

# **Neural Networks**

**COMP3411/9814: Artificial Intelligence  
(2023)**

# Sub-Symbolic Processing



Which one do you see?



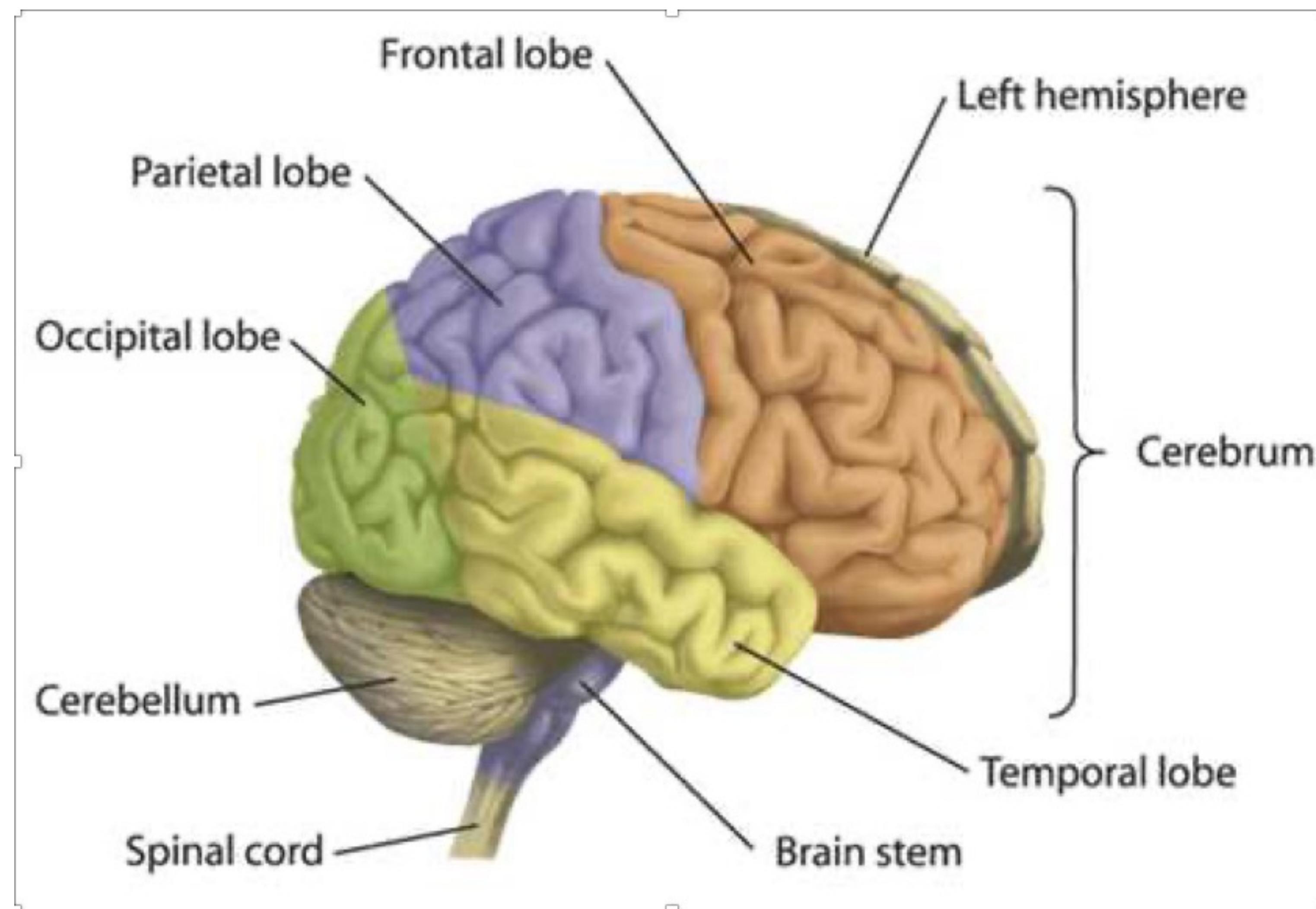
Which one do you see? Dalmatian or Hyenas?



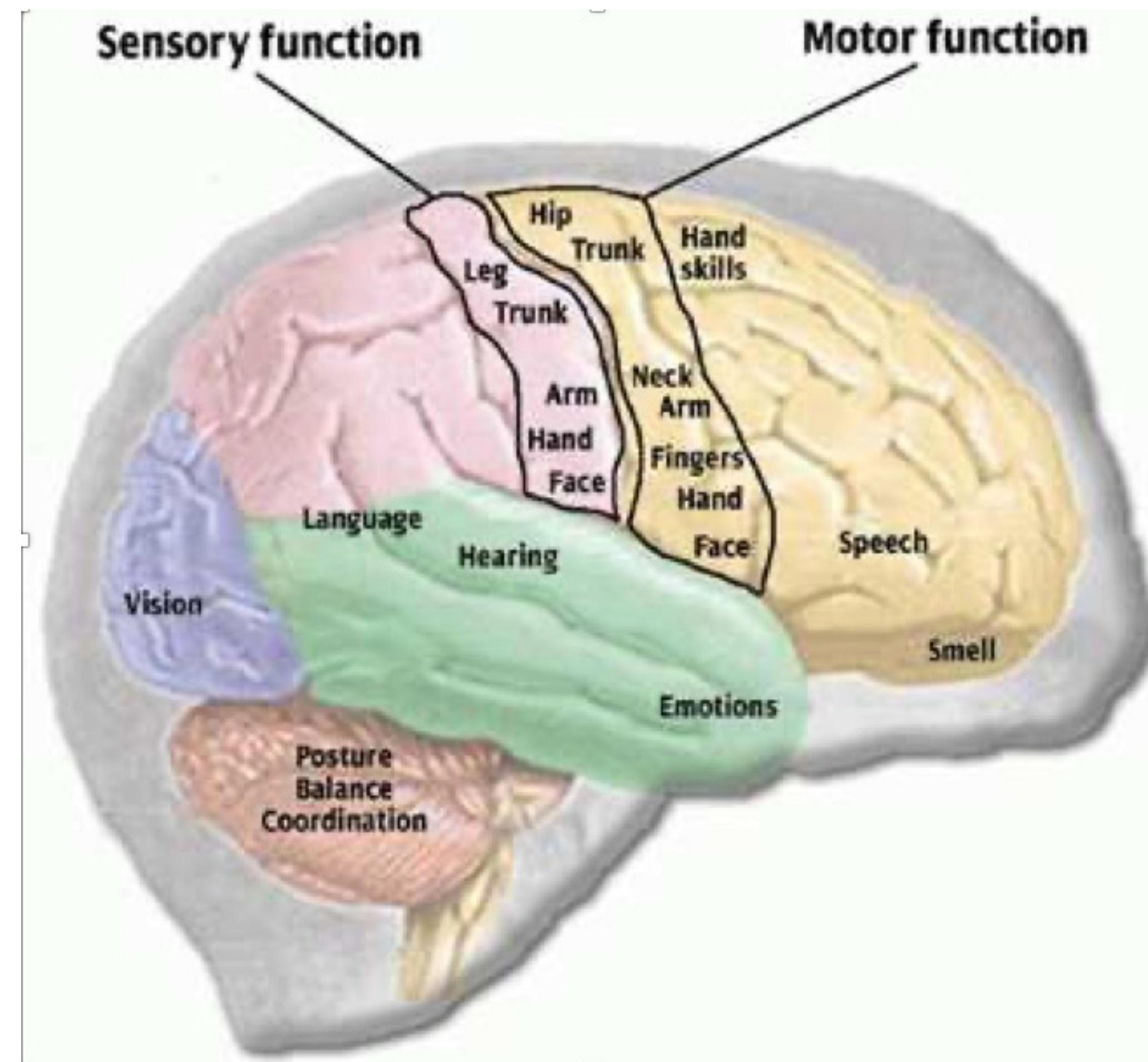
# Lecture Overview

- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

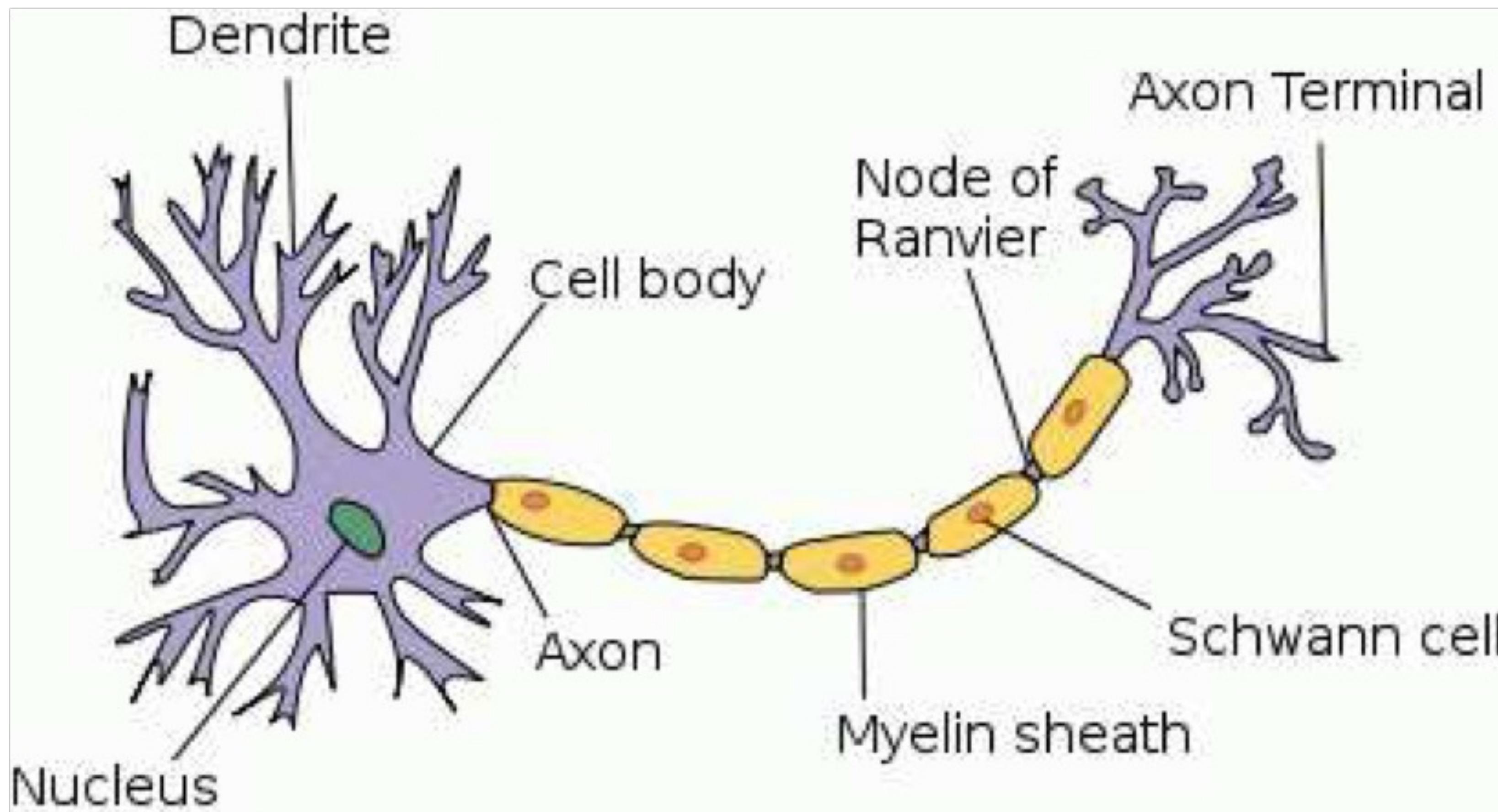
# Brain Regions



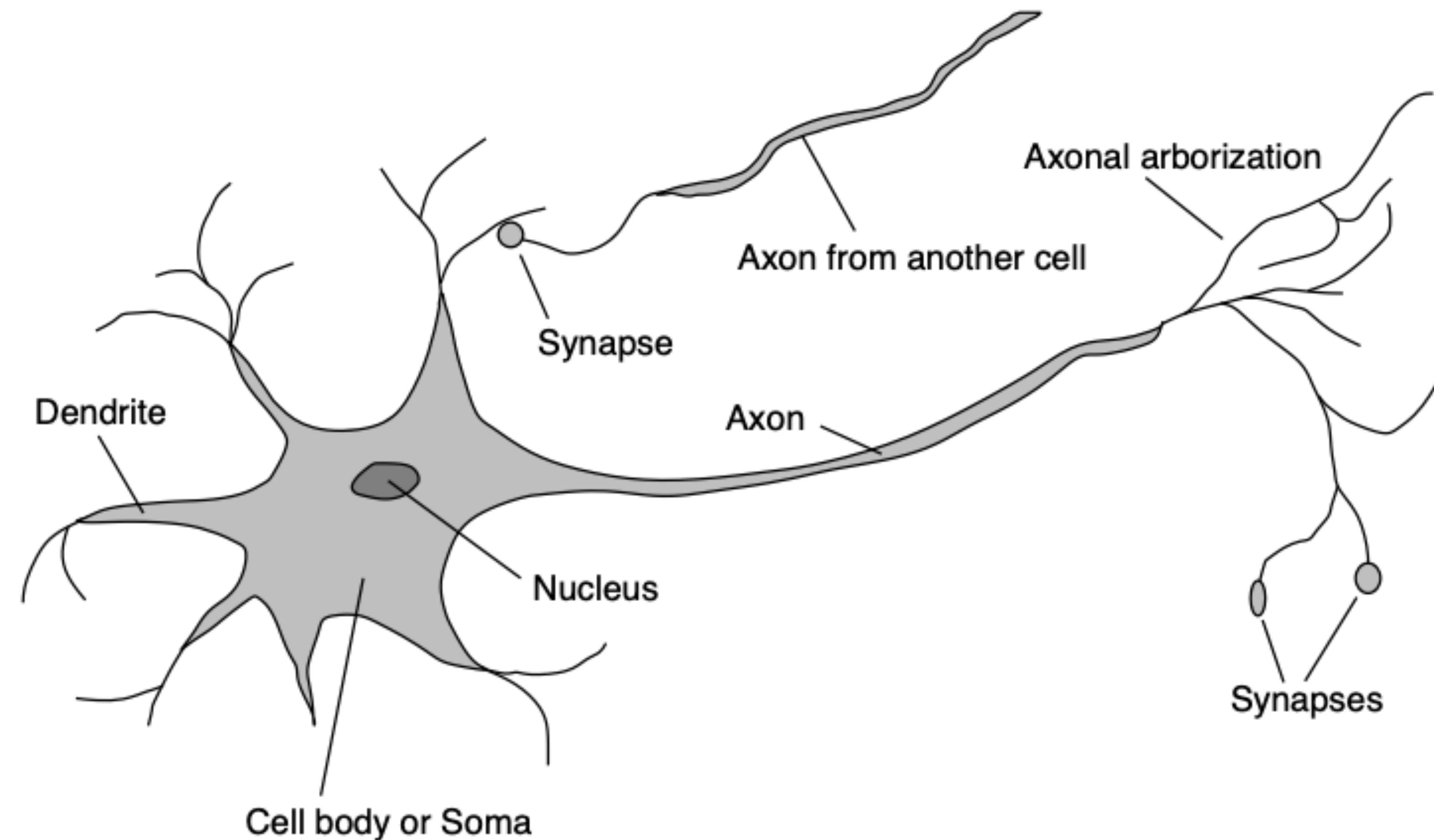
# Brain Regions



# Structure of a Typical Neuron



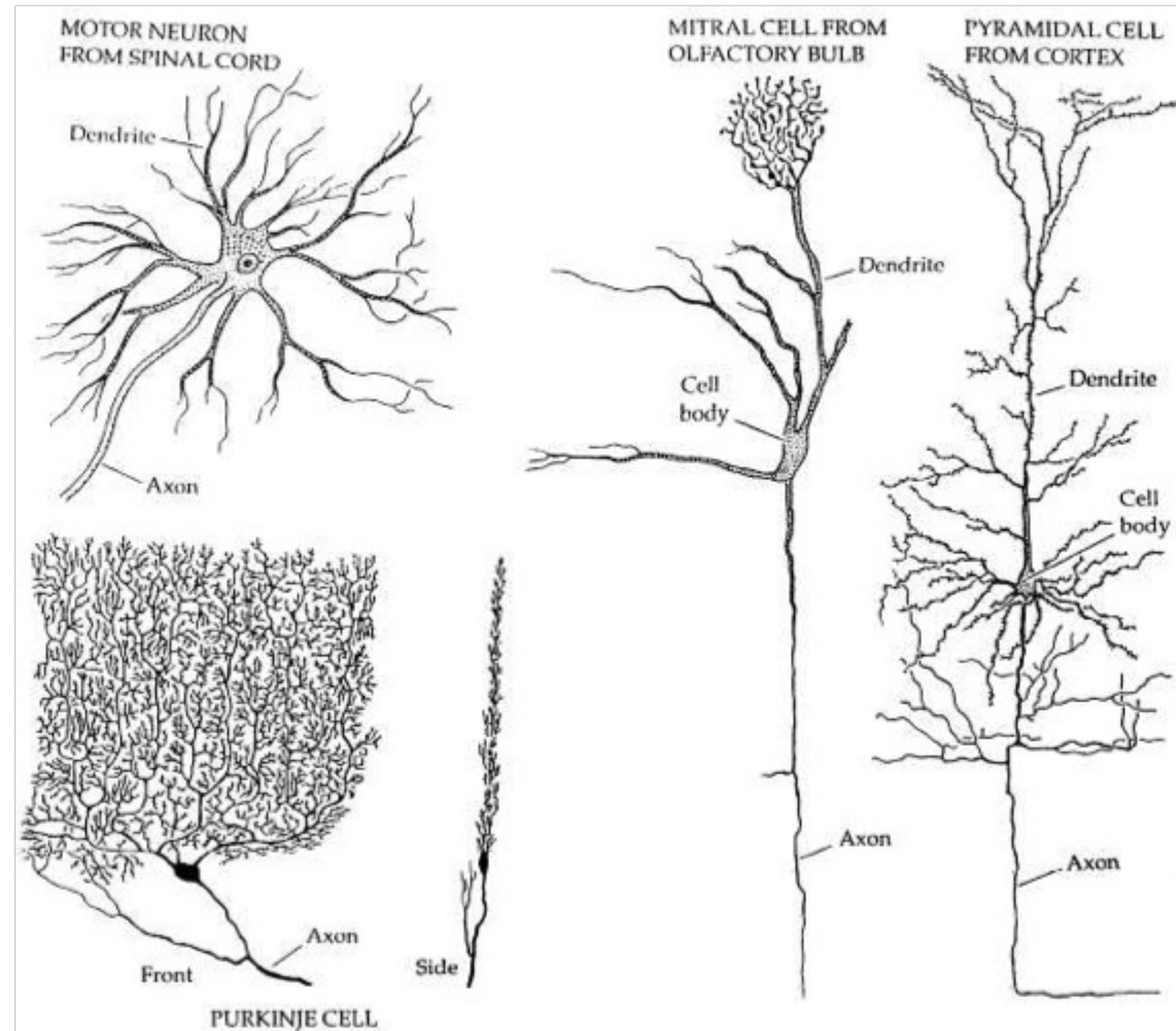
# Brains



# Biological Neurons

- The brain is made up of **neurons** (nerve cells) which have
  - a cell body (soma)
  - **dendrites** (inputs)
  - an **axon** (outputs)
  - **synapses** (connections between cells)
- Synapses can be **excitatory** or **inhibitory** and may change over time.
- When the inputs reach some threshold an **action potential** (electrical pulse) is sent along the axon to the outputs.

