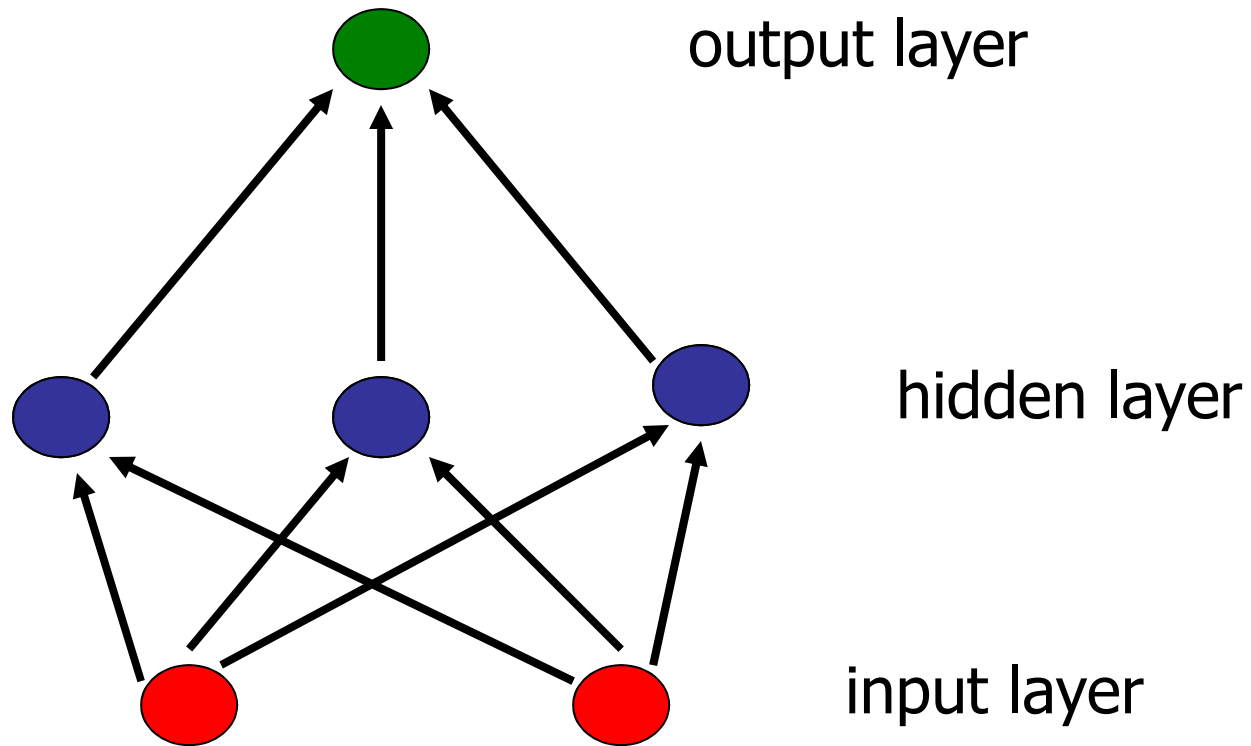
- Training examples are linearly separable
- Sufficiently small learning rate  $\eta$

