

# Week 11 – Introduction to Neural Networks

Advanced Mechatronics System Design – MANU2451

Dr Chow Yin LAI

Edited by Dr Milan Simic

School of Engineering

RMIT University, Victoria, Australia

Email: [milan.simic@rmit.edu.au](mailto:milan.simic@rmit.edu.au)

# New Teaching Schedule

Week		Class Activity Before	Lecture	Team Activity During or After
1			Introduction to the Course / Introduction to LabVIEW	LabVIEW Programming
2			Introduction to LabVIEW / Data Acquisition	LabVIEW Programming
3			Gripper / Introduction to Solidworks / Safety	Gripper Design
4			Sensors I	myRIO Programming for Sensor Signal Reading / Gripper Design
5			Sensors II	myRIO Programming for Sensor Signal Reading
6			Actuators I	LabVIEW Tutorial
7		LabVIEW Assessment.	DC Motors I	Matlab Simulink Simulation
8		Design report submission	DC Motors II	Matlab Simulink Simulation / myRIO Programming for Control
9			Actuators II	Matlab Simulink Simulation
10			Modeling and System Identification	Matlab Simulink Simulation LabVIEW Simulation
11			Artificial Intelligence I	Matlab Simulation LabVIEW Simulation
12			Artificial Intelligent II	Revision

# Contents

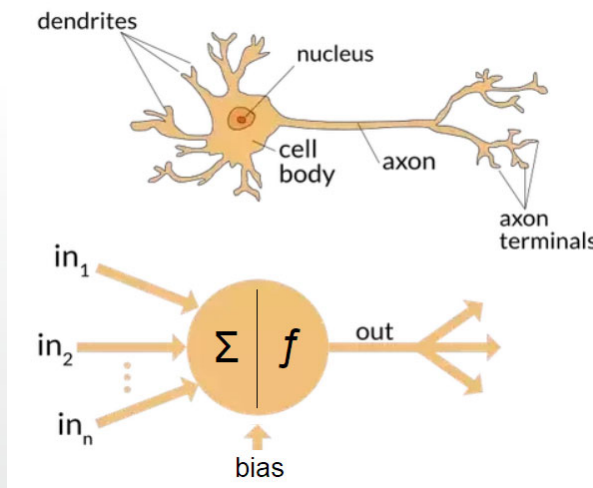
- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- Deep Neural Networks

# Contents

- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- Deep Neural Networks

# What is a Neural Network (NN)?

- A neural network is a **massively parallel distributed** multiprocessing machine that has a natural propensity for:
  - storing experimental knowledge, and
  - making it available for use.
- It is similar to the brain in two ways:
  - **Knowledge is acquired** by the network through a **learning process**.
  - **Knowledge is stored** using interneuron connection strengths known as **synaptic weights**.

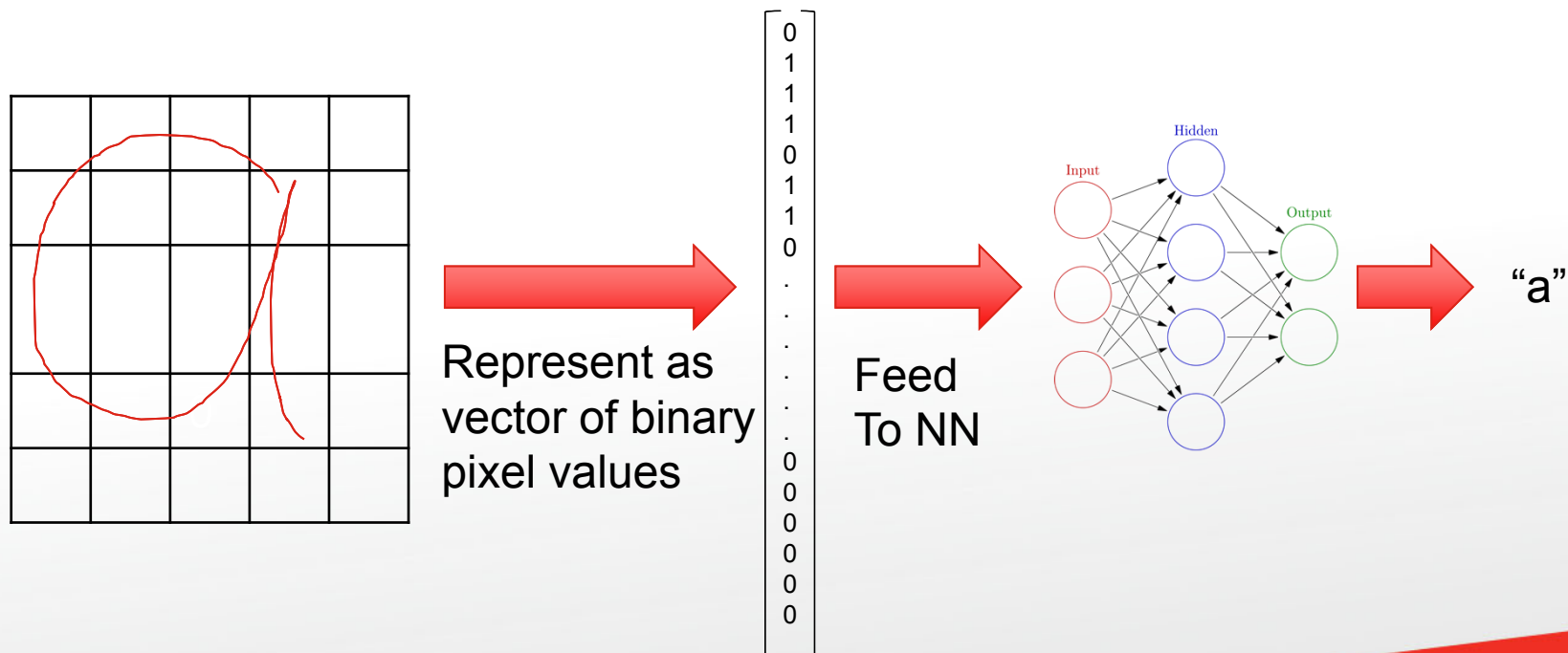


## Biological vs Artificial NN

<https://www.quora.com/What-is-the-differences-between-artificial-neural-network-computer-science-and-biological-neural-network>

# Applications of NNs

- NNs are mainly used for two types of applications:
- 1. Pattern Recognition or Classification
  - Example: Text recognition – Classify a handwritten alphabet as one of the 26 lower case letters.



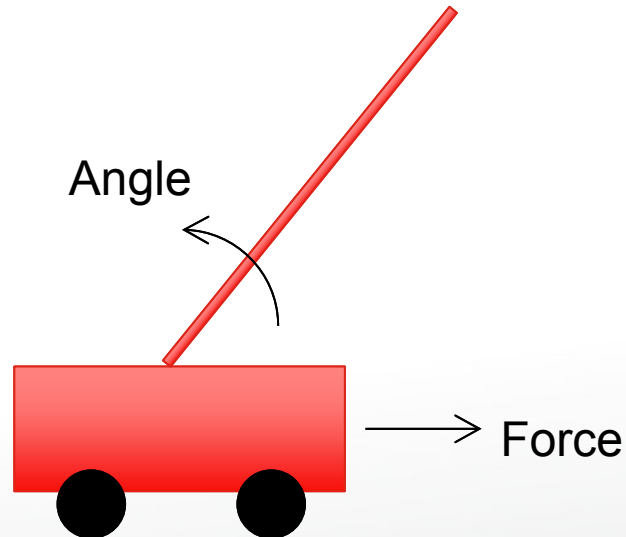
# Applications of NNs

- 2. Regression or Function Approximation

- Example: To understand the input-output relationship of an inverted pendulum (Input = Force, Output = Angle).

- $\text{Angle} = \text{function}(\text{Force})$

Learn by NN  
automatically



- Note: Last week we learnt about modelling, but we had to provide some structure, plus the functions were linear. Here, NN can learn the relationship by itself, even if the function is nonlinear.

# Learning of NNs

- Recall that the knowledge is acquired through a **learning process**.
- For text recognition (e.g. “a”), we need to first show the NN many different handwritings of “a” – This is called **Training**.



- After training, we can test if NN recognizes a new (unseen before) “a” correctly – This is called **Testing**.
- This somewhat resembles human learning: When you went to the kindergarten, you would see many different “a” from different teachers – After some time, you learn to recognize the letter even if it was written by someone new.



# Learning of NNs

- Similarly, for inverted pendulum (broom balancing), one would slowly **learn** the best hand motion in order to keep the broom upright.
- At the end, the brain would have learnt (unknowingly) the **relationship** between force and angular position!



<https://imgur.com/gallery/bZ919>

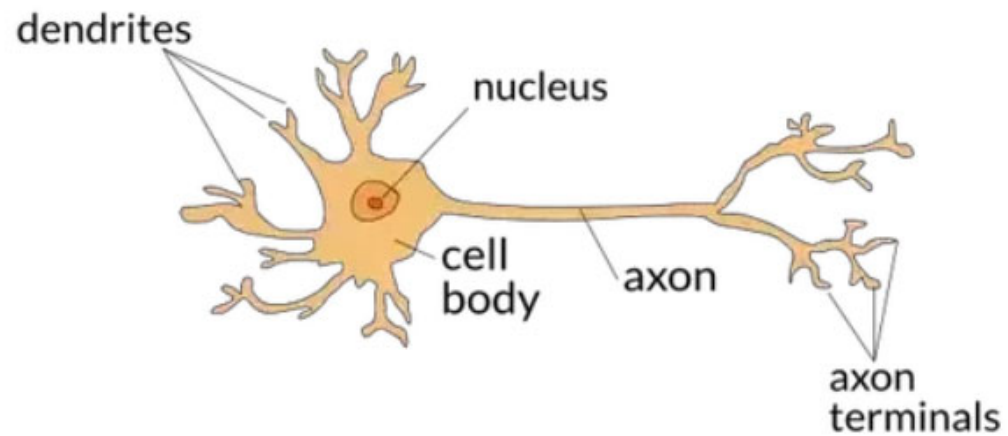
# Historical Perspective

- 1943 McCulloch and Pitts published the **first description of an artificial NN**.
- 1950's – 60's First ANN developed by Marvin Minsky etc.: Single layer networks called **perceptron**, used for weather prediction, vision etc.
- 1970's Research virtually stopped, as many limitations were found e.g. incapable of solving many simple problems.
- 1982 Hopfield proposed associative memory model: NN evolves to minimize an energy function; renewed interest in NNs.
- 1986 **Back propagation** learning rule for **multi-layered networks** proposed, overcoming limitations of simple perceptrons.
- Late 1980's Explosion of research.
- 2000's Research slowed down again with little breakthrough in performance.
- 2010's **Deep neural network** achieved amazing results in vision and research in deep network explodes – It is now the state of the art for object recognition and many other applications.

# Contents

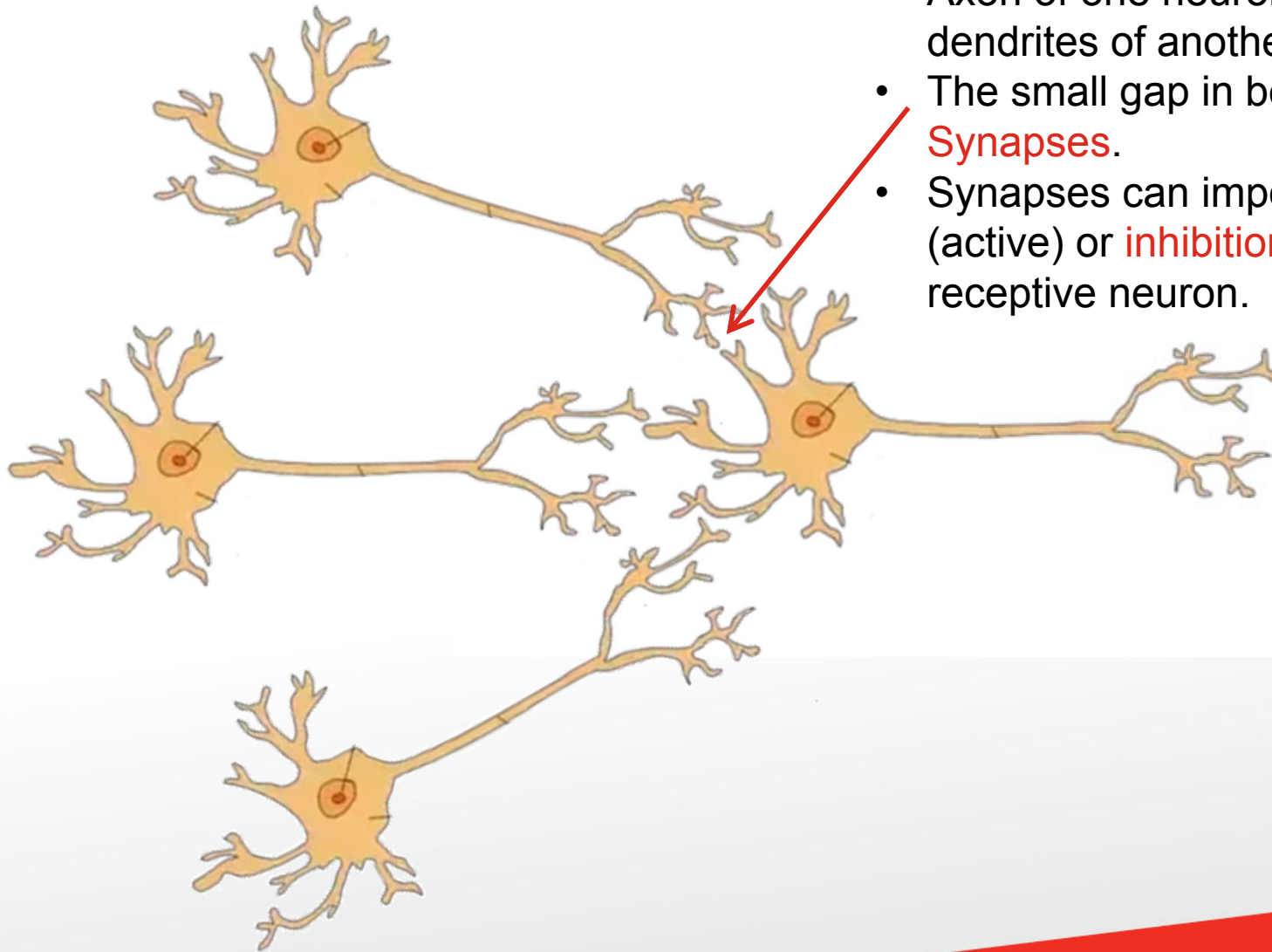
- Introduction
- **The Biological Neuron**
- The Artificial Neuron
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- Deep Neural Networks

# One Biological Neuron



<https://www.quora.com/What-is-the-differences-between-artificial-neural-network-computer-science-and-biological-neural-network>

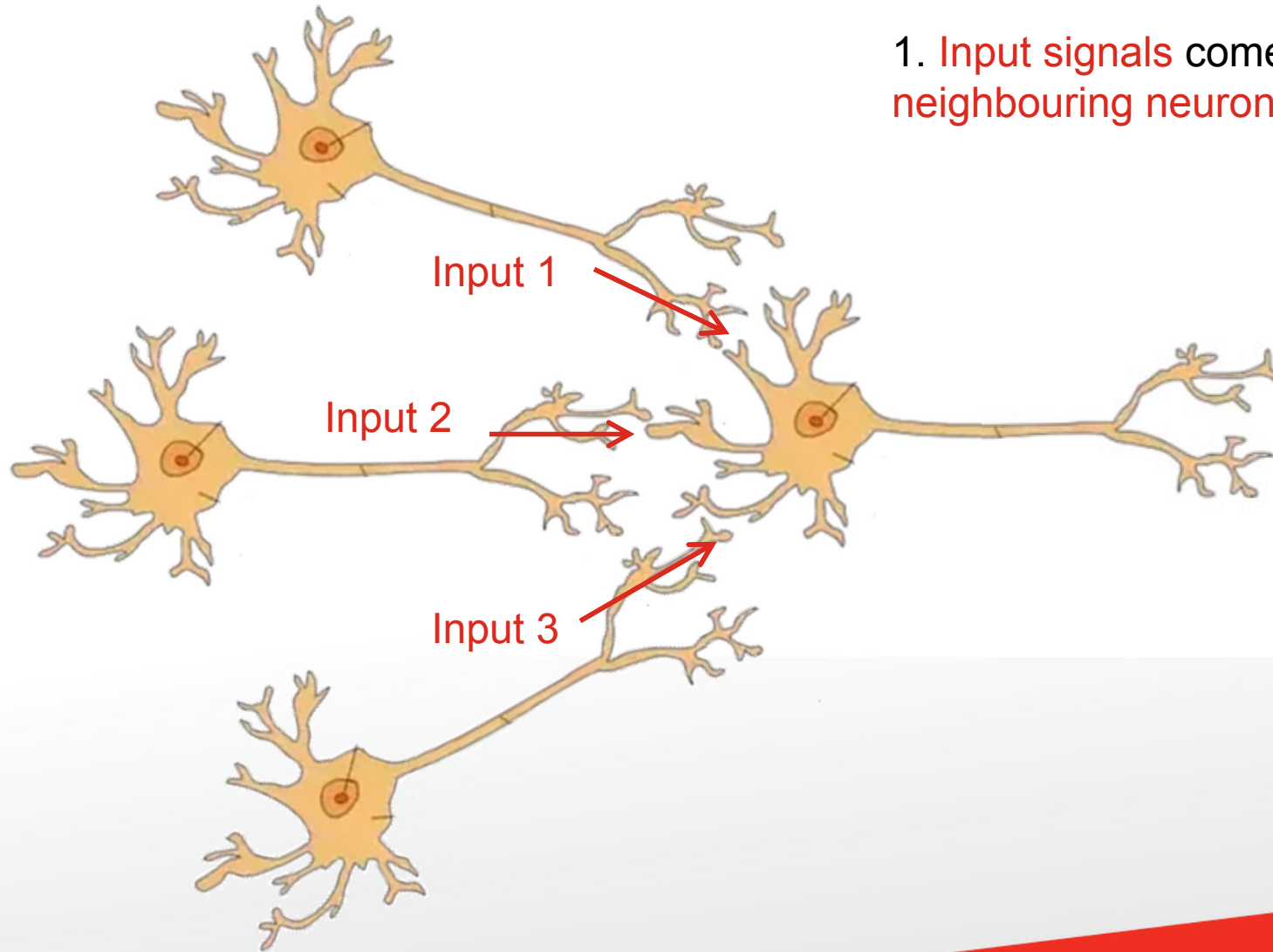
# A Few Neurons Together



- Axon of one neuron almost touches dendrites of another neuron
- The small gap in between is the **Synapses**.
- Synapses can impose **excitation** (active) or **inhibition** (inactive) on the receptive neuron.

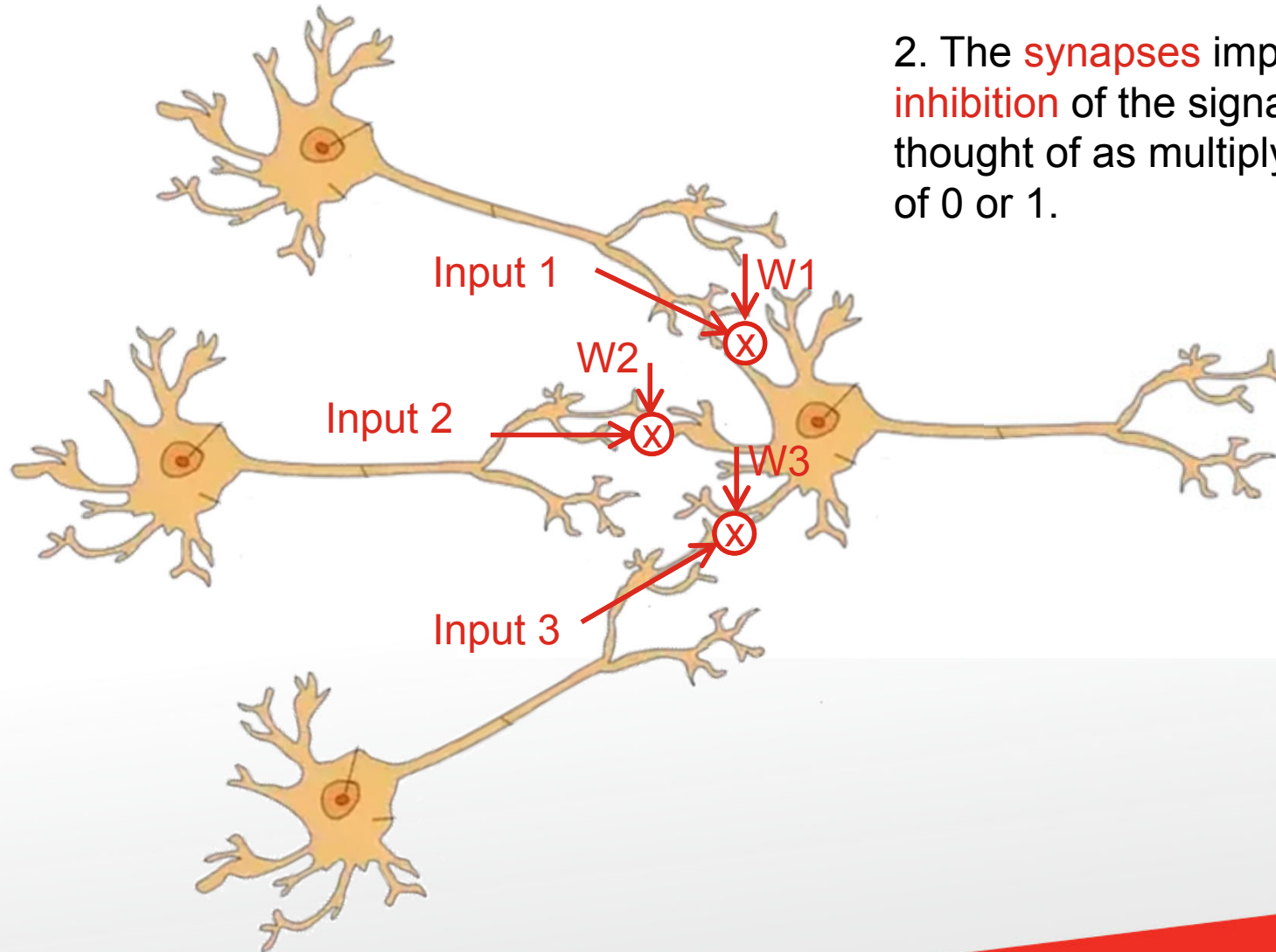
# Looking at the Neuron on the Right

1. **Input signals** come from other neighbouring neurons.