# Variety of Neuron Types



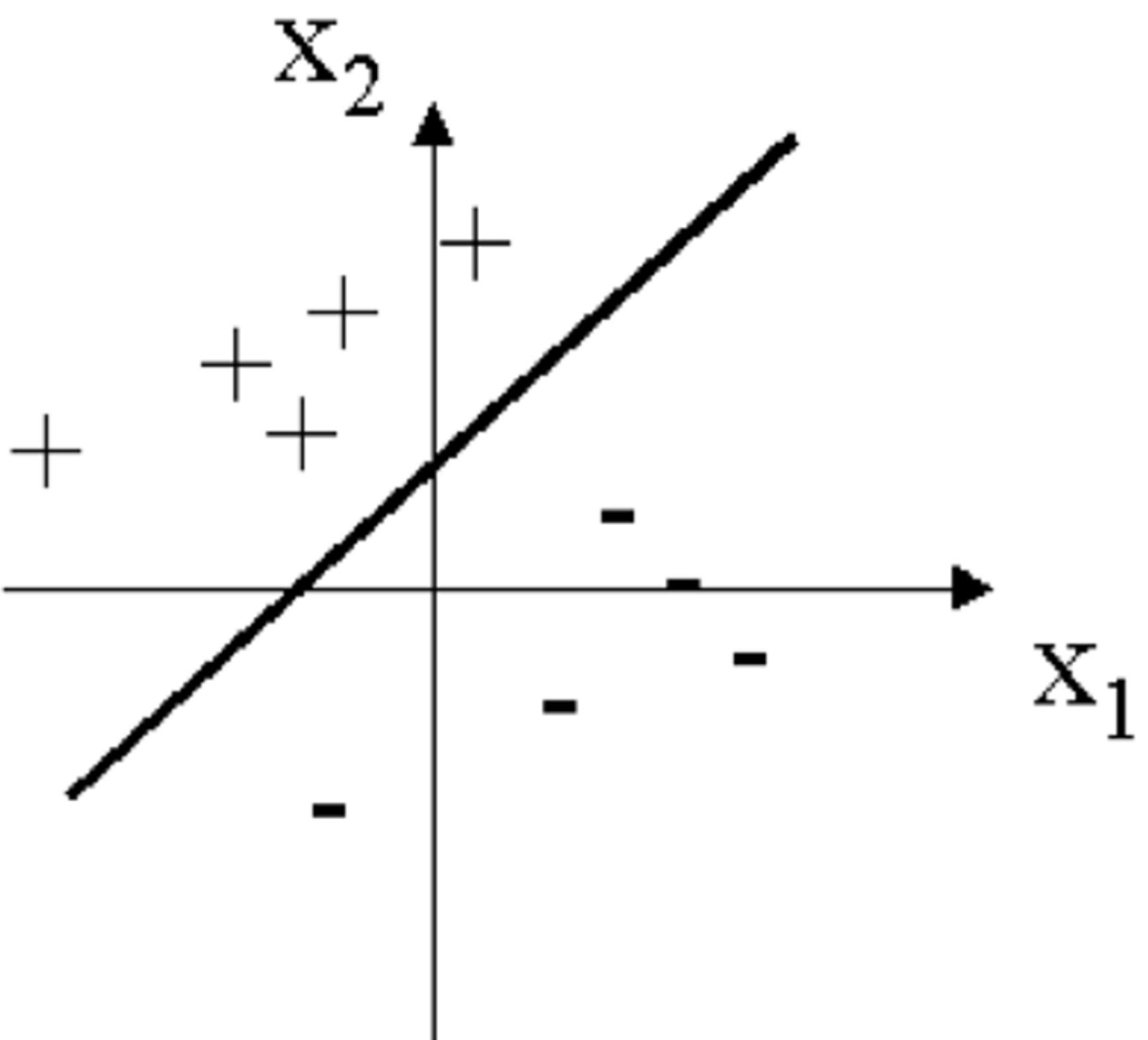
# The Big Picture

- Human brain has 100 billion neurons with an average of 10,000 synapses each
- Latency is about 3-6 milliseconds
- At most a few hundred “steps” in any mental computation, but **massively parallel**

# Lecture Overview

- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

# Perceptron Learning



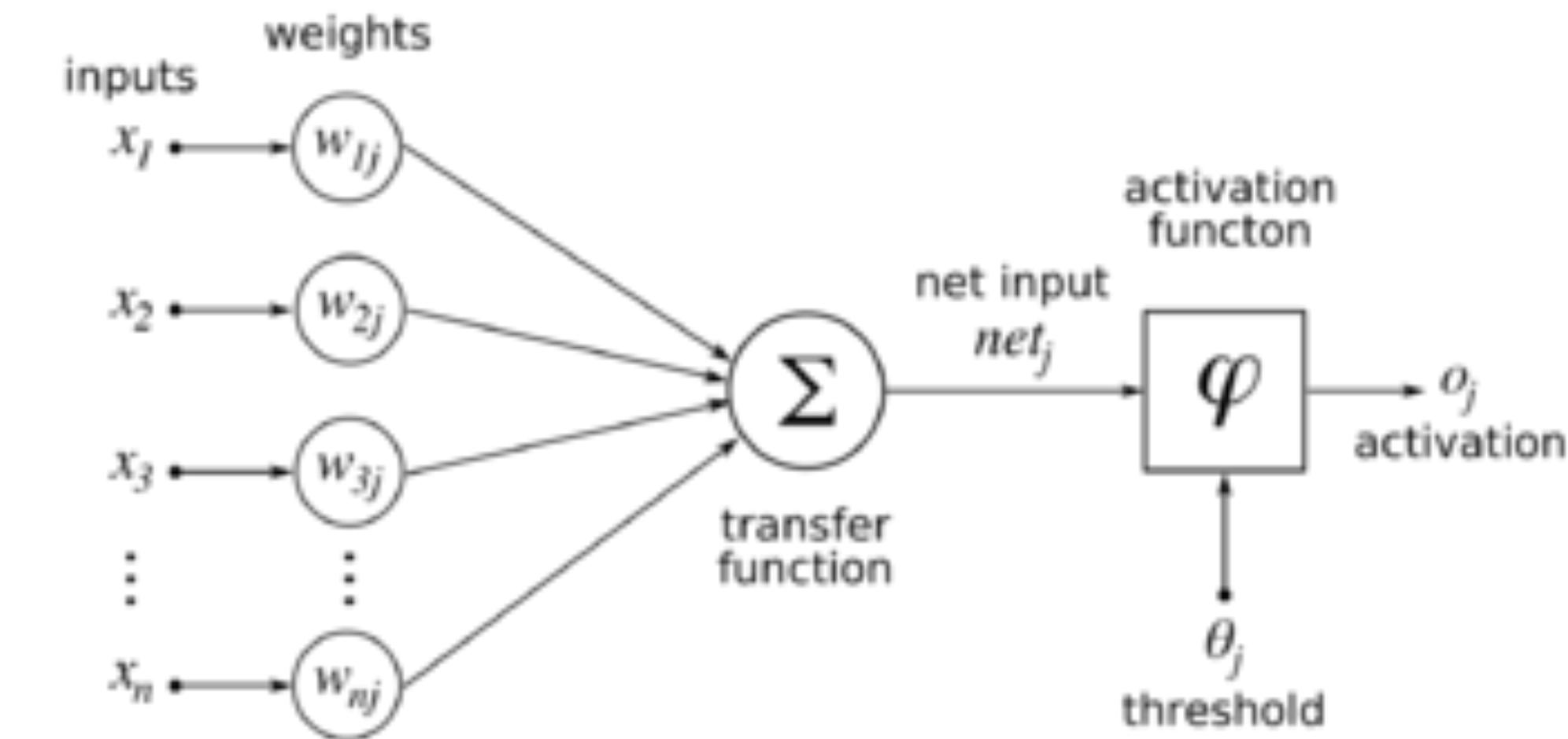
$$y = ax + b$$

# Artificial Neural Networks

- Information processing architecture loosely modelling the brain
- Consists of many interconnected processing units (**neurons**)
  - Work in parallel to accomplish a global task
- Generally used to model relationships between inputs and outputs or to find patterns in data

# Artificial Neural Networks (ANN)

- ANNs nodes have
  - inputs edges with some **weights**
  - outputs edges with **weights**
  - **activation level** (function of inputs)



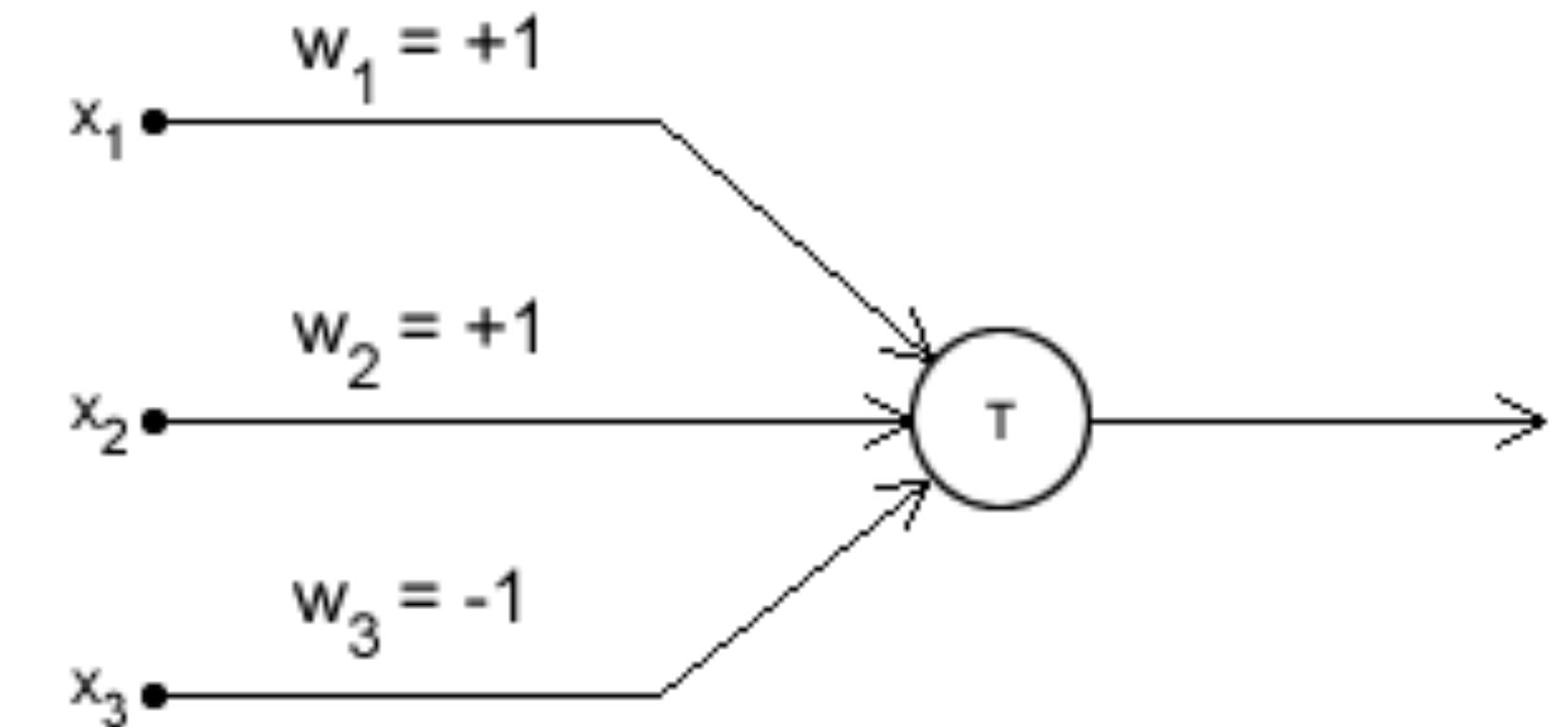
- Weights can be positive or negative and may change over time (learning).
- The **input function** is the weighted sum of the activation levels of inputs.
- The activation level is a non-linear **transfer** function  $g$  of this input:

$$\text{activation}_j = g(s_j) = g\left(\sum_i w_{ij}x_i\right)$$

Some nodes are inputs (sensing), some are outputs (action)

# First artificial neurons: McCulloch-Pitts (1943)

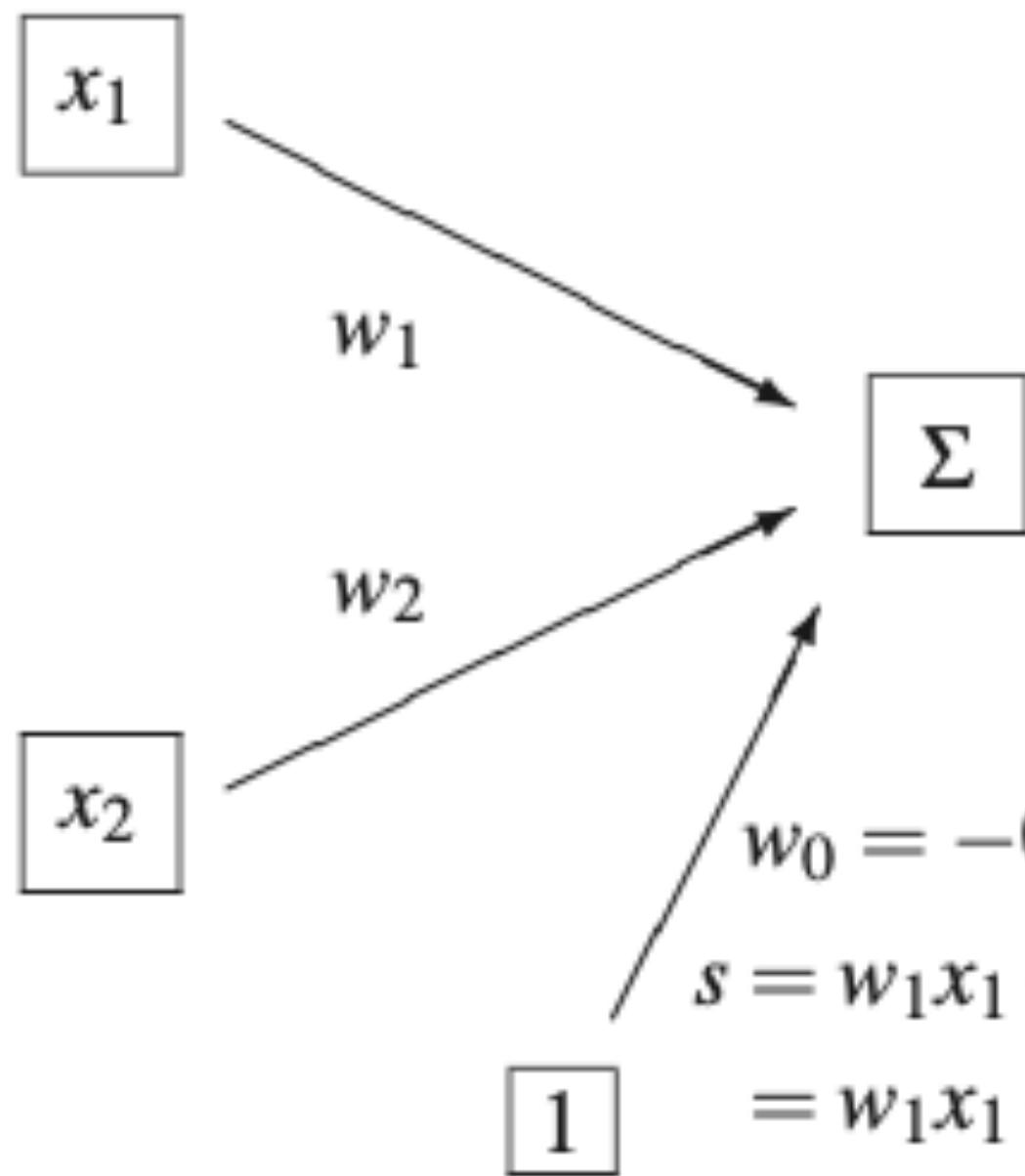
- McCulloch-Pitts model:
  - Inputs either 0 or 1.
  - Output 0 or 1.
  - Input can be either excitatory or inhibitory.
- Summing inputs
  - If input is 1, and is excitatory, add 1 to sum.
  - If input is 1, and is inhibitory, subtract 1 from sum.
- Threshold,
  - if  $\text{sum} < \text{threshold}, T$ , output 0.
  - Otherwise, output 1.