Linear unit training rules uses gradient descent

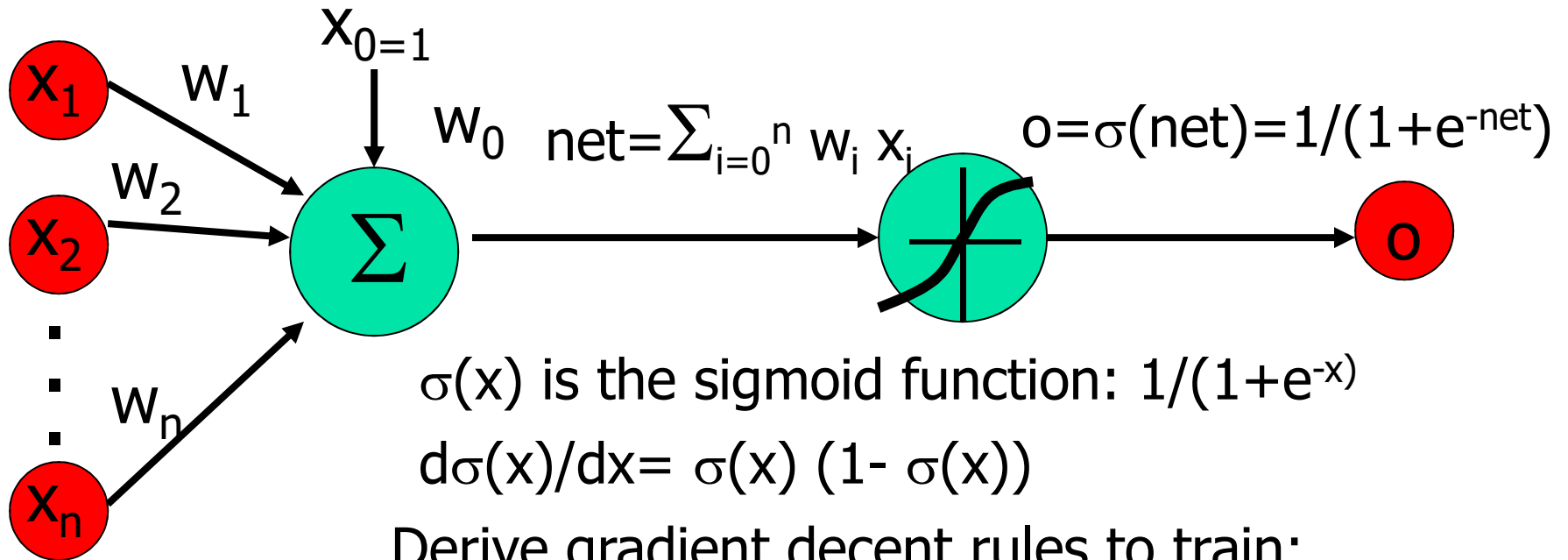
- Guaranteed to converge to hypothesis with minimum squared error
- Given sufficiently small learning rate  $\eta$
- Even when training data contains noise
- Even when training data not separable by H



# Multi-Layer Networks



# Sigmoid Unit



$\sigma(x)$  is the sigmoid function:  $1/(1+e^{-x})$   
 $d\sigma(x)/dx = \sigma(x) (1 - \sigma(x))$

Derive gradient decent rules to train:

- one sigmoid function

$$\partial E / \partial w_i = -\sum_d (t_d - o_d) o_d (1 - o_d) x_i$$

- Multilayer networks of sigmoid units  
backpropagation:



# Backpropagation Algorithm

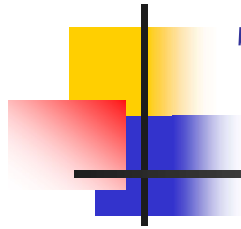
- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - For each training example  $\langle (x_1, \dots, x_n), t \rangle$  Do
    - Input the instance  $(x_1, \dots, x_n)$  to the network and compute the network outputs  $o_k$
    - For each output unit  $k$ 
      - $\delta_k = o_k(1 - o_k)(t_k - o_k)$
    - For each hidden unit  $h$ 
      - $\delta_h = o_h(1 - o_h) \sum_k w_{h,k} \delta_k$
    - For each network weight  $w_{i,j}$  Do
    - $w_{i,j} = w_{i,j} + \Delta w_{i,j}$  where
      - $\Delta w_{i,j} = \eta \delta_j x_{i,j}$



# Backpropagation

---

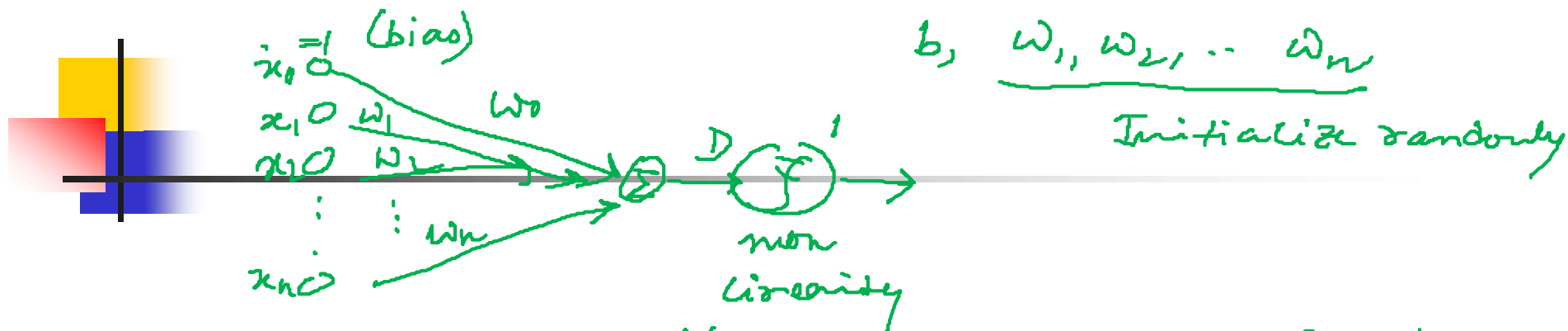
- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum  
-in practice often works well (can be invoked multiple times with different initial weights)
- Often include weight *momentum* term
$$\Delta w_{i,j}(t) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(t-1)$$
- Minimizes error training examples
  - Will it generalize well to unseen instances (over-fitting)?
- Training can be slow typical 1000-10000 iterations  
(use Levenberg-Marquardt instead of gradient descent)
- Using network after training is fast