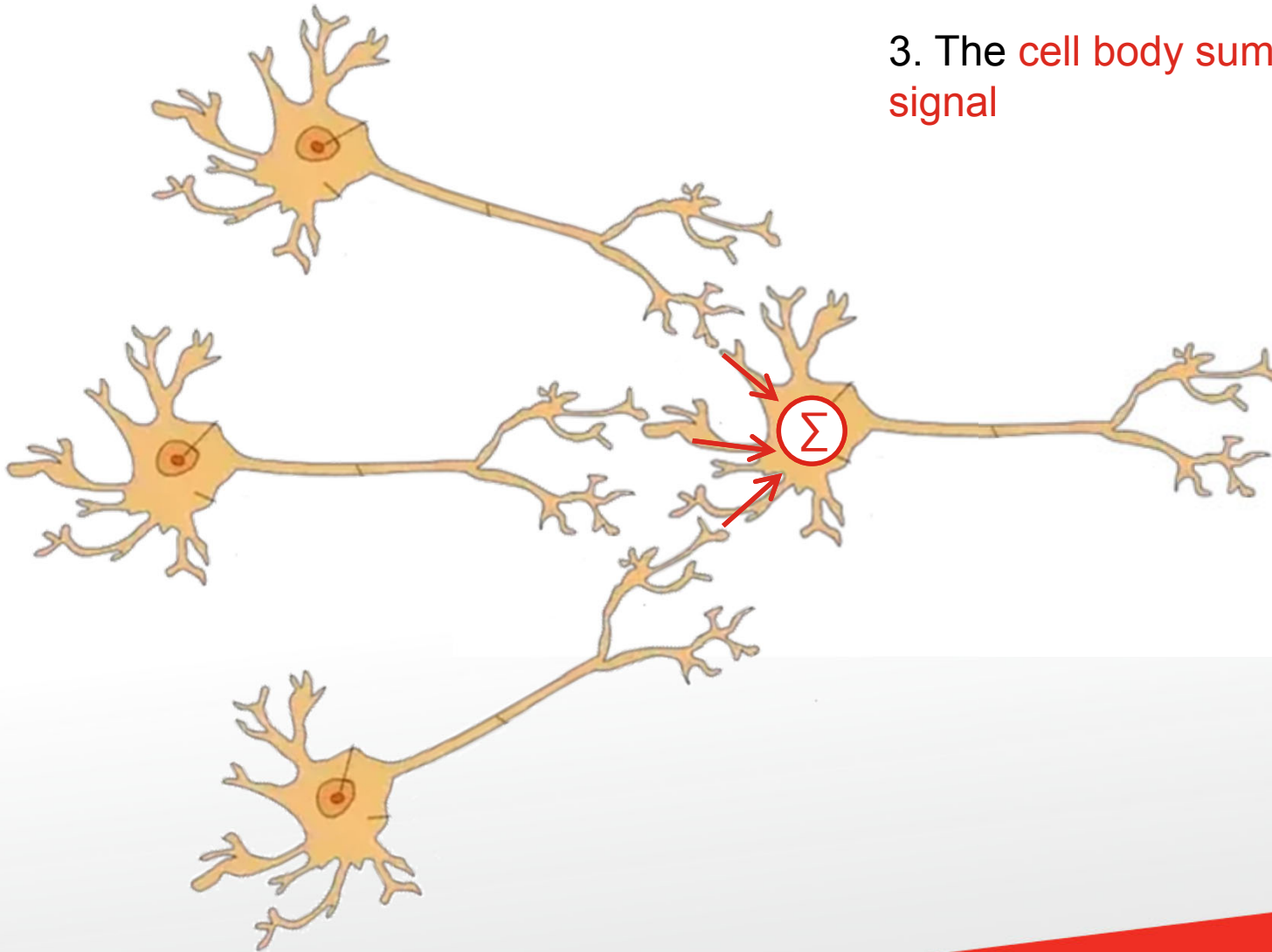
# Looking at the Neuron on the Right

2. The **synapses** impose **excitation** or **inhibition** of the signal. This can be thought of as multiplying with a **weight** of 0 or 1.



# Looking at the Neuron on the Right

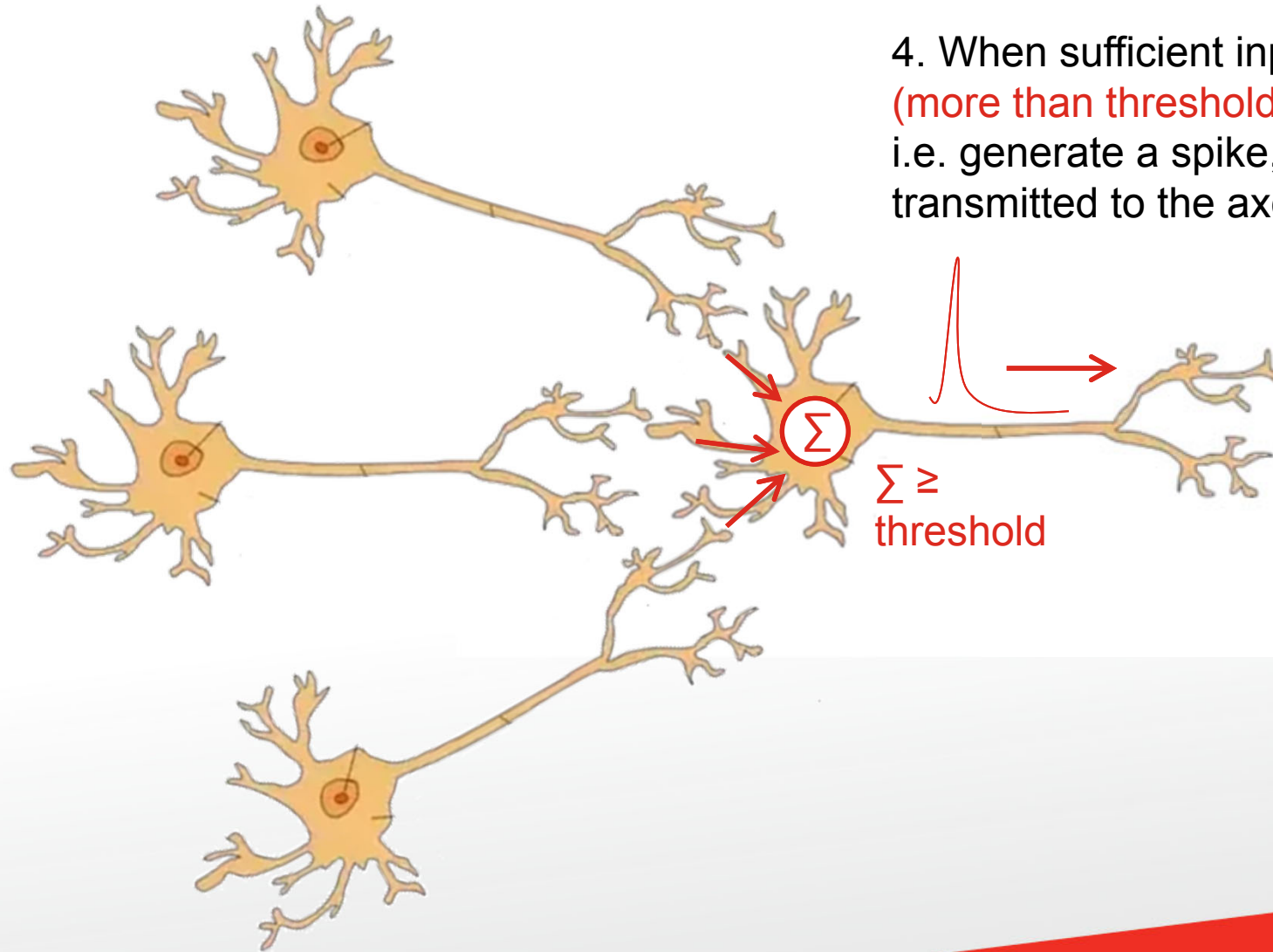
3. The cell body sums the incoming signal





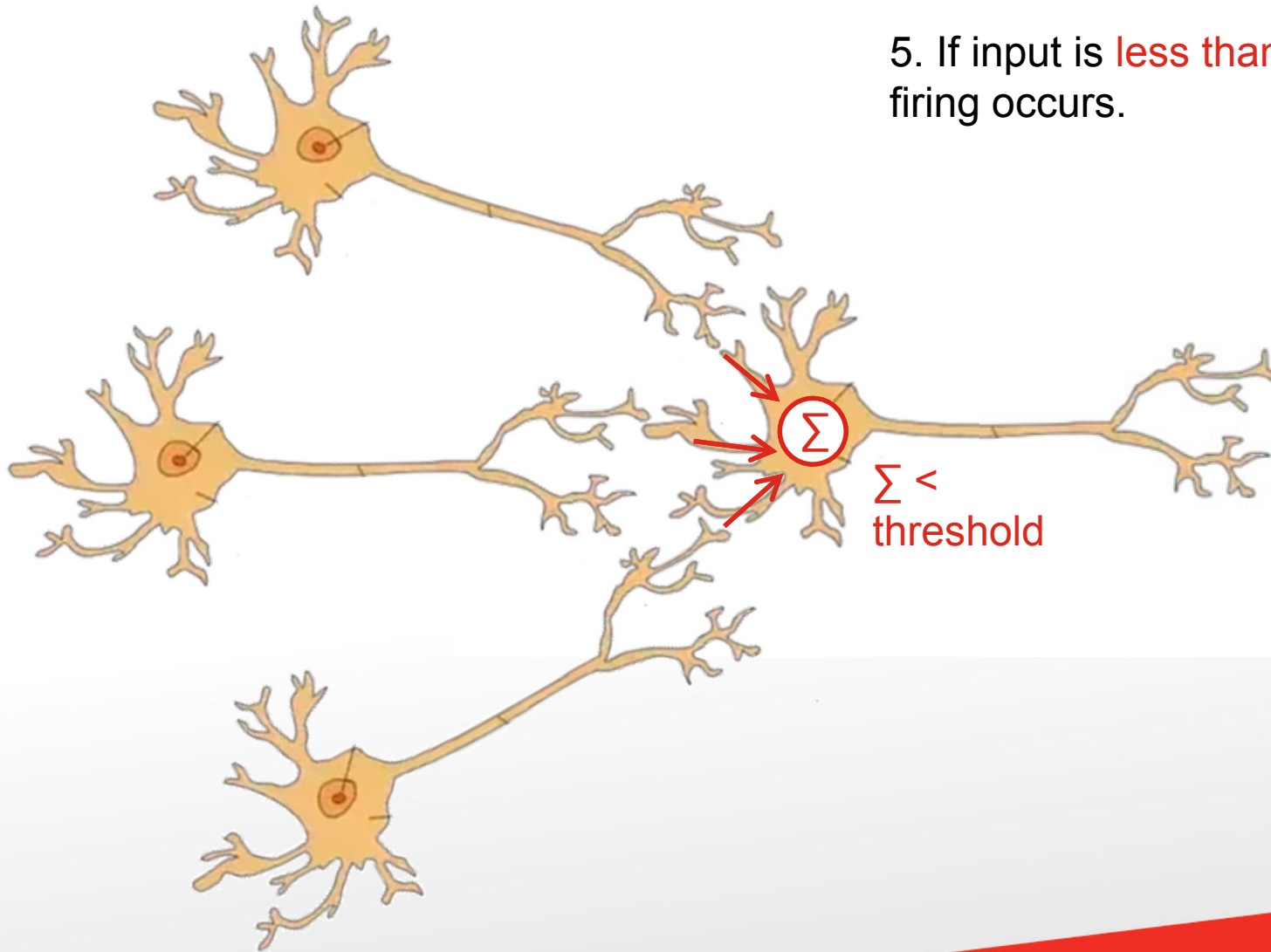
# Looking at the Neuron on the Right

4. When sufficient input is received (more than threshold), the neuron fires, i.e. generate a spike, which is transmitted to the axon.



# Looking at the Neuron on the Right

5. If input is **less than threshold**, then no firing occurs.

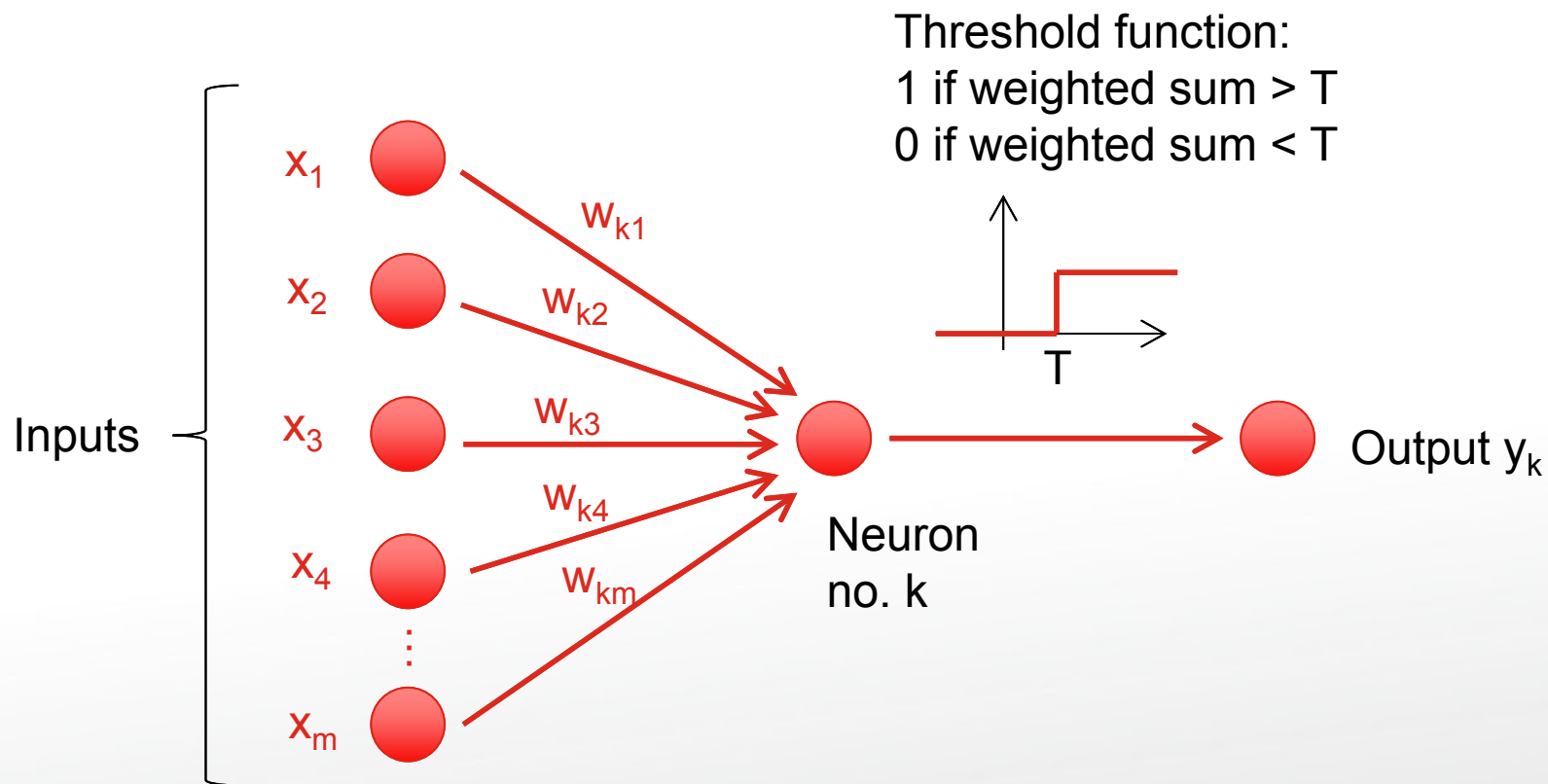


# Contents

- Introduction
- The Biological Neuron
- **The Artificial Neuron**
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- Deep Neural Networks

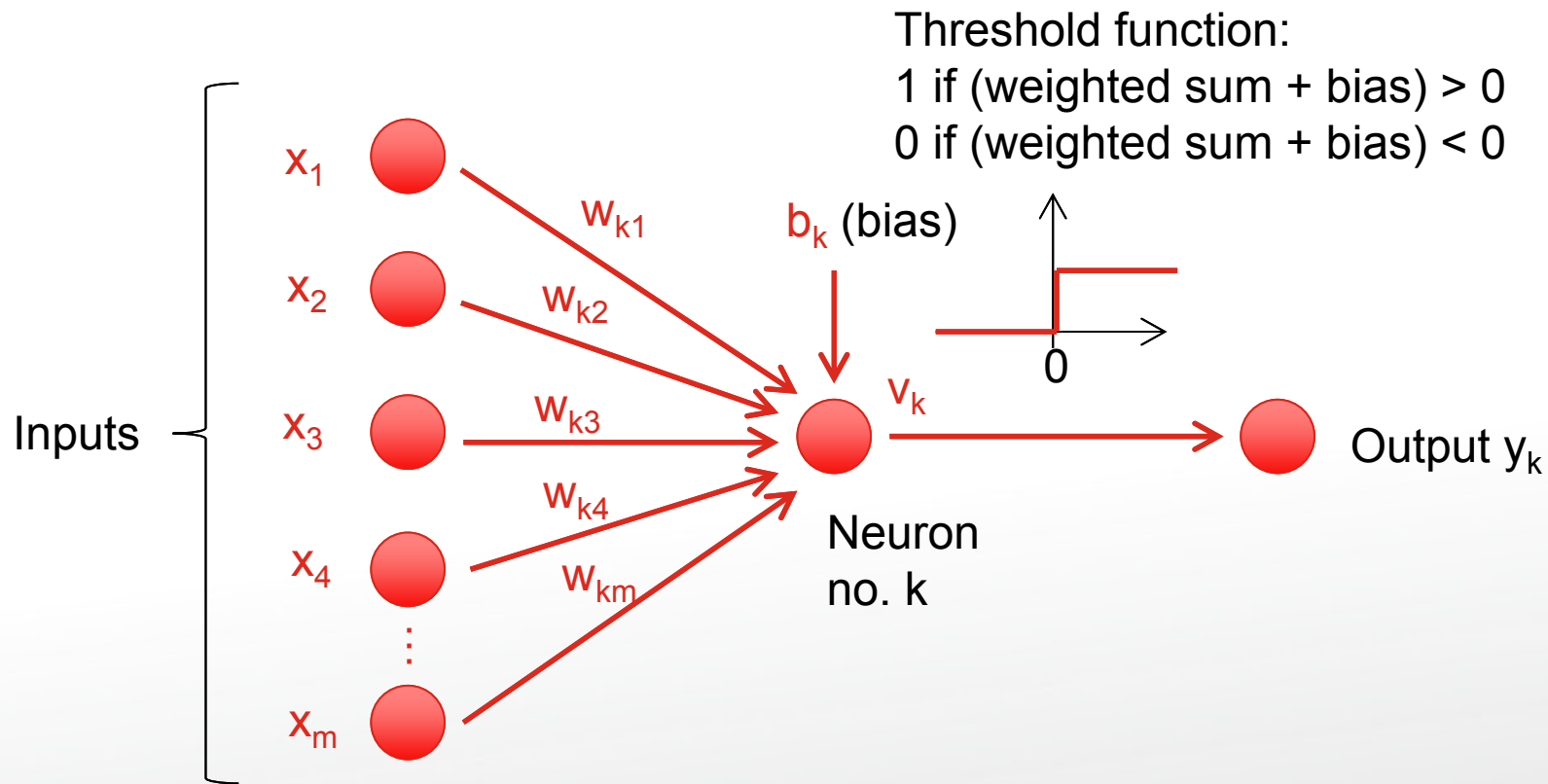
# Early Model of Neuron

Very similar to biological neuron.