$$\text{sum} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \dots$$

**if**  $\text{sum} < T$  **then** *output* is 0  
**else** *output* is 1.

# McCulloch & Pitts Model of a Single Neuron



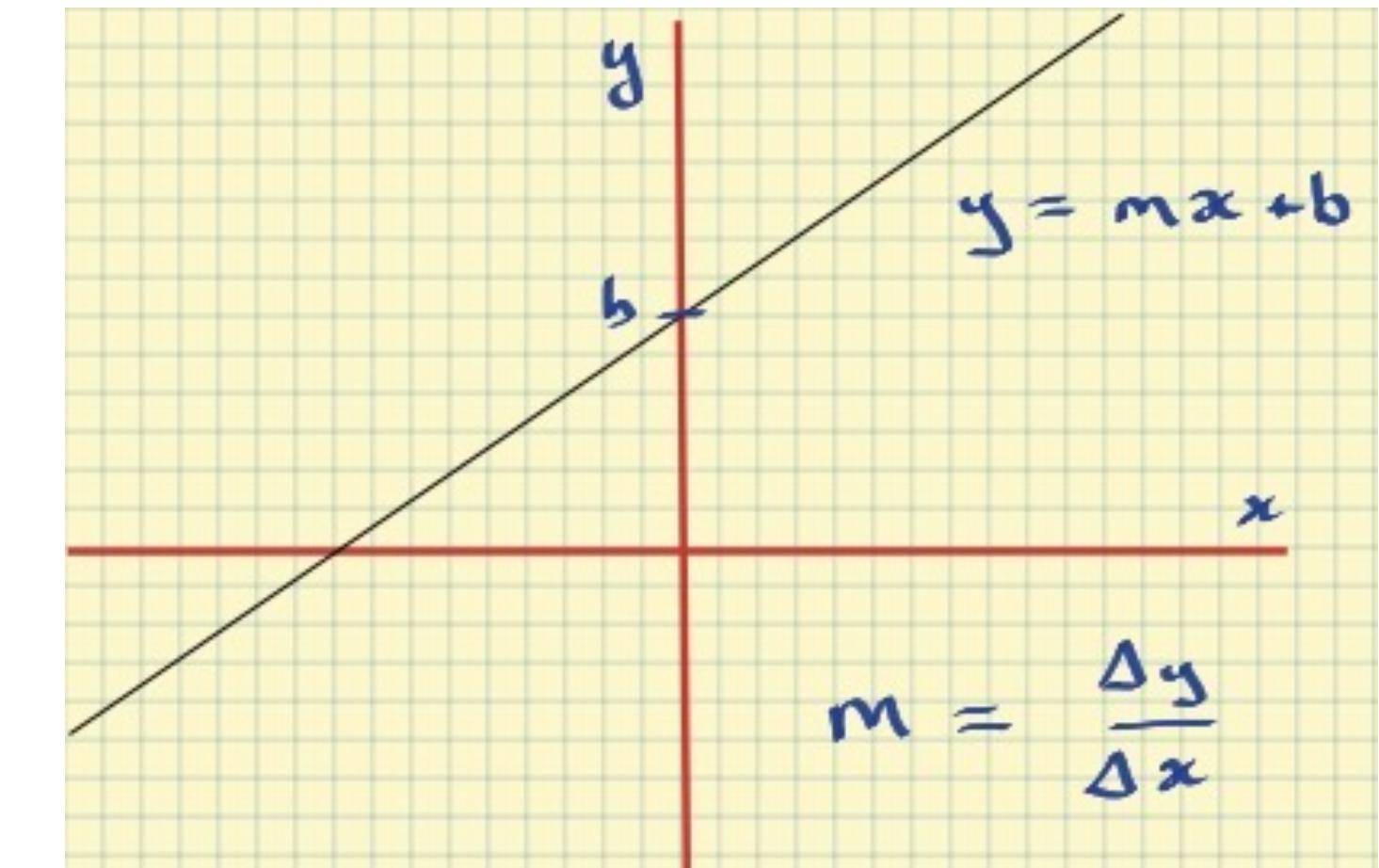
$x_1, x_2$  are inputs

$w_1, w_2$  are synaptic weights

$\theta$  is a threshold

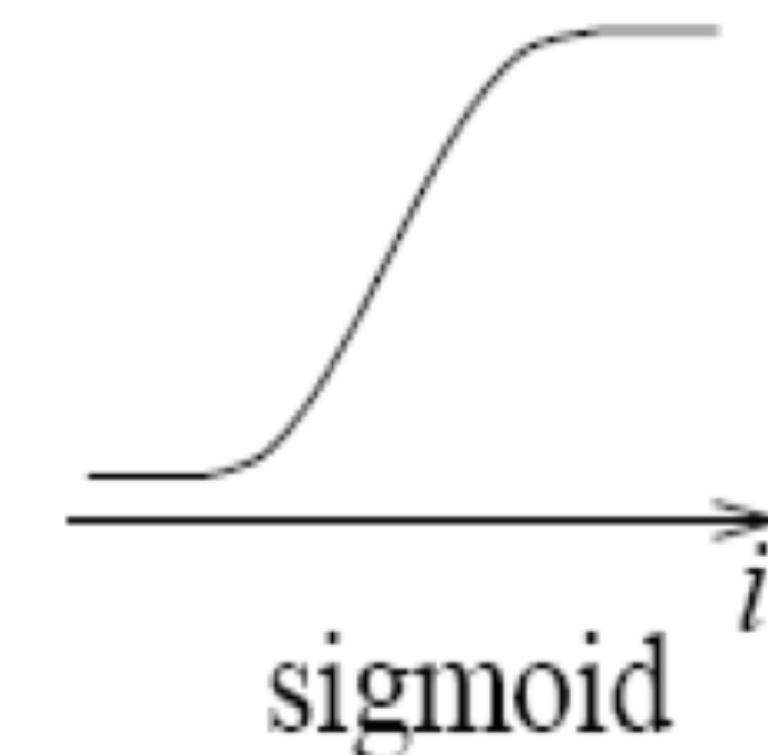
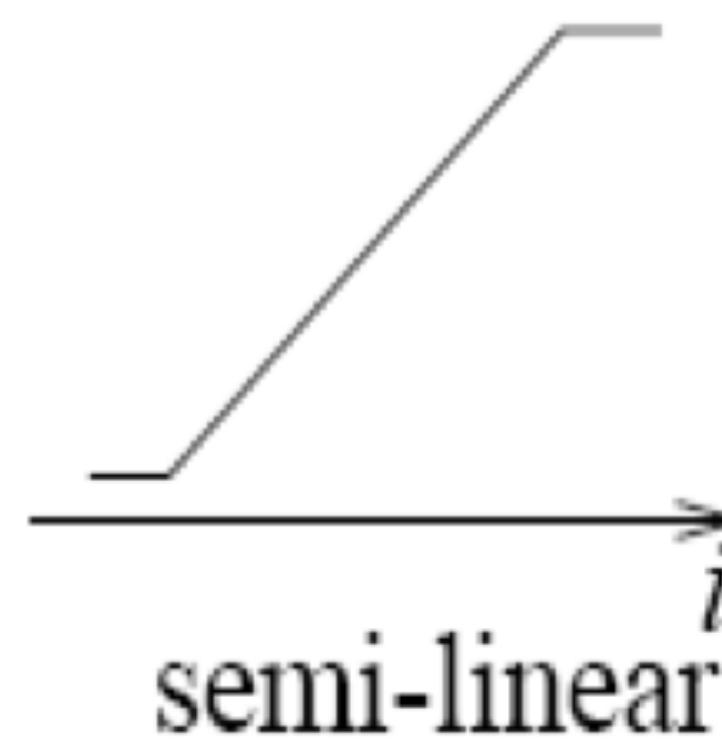
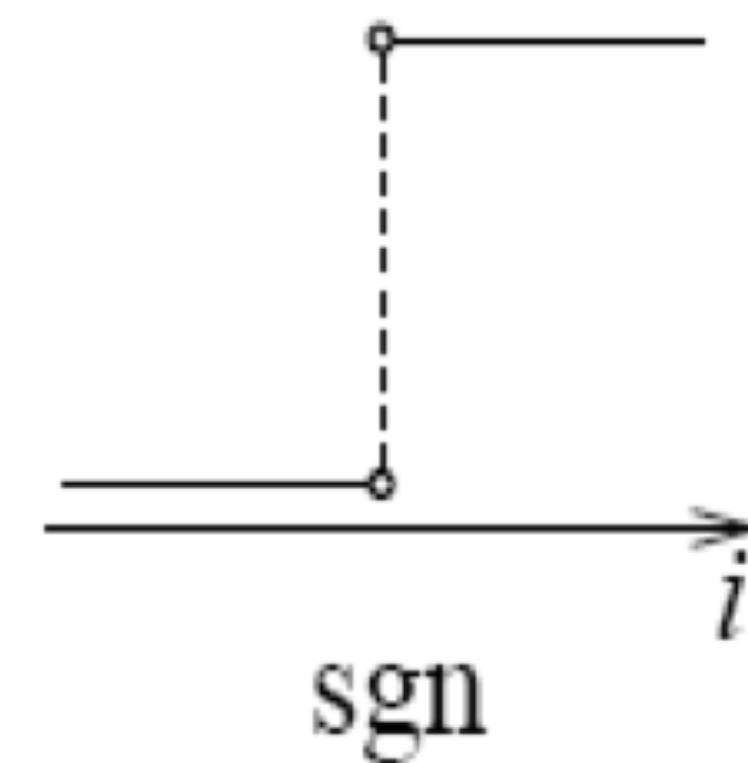
$w_0$  is a **bias** weight

$g$  is transfer function



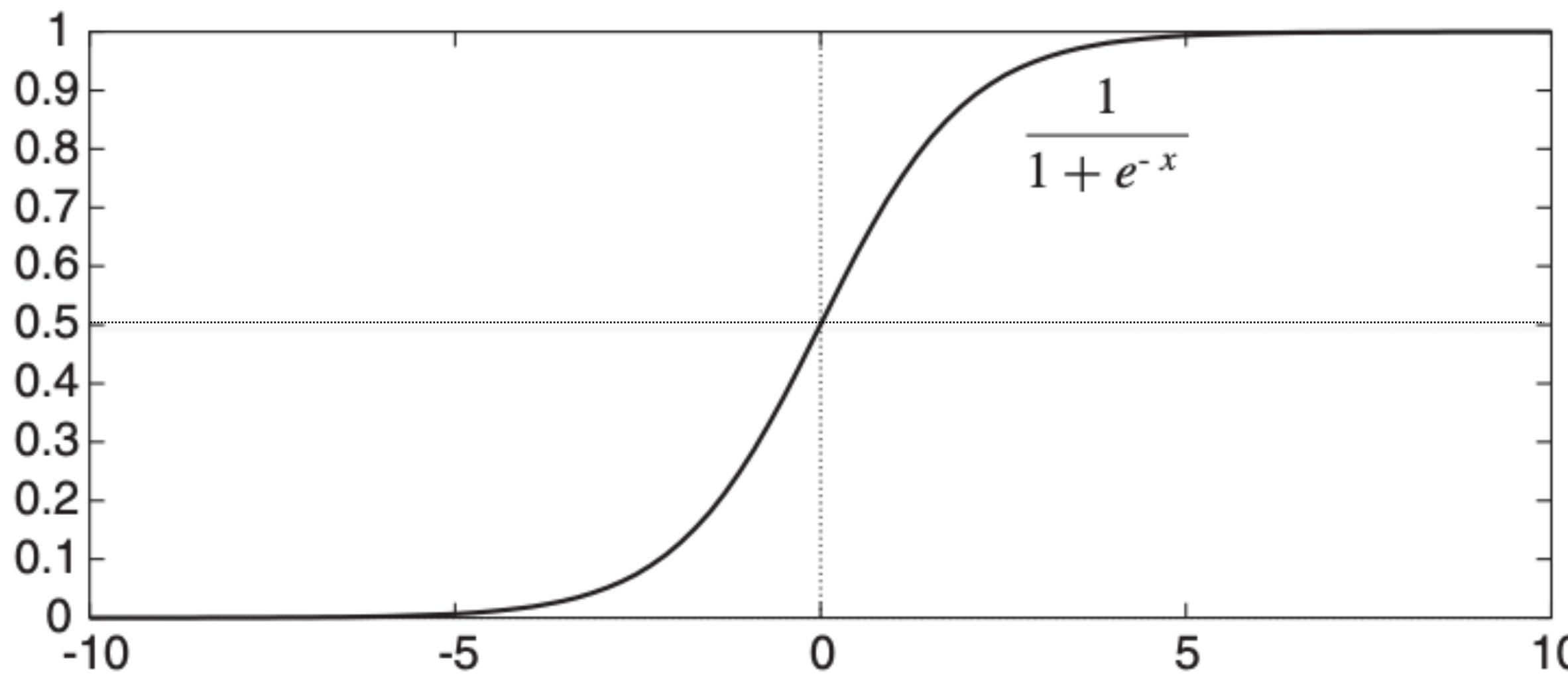
# Activation Functions

Function  $g(s)$  takes the weighted sum of inputs and produces output for node, given some threshold.



$$g(s) = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & \text{if } s < 0 \end{cases}$$

# The sigmoid or logistic activation function

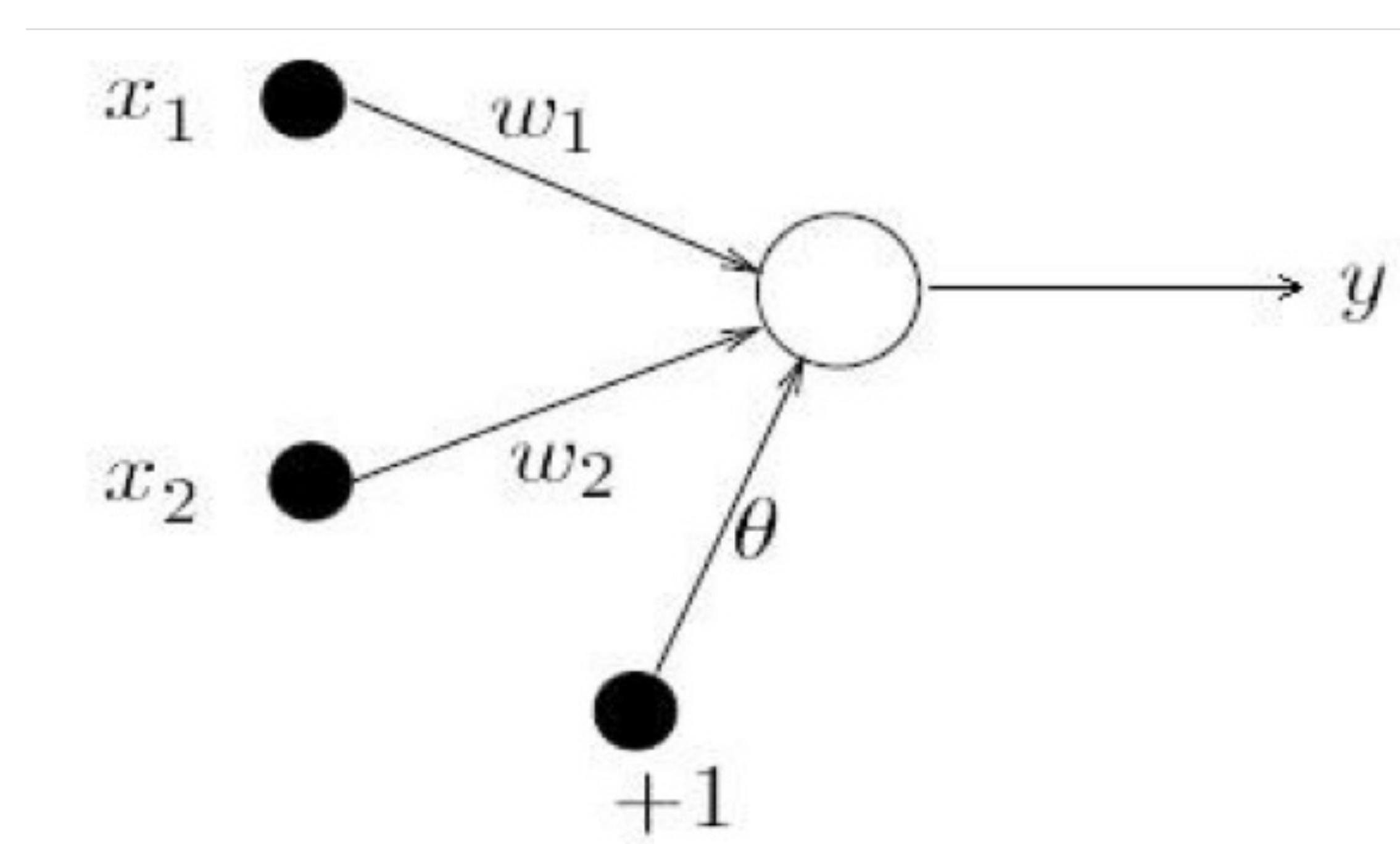


$$f(x) = \frac{1}{1 + e^{-x}}$$

Derivative  $f'(x) = f(x)(1 - f(x))$  is the slope of the function

# Simple Perceptron

The perceptron is a single layer feed-forward neural network.

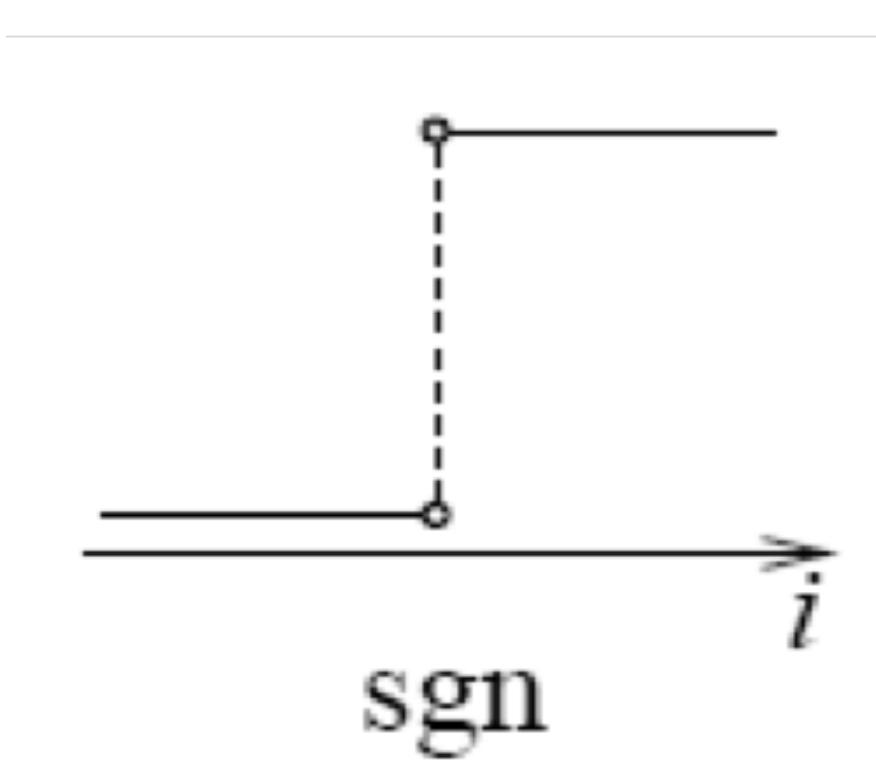


# Simple Perceptron

Simplest output function

$$y = \text{sgn} \left( \sum_{i=1}^2 w_i x_i + \theta \right)$$

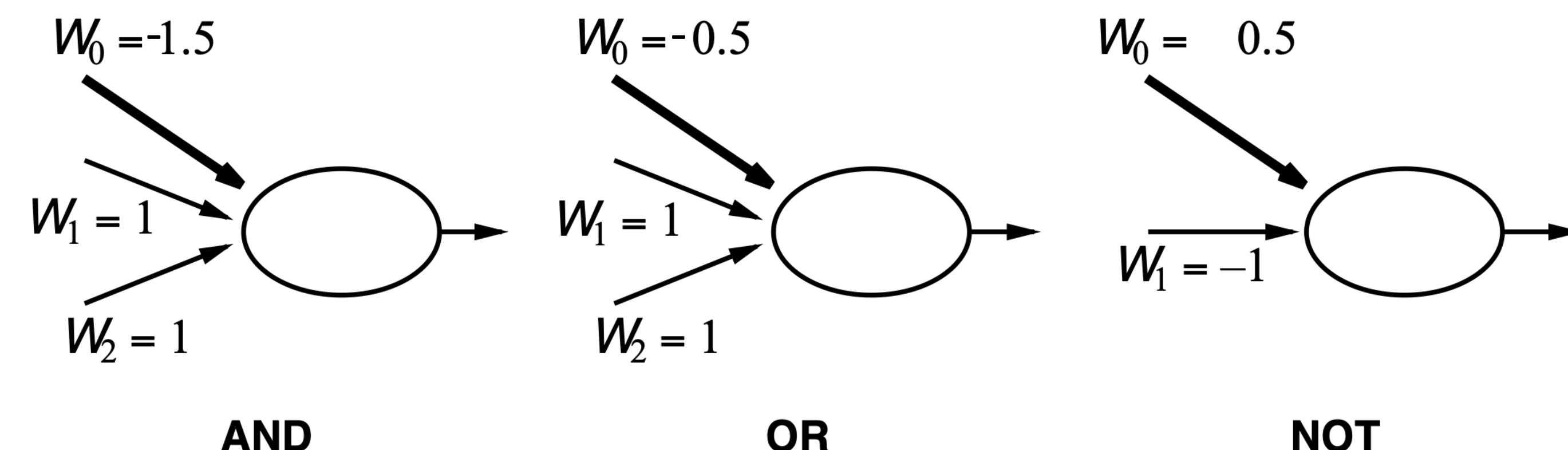
$$\text{sgn}(s) = \begin{cases} 1 & \text{if } s > 0 \\ -1 & \text{otherwise.} \end{cases}$$



Used to classify patterns said to be linearly separable

# Implementing logical functions

McCulloch and Pitts - every Boolean function can be implemented:



Threshold = 0.5

# Lecture Overview

- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

# Linear Separability

Examples:

AND             $w_1 = w_2 = 1.0, \quad w_0 = -1.5$

OR             $w_1 = w_2 = 1.0, \quad w_0 = -0.5$

NOR             $w_1 = w_2 = -1.0, \quad w_0 = 0.5$

Can we train a perceptron net to learn a new function?