# The Structure of Neurons

---

- A neuron only fires if its input signal exceeds a certain amount (the **threshold**) in a short time period.
- Synapses vary in strength
  - Good connections allowing a large signal
  - Slight connections allow only a weak signal.
  - Synapses can be either **excitatory** or **inhibitory**.



$$D = \sum_{i=0}^N w_i x_i, \quad \text{target of } d \checkmark$$

mean square error

$$E = \frac{1}{2} (D - d)^2, \quad \text{minimize Error (E) with respect to } w_i \text{'s.}$$

Error gradient

$$\frac{\partial E}{\partial w_i} = (D - d) \left[ \frac{\partial (D - d)}{\partial w_i} \right]$$

$d = \text{const.}$   
independent of  $i$

$$\frac{\partial E}{\partial w_i} = (D - d) \frac{\partial (D - d)}{\partial (w_i)} = \frac{\partial (D)}{\partial w_i} = \frac{\partial}{\partial w_i} \left[ \sum_{i=0}^N x_i w_i \right] = x_i$$





$w_i(0)$  = initialized randomly.

$$\begin{aligned} w_i(k+1) &= w_i(k) - \eta \frac{\partial E}{\partial w_i} \\ &= w_i(k) - \eta (D-d)x_i \end{aligned}$$

2-class  
Problem

update rule of weight  
at Step  $k+1$

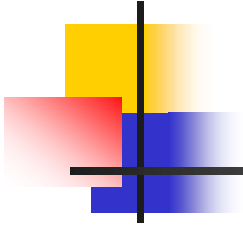
$\eta$  = learning  
rate

Extend to Multi-class  
Problem

= rate of convergence.

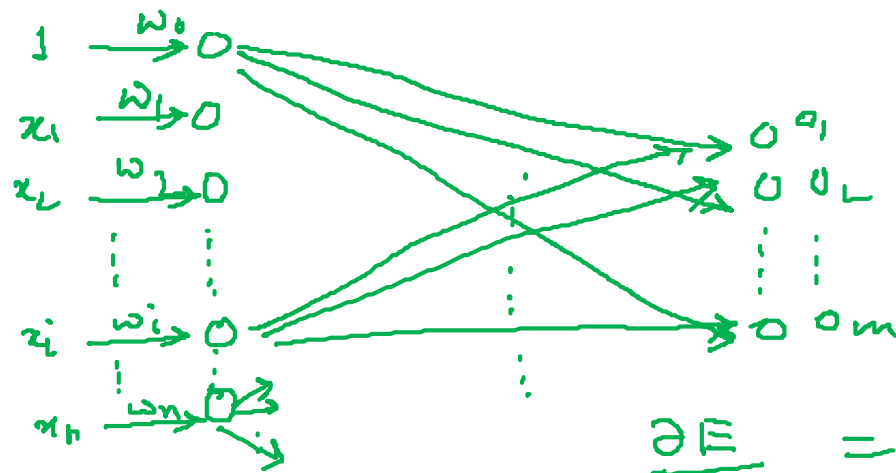
for  $\forall w_i$

$$g_i(x) > g_j(x) \quad \forall j \neq i \\ \Rightarrow x \in w_i$$



to continue...

# multi-class classification



$$E = \frac{1}{2} \sum_{j=1}^m (D_j - d_j)^2$$

$$\frac{\partial E}{\partial w_{ij}} = (D_j - d_j) x_i$$

Training algo

$w_{ij}(0) \leftarrow$  initialised randomly

$$w_{ij}(k+1) = w_{ij}(k) - \eta (D_j - d_j) x_i$$

weight updation rule.