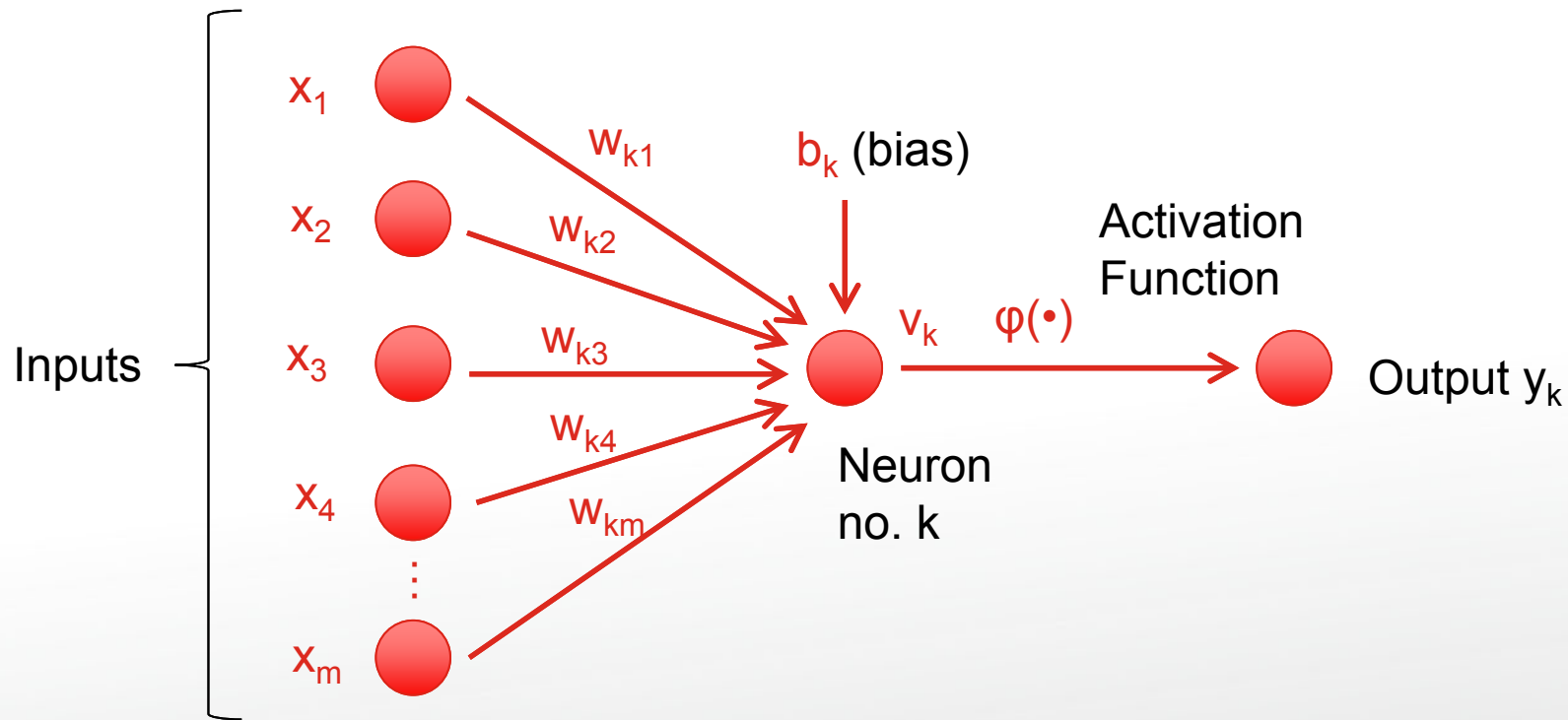
# Improved Model of Neuron

Having **threshold function not symmetrical about 0** makes some calculations difficult. Therefore, a **bias is added**.



# Further Improvements

Various **activation functions** were also proposed in place of the threshold function.

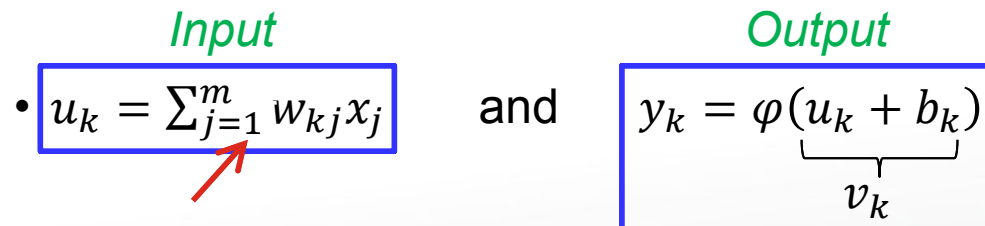


# Model of a Neuron

- The model of a neuron has **three basic components**:
  - A set of **synapses** or connecting links, characterized by **weights**.
  - An **adder for summing** the weighted input signals.
  - An **activation function** for limiting the amplitude of the output of a neuron, e.g. within the range of  $[0, 1]$  or  $[-1, 1]$ .
- **Mathematically**, for a neuron  $k$ ,

*Input* *Output*

•  $u_k = \sum_{j=1}^m w_{kj} x_j$  and  $y_k = \varphi(\underbrace{u_k + b_k}_{v_k})$



- Alternatively, we can think of the **bias** as input  $x_0 = 1$  multiplied by weight  $w_{k0} = b_k$ .

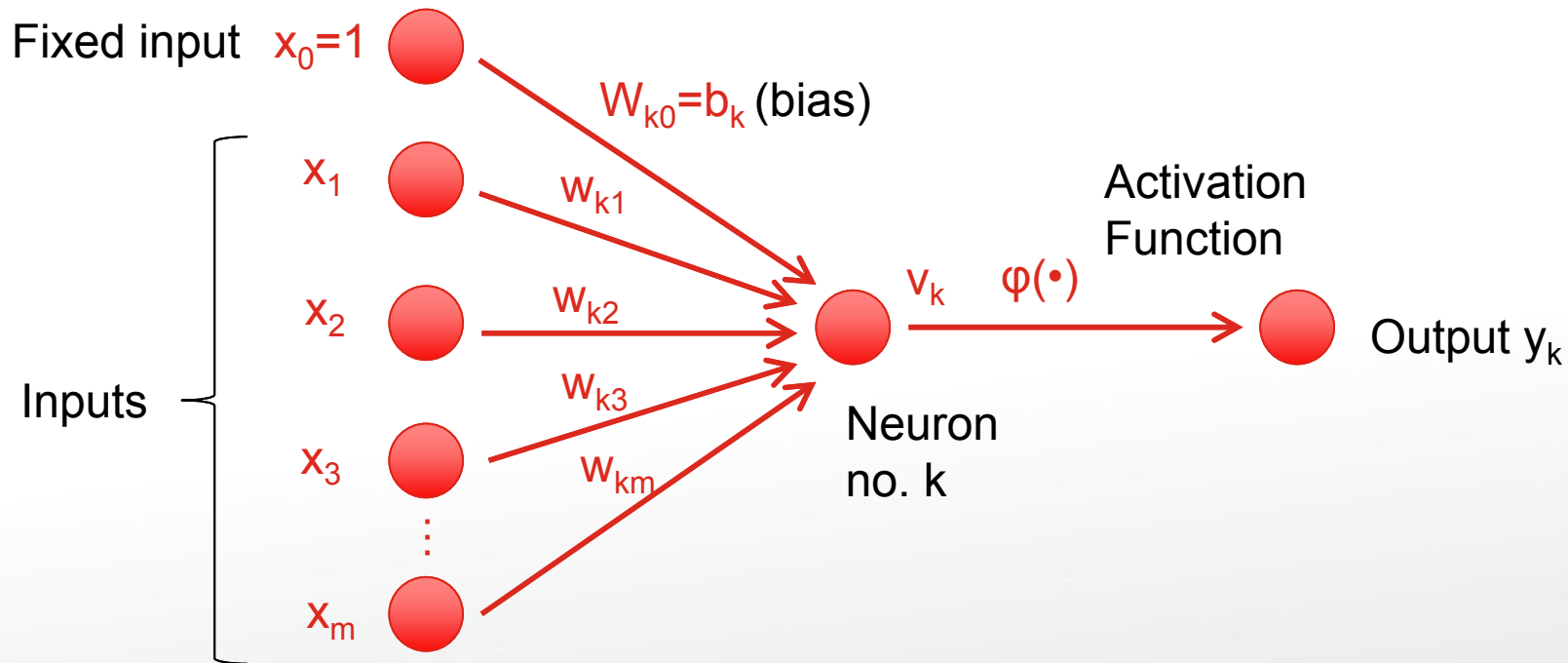
•  $v_k = \sum_{j=0}^m w_{kj} x_j$  and  $y_k = \varphi(v_k)$



**Note!**

# Bias as Input x Weight

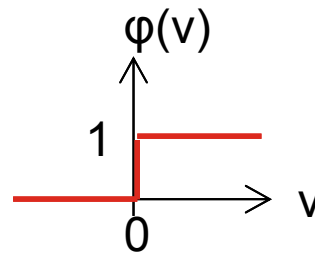
With the last formulation, the model of a neuron looks as follows:



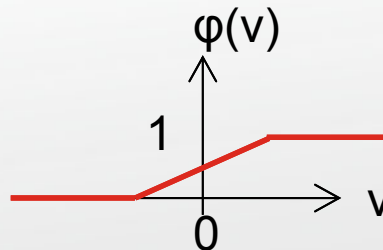


# Activation Functions

- Various activation functions have been proposed for neural networks.
- **Threshold function (hard-limiter):**
  - Note: McCulloch-Pits model (1943) of neuron used this form of function.

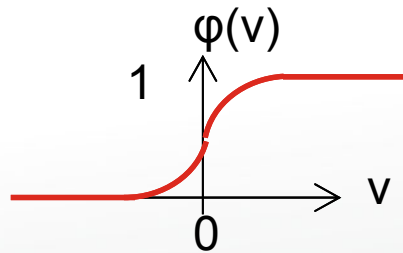


- **Piecewise linear function:**
  - Linear combiner within certain range, then saturated to 0 or 1.
  - If gradient of linear region is very high, it would reduce to threshold function.

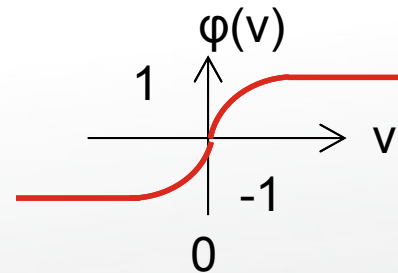


# Activation Functions

- **Sigmoid function** (s-shaped):
  - Most common
  - Strictly increasing function
  - Asymptotically approach the saturation values
  - **Continuous & Differentiable everywhere** (very useful for optimisation later)
  - Example: **Logistic** function (left) and **hyperbolic tangent** function (right).



$$\varphi(v) = \frac{1}{1 + e^{-av}}$$

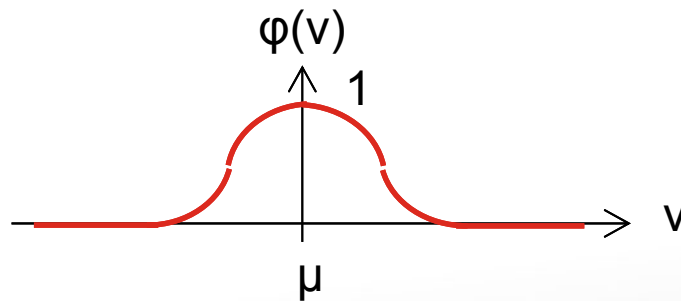


$$\varphi(v) = \tanh(v)$$

# Activation Functions

## 1. Gaussian function:

- Continuous & Differentiable everywhere
- Single maximum at  $v = \mu$ .
- Used in Radial Basis Function networks.



$$\phi(v) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2}\left(\frac{v - \mu}{\sigma}\right)^2\right)$$

## 2. Linear:

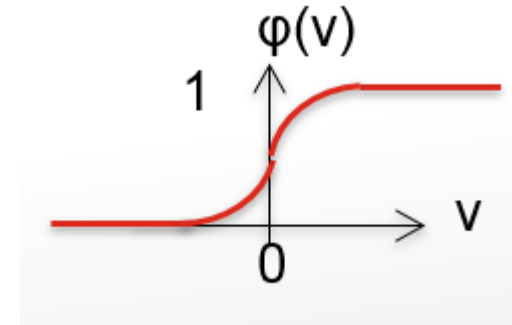
$$\phi(v) = v$$

- This is mostly used in the **output layer** of a multilayer network (later).

# Activation Functions

- The **logistic function** is widely used because **its differentiation** (again, very useful for optimisation) is very easy:

$$\varphi(v) = \frac{1}{1 + e^{-av}} = (1 + e^{-av})^{-1}$$



- The **derivative** of  $\varphi(v)$  with respect to  $v$  is:

$$\begin{aligned}\varphi'(v) &= -(1 + e^{-av})^{-2} \cdot e^{-av} \cdot (-a) = \frac{ae^{-av}}{(1 + e^{-av})^2} \\ &= a \cdot \frac{e^{-av}}{(1 + e^{-av})} \cdot \frac{1}{(1 + e^{-av})} = a \cdot \frac{(1 + e^{-av} - 1)}{(1 + e^{-av})} \cdot \frac{1}{(1 + e^{-av})} \\ &= a \cdot \left(1 - \frac{1}{(1 + e^{-av})}\right) \cdot \frac{1}{(1 + e^{-av})} = a \cdot (1 - \varphi(v)) \cdot \varphi(v)\end{aligned}$$

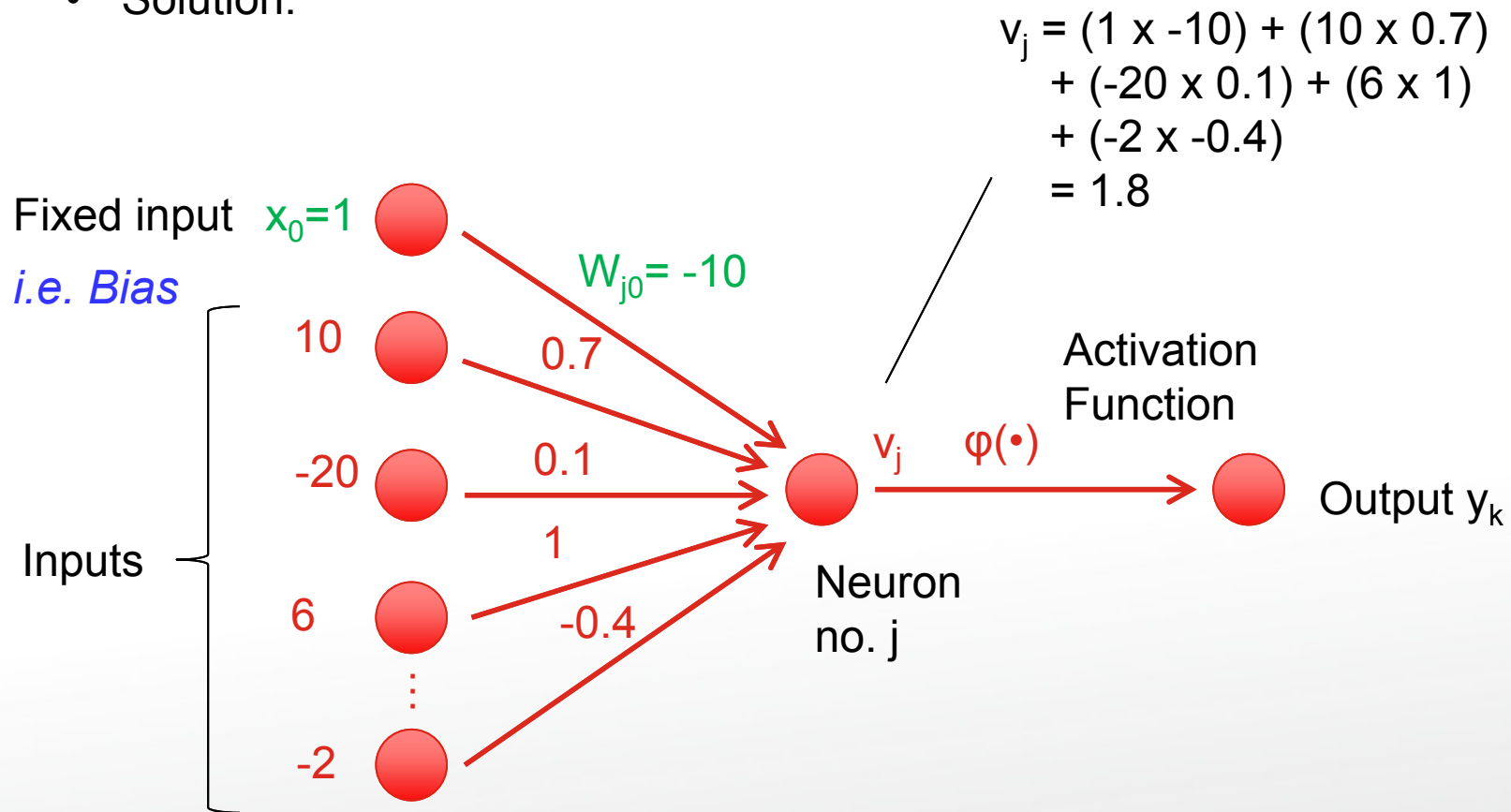
- That means, if you have the output value  $\varphi(v)$ , then you can almost automatically get the derivative.

# Example

- A neuron  $j$  receives **inputs** from four other neurons whose activity levels are 10, -20, 6 and -2. The respective synaptic **weights** of neuron  $j$  are 0.7, 0.1, -1 and -0.4. Calculate the output of neuron  $j$  for the following two situations:
  - a) The neuron is linear
  - b) The neuron is represented by a hard limiter.
- Assume that the bias applied to the neuron is -10

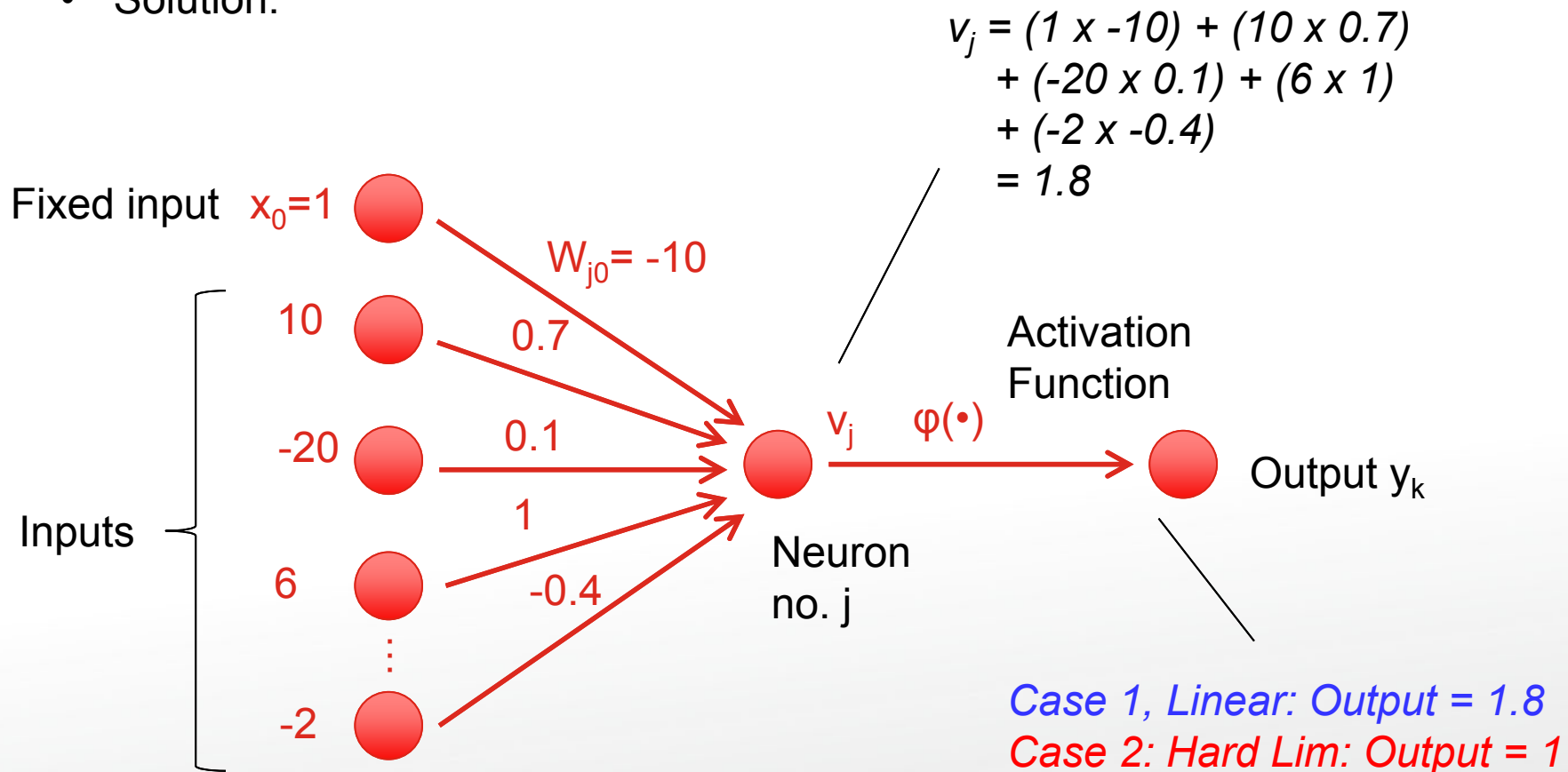
# Example

- Solution:



# Example

- Solution:



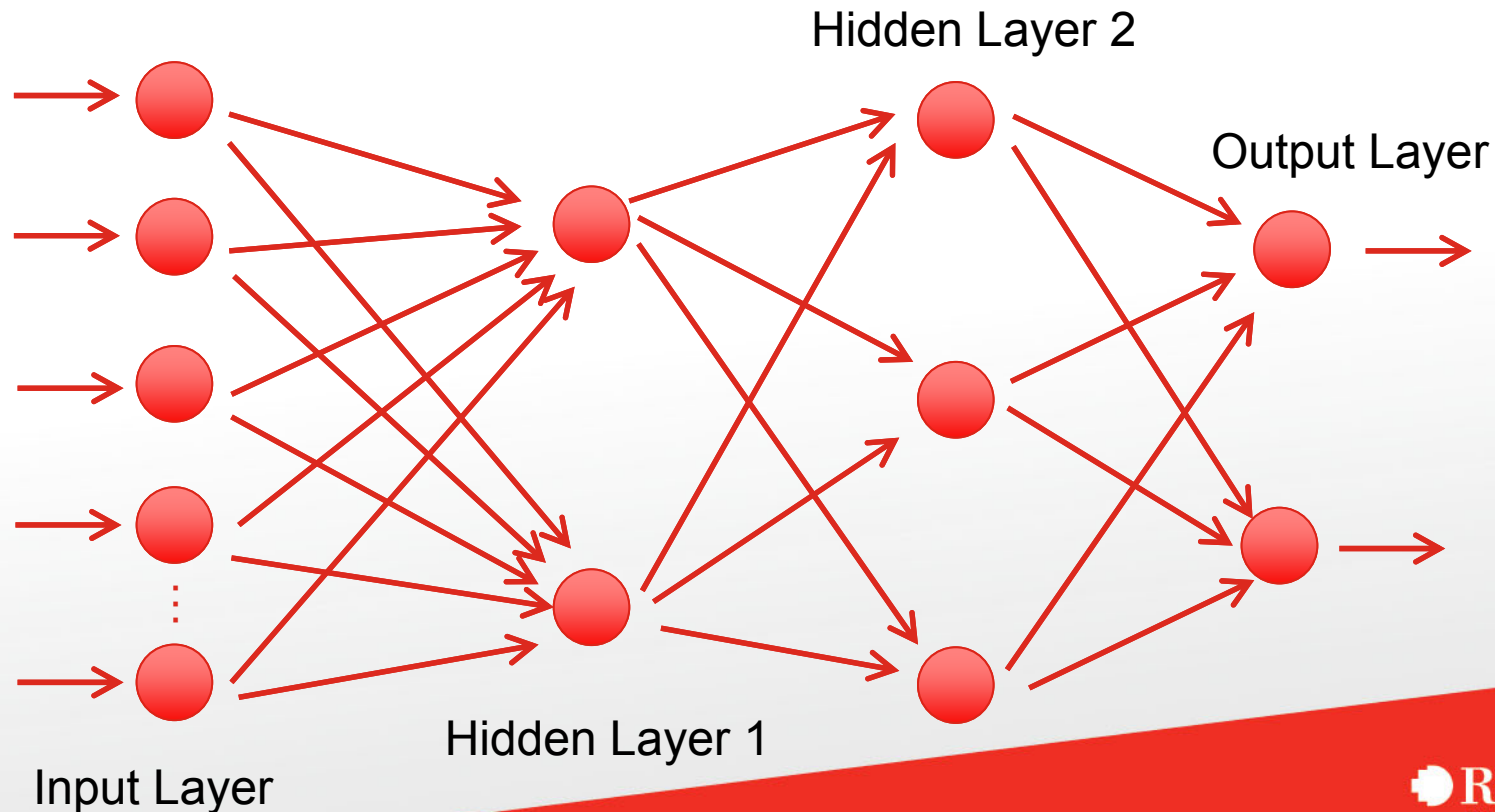
# Contents

- Introduction
- The Biological Neuron
- The Artificial Neuron
- **Network Architectures**
- Perceptron
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- Deep Neural Networks



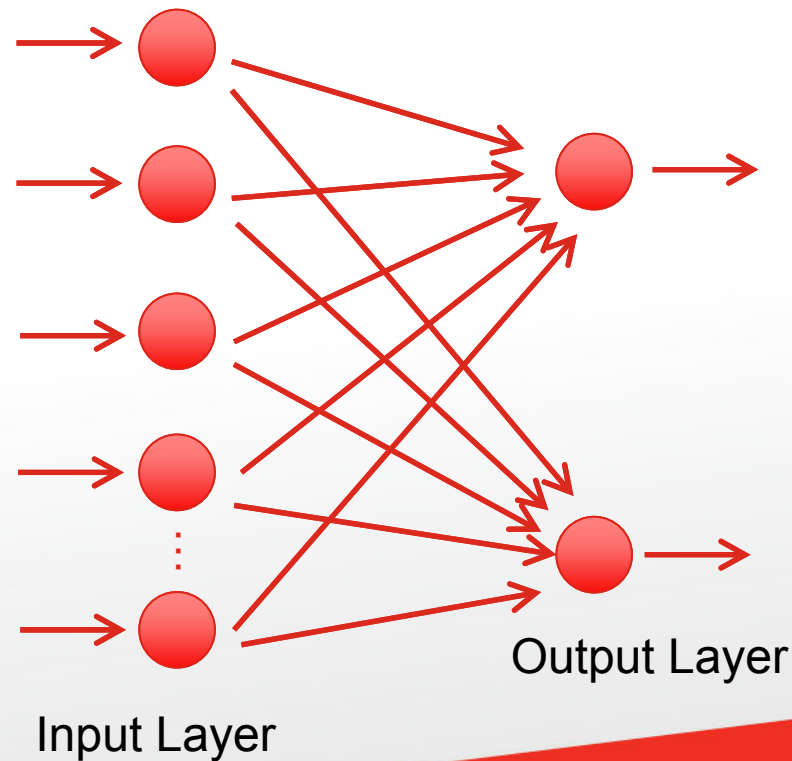
# Network Architectures

- One single neuron is only able to solve some **very simple problems**.
- To solve for **more complex problems**, networks with large number of neurons are required.
- **Multilayer feedforward neural networks**: Connections only from left to right.



# Network Architectures

- The **number of hidden layer** is a design parameter. It can be 0, 1, 2...
- The network on the page before has 2 hidden layers.
- On this page, the example below has no hidden layers.
  - It is called “**Single-layer** Feedforward Network”. The layer refers to **output layer**.



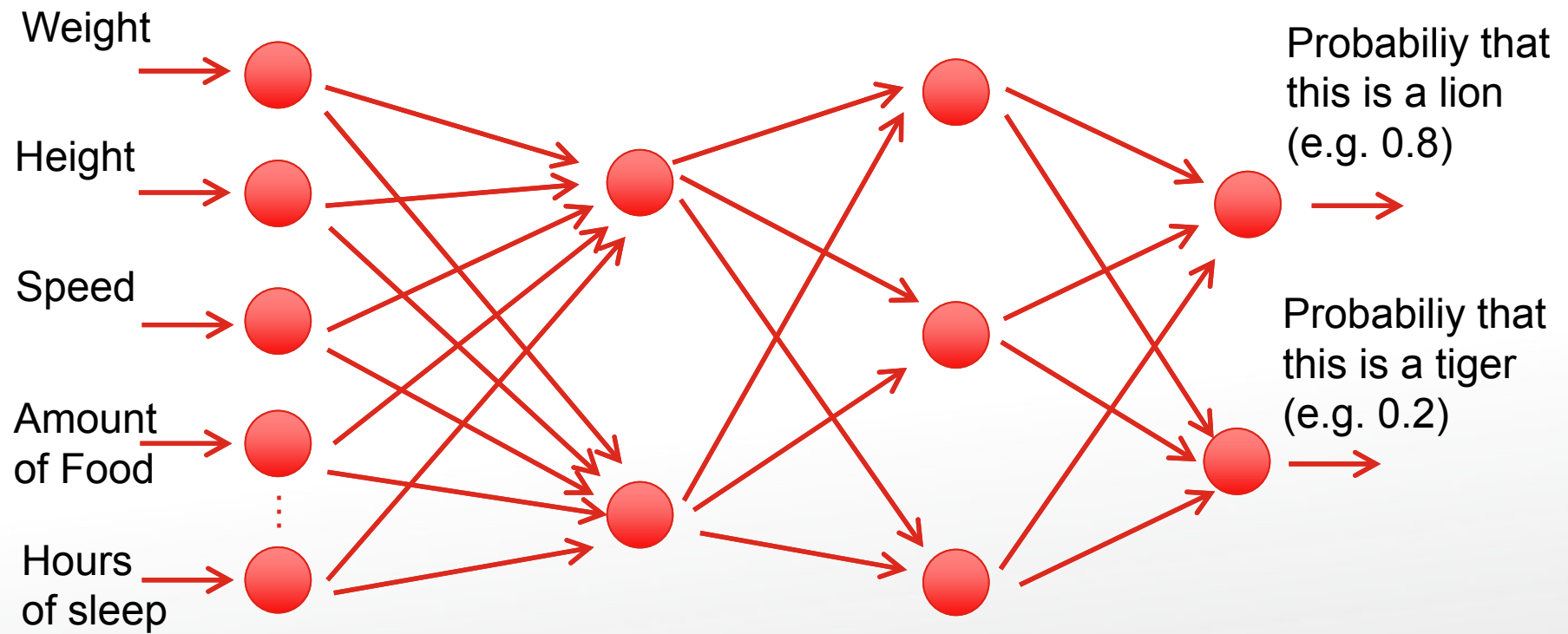
# Network Architectures

- It can be mathematically proven that a **multilayer network is equivalent to single-layer network** (i.e. can solve the same problem).
- **Advantage** of multilayer network:
  - The **total number of synapses weight** in multilayer network is less → less parameters to tune
    - E.g. 1-9-1 network: 28 weights (including biases)
    - 1-3-3-1 network: 22 weights (including biases)
- **Disadvantage** of multilayer network:
  - More prone to local minima due to its more complicated structure.

# Network Architectures

- Design parameters:
  - Number of **hidden layers** (we already saw this).
  - Number of **neurons** in each hidden layer.
  - The **activation function** in each layer.
    - Can be different for different layers.
- Not design parameters:
  - Number of input nodes – One for each input signal.
  - Number of output nodes:
    - For single function approximation, only one output node.
    - For **classification**, same as the **number of expected classes**. See next page.

# Network Architectures

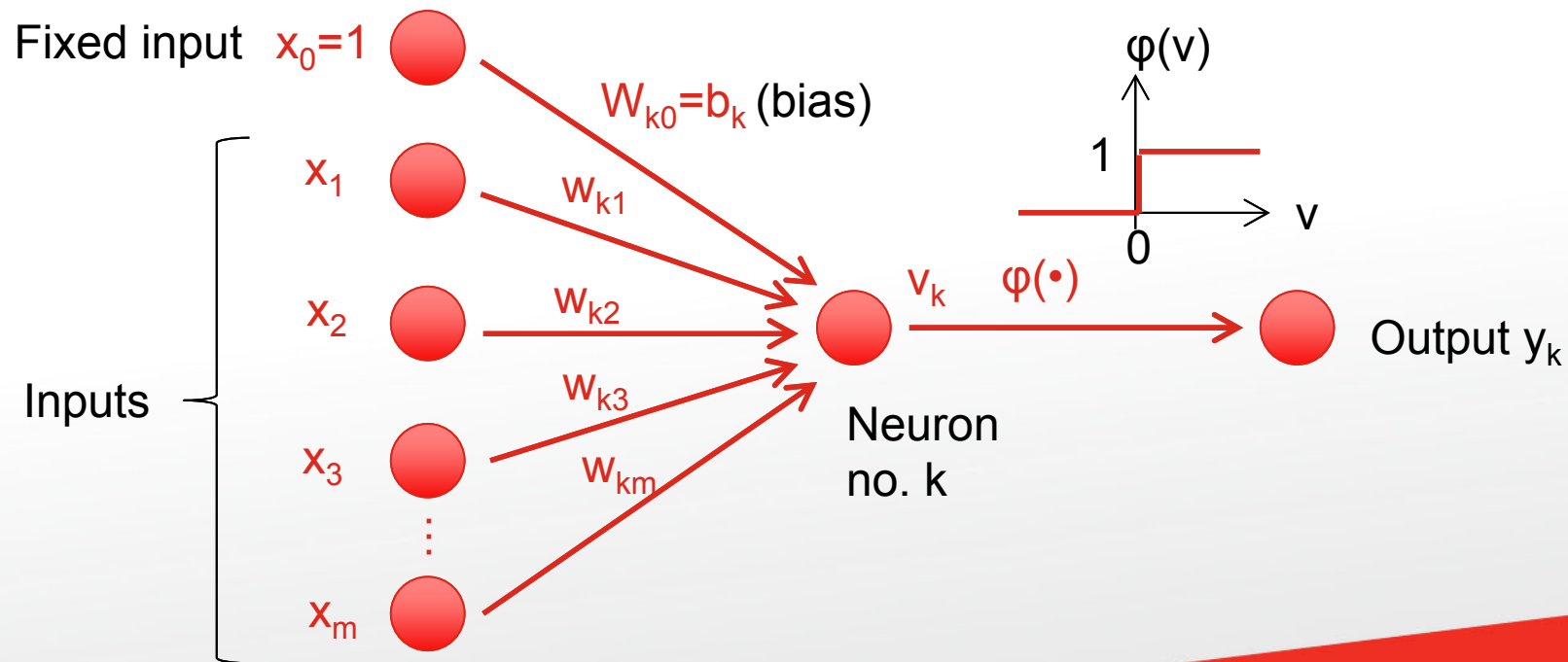


# Contents

- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- **Perceptron**
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- Deep Neural Networks

# Perceptron

- It is a **single-layer** feedforward network.
  - Simplest form of a NN for classification of patterns.
  - It learns via examples, how to **assign input vectors** (samples) to different **classes** (Rosenblatt, 1958).



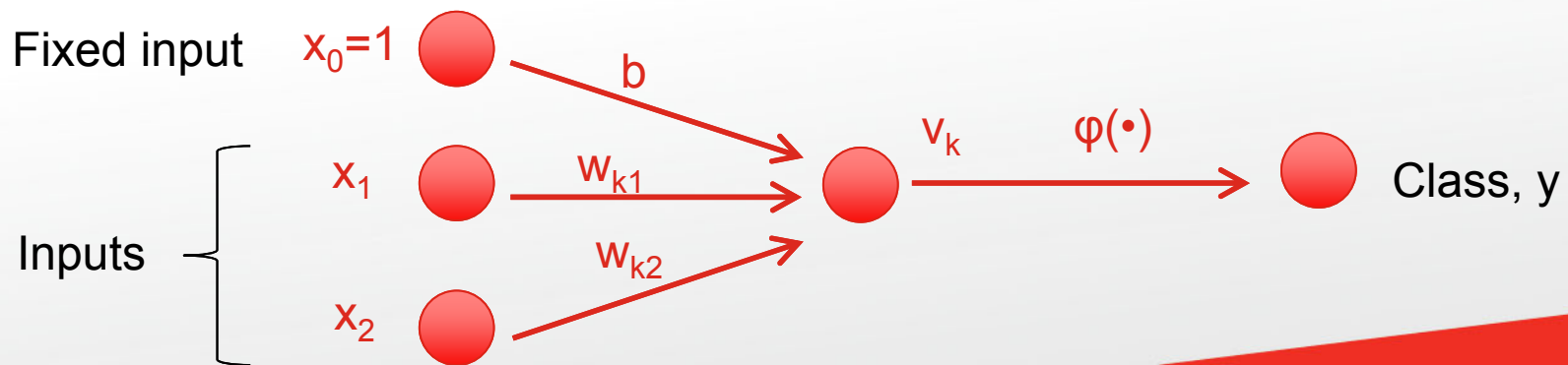
# 2D Example

- A **2-dimensional** example: To correctly classify the external inputs  $\{x_1, x_2\}$  into one of two classes  $\{C_1 \text{ or } C_0\}$ .
- E.g. **AND binary function**:

x1	0	0	1	1
x2	0	1	0	1
y	0	0	0	1

2D

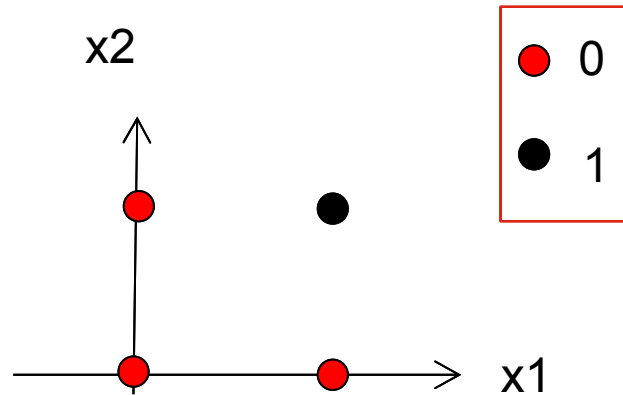
- The perceptron is shown below. We want to find  **$b$ ,  $w_{k1}$  and  $w_{k2}$** , so that when given  $x_1$  and  $x_2$ , the correct  $y$  will be calculated.



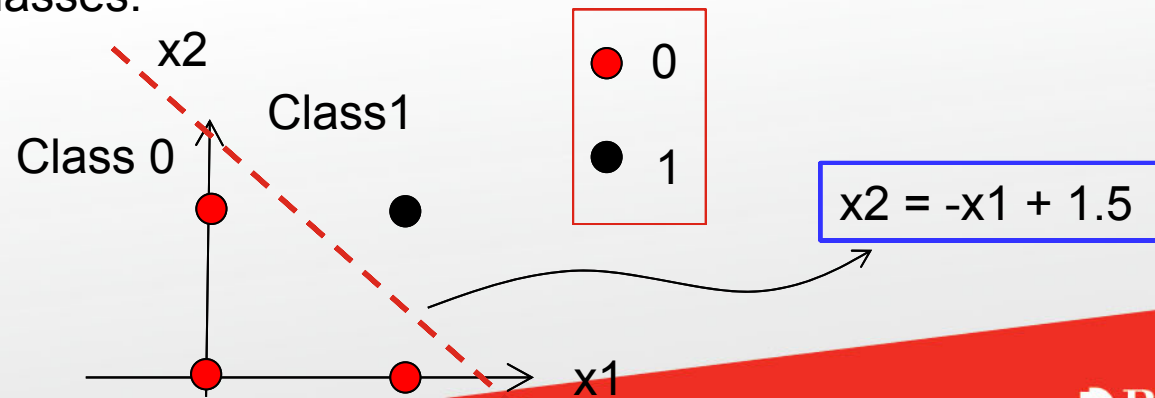


# 2D Example

- This is a simple **2-dimensional** problem, thus we can sketch the input-output space:

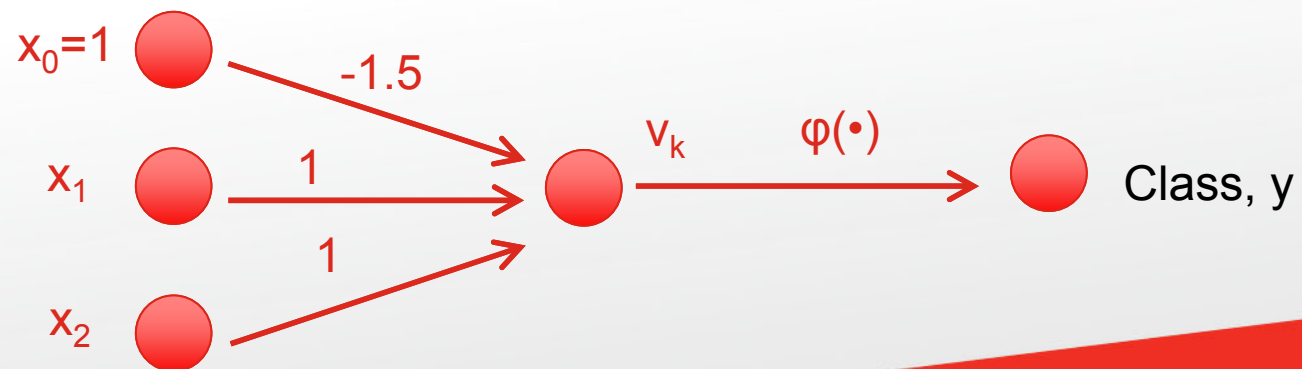


- The classes are **linearly separable** and we can easily get a straight line to separate the two classes.



# 2D Example

- From the line equation, we would either have:
  - $v_k = -x_1 - x_2 + 1.5$ , or
  - $v_k = x_1 + x_2 - 1.5$ .
- Which one is correct?
- Try with  $x_1 = 0, x_2 = 0$ :
  - First equation gives  $v_k = 1.5$ , then  $\phi(1.5) = 1 \rightarrow$  Wrong
  - Second equation gives  $v_k = -1.5$ , then  $\phi(-1.5) = 0 \rightarrow$  **Correct!**
- Thus, the complete perceptron is as follows:



# Revision

- What are binary logic functions used to build all digital systems, computers...