Yes, as long as it is linearly separable

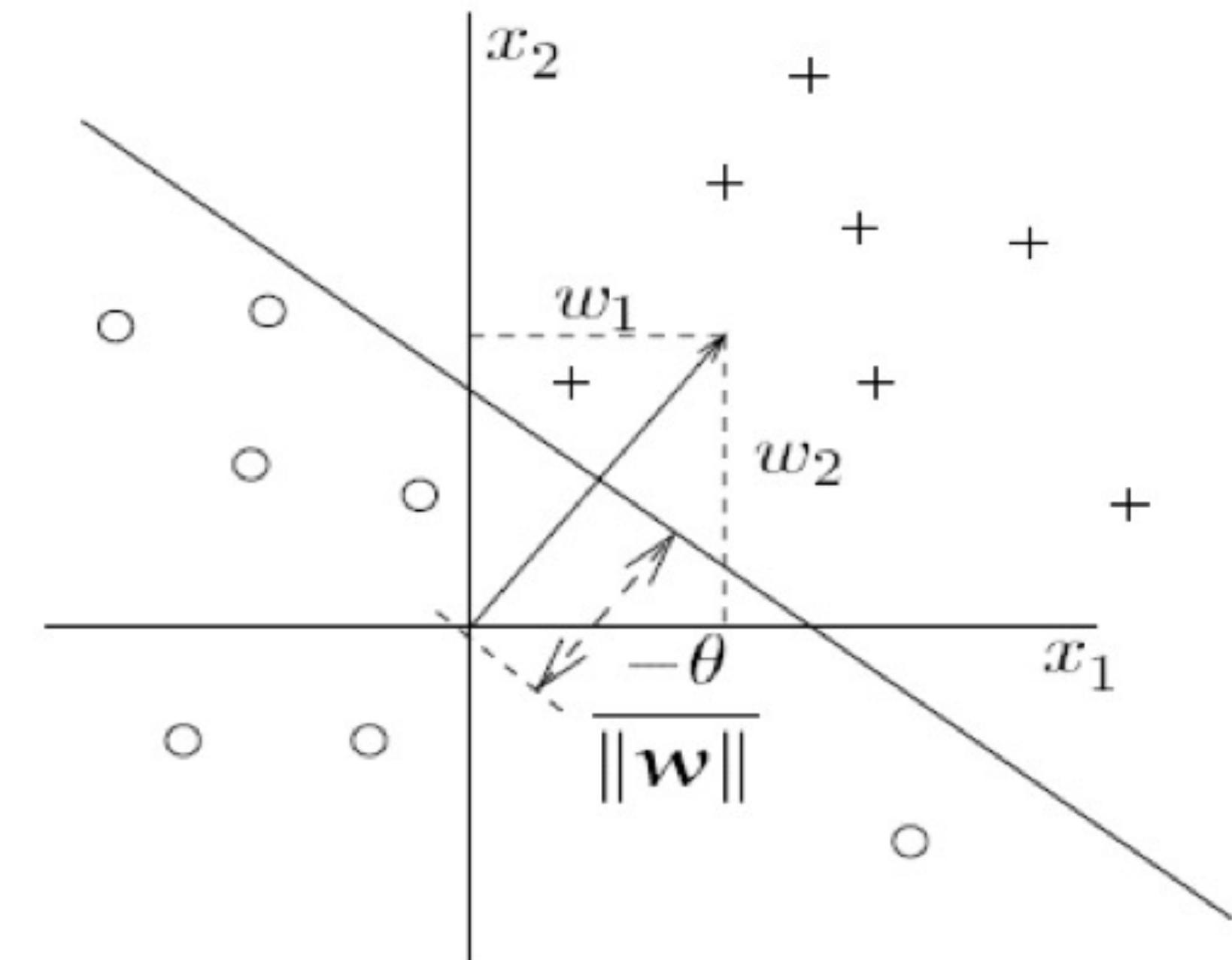
# Linear Separability

The **bias** is proportional to the offset of the plane from the origin

The weights determine the slope of the line

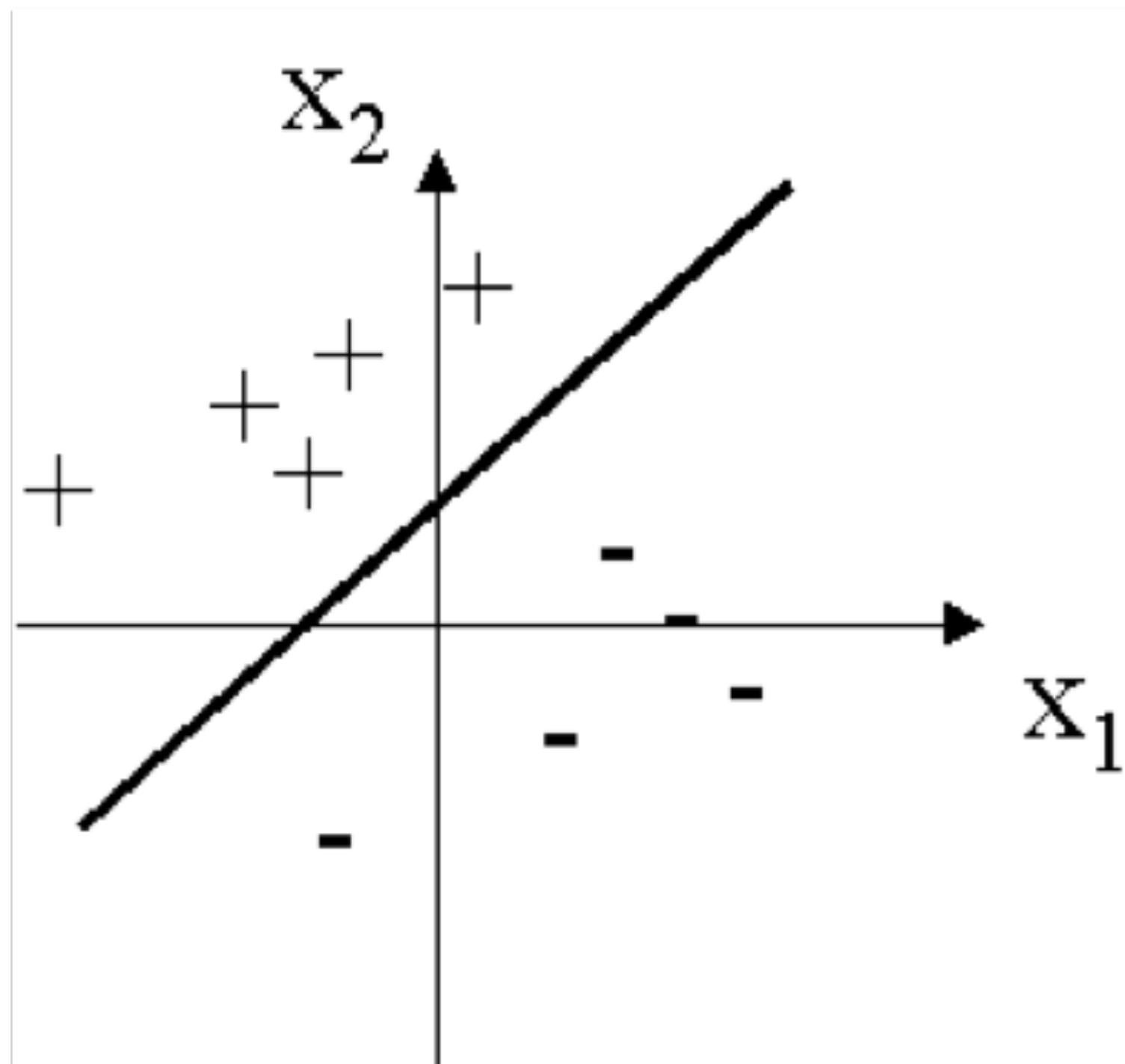
The weight vector is perpendicular to the plane

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



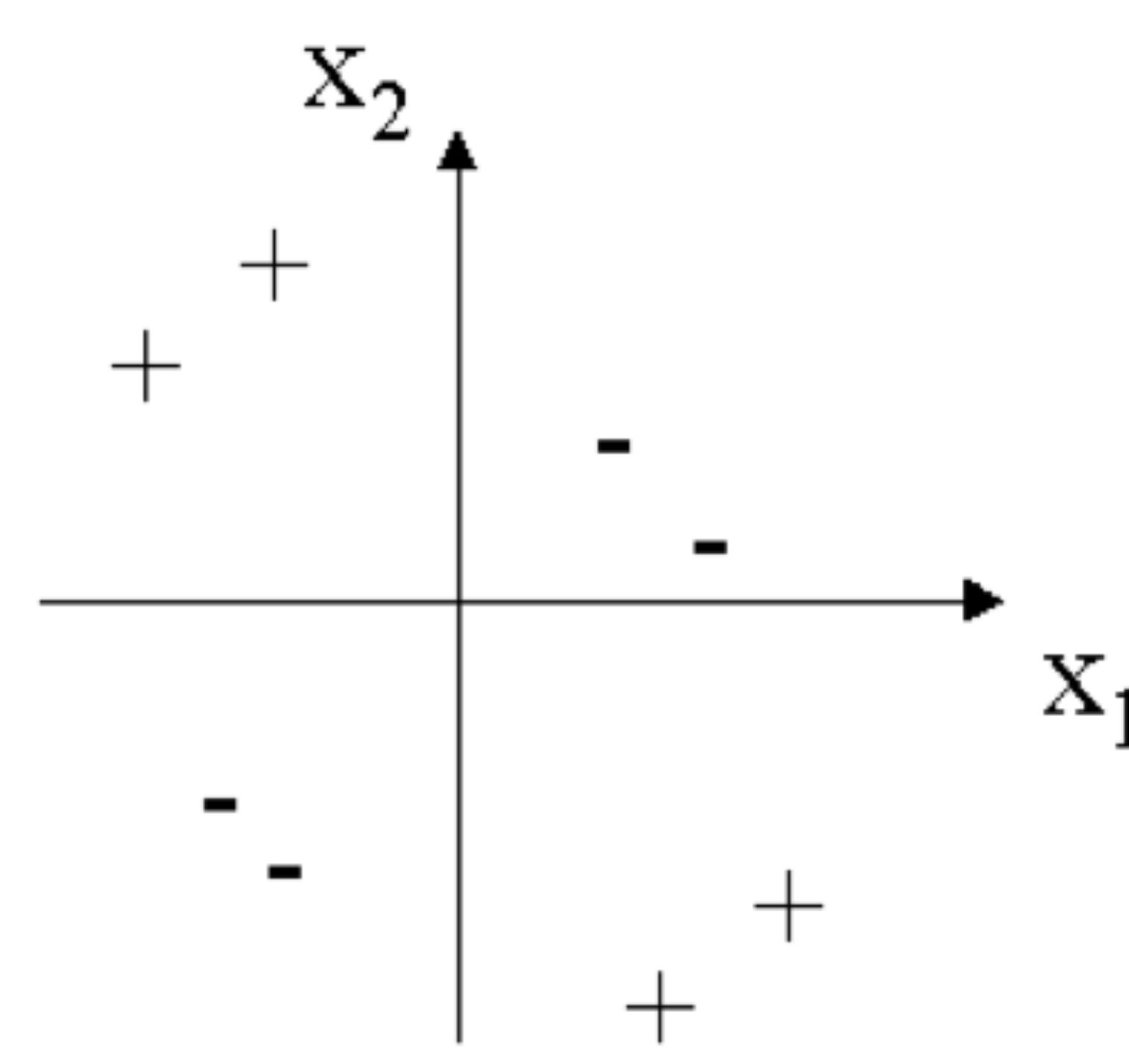
# Linear Separability

What kind of functions can a perceptron compute?



Linearly Separable

$$w_1x_1 + w_2x_2 + \theta = 0$$



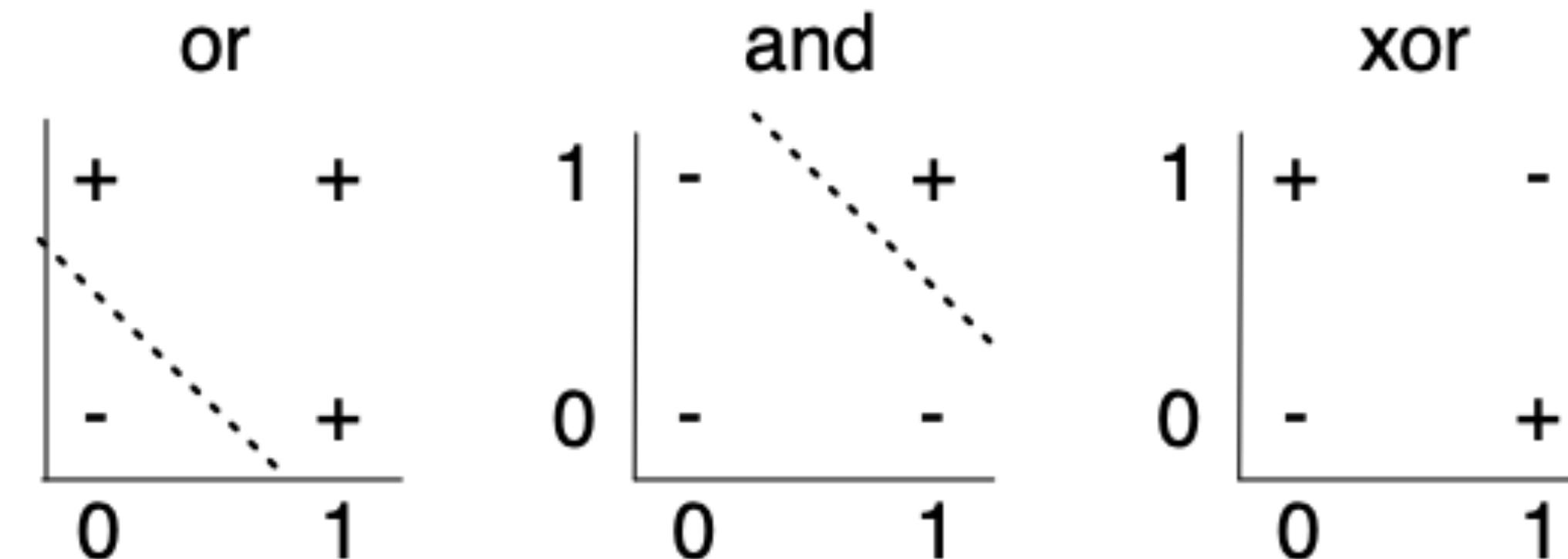
Not Linearly Separable

# Linearly Separable

- **Linearly separable** if there is a hyperplane where classification is true on one side of hyperplane and false on other side
- For the sigmoid function, when the hyperplane is:

$$x_1 \cdot w_1 + \cdots + x_n \cdot w_n = 0$$

separates the predictions  $> 0.5$  and  $< 0.5$ .

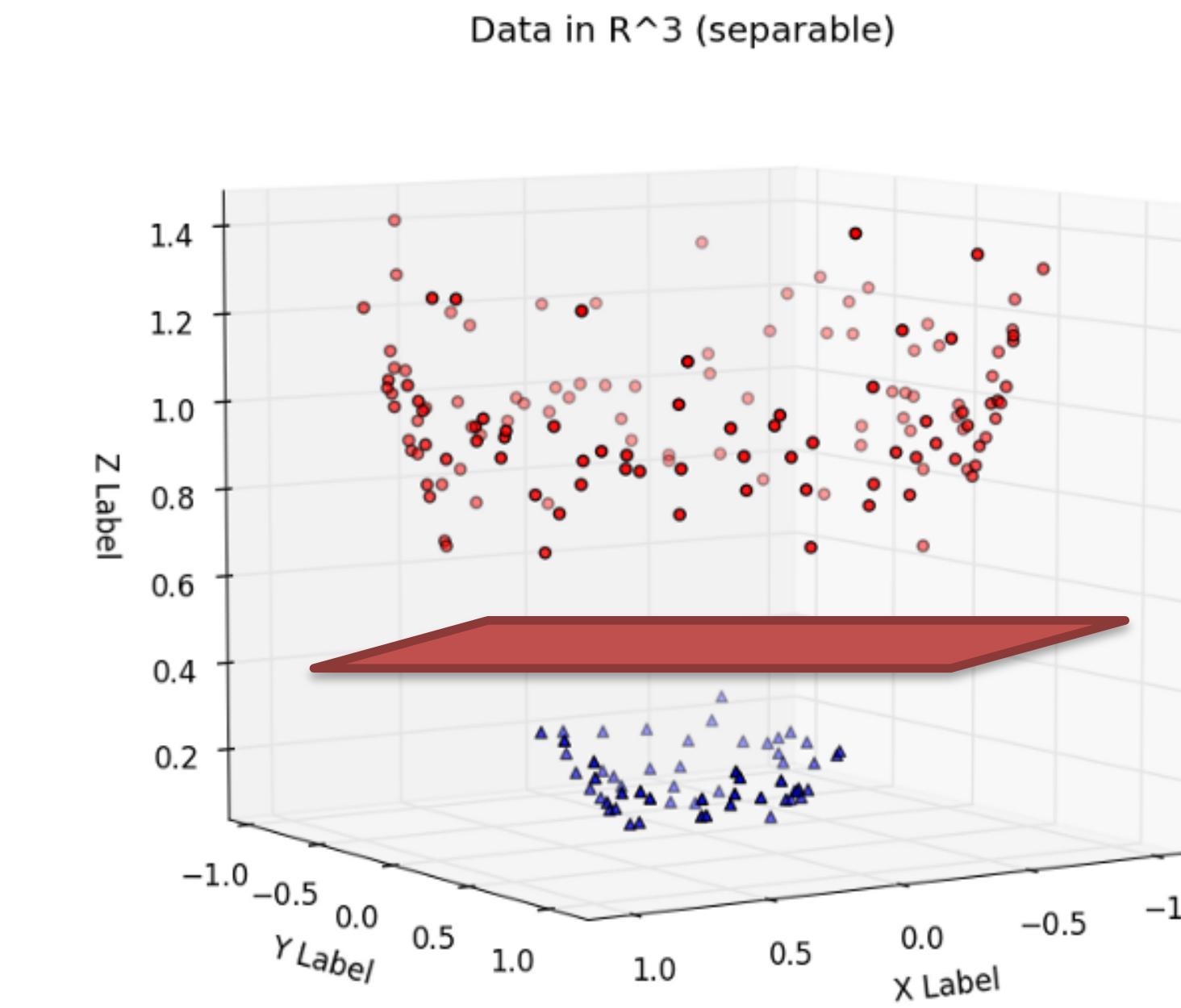
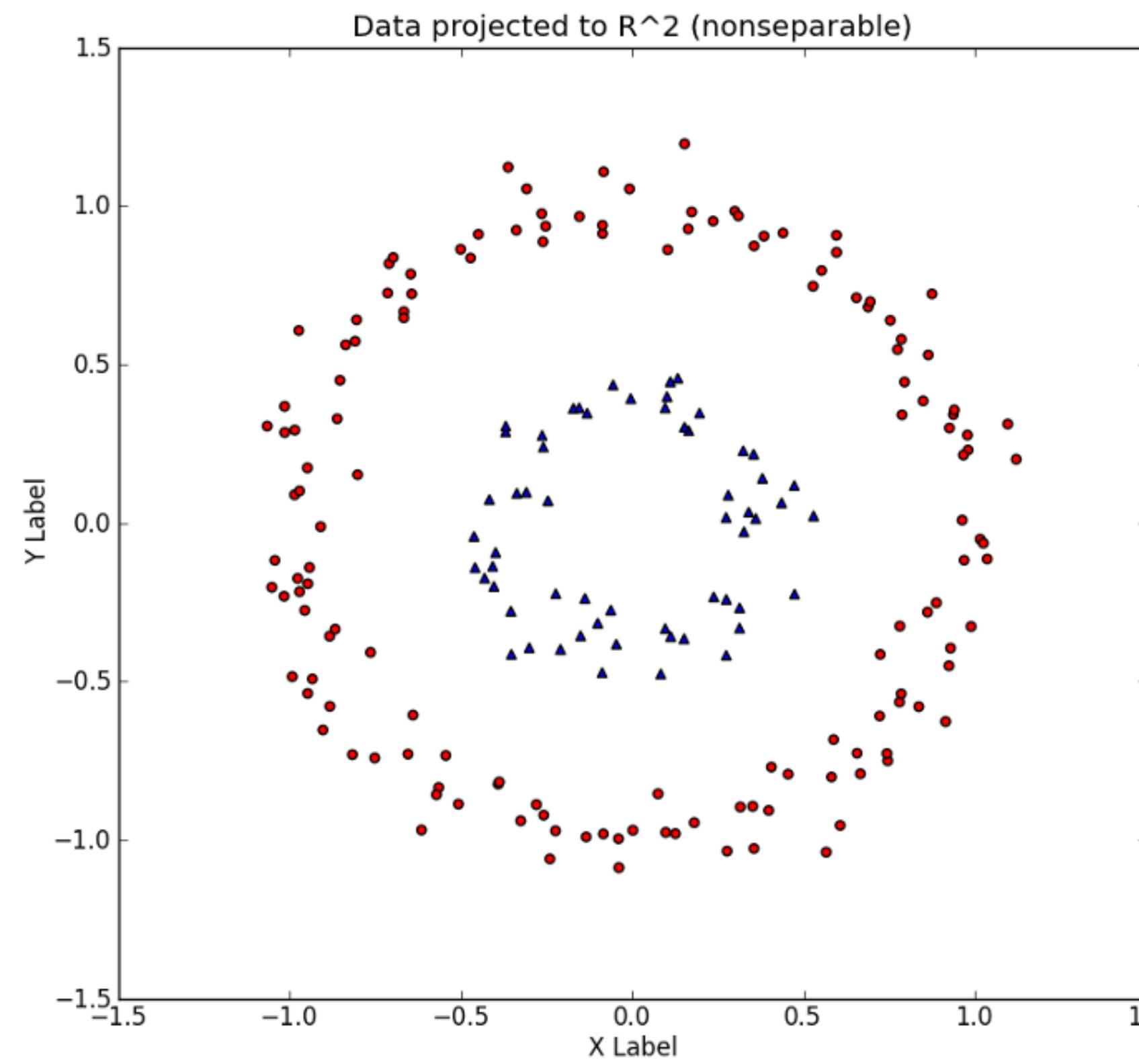


# Variants in Linear Separators

- Which linear separator to use can result in various algorithms:
  - Perceptron
  - Logistic Regression
  - Support Vector Machines (SVMs) ...

# Kernel Trick

- Project points onto an higher dimensional space
- Becomes linearly separable



# Perceptron Learning Algorithm

- Want to train perceptron to classify inputs correctly
- Compare output of network with correct output and **adjust** the **weights** and **bias** to minimise the error
- So learning is parameter optimisation
- Can only handle linearly separable sets

# Perceptron Learning Algorithm

- Training set: set of input vectors to train perceptron.
- During training,  $w_i$  and  $\theta$  (*bias*) are modified
  - ▶ for convenience, let  $w_0 = \theta$  and  $x_0 = 1$
- $\eta$ , is the *learning rate*, a small positive number
  - ▶ small steps lessen possibility of destroying correct classifications
- Initialise  $w_i$  to some values

# Perceptron Learning Rule

- Repeat for each training example
  - Adjust the weight,  $w_i$ , for each input,  $x_i$ .
$$w_i \leftarrow w_i + \eta(d - y) \cdot x_i$$

$\eta > 0$  is the learning rate  
 $d$  is the desired output  
 $y$  is the actual output
- If output correct, no change
- If  $d=1$  but  $y=0$ ,  $w_i$  is increased when  $x_i$  is positive and decreased when  $x_i$  is negative (want to increase  $\mathbf{w} \cdot \mathbf{x}$ )
- If  $d=0$  but  $y=1$ ,  $w_i$  is decreased when  $x_i$  is positive and increased when  $x_i$  is negative (want to decrease  $\mathbf{w} \cdot \mathbf{x}$ )

# Perceptron Convergence Theorem

For any data set that is linearly separable, perceptron learning rule is guaranteed to find a solution in a finite number of iterations.

# Historical Context

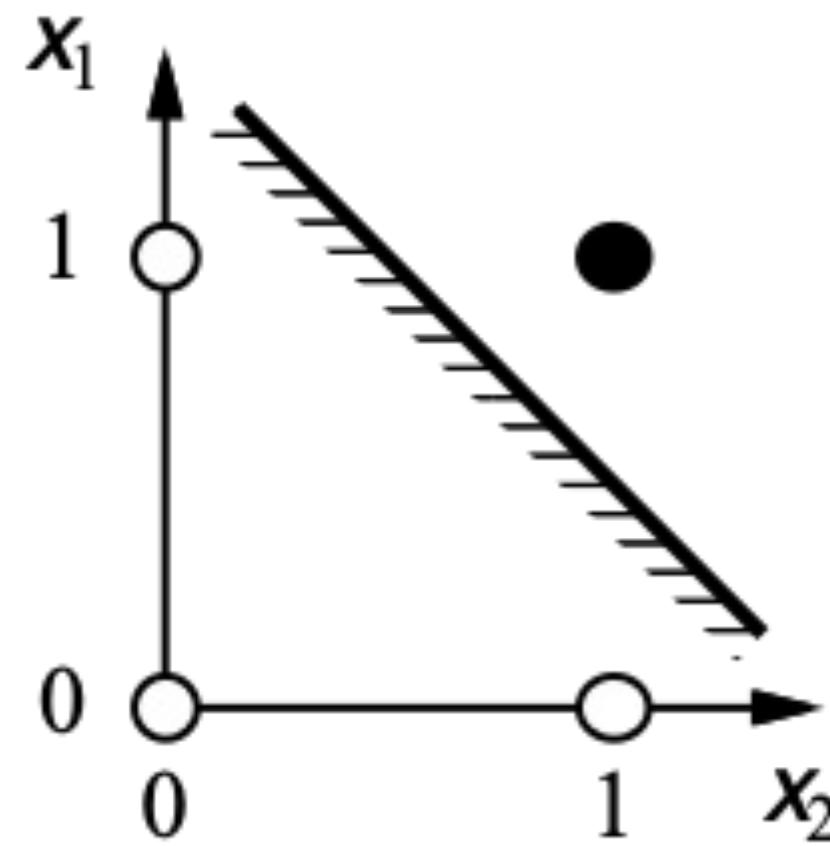
- In 1969, Minsky and Papert published book highlighting limitations of perceptrons
  - Funding agencies redirected funding away from neural network research,
  - preferring instead logic-based methods such as expert systems.
- Known since 1960's that any logical function could be implemented in a 2-layer neural network with step function activations.
- Problem was how to learn weights of multi-layer neural network from training examples
- Solution found in 1976 by Paul Werbos
  - not widely known until rediscovered in 1986 by Rumelhart, Hinton and Williams.

# Lecture Overview

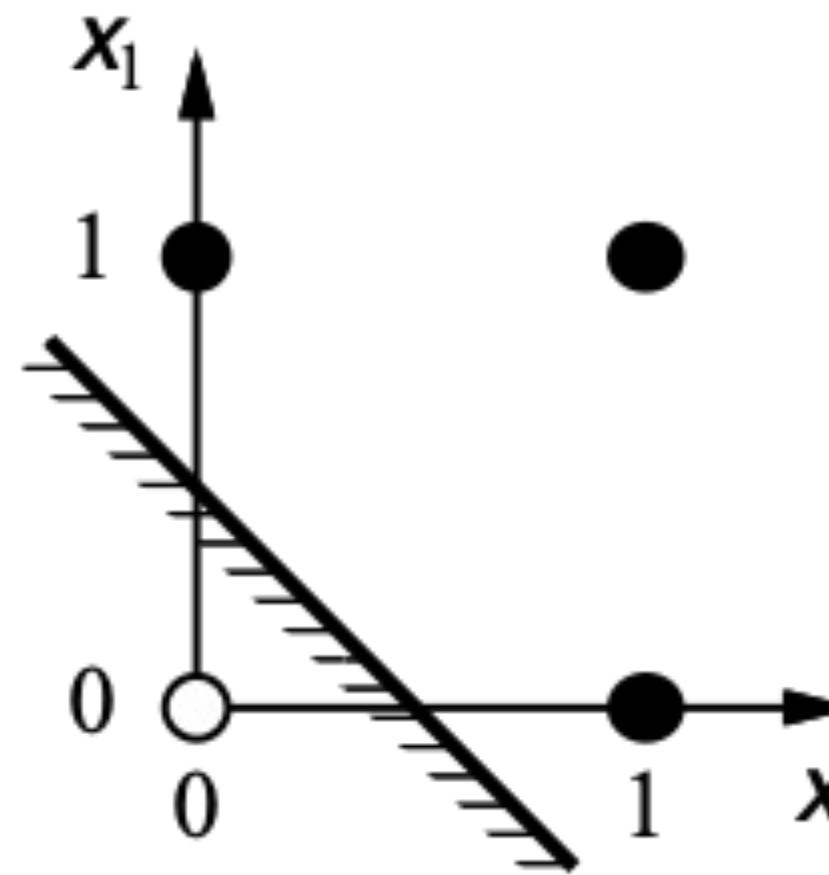
- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

# Limitations of Perceptrons

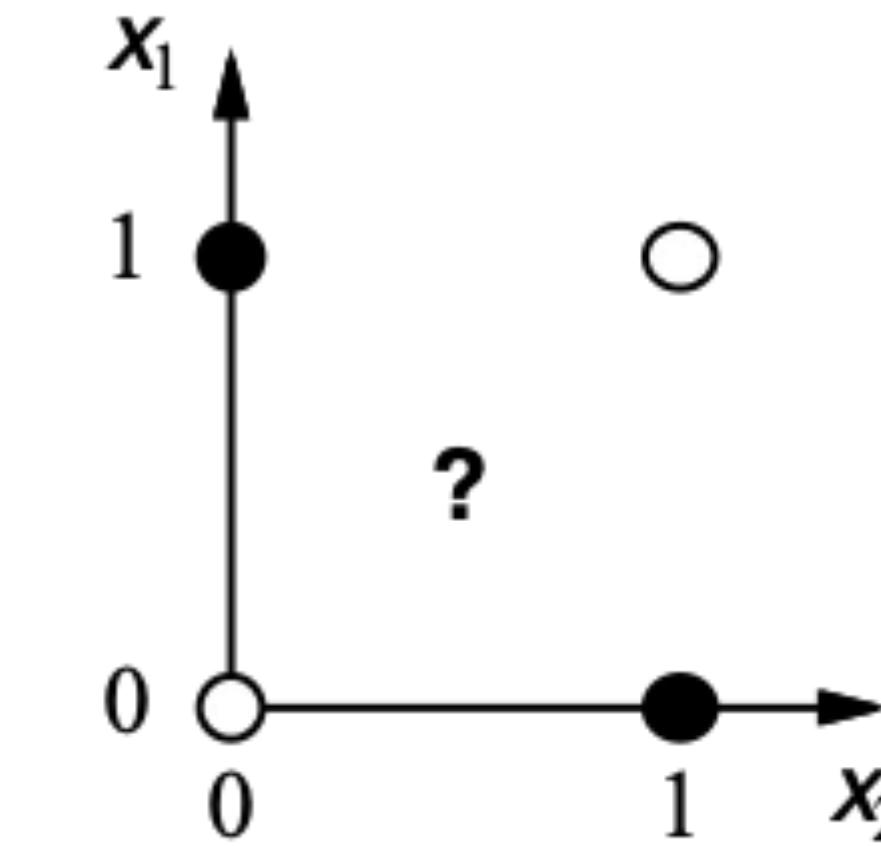
Problem: many useful functions are not linearly separable (e.g. XOR)



(a)  $x_1$  **and**  $x_2$



(b)  $x_1$  **or**  $x_2$



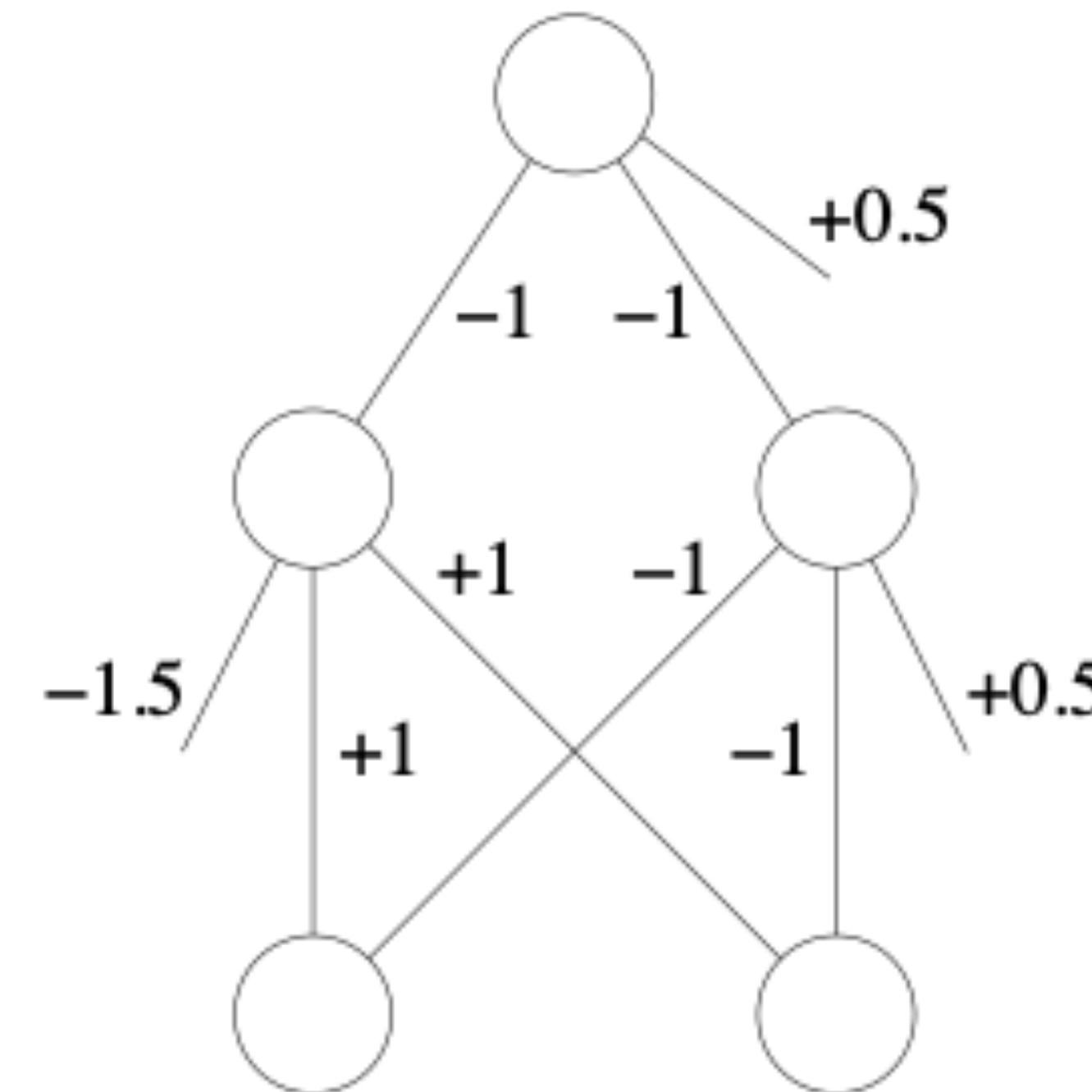
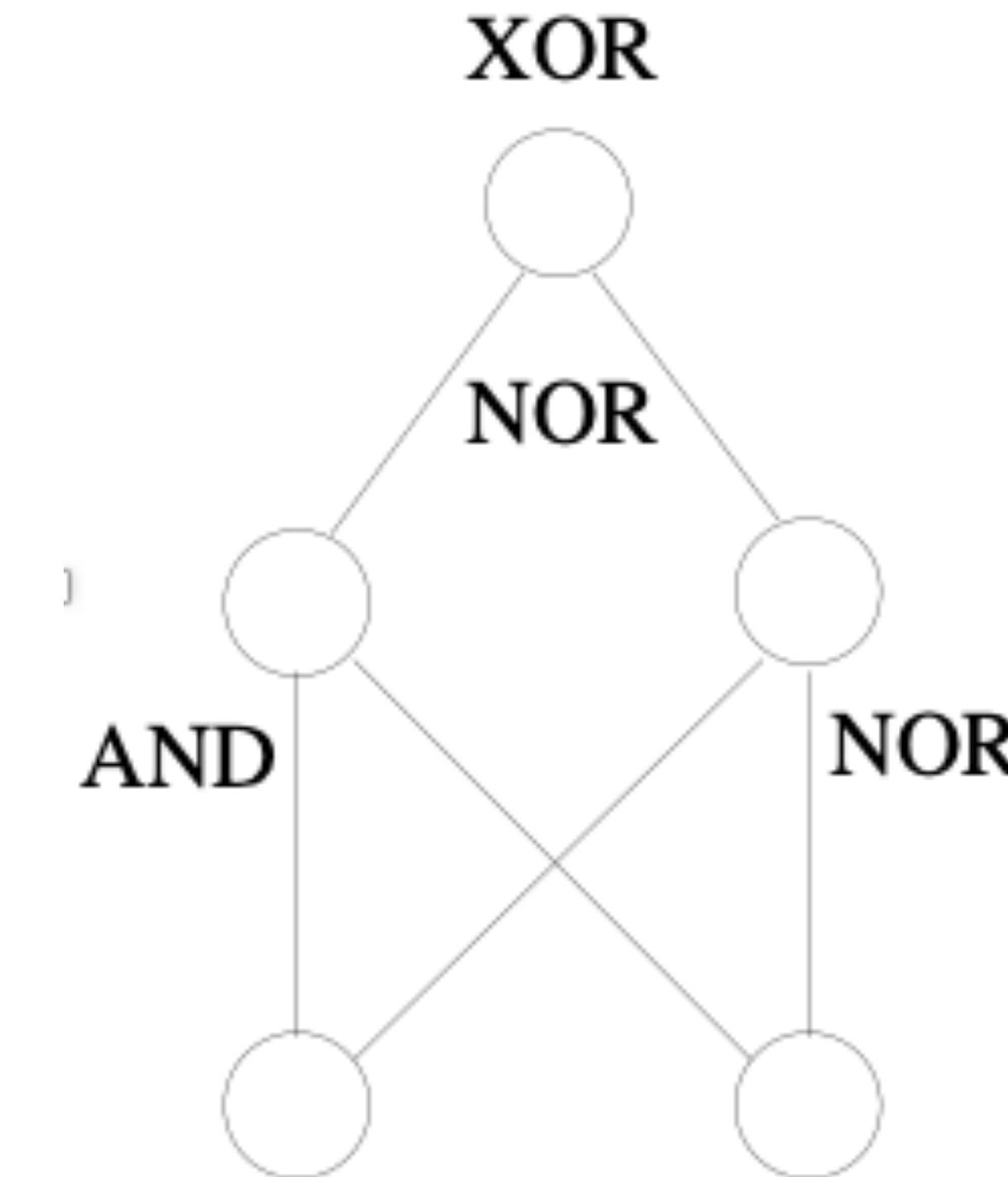
(c)  $x_1$  **xor**  $x_2$

Possible solution:

$x_1 \text{ XOR } x_2$  can be written as:  $(x_1 \text{ AND } x_2) \text{ NOR } (x_1 \text{ NOR } x_2)$

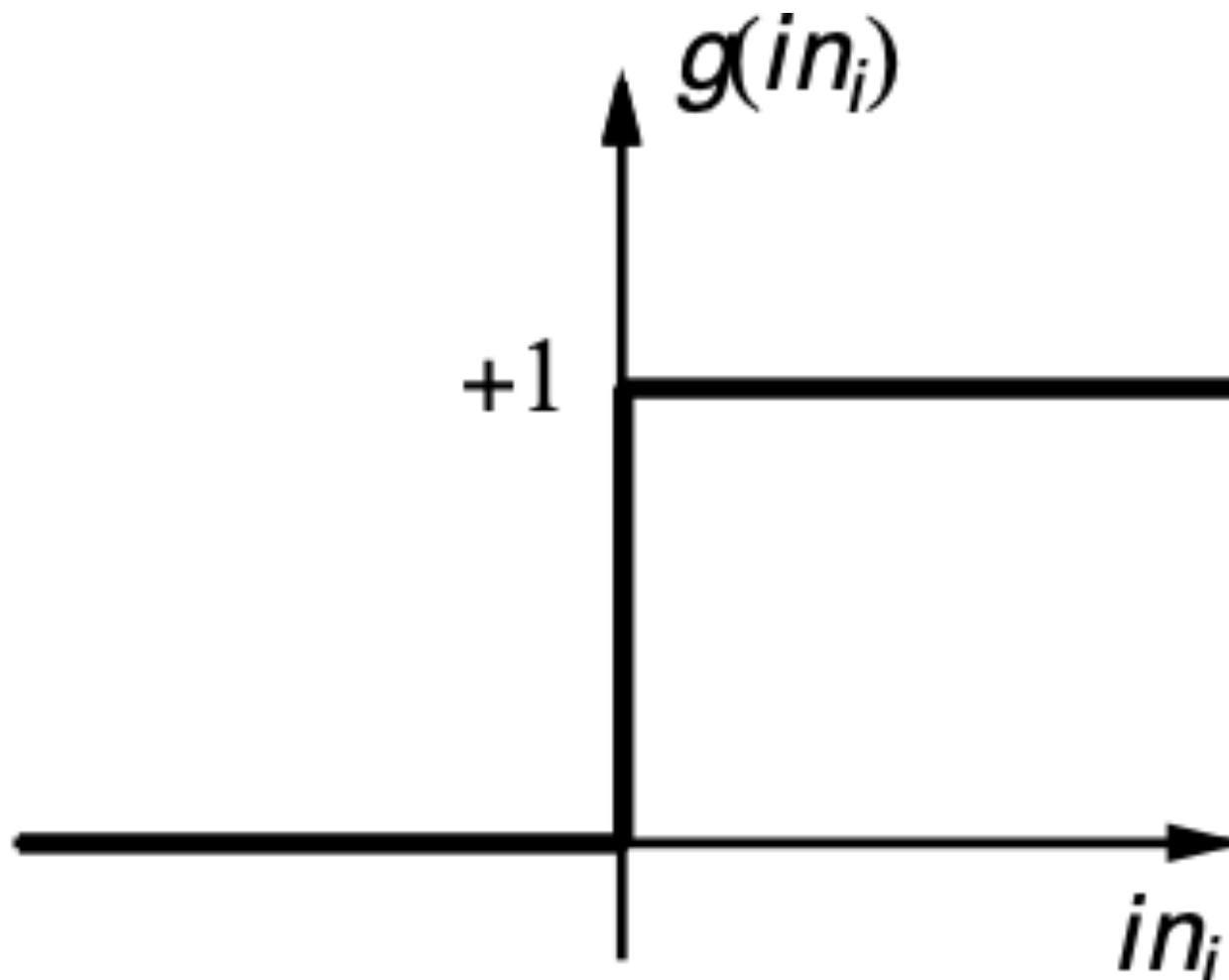
Recall that AND, OR and NOR can be implemented by perceptrons.