# Revision

- What are binary logic functions used to build all digital systems, computers...
- **AND, OR, NOT**

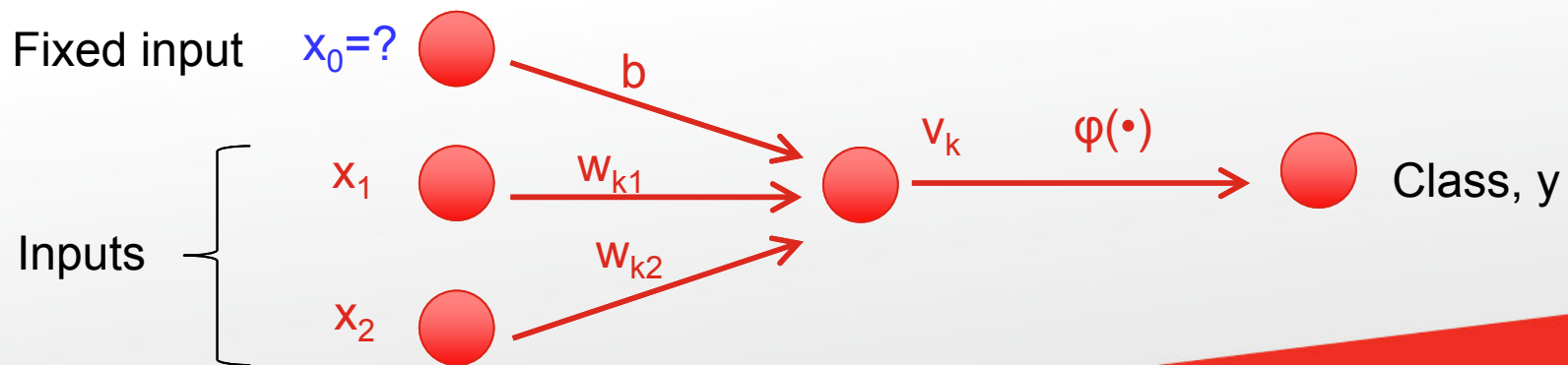
# 2D Example - Exercise

- A **2-dimensional** example: To correctly classify the external inputs  $\{x_1, x_2\}$  into one of two classes  $\{C_1 \text{ or } C_0\}$ .
- E.g. **OR binary function**:

x1	0	0	1	1
x2	0	1	0	1
y				

2D

- The perceptron is shown below. We want to find  **$b$ ,  $w_{k1}$  and  $w_{k2}$** , so that when given  $x_1$  and  $x_2$ , the correct  $y$  will be calculated.



# More General Case

- The **input vector** is:

$$x'(n) = [1, x_1(n), x_2(n), \dots, x_m(n)]^T$$

- The **weight vector** is:

$$w'(n) = [b(n), w_1(n), w_2(n), \dots, w_m(n)]^T$$

- Where  $n$  denotes the **iteration step**.
- The **intermediate output** (before the hard limiter) is:

$$v(n) = w'^T(n) \cdot x'(n)$$

- The equation  $w'^T \cdot x' = 0$ , plotted in an  $m$ -dimensional space with coordinates  $x_1, x_2, \dots, x_m$ , defines a **hyperplane** which separates the two classes.
- With the help of hard limiter, we finally get:

$$w'^T \cdot x' \geq 0 \text{ Class 1}$$

$$w'^T \cdot x' < 0 \text{ Class 0}$$

# More General Case

- Unlike in the 2D case, it **would not be easy** to calculate the weights **manually**.
  - In fact, it is hard to even “visualize” the data space.
- Therefore, a **learning procedure** is normally used to **train the perceptron**.
  - A **training set** of **input-output vectors**, i.e. exemplars, is given.
  - The **weight vector will be tuned** in such a way that the **best classification** of the training vectors is achieved.

# Perceptron Learning Algorithm

- Start with randomly chosen weight vector  $w'(0)$ .
- Let  $n = 1$ .
- **While** there exist input vectors that are wrongly classified by  $w'(n - 1)$ , **do**
  - If  $x'$  is a misclassified input vector,
  - Update the weight vector to
$$w'(n) = w'(n - 1) + \eta(d - y)x'$$
  - Where  $\eta > 0$  and  $d = \begin{cases} 1 & \text{if } x \text{ belongs to Class 1} \\ 0 & \text{if } x \text{ belongs to Class 0} \end{cases}$  and  $d = \text{desired output}$   
 $y = \text{network output}$
  - Increment  $n$
- **End While**



# Perceptron Learning Algorithm

- Example: **AND** problem re-visited.

x1	0	0	1	1
x2	0	1	0	1
d	0	0	0	1

- Randomly initiated weight:  $w'(0) = [0.5, 0.5, 0.5]^T$

- First epoch, first column of data

$$w'(0)^T \cdot x = [0.5, 0.5, 0.5] \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0.5 \rightarrow y = \varphi(0.5) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(1) = w'(0) + \eta(d - y)x' = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.5 \\ 0.5 \end{bmatrix}$$

Chosen to be 0.1

# Perceptron Learning Algorithm

- First epoch, second column of data

$$w'(1)^T \cdot x = [0.4, 0.5, 0.5] \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 0.9 \rightarrow y = \varphi(0.9) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(2) = w'(1) + \eta(d - y)x' = \begin{bmatrix} 0.4 \\ 0.5 \\ 0.5 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.5 \\ 0.4 \end{bmatrix}$$

- 
- First epoch, third column of data

$$w'(2)^T \cdot x = [0.3, 0.5, 0.4] \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = 0.8 \rightarrow y = \varphi(0.8) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(3) = w'(2) + \eta(d - y)x' = \begin{bmatrix} 0.3 \\ 0.5 \\ 0.4 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.4 \\ 0.4 \end{bmatrix}$$

# Perceptron Learning Algorithm

- First epoch, fourth column of data

$$w'(3)^T \cdot x = [0.2, 0.4, 0.4] \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 1 \rightarrow y = \varphi(1) = 1, d = 1 \rightarrow \text{correct}$$

- Because the classification is correct,  $w$  remains the same.

$$w'(4) = w'(3) = \begin{bmatrix} 0.2 \\ 0.4 \\ 0.4 \end{bmatrix}$$

- 
- We have **completed the first epoch**, i.e. all the data has been presented once.

# Perceptron Learning Algorithm

- Continue to the **second epoch, first column of data**

$$w'(4)^T \cdot x = [0.2, 0.4, 0.4] \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0.2 \rightarrow y = \varphi(0.2) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(5) = w'(4) + \eta(d - y)x' = \begin{bmatrix} 0.2 \\ 0.4 \\ 0.4 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.4 \\ 0.4 \end{bmatrix}$$

- 
- Second epoch, second column of data**

$$w'(5)^T \cdot x = [0.1, 0.4, 0.4] \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 0.5 \rightarrow y = \varphi(0.5) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(6) = w'(5) + \eta(d - y)x' = \begin{bmatrix} 0.1 \\ 0.4 \\ 0.4 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.4 \\ 0.3 \end{bmatrix}$$

# Perceptron Learning Algorithm

- Second epoch, third column of data

$$w'(6)^T \cdot x = [0, 0.4, 0.3] \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = 0.4 \rightarrow y = \varphi(0.4) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(7) = w'(6) + \eta(d - y)x' = \begin{bmatrix} 0 \\ 0.4 \\ 0.3 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.1 \\ 0.3 \\ 0.3 \end{bmatrix}$$

- 
- Second epoch, fourth column of data

$$w'(7)^T \cdot x = [-0.1, 0.3, 0.3] \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 0.5 \rightarrow y = \varphi(0.5) = 1, d = 1 \rightarrow \text{correct}$$

- Because the classification is correct,  $w$  remains the same.

$$w'(8) = w'(7) = \begin{bmatrix} -0.1 \\ 0.3 \\ 0.3 \end{bmatrix}$$

- Second epoch completed.

# Perceptron Learning Algorithm

- Continue to **third epoch, first column of data**

$$w'(8)^T \cdot x = [-0.1, 0.3, 0.3] \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = -0.1 \rightarrow y = \varphi(-0.1) = 0, d = 0 \rightarrow \text{correct}$$

- Because the classification is correct,  $w$  remains the same.

$$w'(9) = w'(8) = \begin{bmatrix} -0.1 \\ 0.3 \\ 0.3 \end{bmatrix}$$

- 
- Third epoch, second column of data**

$$w'(9)^T \cdot x = [-0.1, 0.3, 0.3] \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 0.2 \rightarrow y = \varphi(0.2) = 1, d = 0 \rightarrow \text{misclassified}$$

- Because of misclassification, update  $w$  to:

$$w'(10) = w'(9) + \eta(d - y)x' = \begin{bmatrix} -0.1 \\ 0.3 \\ 0.3 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.3 \\ 0.2 \end{bmatrix}$$

# Perceptron Learning Algorithm

- Third epoch, third column of data

$$w'(10)^T \cdot x = [-0.2, 0.3, 0.2] \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = 0.1 \rightarrow y = \varphi(0.1) = 1, d = 0 \rightarrow \text{misclassified}$$

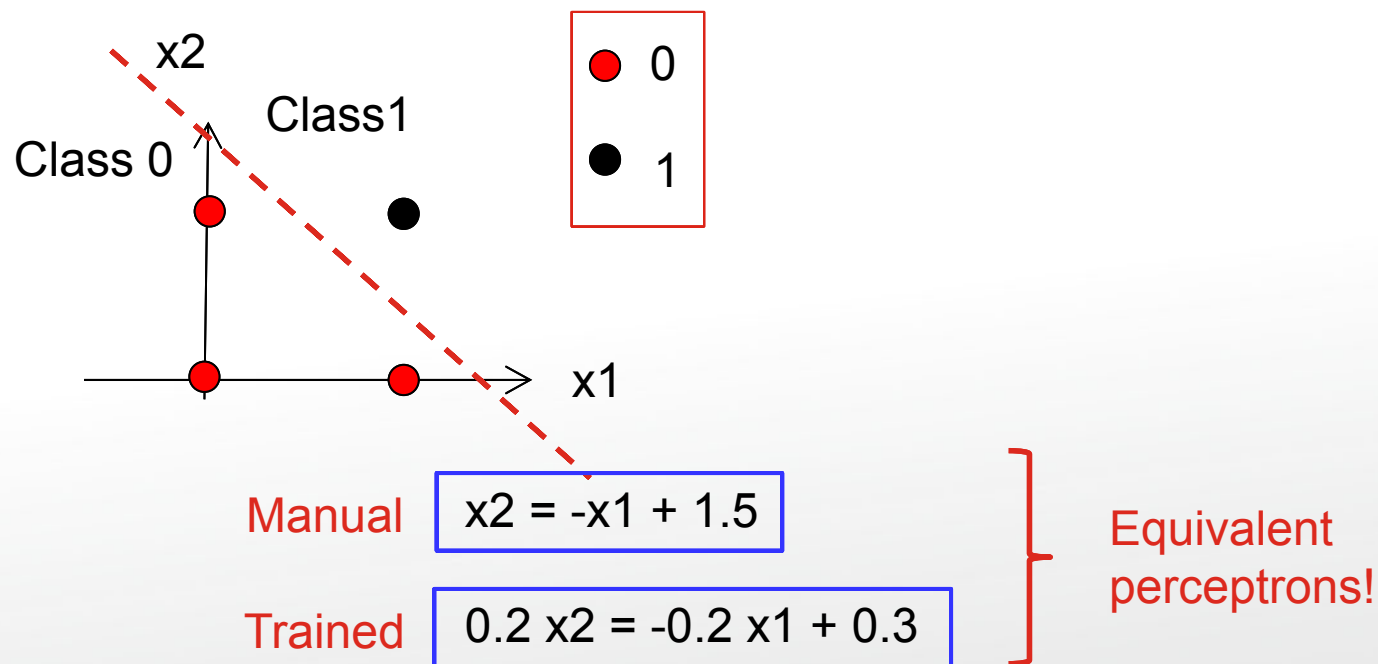
- Because of misclassification, update  $w$  to:

$$w'(11) = w'(10) + \eta(d - y)x' = \begin{bmatrix} -0.2 \\ 0.3 \\ 0.2 \end{bmatrix} + 0.1(0 - 1) \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.3 \\ 0.2 \\ 0.2 \end{bmatrix}$$

- 
- We can continue doing the same, and it will be observed that with  $w' = [-0.3, 0.2, 0.2]^T$  always gives the **correct classification**!
  - The perceptron is successfully trained after 3 epochs!

# Perceptron Learning Algorithm

- **Summary:** After three epochs, we arrive at the weight  $w' = [-0.3, 0.2, 0.2]^T$  which **correctly classifies** all the data point.
- It **looks** different from what we manually calculated ( $w'_{\text{manual}} = [-1.5, 1, 1]^T$ ) but is actually the same!





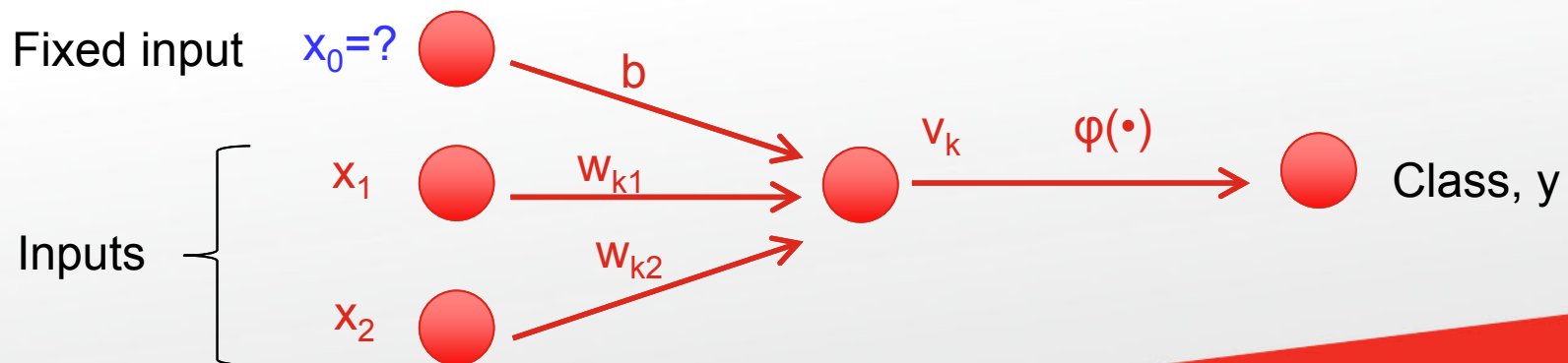
# 2D Example - Exercise

- A **2-dimensional** example: To correctly classify the external inputs  $\{x_1, x_2\}$  into one of two classes  $\{C_1 \text{ or } C_0\}$ .
- E.g. **OR binary function**:

x1	0	0	1	1
x2	0	1	0	1
d				

2D

- The perceptron is shown below. We want to find  **$b$ ,  $w_{k1}$  and  $w_{k2}$** , so that when given  $x_1$  and  $x_2$ , the correct  $y$  will be calculated.



# MATLAB Code

```
clear all
close all
clc

%%%%%%%%%
% Data %
%%%%%%%%%

% In the order of [x0 x1 x2 d]
Data = [1 0 0 0;
        1 0 1 0;
        1 1 0 0;
        1 1 1 1];

%%%%%%%%%
% Parameters %
%%%%%%%%%

w = [0.5, 0.5, 0.5]'; % [bias w1 w2]
eta = 0.1; % Try changing this and observe results
epochs = 4;
wrecord = w;
```

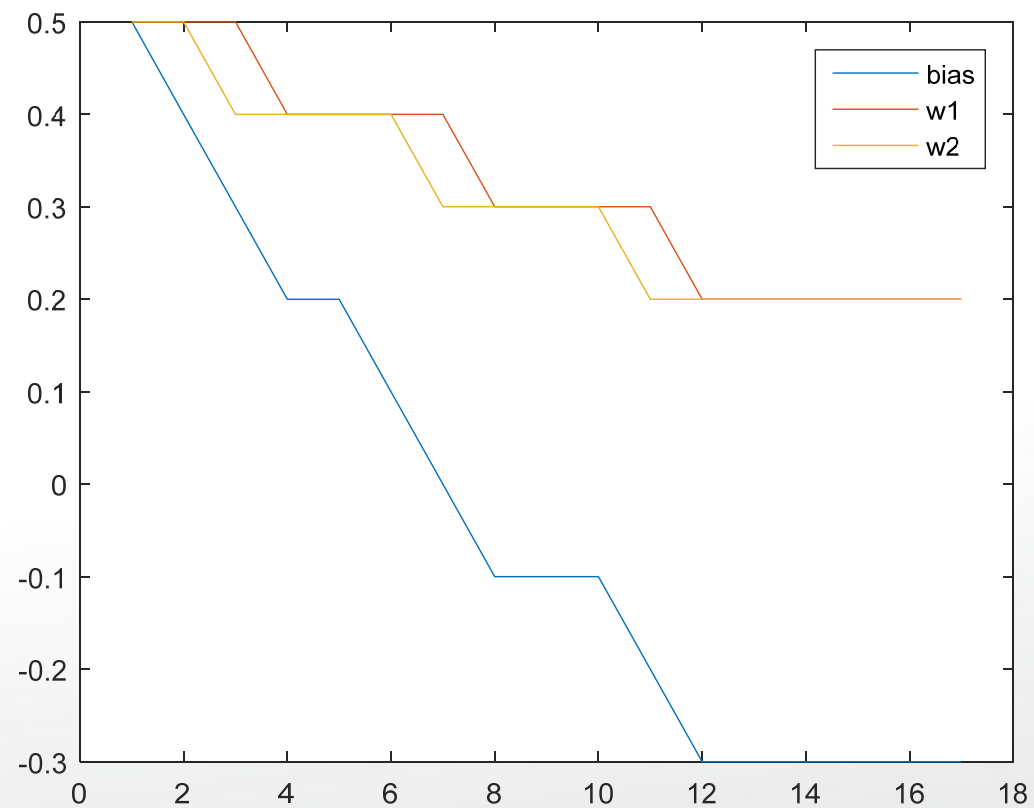
```
%%%%%%%%%
% Algorithm %
%%%%%%%%%

[ndata, mdata] = size(Data);

for i = 1:epochs
    for j = 1:ndata
        x = Data(j, 1:3)';
        v = w'*x;
        if v >= 0
            y = 1;
        else
            y = 0;
        end
        d = Data(j, 4);
        w = w + eta*(d-y)*x;
        wrecord = [wrecord w];
    end
end
figure, plot(wrecord(1,:))
hold on, plot(wrecord(2,:))
hold on, plot(wrecord(3,:))
legend('bias', 'w1', 'w2')
```

# MATLAB Code

- Progress of the weights during training.

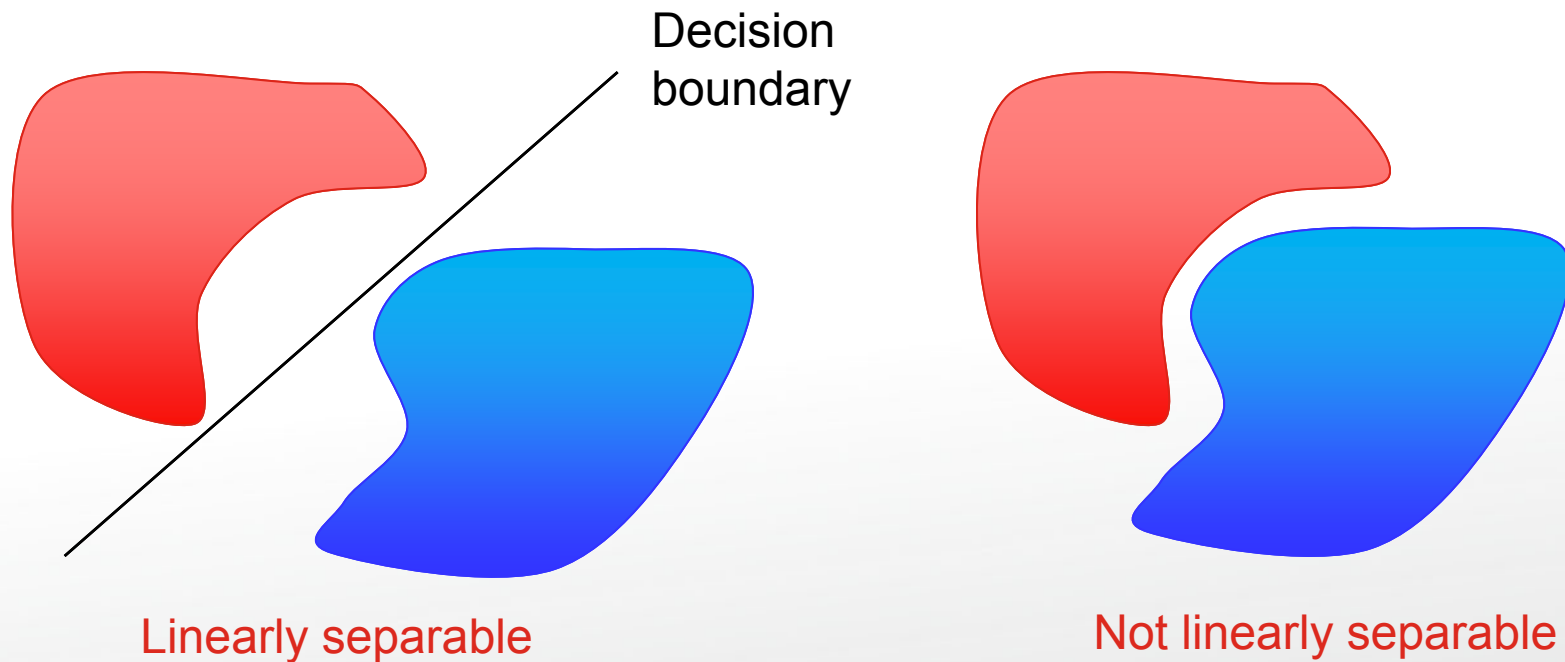


# Learning Rate

- The parameter  $\eta > 0$  influences the **learning rate**.
- Small value leads to slow learning.
- Large value can “spoil” the learning that has taken place earlier with respect to other data points.
- Therefore, some **medium** value is the best.
  - What “medium” means depends on the problem being solved.
- **Exercise:** Try changing the  $\eta$  value in the MATLAB code earlier, and observe the results.
  - Note: You might need to increase the epochs to allow the weights to converge.