# Multi-Layer Neural Networks

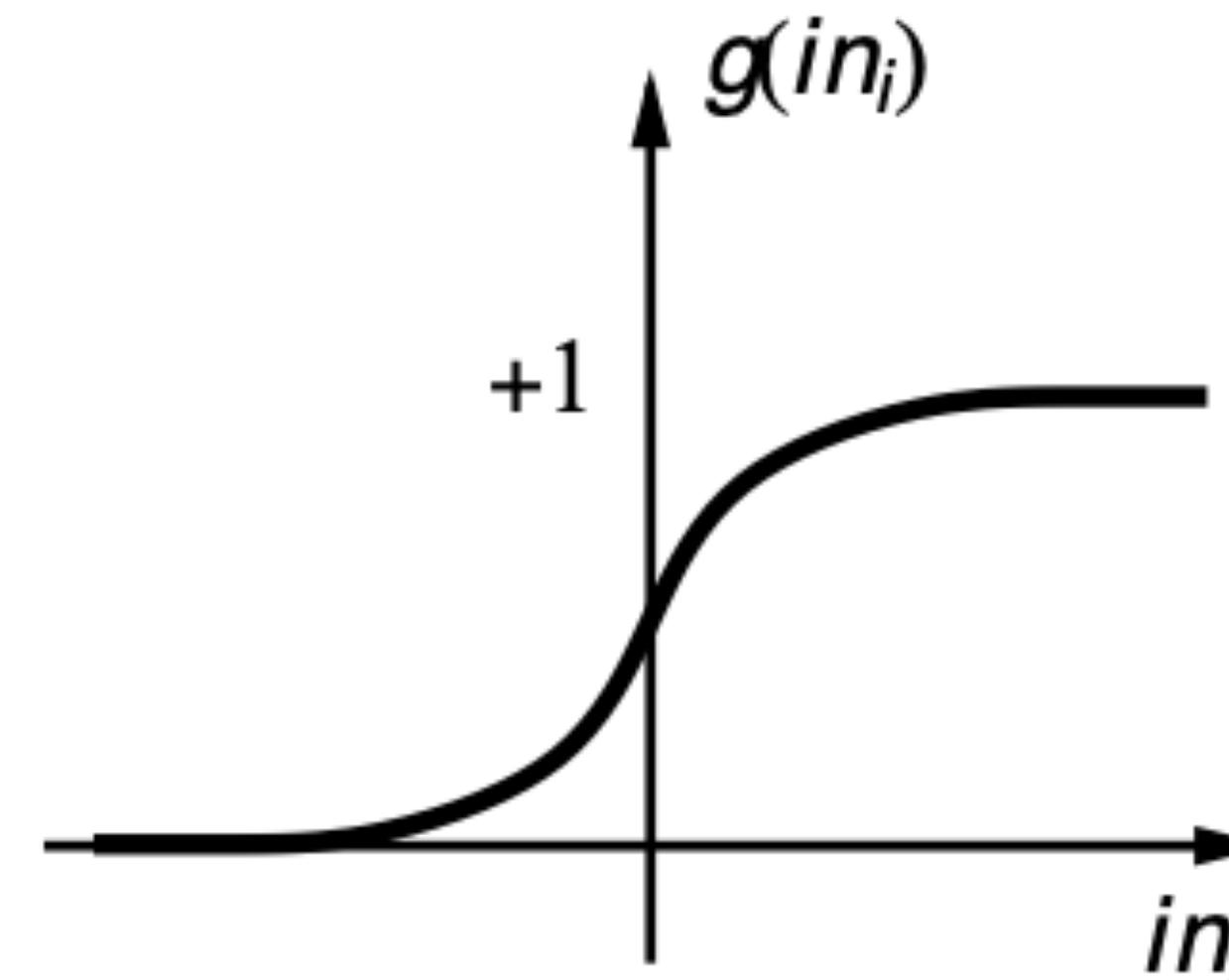


- Given an explicit logical function, we can design a multi-layer neural network by hand to compute that function.
- But, if we are just given a set of training data, can we train a multi-layer network to fit these data?

# Activation functions



(a)



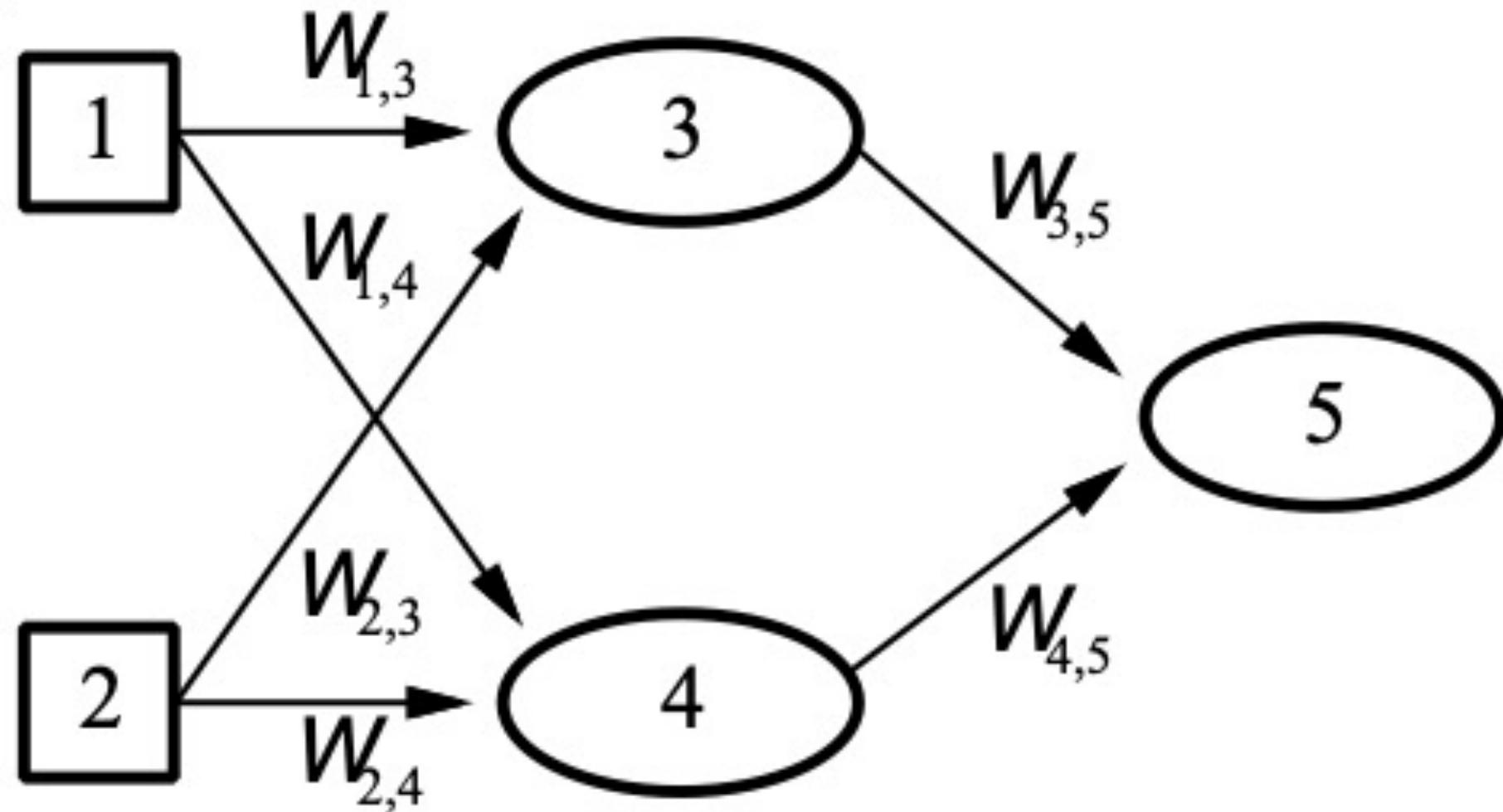
(b)

(a) is a **step function** or **threshold function**

(b) is a **sigmoid** function  $\frac{1}{1+e^{-x}}$

Changing the bias weight  $w_{0,i}$  moves the threshold

# Feed-Forward Example



$w_{i,j} \equiv$  weight between node  $i$  and node  $j$

- Feed-forward network = a parameterised family of nonlinear functions:  
$$a_5 = g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4)$$
$$= g\left(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)\right)$$
- Adjusting weights changes the function

# ANN Training as Cost Minimisation

- Define error function Mean Squared Error,  $E$

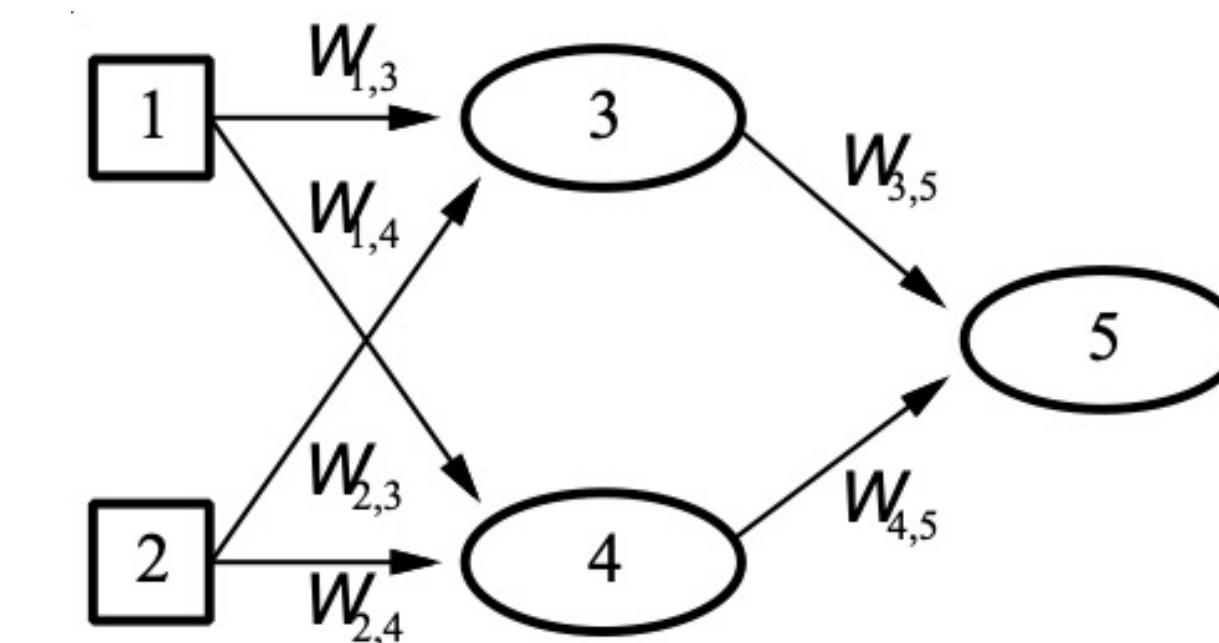
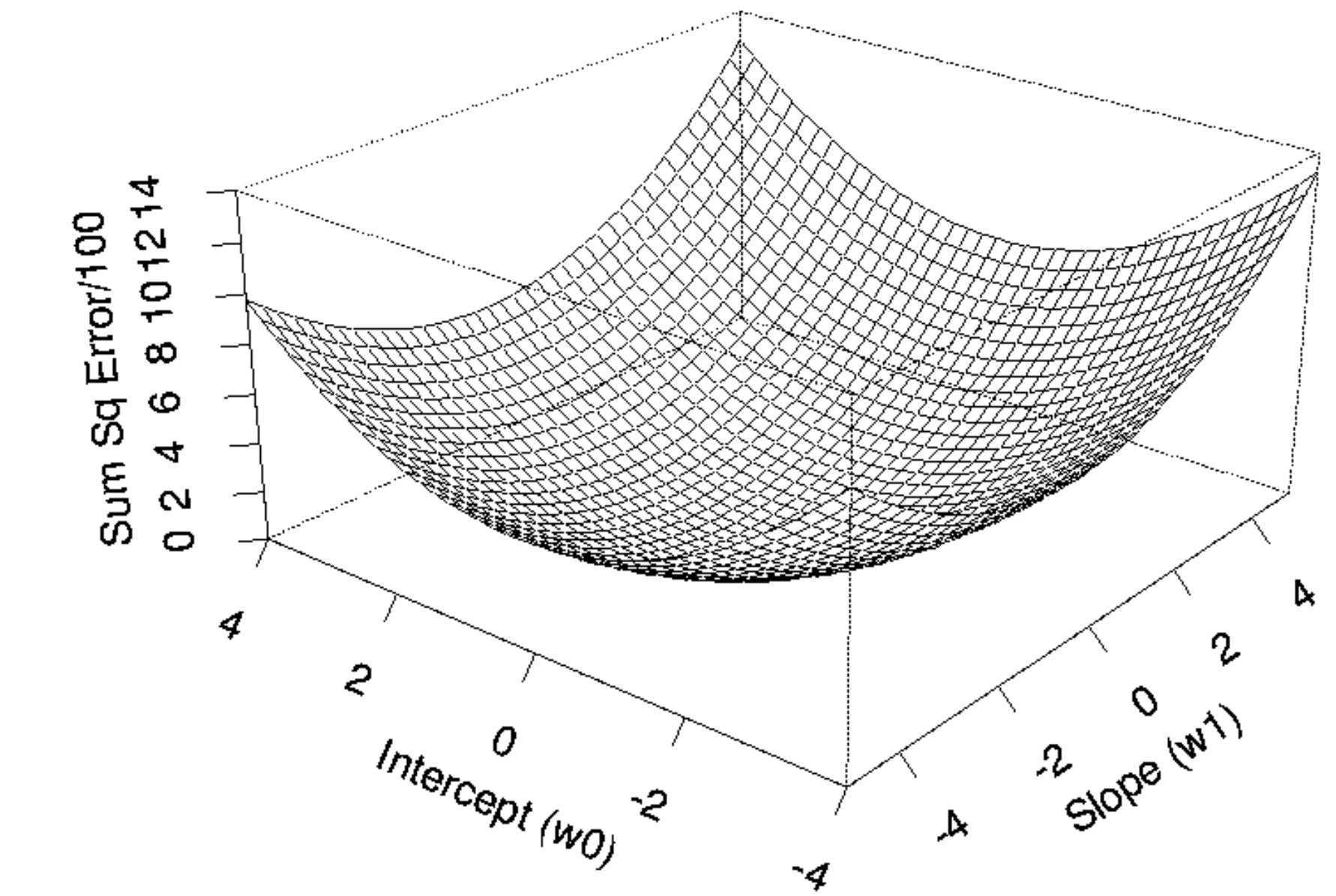
$$E = \frac{1}{2N} \sum (d - y)^2$$

$N$  = number of examples

$y$  = actual output

$d$  = desired output

- Think of  $E$  as height of error landscape of weight space.
- Aim to find a set of weights for which  $E$  is very low.

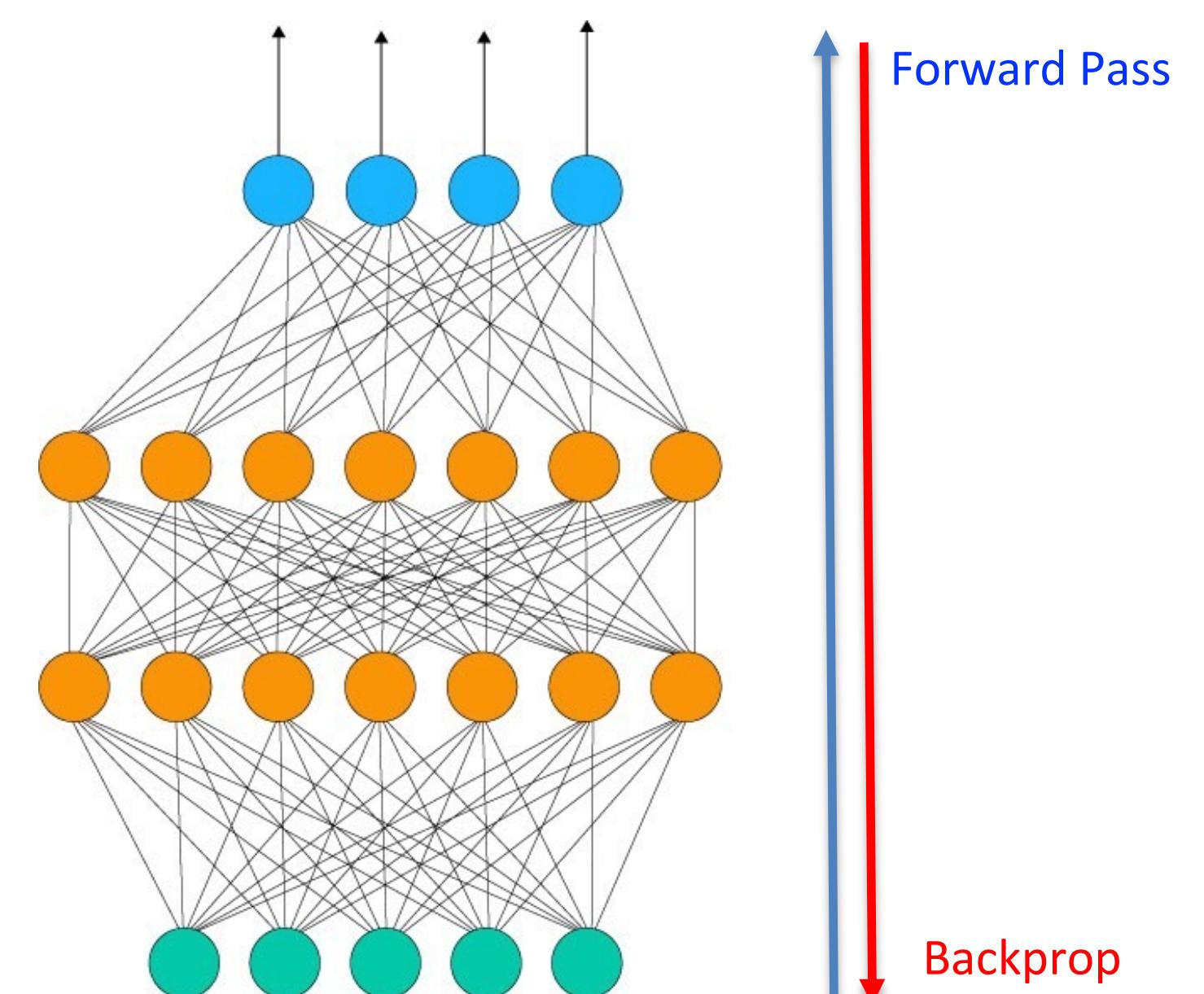


# Lecture Overview

- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

# Backpropagation

- 1. Forward pass:** apply inputs to “lowest layer” and feed activations forward to get output
- 2. Calculate error:** difference between desired output and actual output
- 3. Backward pass:** Propagate errors back through network to adjust weights



# Gradient Descent

$$E = \frac{1}{2N} \sum (d - y)^2$$

If transfer functions are smooth, can use multivariate calculus to adjust weights by taking steepest downhill direction.

$$w \leftarrow w + \eta \frac{\partial E}{\partial w}$$

Parameter  $\eta$  is the **learning rate**

- How the cost function affects the particular weight
- Find the weight

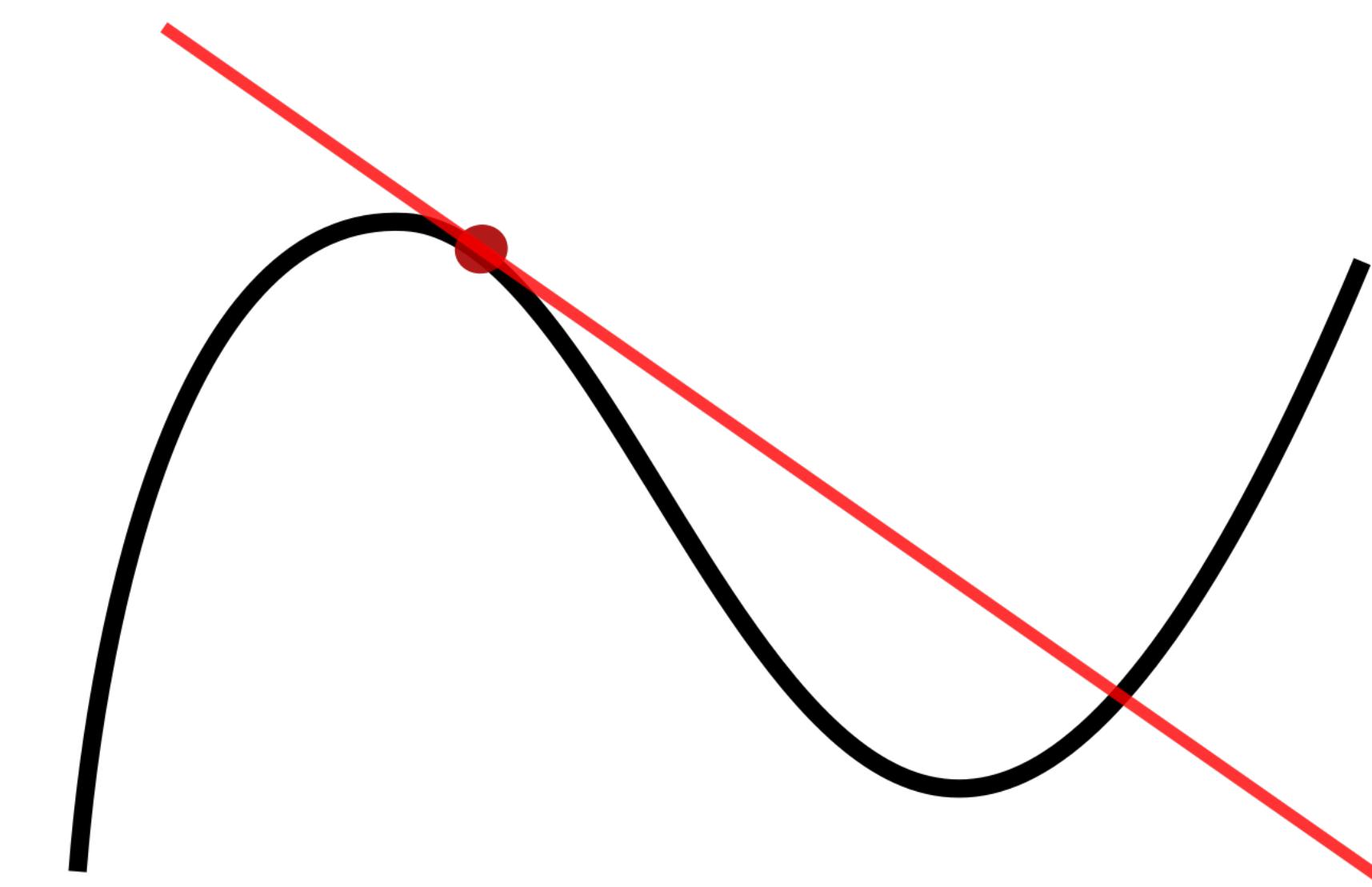
# Derivative of a Function

The derivative of a function is the slope of the tangent at a point

$$y = f(x) = mx + b$$

$$m = \frac{\text{change in } y}{\text{change in } x} = \frac{\Delta y}{\Delta x}$$

Written  $\frac{dy}{dx}$



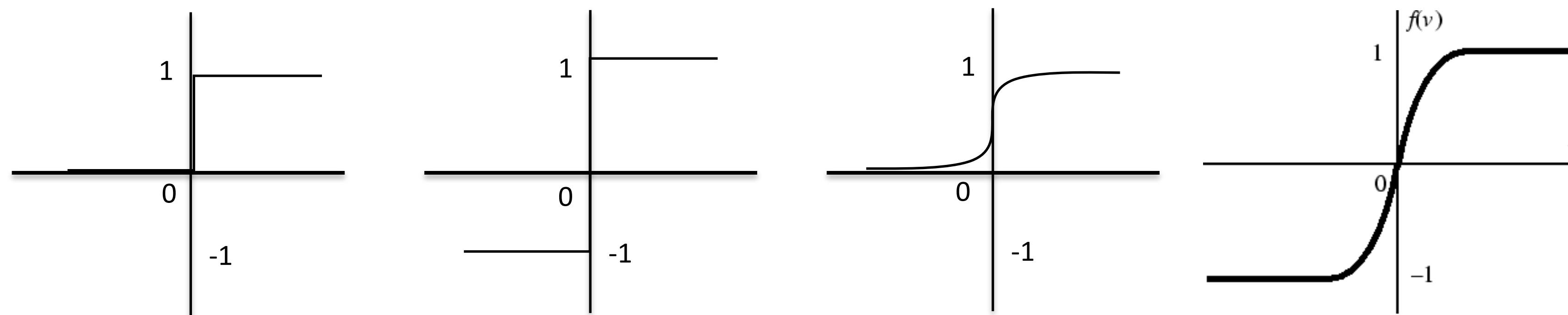
# Partial Derivative

Derivative of a function of several variables with respect to one of this variables

If  $z = f(x, y, \dots)$

Derivative with respect to  $x$  is written:  $\frac{\partial z}{\partial x}$

# Function must be continuous to be differentiable



Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

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

or hyperbolic tangent

$$g(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} = 2 \left( \frac{1}{1 + e^{-s}} \right) - 1 \quad (-1 \text{ to } 1)$$

# Chain Rule

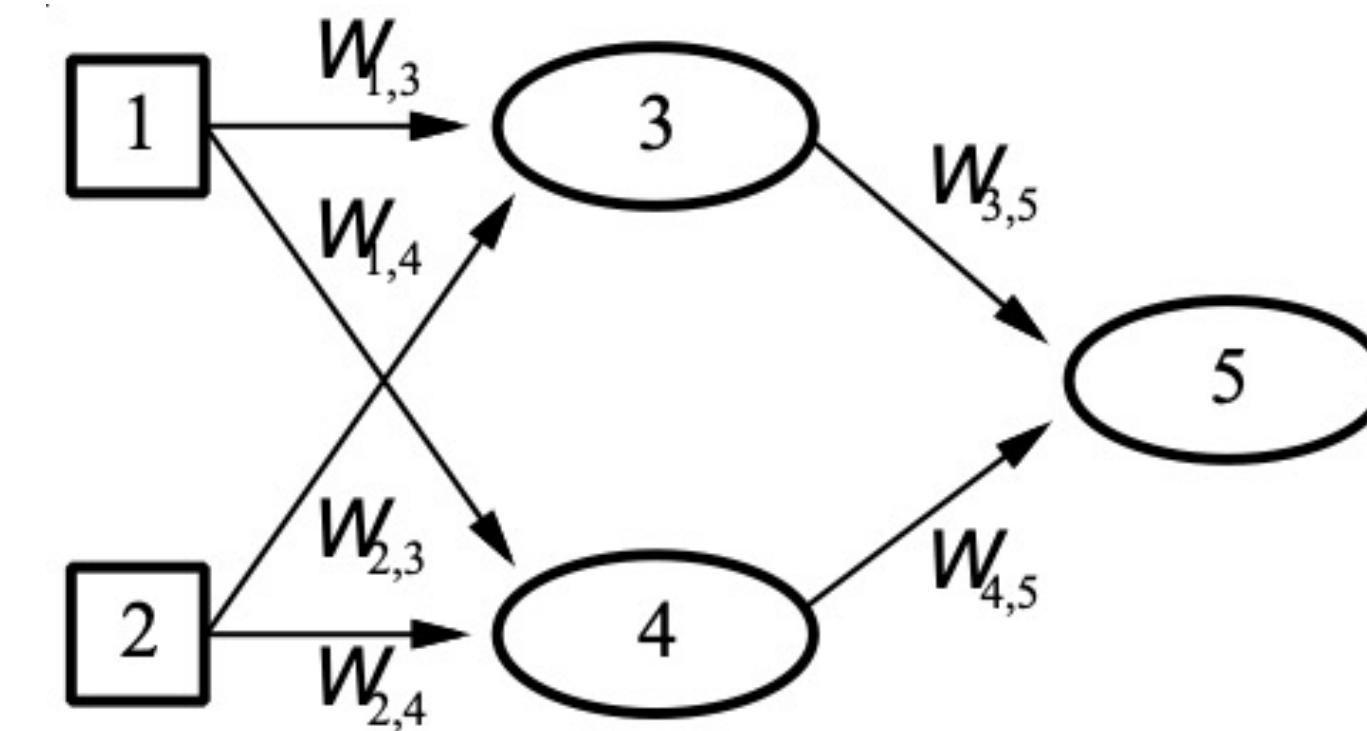
$$g \left( W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2) \right)$$

If

$$y = g(f(x)) \equiv \begin{cases} y = g(u) \\ u = f(x) \end{cases}$$

then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$



Used chain rule to compute partial derivatives efficiently.

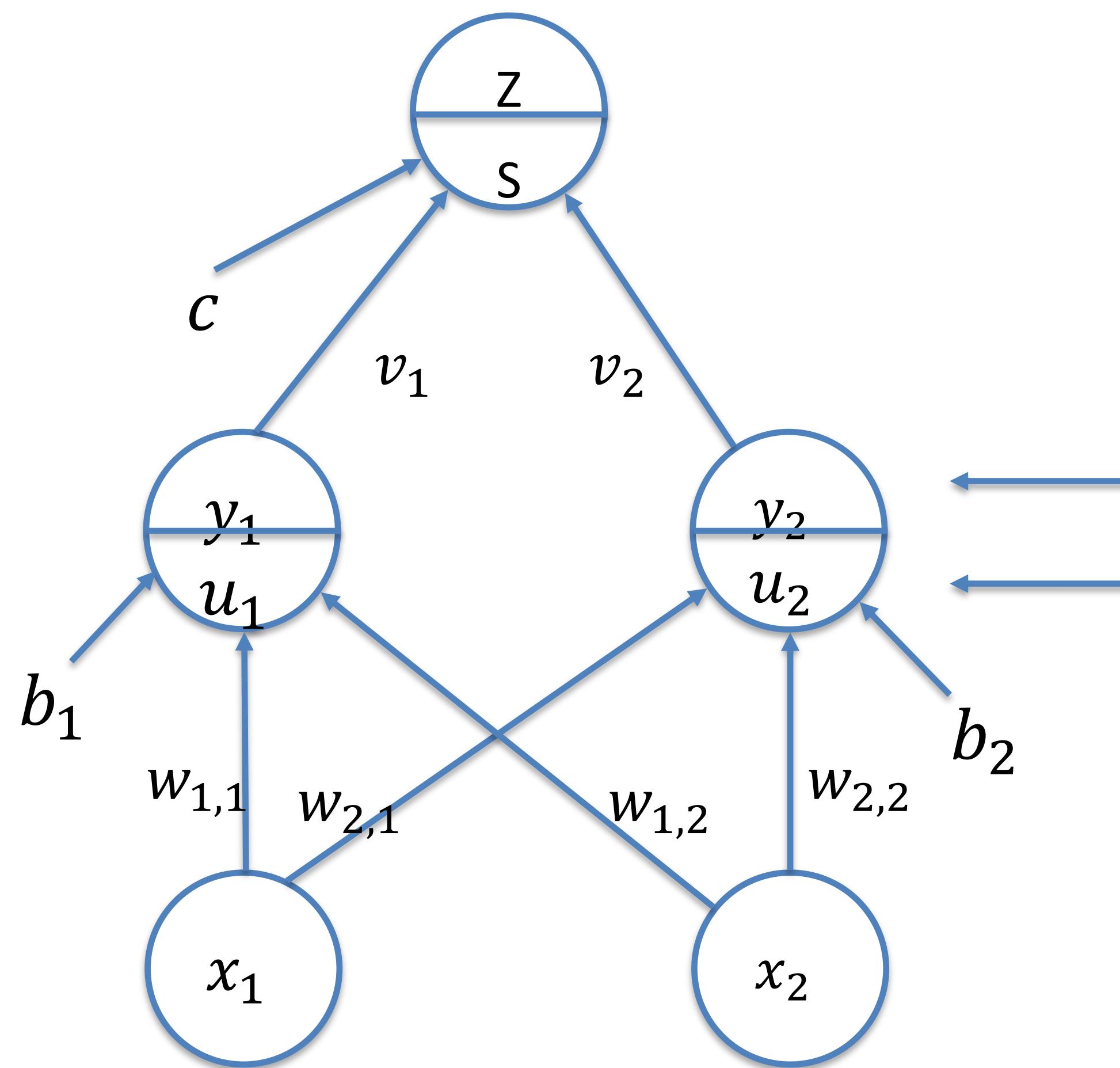
Transfer function must be differentiable (usually sigmoid, or tanh).

Note: if  $z(s) = \frac{1}{1+e^{-s}}$  derivative  $\boxed{z'(s) = z(1 - z)}$

Slope of error function

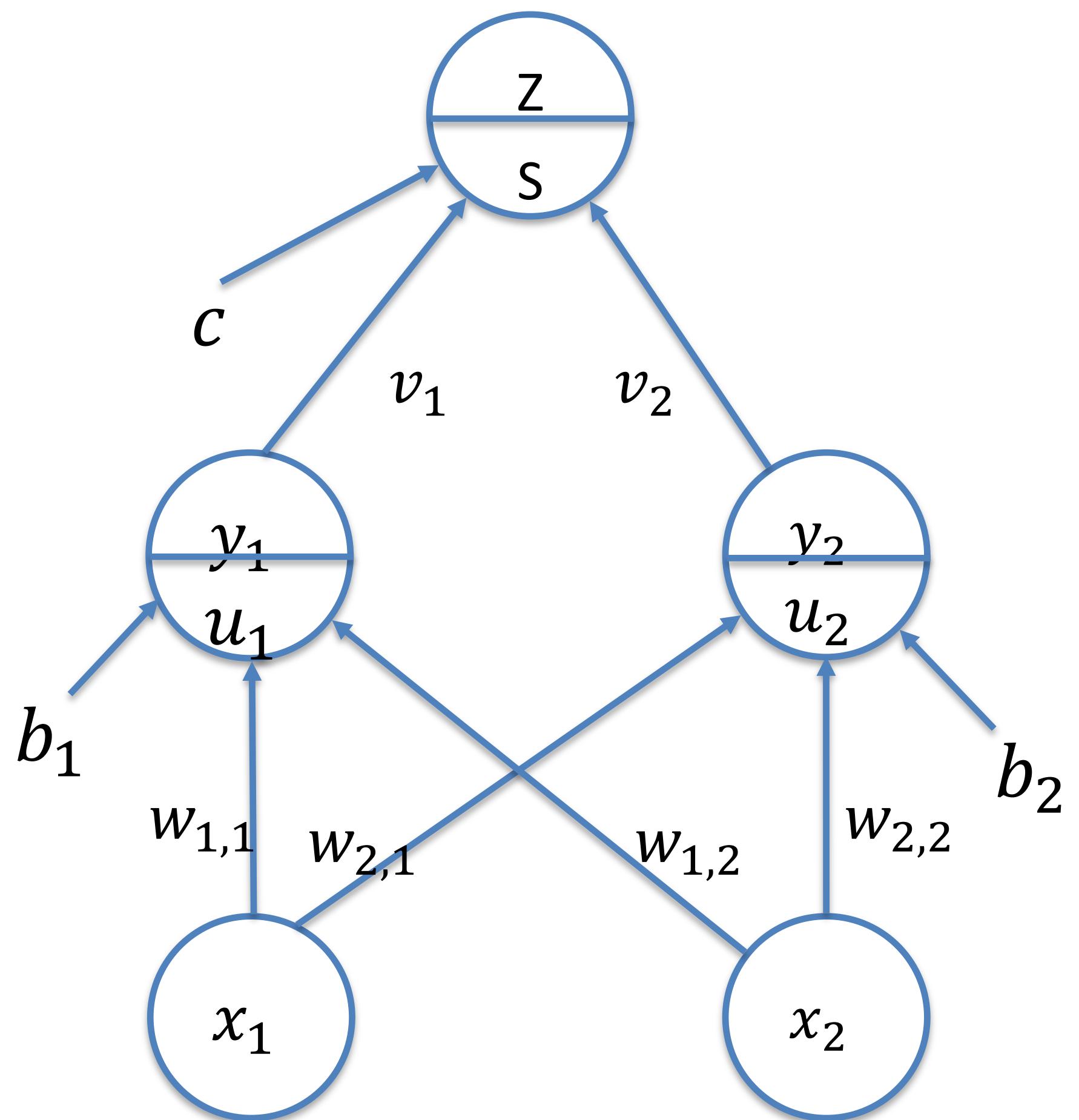
if  $z(s) = \tanh(s)$  derivative  $z'(s) = 1 - z^2$

# Backpropagation



Convenient to split layers into:  
*sigmoid layer* and  
*linear complete layer*

# Forward Pass



$$u_1 = b_1 + w_{1,1}x_1 + w_{1,2}x_2$$

$$y_1 = g(u_1)$$

$$s = c + v_1y_1 + v_2y_2$$

$$z = g(s)$$

$$E = \frac{1}{2} \sum (z - t)^2 \quad (t = \text{target})$$

# Backpropagation Algorithm

```
procedure BackpropLearner( $X_s, Y_s, E_s, layers, \eta$ )  
    repeat  
        for each example  $e$  in  $E_s$  in random order do  
             $values[i] \leftarrow X_i(e)$  for each input unit  $i$   
            for each layer from lowest to highest do  
                 $values \leftarrow OutputValues(layer, values)$   
                 $error \leftarrow SumSqError(Y_s(e), values)$   
            for each layer from highest to lowest do  
                 $error \leftarrow Backprop(layer, error)$   
    until termination
```

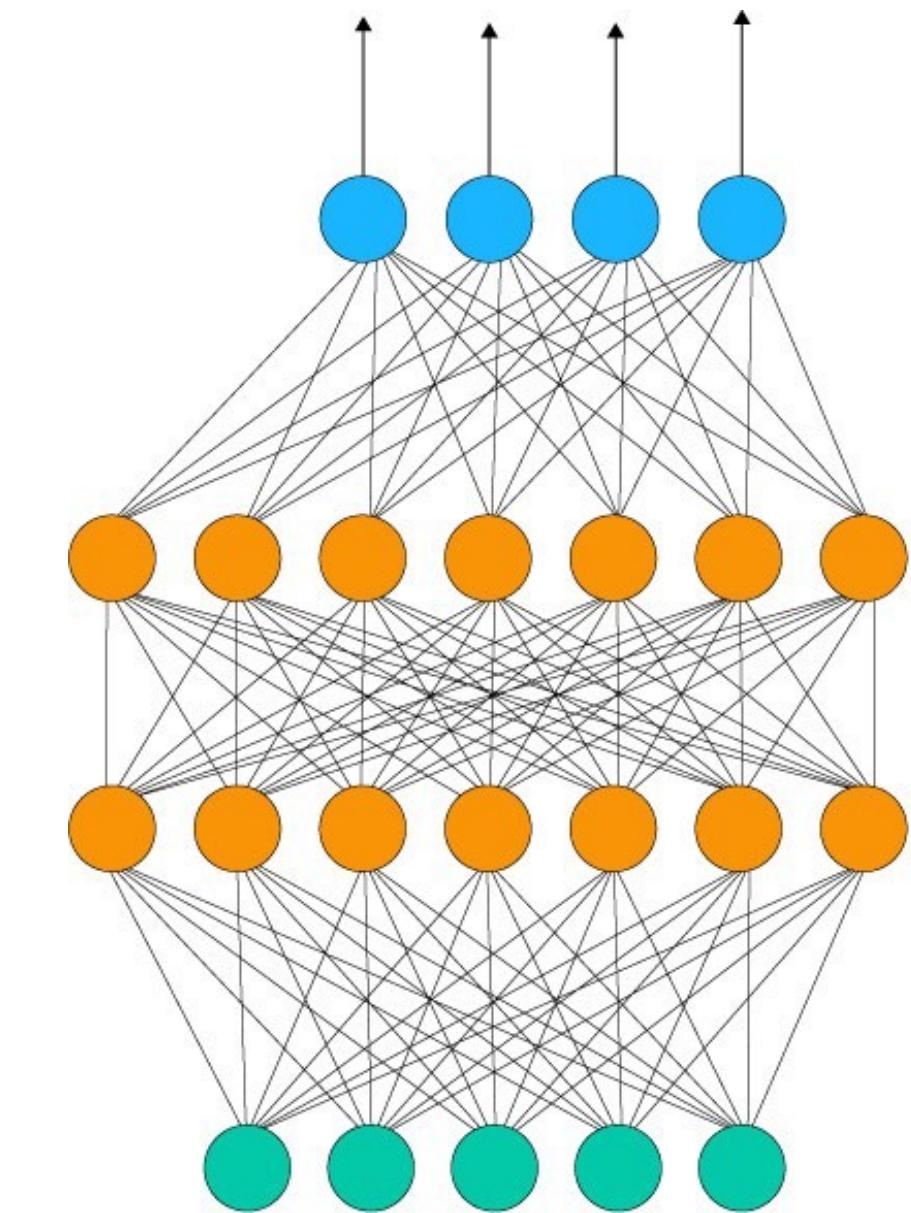
$X_s$ : set of input features,  $X_s = \{X_1, \dots, X_n\}$

$Y_s$ : target features

$E_s$ : set of examples from which to learn

$layers$ : a sequence of layers

$\eta$ : learning rate (gradient descent step size)