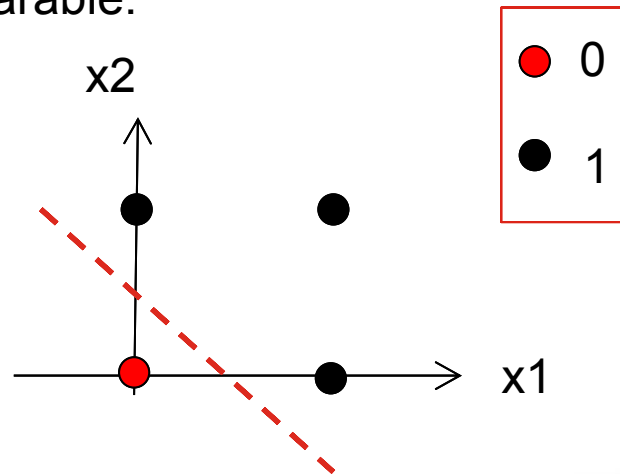
# Limitations of Perceptrons

- Perceptrons can only classify two classes which are **linearly separable**:
  - E.g. **2-D case**, if the classes cannot be separated by a **straight line**, then they cannot be classified by simple perceptrons.

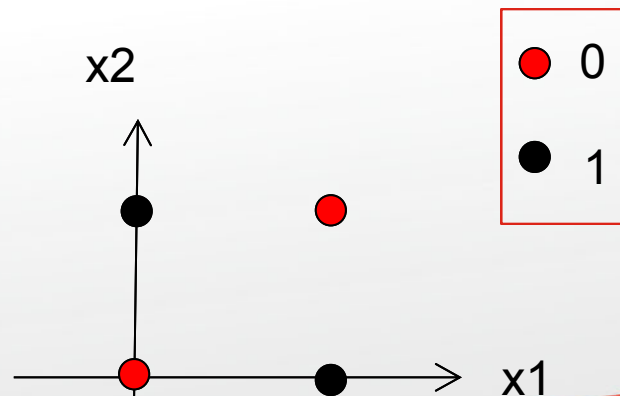


# Limitations of Perceptrons

- AND – Linearly separable, as already seen earlier.
- OR – also linearly separable:



- XOR - ???

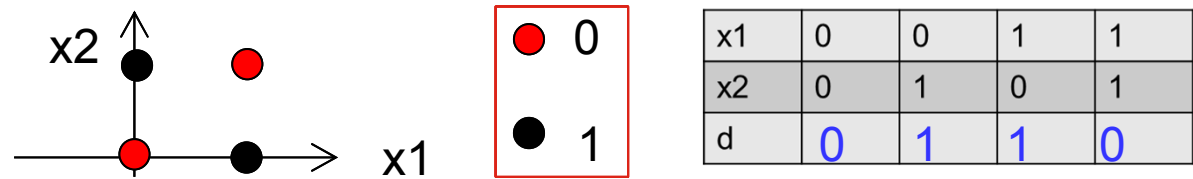


# Contents

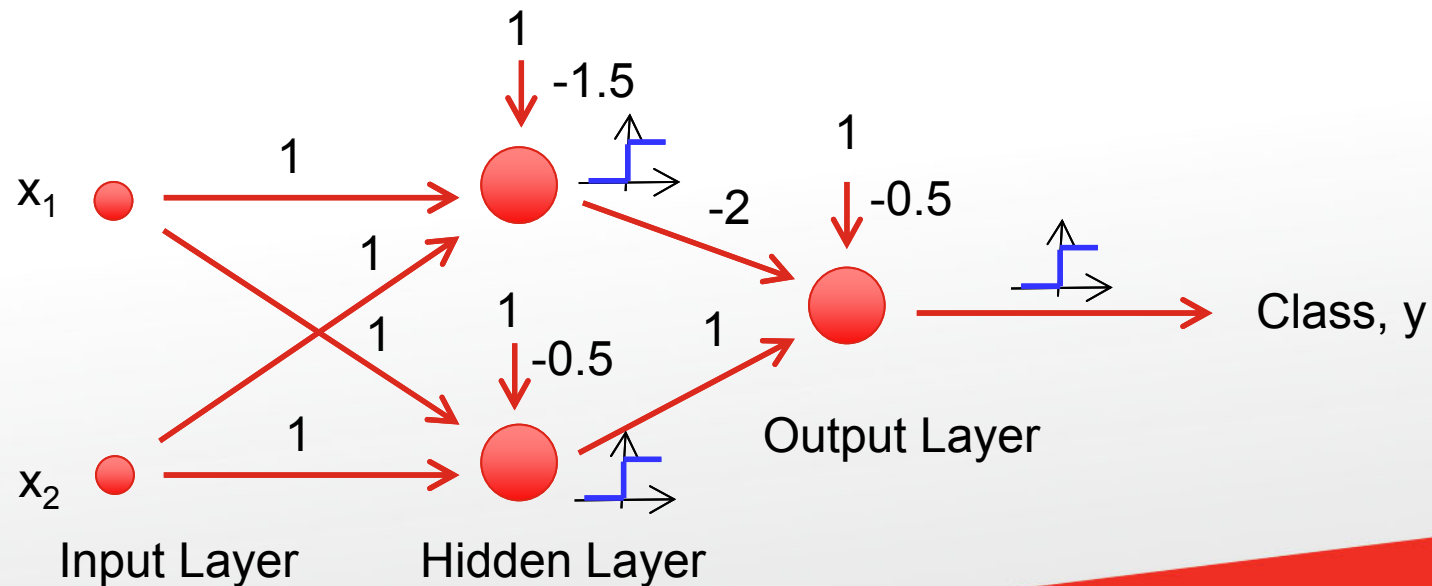
- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- Perceptron
- **Multilayer Perceptrons**
- Generalization
- MATLAB Example
- Deep Neural Networks

# XOR Problem

- We saw that a single perceptron is not able to solve the **XOR** function.



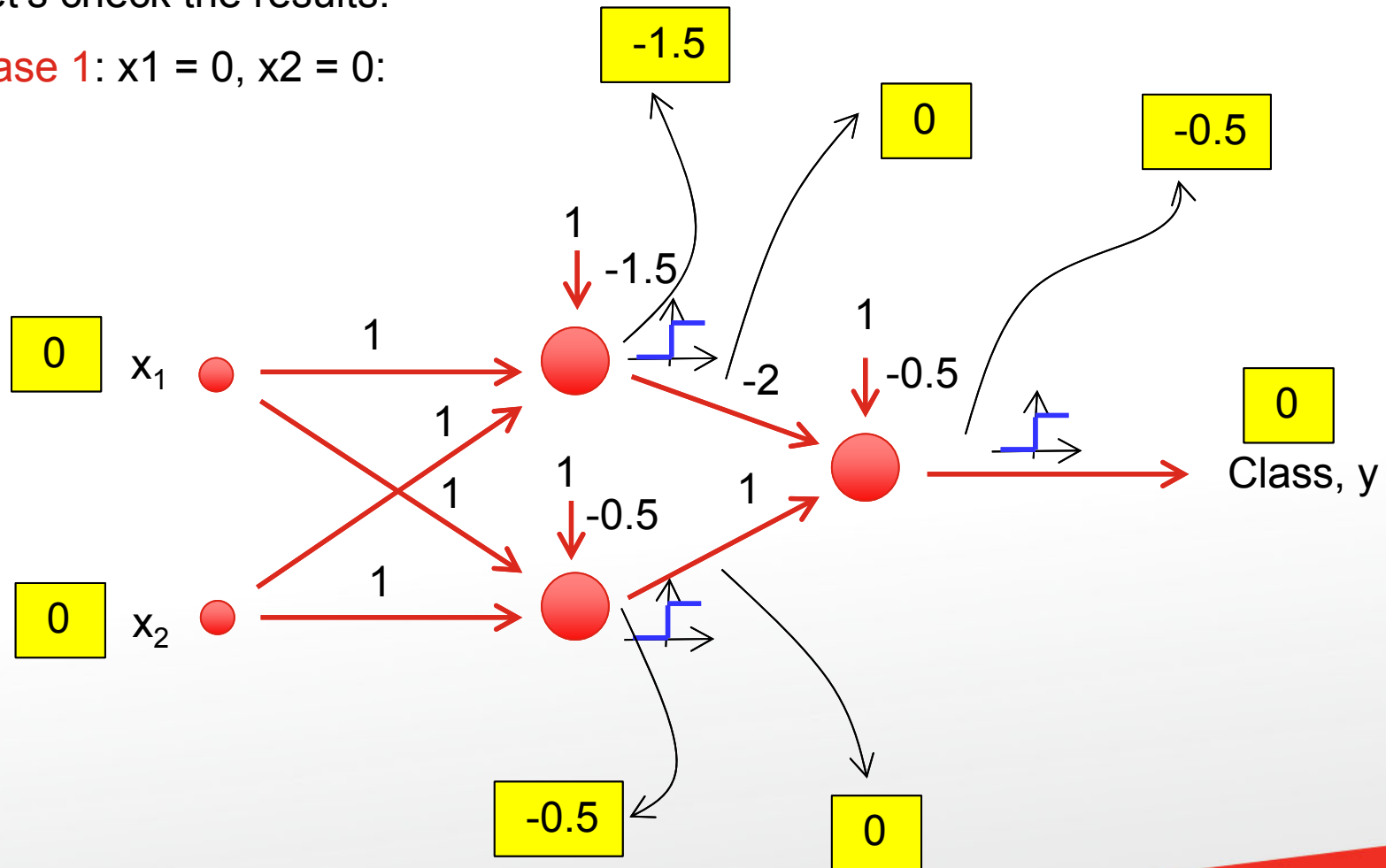
- In order to do that, we need **a few perceptrons** working together. For instance, consider the following network by Touretzky and Pomerlau (1989):





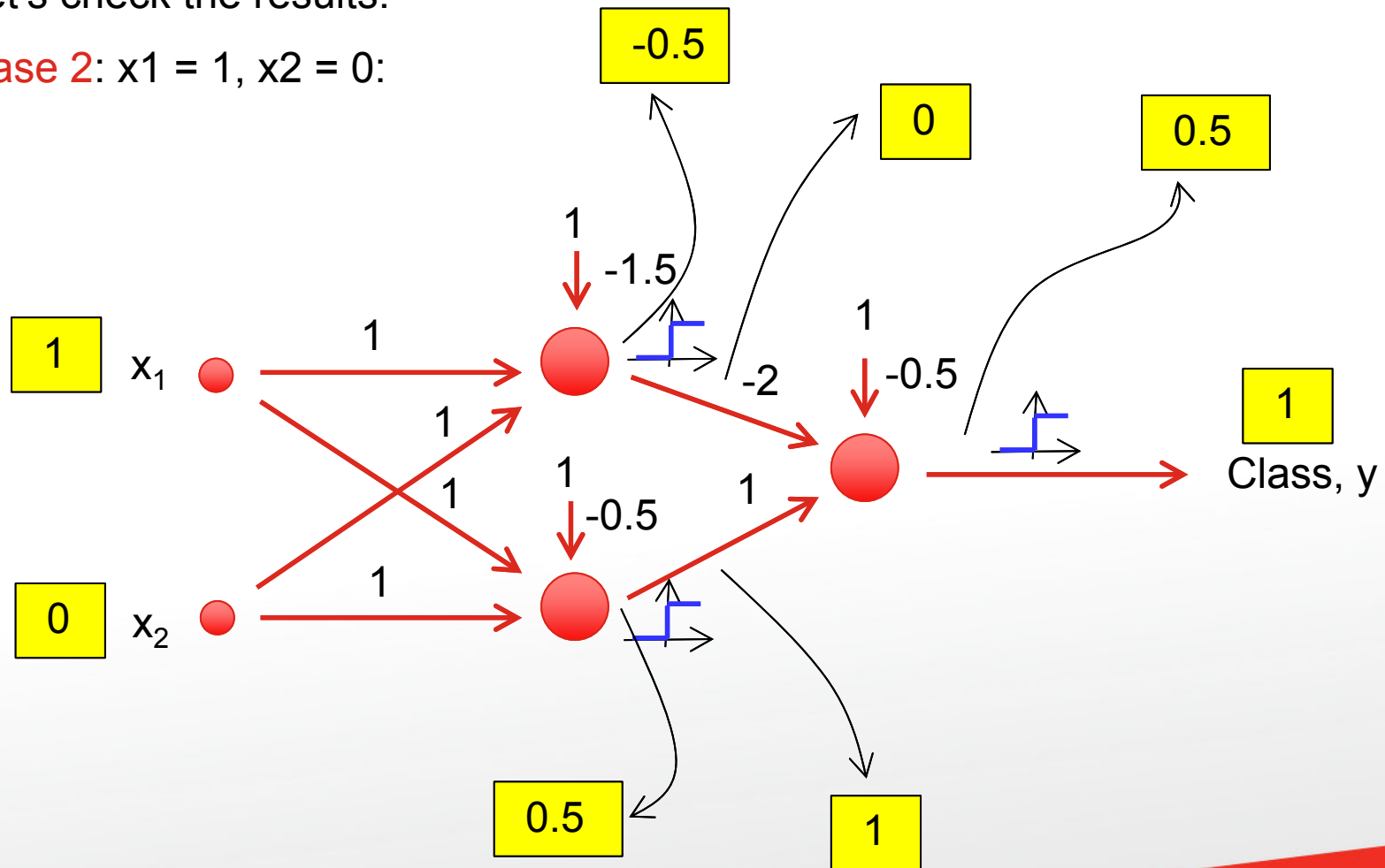
# XOR Problem

- Let's check the results:
- Case 1:  $x_1 = 0, x_2 = 0$ :



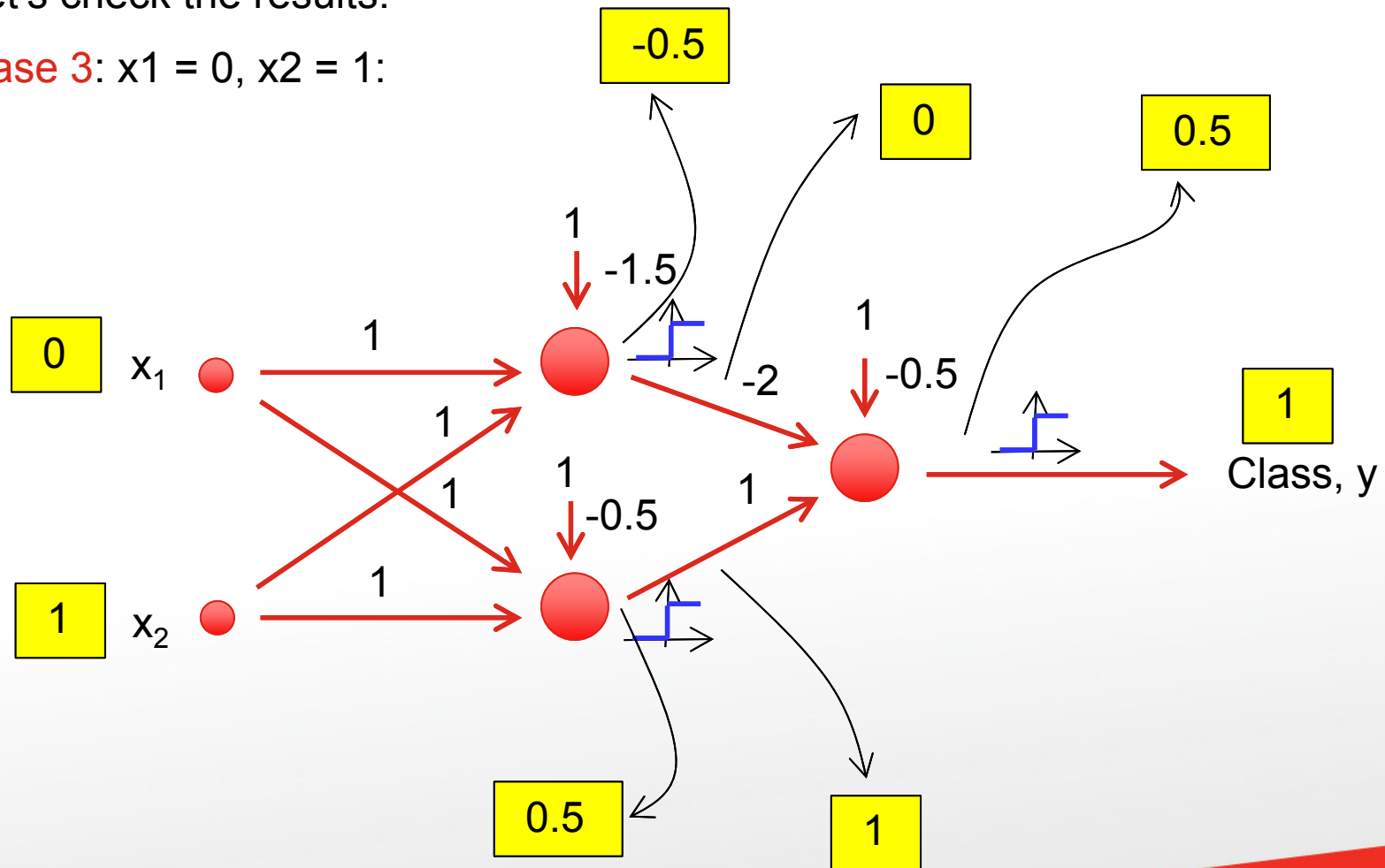
# XOR Problem

- Let's check the results:
- Case 2:**  $x_1 = 1, x_2 = 0$ :



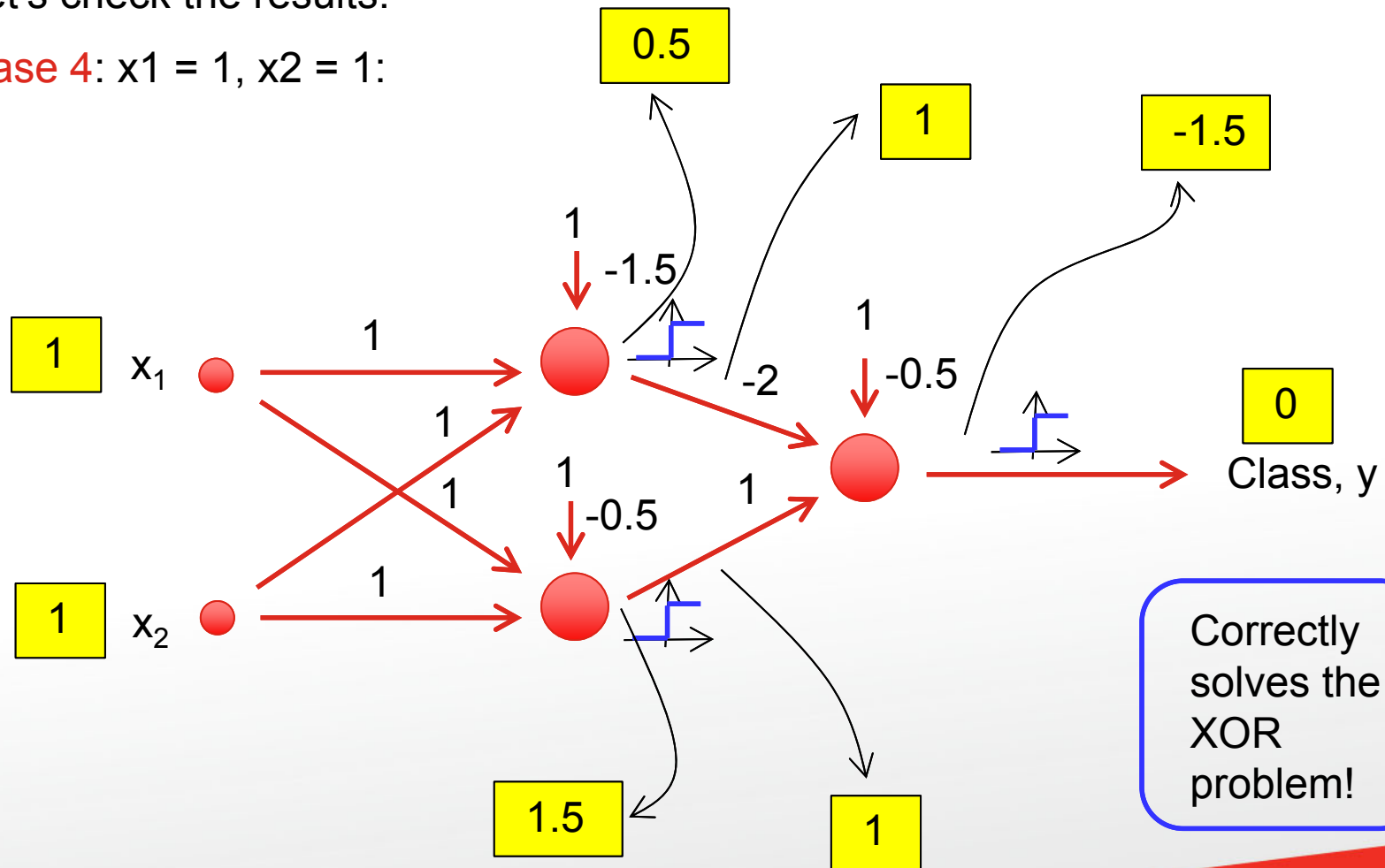
# XOR Problem

- Let's check the results:
- Case 3:  $x_1 = 0$ ,  $x_2 = 1$ :



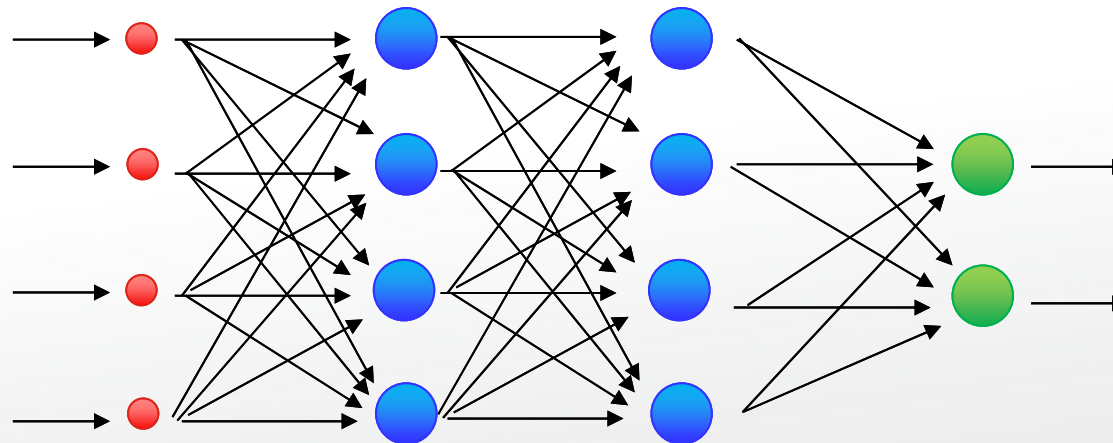
# XOR Problem

- Let's check the results:
- Case 4:  $x_1 = 1, x_2 = 1$ :



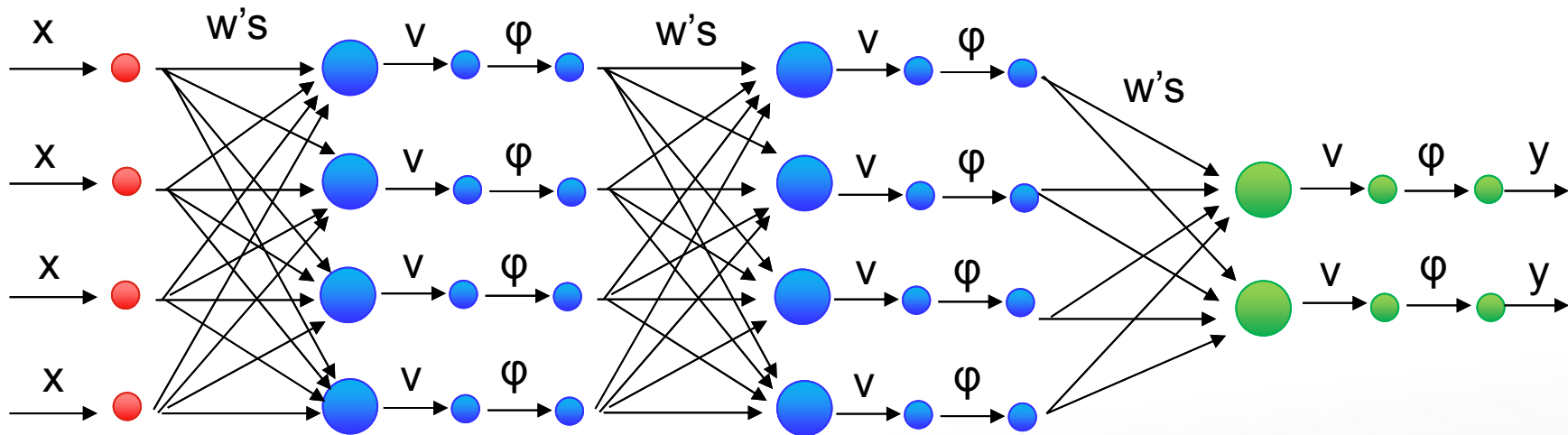
# Multilayer Perceptron

- As can be seen, by connecting multiple perceptrons together, more complicated problems can be solved.
- A **Multilayer Perceptron (MLP)** consists of:
  - An **input layer** (note: no computation here)
  - One or more **hidden layers** of computation nodes
  - An **output layer** of computation nodes
- E.g.



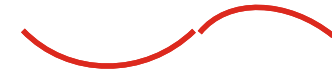
# Multilayer Perceptron

- It is sometimes advantageous to “explode up” each layer and show the intermediate results, for the ease of tracing the signals:



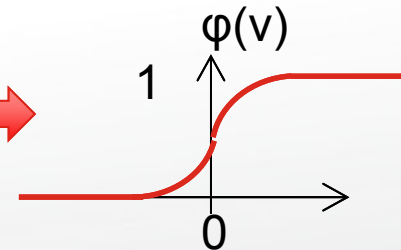
# Multilayer Perceptron

- Generally uses **differentiable activation functions** instead of threshold function.
  - This is because for training of the weights, we perform **optimization** which require the functions to be differentiable.
  - Just in case you have forgotten: optimization (maximization or minimization) usually requires calculating  $dy/dx=0$ .



- Popular activation functions:

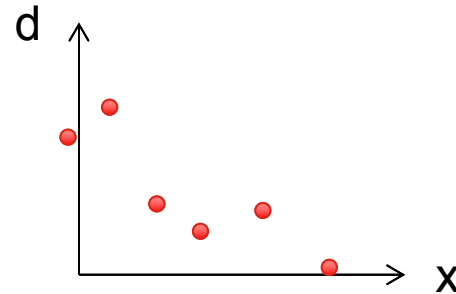
- **Logistic function** for hidden layer;
- **Linear** for output layer.



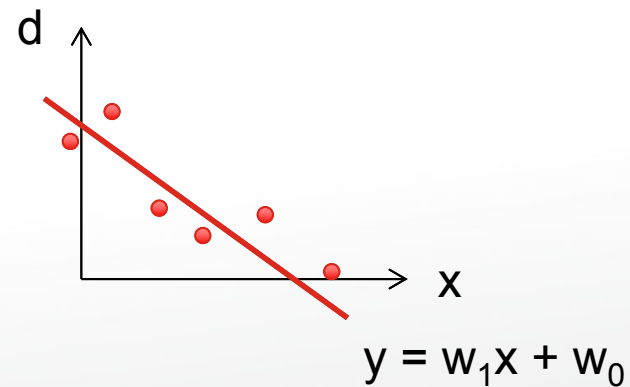
$$\varphi(v) = \frac{1}{1 + e^{-av}}$$

# Revision: Steepest Descent

- Consider the following data:



- We want to find a **best-fit straight line** so that the total error is the least.

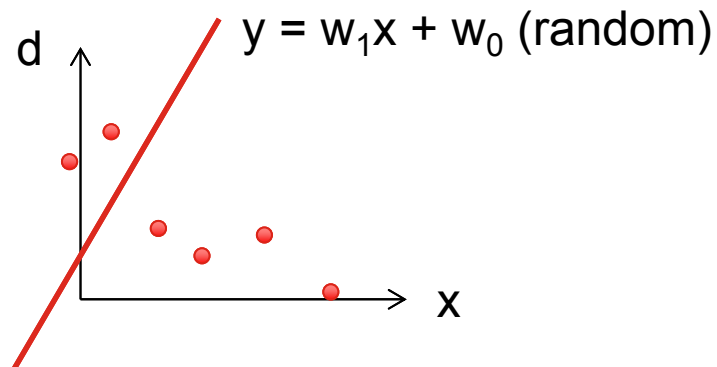


- How do we do that?

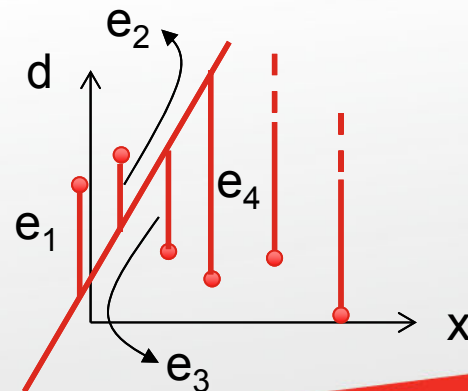


# Revision: Steepest Descent

- Firstly, we randomly generate any  $w_1$  and  $w_0$ .
- This will obviously give a very poor fit.



- We get the **modelling error** at each data point, then calculate the **sum of squared errors**.



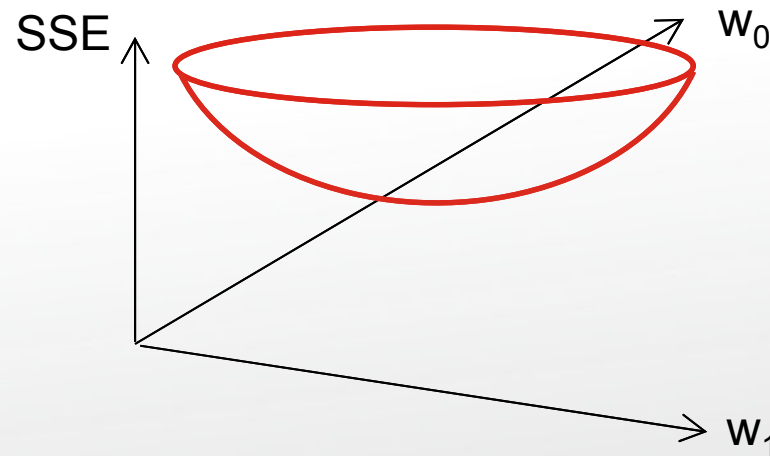
$$\text{SSE} = e_1^2 + e_2^2 + e_3^2 + \dots e_m^2$$

# Revision: Steepest Descent

- The **sum of squared errors**, in terms of the weights, is:

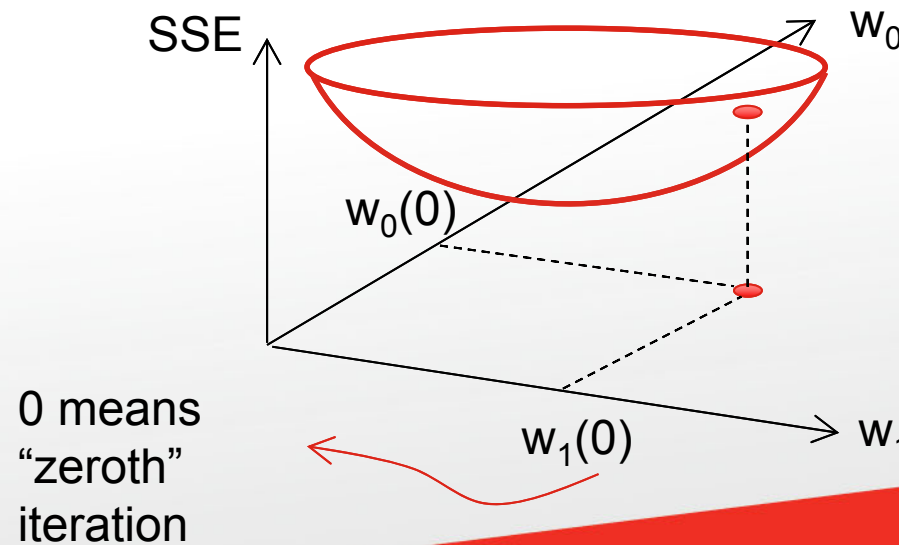
$$\begin{aligned}\text{SSE} &= e_1^2 + e_2^2 + e_3^2 + \dots + e_m^2 \\ &= (d_1 - y_1)^2 + \dots + (d_m - y_m)^2 \\ &= (d_1 - w_1x_1 - w_0)^2 + \dots + (d_m - w_1x_m - w_0)^2\end{aligned}$$

- If we plot a graph of **SSE vs all possible combinations of  $w_1$  and  $w_0$** , it would look like:



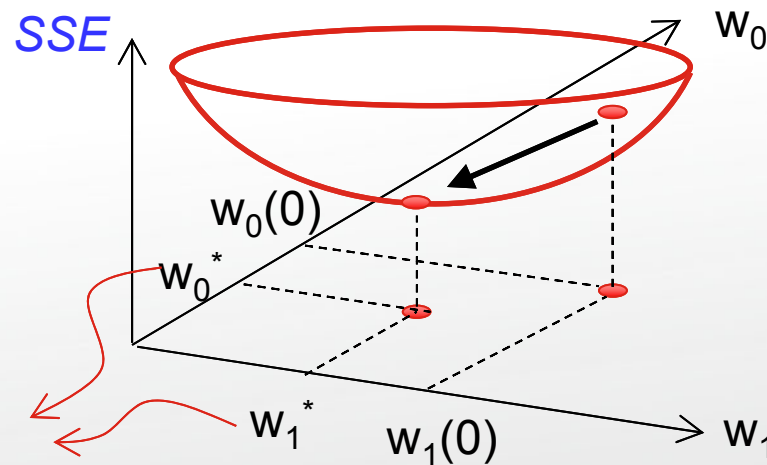
# Revision: Steepest Descent

- At the current moment, with the **randomly initialized weights**, we could be having the following SSE:



# Revision: Steepest Descent

- We want to use an algorithm such that the **weights move** to the optimal values.



\* Means optimal values, which gives least **SSE**  
*sum of squared errors.*

# Revision: Steepest Descent

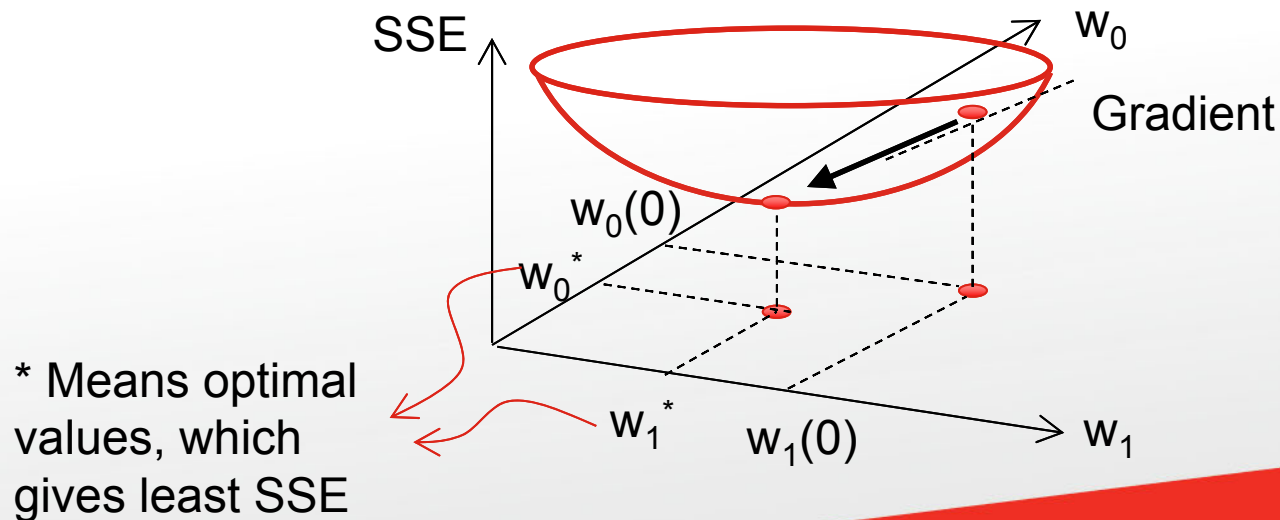
- The idea is to **adjust the weight in the direction of steepest descent**.
  - Where the direction of steepest descent is **opposite to the gradient**.

$$w_0(n+1) = w_0(n) - \eta \frac{\partial(\text{SSE})}{\partial(w_0)}$$

and

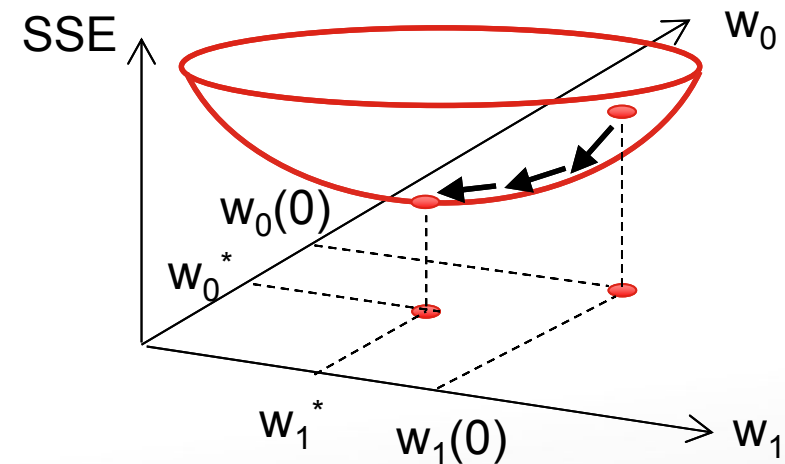
$$w_1(n+1) = w_1(n) - \eta \frac{\partial(\text{SSE})}{\partial(w_1)}$$

- $\eta$  is the learning rate parameter.

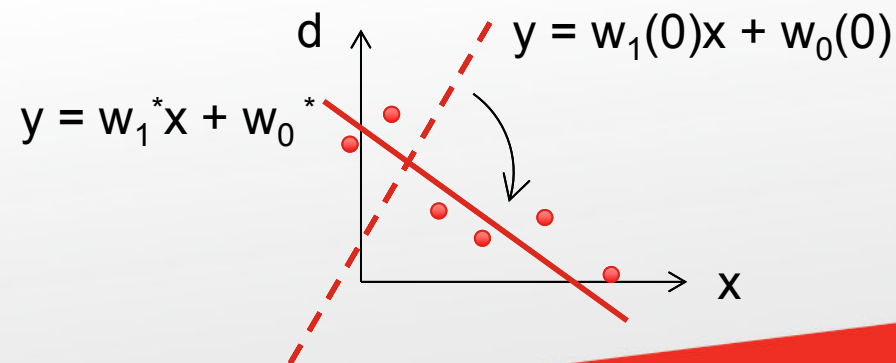


# Revision: Steepest Descent

- Depending on  $\eta$ , it is very likely that it will take several iterations to reach the optimum point.
- Therefore, this is an **iterative process**.

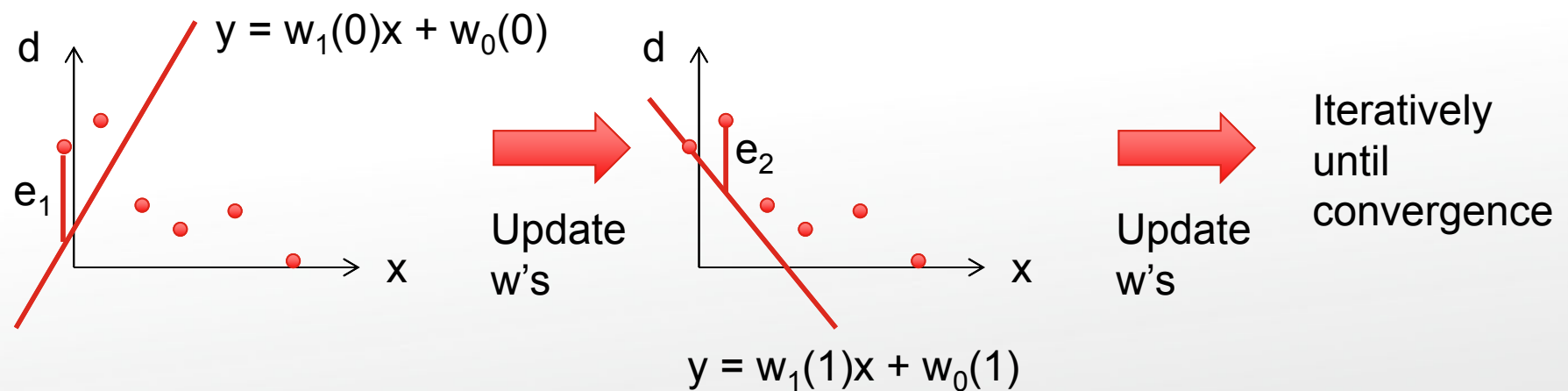


- At the end, the optimal values would give the best-fit line:



# Revision: Steepest Descent

- What you have seen so far is “**batch**” mode, i.e. all the data are presented at once.
- Another possibility is the “**sequential**” mode, where data is presented **one at a time**, and the weights are updated after presentation of each data.
  - Similar to the perceptron learning algorithm earlier.
- Pictorially, it means:



# Revision: Steepest Descent

- For sequential mode, since only one data is presented at a particular instance, we do not have \*sum\* of squared errors. Rather, we have the **instantaneous** squared error.

$$E(w, i) = \frac{1}{2} e(i)^2$$

where  $i$  refers to time  $i$ .