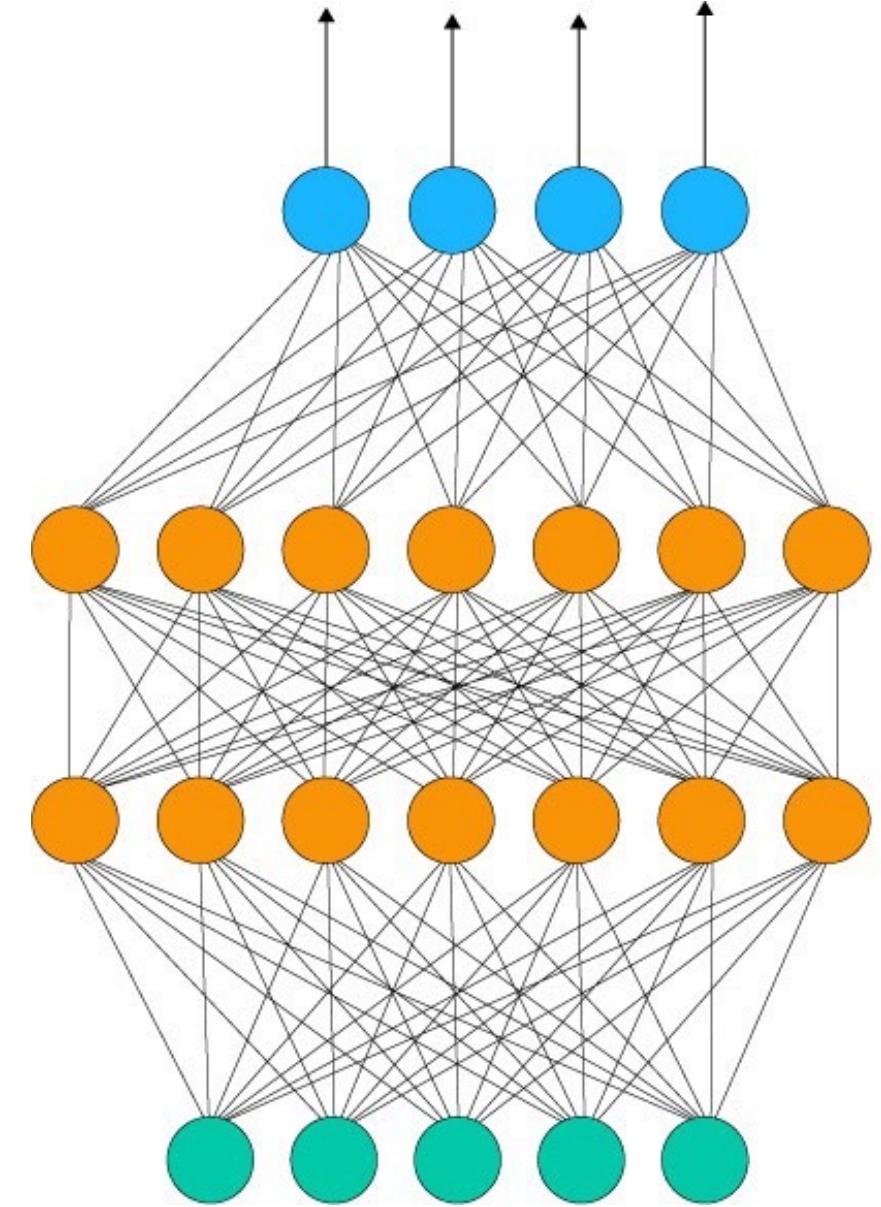
# Backpropagation Algorithm

```
function  $g(x) = 1/(1 + e^{-x})$ 
```

```
function  $SumSqError(Ys, predicted) = [\frac{1}{2} \sum (Ys[j] - predicted[j])^2 \text{ for each output unit } j]$ 
```

```
function OutputValues(layer, input) //  $input$  is array with length  $n_i$ 
    define  $input[n]$  to be 1
     $output[j] \leftarrow g(\sum_{i=0}^n w_{ij} \cdot input[i]) \text{ for each } j$  // update output for layer
    return  $output$ 
```

```
function Backprop(layer, error) // each layer has an input and output array
     $delta[i] \leftarrow output[i] \cdot (1 - output[i]) \cdot error[i] \text{ for each } i$ 
     $w_{ji} \leftarrow w_{ji} + \eta \cdot delta[j] \cdot input[i] \text{ for each } i, j, \text{ for learning rate } \eta$  // perceptron rule
     $delta[i] \leftarrow \sum_j w_{ji} \cdot delta[j] \text{ for each } i$ 
    return  $delta$ 
```



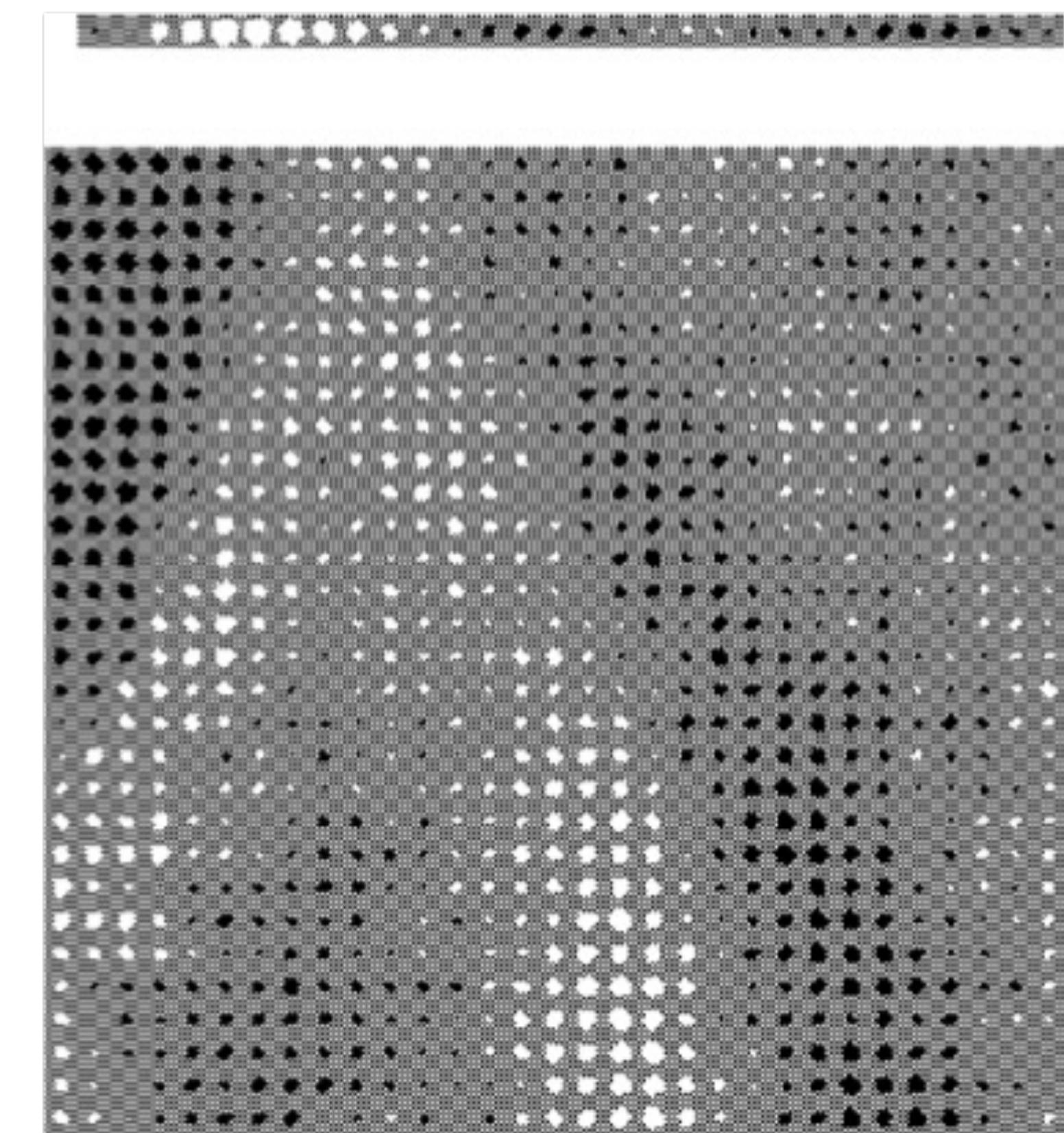
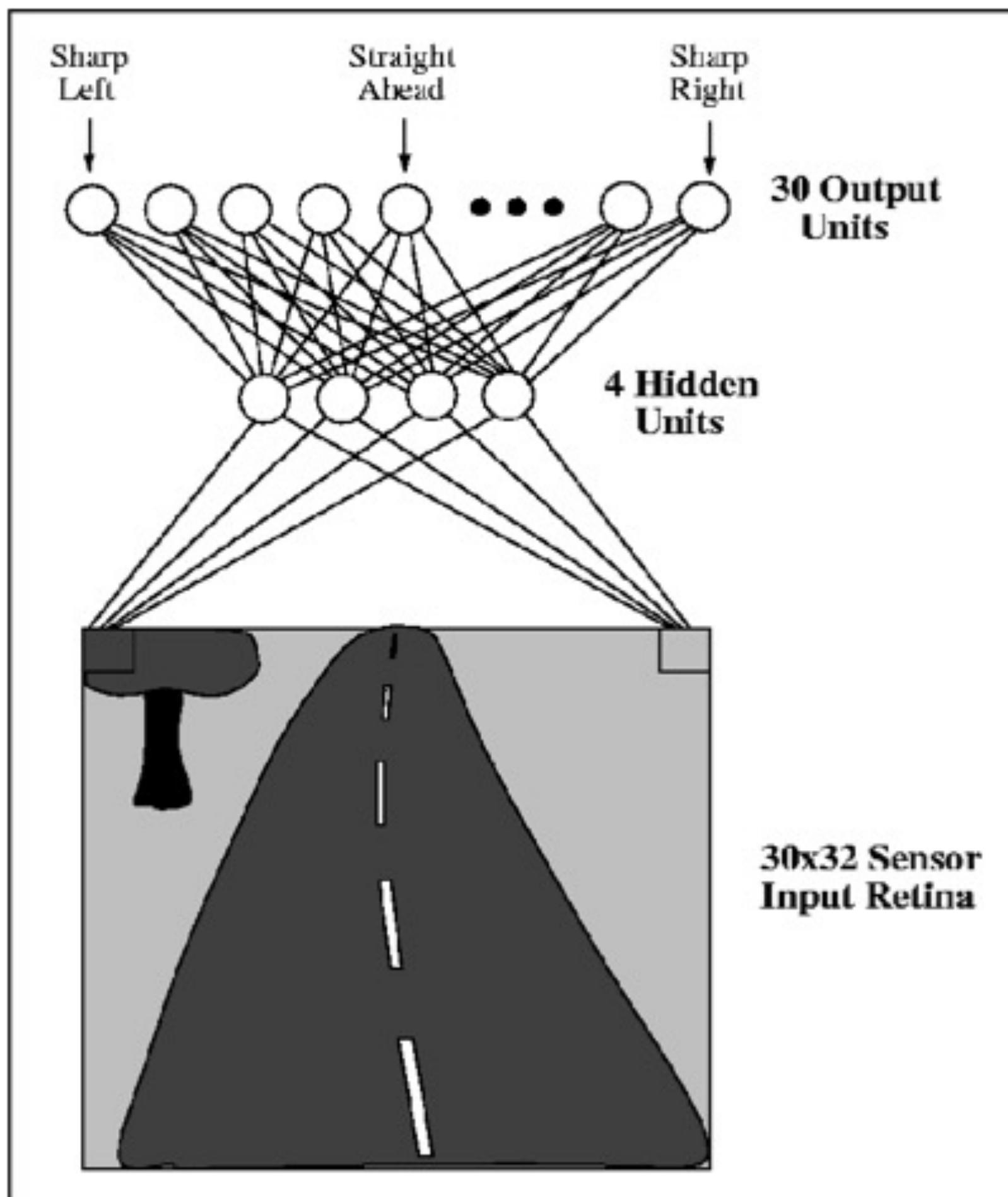
# Lecture Overview

- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Back propagation
- Applications of neural networks

# Neural Network – Applications

- Object Recognition
- Autonomous Driving
- Language Models
- Speech Recognition
- Handwriting Recognition
- Game Playing
- Credit Card Fraud Detection
- Financial Prediction

# ALVINN (First demo of long distance autonomous driving)



# ALVINN

- Autonomous Land Vehicle In a Neural Network
- later version included a sonar range finder
  - 8×32 rangefinder input retina
  - 29 hidden units
  - 45 output units
- Supervised Learning, from human actions (Behavioural Cloning)
  - additional “transformed” training items to cover emergency situations
- Drove (mostly) autonomously from coast to coast in USA

# Training Tips

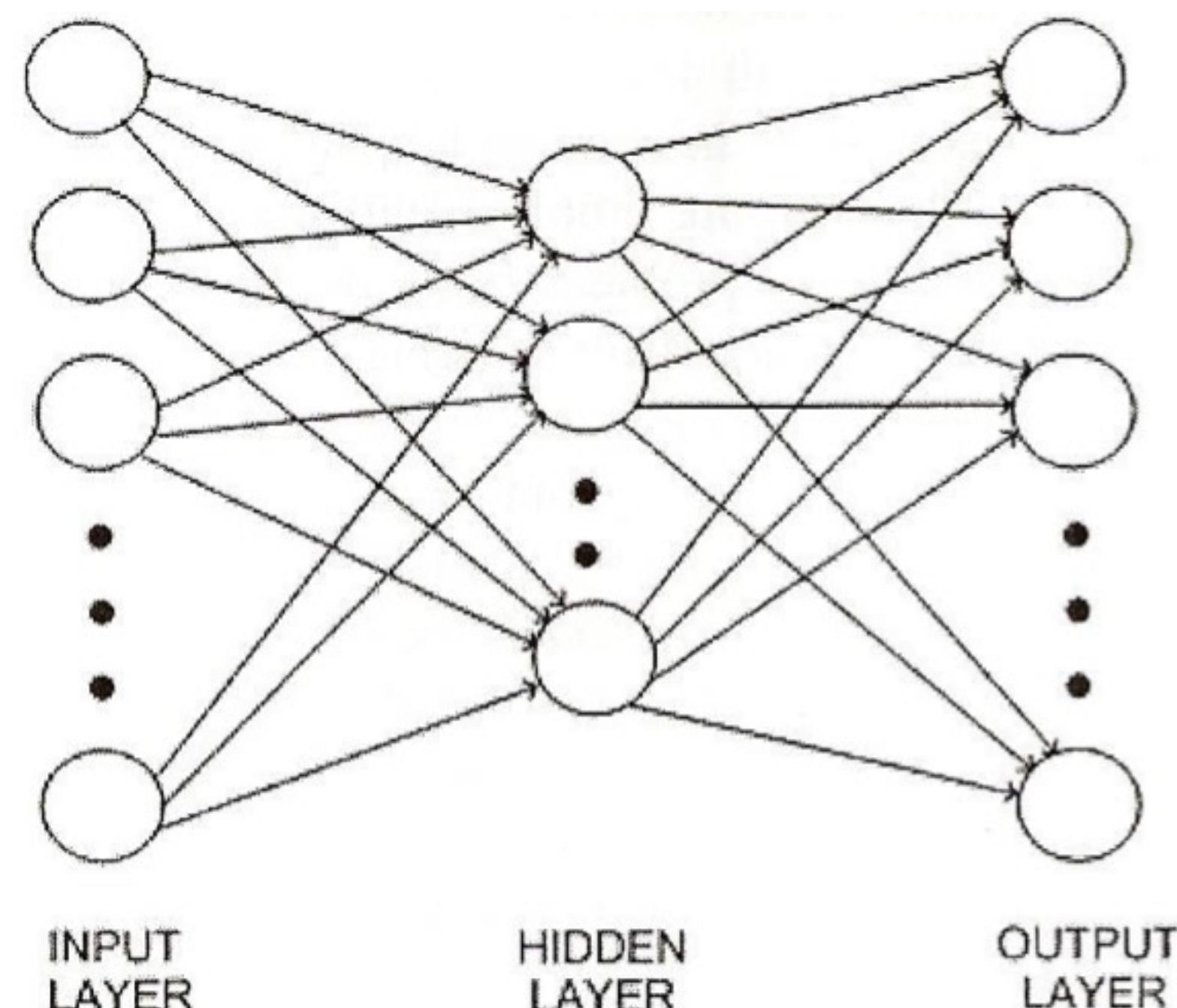
- Re-scale inputs and outputs to be in the range 0 to 1 or –1 to 1
- Initialise weights to very small random values
- On-line or batch learning
- Three different ways to prevent overfitting:
  - limit the number of hidden nodes or connections
  - limit the training time, using a validation set
  - weight decay
- Adjust the parameters: learning rate (and momentum) to suit the particular task

# Neural Network Structure

Two main network structures

1. Feed-Forward  
Network

2. Recurrent  
Network

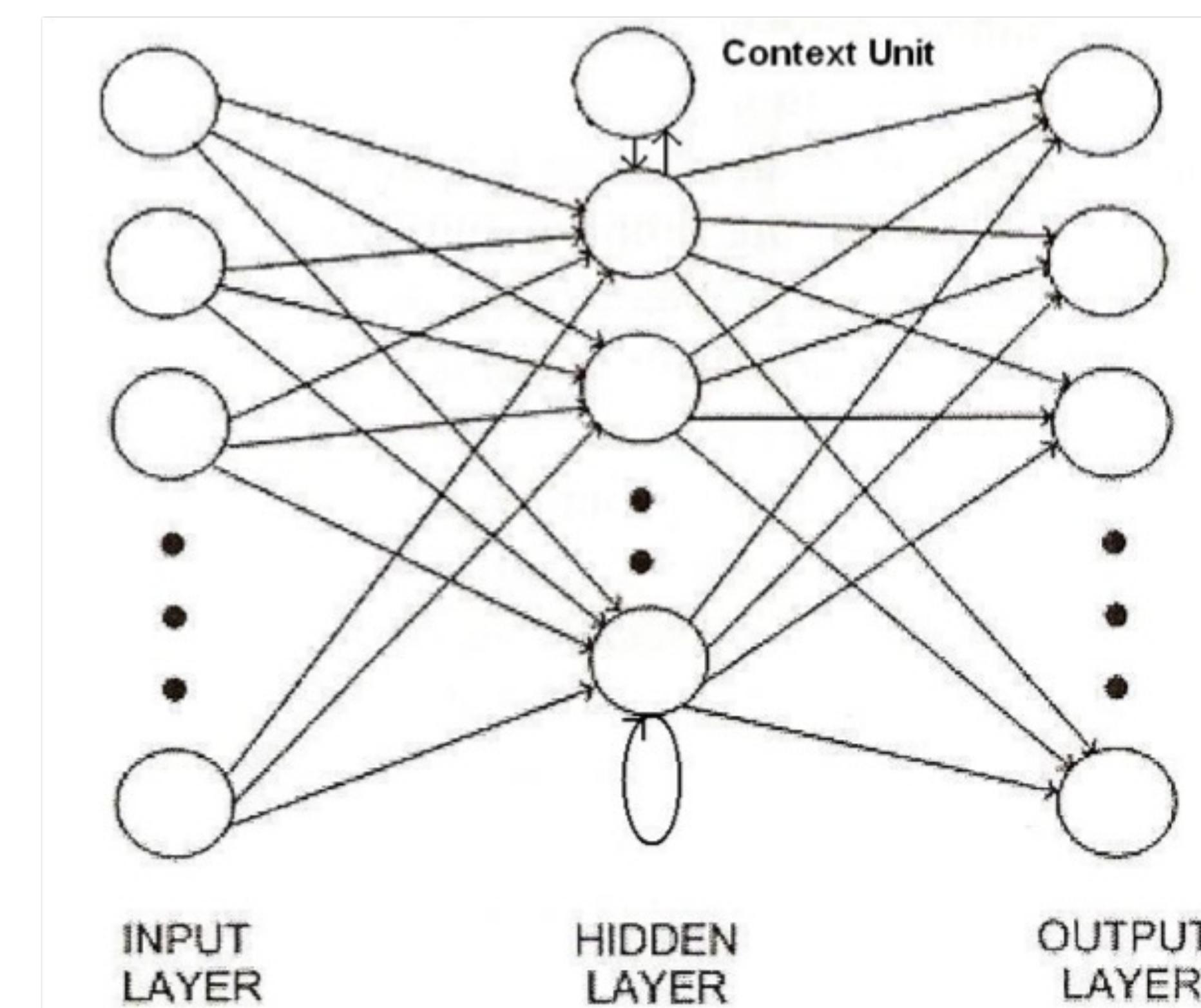


# Neural Network Structure

Two main network structures

1. Feed-Forward  
Network

2. Recurrent  
Network



# Neural Network Structures

**Feed-forward network** has connections only in one direction

- Every node receives input from “upstream” nodes; delivers output to “downstream” nodes
  - no loops.
- Represents a function of its current input
  - has no internal state other than the weights themselves.

**Recurrent network** feeds outputs back into its own inputs

- Activation levels of network form a dynamical system
  - may reach a stable state or exhibit oscillations or even chaotic behaviour
- Response of network to an input depends on its initial state
  - which may depend on previous inputs.
- Can support short-term memory

# Neural Networks

- Multiple layers form a hierarchical model, known as **deep learning**
  - **Convolutional neural networks** are specialised for vision tasks
  - **Recurrent neural networks** are used for time series
- Typical real-world network can have 10 to 20 layers with hundreds of millions of weights
  - can take hours, days, months to learn on machines with thousands of cores

# Summary

- Vector-valued inputs and outputs
- Multi-layer networks can learn non-linearly separable functions
- Hidden layers learn intermediate representation
  - ◆ How many to use?
- Prediction – Forward propagation
- Gradient descent (Back-propagation)
  - ◆ Local minima problems
- Kernel trick can be introduced through a deep belief network

# References

- Poole & Mackworth, Artificial Intelligence: Foundations of Computational Agents, Chapter 7.
- Russell & Norvig, *Artificial Intelligence: a Modern Approach*, Chapters 18.6, 18.7 .