- For the same example,  $E(w, i) = \frac{1}{2} e(i)^2 = \frac{1}{2} (d_i - w_1 x_i - w_0)^2$
- The weights are updated as follows:

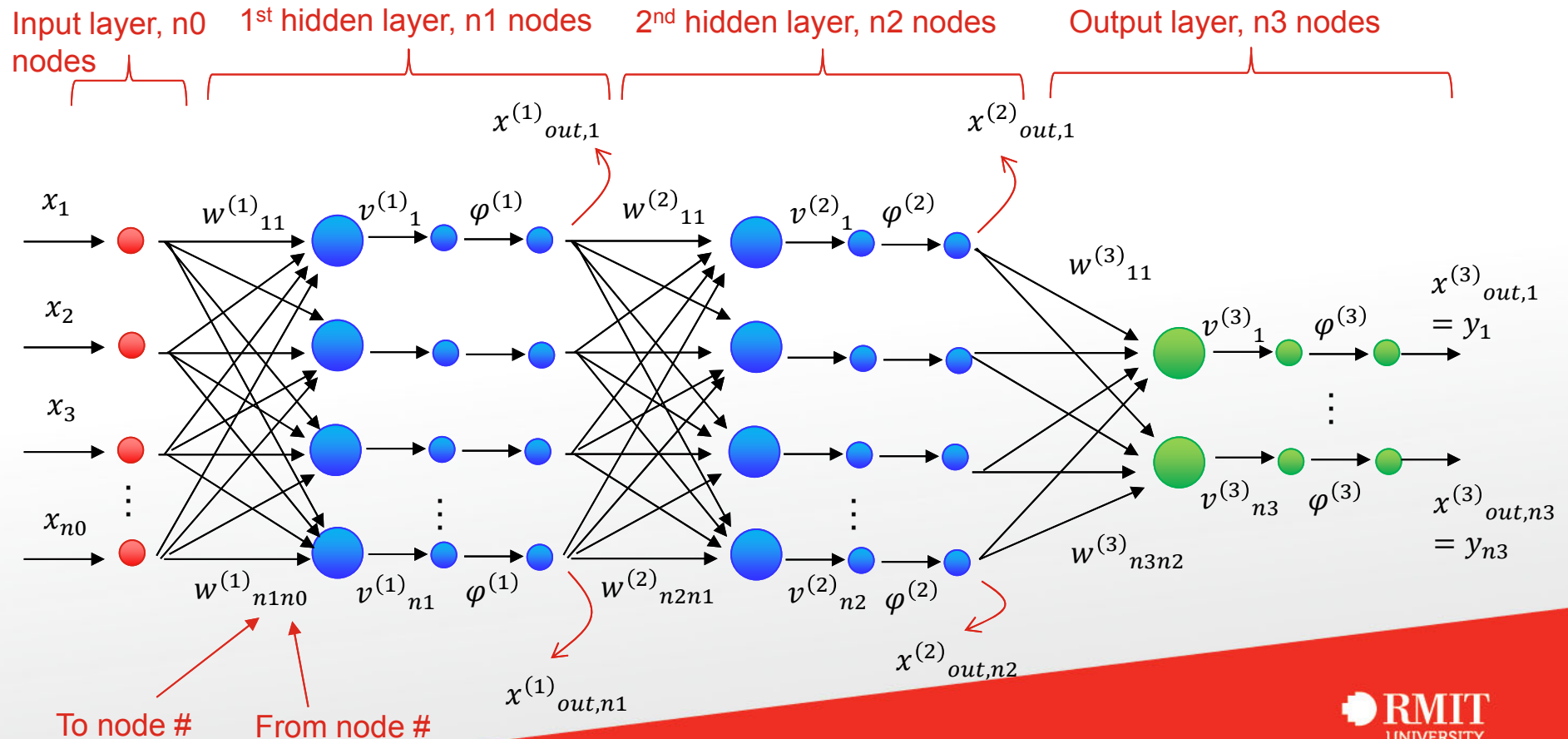
$$\begin{aligned} w_0(n+1) &= w_0(n) - \eta \frac{\partial(E)}{\partial(w_0)} \\ &= w_0(n) + \eta (d_i - w_1 x_i - w_0) \\ &= w_0(n) + \eta e(i) \end{aligned}$$

$$\begin{aligned} w_1(n+1) &= w_1(n) - \eta \frac{\partial(E)}{\partial(w_1)} \\ &= w_1(n) + \eta x_i (d_i - w_1 x_i - w_0) \\ &= w_1(n) + \eta x_i e(i) \end{aligned}$$



# Backpropagation (BP) Algorithm for MLP

- Now that you have understood the method of steepest descent, particularly for the sequential mode, it would be quite easy to understand the algorithm to **update the weights in Multilayer Perceptron (MLP)**.
- Consider the following MLP, with proper labelling of the signals and weights.



# BP for MLP – Output Layer

- The instantaneous squared errors of all the outputs for one instance of input data is:

$$E = \frac{1}{2}(d_1 - y_1)^2 + \frac{1}{2}(d_2 - y_2)^2 + \dots + \frac{1}{2}(d_{n3} - y_{n3})^2$$

- To update the output layer weights:
- Write the  $y$ 's in terms of output layer weights:

$$E = \frac{1}{2}\left(d_1 - \varphi^{(3)}(v^{(3)}_1)\right)^2 + \frac{1}{2}\left(d_2 - \varphi^{(3)}(v^{(3)}_2)\right)^2 + \dots + \frac{1}{2}\left(d_{n3} - \varphi^{(3)}(v^{(3)}_{n3})\right)^2$$

- The  $v$ 's are:

$$\begin{aligned} v^{(3)}_1 &= w^{(3)}_{11}x^{(2)}_{out,1} + w^{(3)}_{12}x^{(2)}_{out,2} + \dots + w^{(3)}_{1n2}x^{(2)}_{out,n2} \\ &\vdots \\ v^{(3)}_{n3} &= w^{(3)}_{n31}x^{(2)}_{out,1} + w^{(3)}_{n32}x^{(2)}_{out,2} + \dots + w^{(3)}_{n3n2}x^{(2)}_{out,n2} \end{aligned}$$

# BP for MLP – Output Layer

- Let's use  $w_{11}^{(3)}$  as example:

$$w_{11,new}^{(3)} = w_{11,old}^{(3)} - \eta^{(3)} \frac{\partial(E)}{\partial(w_{11}^{(3)})}$$

- But:

$$\frac{\partial(E)}{\partial(w_{11}^{(3)})} = \frac{\partial(E)}{\partial(v_1^{(3)})} \cdot \frac{\partial(v_1^{(3)})}{\partial(w_{11}^{(3)})}$$

Chain rule

$$\begin{aligned} \frac{\partial(E)}{\partial(v_1^{(3)})} &= \frac{\partial\left(\frac{1}{2}(d_1 - \varphi^{(3)}(v_1^{(3)}))^2 + \dots\right)}{\partial(v_1^{(3)})} \\ &= -\left(d_1 - \varphi^{(3)}(v_1^{(3)})\right) \varphi^{(3)'}(v_1^{(3)}) \\ &\triangleq -\delta_1^{(3)} \end{aligned}$$

$$\frac{\partial(v_1^{(3)})}{\partial(w_{11}^{(3)})} = x_{out,1}^{(2)}$$

- Therefore:

$$w_{11,new}^{(3)} = w_{11,old}^{(3)} + \eta^{(3)} \delta_1^{(3)} x_{out,1}^{(2)}$$

# BP for MLP – Output Layer

- Similarly, for all other **weights in the output layer**, we have:

$$w^{(3)}_{ji,new} = w^{(3)}_{ji,old} + \eta^{(3)} \delta_j^{(3)} x^{(2)}_{out,i}$$

- where

$$\delta_j^{(3)} = \left( d_j - \varphi^{(3)}(v^{(3)}_j) \right) \varphi^{(3)'}(v^{(3)}_j)$$

- is called the **local gradient** (or local error) at the  $j$ -th neuron.

# BP for MLP – Hidden Layer

- Next, let's update the weights of the 2<sup>nd</sup> hidden layer.
- The **instantaneous squared errors** of all the outputs for one instance of input data is:

$$E = \frac{1}{2}(d_1 - y_1)^2 + \frac{1}{2}(d_2 - y_2)^2 + \dots + \frac{1}{2}(d_{n3} - y_{n3})^2$$

- Write the  $y$ 's in terms of **output layer weights**:

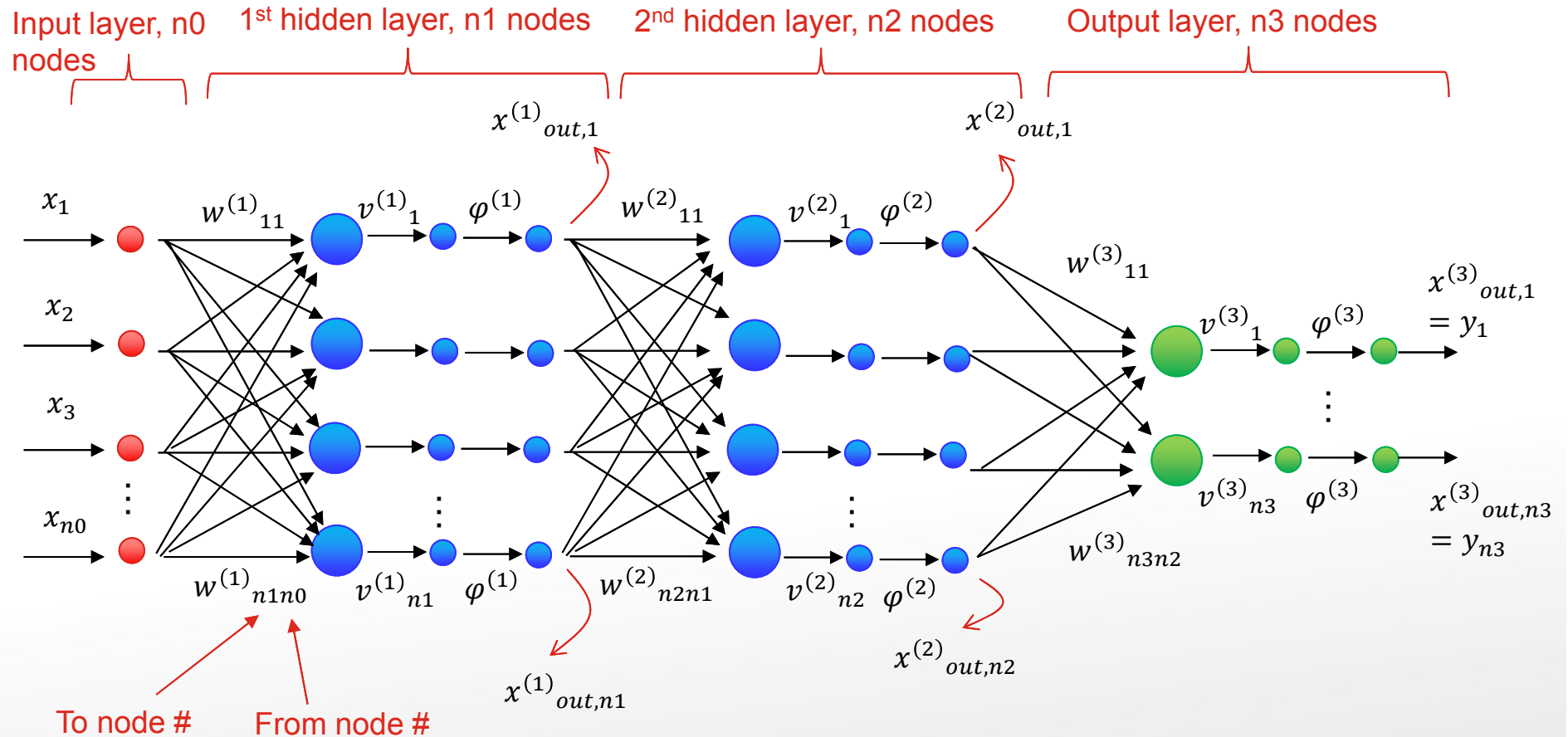
$$E = \frac{1}{2}\left(d_1 - \varphi^{(3)}(v^{(3)}_1)\right)^2 + \frac{1}{2}\left(d_2 - \varphi^{(3)}(v^{(3)}_2)\right)^2 + \dots + \frac{1}{2}\left(d_{n3} - \varphi^{(3)}(v^{(3)}_{n3})\right)^2$$

- The  $v$ 's are:

$$\begin{aligned} v^{(3)}_1 &= w^{(3)}_{11}x^{(2)}_{out,1} + w^{(3)}_{12}x^{(2)}_{out,2} + \dots + w^{(3)}_{1n2}x^{(2)}_{out,n2} \\ &\vdots \\ v^{(3)}_{n3} &= w^{(3)}_{n31}x^{(2)}_{out,1} + w^{(3)}_{n32}x^{(2)}_{out,2} + \dots + w^{(3)}_{n3n2}x^{(2)}_{out,n2} \end{aligned}$$

Note: These are the same as 3 slides earlier

# BP for MLP – Hidden Layer



# BP for MLP – Hidden Layer

- We continue to express the terms using signals/weights of 2<sup>nd</sup> hidden layers:

$$x^{(2)}_{out,1} = \varphi^{(2)}(v^{(2)}_1)$$

$$v^{(2)}_1 = w^{(2)}_{11}x^{(1)}_{out,1} + w^{(2)}_{12}x^{(1)}_{out,2} + \cdots + w^{(2)}_{1n1}x^{(1)}_{out,n1}$$

$$x^{(2)}_{out,2} = \varphi^{(2)}(v^{(2)}_2)$$

$$v^{(2)}_2 = w^{(2)}_{21}x^{(1)}_{out,1} + w^{(2)}_{22}x^{(1)}_{out,2} + \cdots + w^{(2)}_{2n1}x^{(1)}_{out,n1}$$

$\vdots$

$$x^{(2)}_{out,n2} = \varphi^{(2)}(v^{(2)}_{n2})$$

$$v^{(2)}_{n2} = w^{(2)}_{n21}x^{(1)}_{out,1} + w^{(2)}_{n22}x^{(1)}_{out,2} + \cdots + w^{(2)}_{n2n1}x^{(1)}_{out,n1}$$

# BP for MLP – Hidden Layer

- Let's use  $w^{(2)}_{11}$  as example:

$$w^{(2)}_{11,new} = w^{(2)}_{11,old} - \eta^{(2)} \frac{\partial(E)}{\partial(w^{(2)}_{11})}$$

- If we look at the equations in the previous two slides carefully, we see that:
  - $w^{(2)}_{11}$  appears in the  $v^{(2)}_1$  equation.
  - $v^{(2)}_1$  appears in the  $x^{(2)}_{out,1}$  equation.
  - $x^{(2)}_{out,1}$  appears several times in  $v^{(3)}_1, v^{(3)}_2, \dots, v^{(3)}_{n3}$  equations.
  - $v^{(3)}_1, v^{(3)}_2, \dots, v^{(3)}_{n3}$  appear in  $E$  equation.
- By using **chain rule**, we can get:

$$\begin{aligned} \frac{\partial(E)}{\partial(w^{(2)}_{11})} = & \frac{\partial(E)}{\partial(v^{(3)}_1)} \frac{\partial(v^{(3)}_1)}{\partial(x^{(2)}_{out,1})} \frac{\partial(x^{(2)}_{out,1})}{\partial(v^{(2)}_1)} \frac{\partial(v^{(2)}_1)}{\partial(w^{(2)}_{11})} + \frac{\partial(E)}{\partial(v^{(3)}_2)} \frac{\partial(v^{(3)}_2)}{\partial(x^{(2)}_{out,1})} \frac{\partial(x^{(2)}_{out,1})}{\partial(v^{(2)}_1)} \frac{\partial(v^{(2)}_1)}{\partial(w^{(2)}_{11})} \\ & + \dots + \frac{\partial(E)}{\partial(v^{(3)}_{n3})} \frac{\partial(v^{(3)}_{n3})}{\partial(x^{(2)}_{out,1})} \frac{\partial(x^{(2)}_{out,1})}{\partial(v^{(2)}_1)} \frac{\partial(v^{(2)}_1)}{\partial(w^{(2)}_{11})} \end{aligned}$$



# BP for MLP – Hidden Layer

- Combining **similar terms**, we get:

$$\frac{\partial(E)}{\partial(w^{(2)}_{11})} = \left( \begin{aligned} &\frac{\partial(E)}{\partial(v^{(3)}_1)} \frac{\partial(v^{(3)}_1)}{\partial(x^{(2)}_{out,1})} \\ &+ \frac{\partial(E)}{\partial(v^{(3)}_2)} \frac{\partial(v^{(3)}_2)}{\partial(x^{(2)}_{out,1})} \\ &+ \dots + \frac{\partial(E)}{\partial(v^{(3)}_{n3})} \frac{\partial(v^{(3)}_{n3})}{\partial(x^{(2)}_{out,1})} \end{aligned} \right) \frac{\partial(x^{(2)}_{out,1})}{\partial(v^{(2)}_1)} \frac{\partial(v^{(2)}_1)}{\partial(w^{(2)}_{11})}$$

- Good news!**  $\frac{\partial(E)}{\partial(v^{(3)}_1)}$ ,  $\frac{\partial(E)}{\partial(v^{(3)}_2)}$ , ...,  $\frac{\partial(E)}{\partial(v^{(3)}_{n3})}$  have already been calculated when we updated output layer just now. They are:

$$\frac{\partial(E)}{\partial(v^{(3)}_1)} = -\delta_1^{(3)} \quad \frac{\partial(E)}{\partial(v^{(3)}_2)} = -\delta_2^{(3)} \quad \dots \quad \frac{\partial(E)}{\partial(v^{(3)}_{n3})} = -\delta_{n3}^{(3)}$$

# BP for MLP – Hidden Layer

- Next,

$$\frac{\partial(v^{(3)}_1)}{\partial(x^{(2)}_{out,1})} = w^{(3)}_{11} \quad \frac{\partial(v^{(3)}_2)}{\partial(x^{(2)}_{out,1})} = w^{(3)}_{21} \quad \dots \quad \frac{\partial(v^{(3)}_{n3})}{\partial(x^{(2)}_{out,1})} = w^{(3)}_{n31}$$

$$\frac{\partial(x^{(2)}_{out,1})}{\partial(v^{(2)}_1)} = \varphi'^{(2)}(v^{(2)}_1)$$

$$\frac{\partial(v^{(2)}_1)}{\partial(w^{(2)}_{11})} = x^{(1)}_{out,1}$$

# BP for MLP – Hidden Layer

- Putting all the terms together in the **chain rule**, we get:

$$\begin{aligned}w_{11,new}^{(2)} &= w_{11,old}^{(2)} - \eta^{(2)} \frac{\partial(E)}{\partial(w_{11}^{(2)})} \\&= w_{11,old}^{(2)} - \eta^{(2)} \left( \begin{aligned} &\frac{\partial(E)}{\partial(v_{11}^{(3)})} \frac{\partial(v_{11}^{(3)})}{\partial(x_{out,1}^{(2)})} \\ &+ \frac{\partial(E)}{\partial(v_{21}^{(3)})} \frac{\partial(v_{21}^{(3)})}{\partial(x_{out,1}^{(2)})} \\ &+ \dots + \frac{\partial(E)}{\partial(v_{n3}^{(3)})} \frac{\partial(v_{n3}^{(3)})}{\partial(x_{out,1}^{(2)})} \end{aligned} \right) \frac{\partial(x_{out,1}^{(2)})}{\partial(v_{11}^{(2)})} \frac{\partial(v_{11}^{(2)})}{\partial(w_{11}^{(2)})} \\&= w_{11,old}^{(2)} + \eta^{(2)} \left( \begin{aligned} &\delta_1^{(3)} w_{11}^{(3)} \\ &+ \delta_2^{(3)} w_{21}^{(3)} \\ &+ \dots + \delta_{n3}^{(3)} w_{n31}^{(3)} \end{aligned} \right) \varphi'^{(2)}(v_{11}^{(2)}) x_{out,1}^{(1)}\end{aligned}$$

# BP for MLP – Hidden Layer

- Similarly, for all other weights in the hidden layer, we have:

$$\begin{aligned}w_{ji,new}^{(s)} &= w_{ji,old}^{(s)} - \eta^{(s)} \frac{\partial(E)}{\partial(w_{ji}^{(s)})} \\&= w_{ji,old}^{(s)} + \eta^{(s)} \left( \begin{aligned} &\delta_1^{(s+1)} w_{11}^{(s+1)} \\ &+ \delta_2^{(s+1)} w_{21}^{(s+1)} \\ &+ \dots + \delta_{ns+1}^{(s+1)} w_{ns+1,1}^{(s+1)} \end{aligned} \right) \varphi'^{(s)}(v_j^{(s)}) x_{out,i}^{(s-1)} \\&= w_{ji,old}^{(s)} + \eta^{(s)} \underbrace{\left( \sum_{k=1}^{ns+1} \delta_k^{(s+1)} w_{kj}^{(s+1)} \right)}_{\delta_j^{(s)}} \varphi'^{(s)}(v_j^{(s)}) x_{out,i}^{(s-1)} \\&= w_{ji,old}^{(s)} + \eta^{(s)} \delta_j^{(s)} x_{out,i}^{(s-1)}\end{aligned}$$

# BP for MLP - Summary

- In summary, for both output and hidden layers, the **weight updates** are:

$$w_{ji,new}^{(s)} = w_{ji,old}^{(s)} + \eta^{(s)} \delta_j^{(s)} x_{out,i}^{(s-1)}$$

- The difference lies in how the **local gradients** are calculated:

- Output layer:

$$\begin{aligned} \delta_j^{(s)} &= \left( d_j - \varphi^{(s)}(v^{(s)}_j) \right) \varphi'^{(s)}(v^{(s)}_j) \\ &= \left( d_j - x_{out,j}^{(s)} \right) \varphi'^{(s)}(v^{(s)}_j) \end{aligned}$$

- Hidden layer:

$$\delta_j^{(s)} = \left( \sum_{k=1}^{ns+1} \delta_k^{(s+1)} w_{kj}^{(s+1)} \right) \varphi'^{(s)}(v^{(s)}_j)$$

# BP for MLP - Summary

- At this point, it is good to remind ourselves that if the activation function is a logistic function,

$$\varphi(v) = \frac{1}{1 + e^{-av}} = (1 + e^{-av})^{-1}$$

- Then the derivative is:

$$\varphi'(v) = a \cdot (1 - \varphi(v)) \cdot \varphi(v)$$

# Backpropagation Algorithm

- Present first set of data.
- • **Forward pass:**
  - Use previously calculated weights. (For first iteration, randomly initialized).
  - Compute signals for each neuron
- **Backward pass:**
  - Starting from the output layer, the local gradient is calculated towards the first layer (hence “**backpropagation**”).
  - At each layer, the weights are updated based on the formula on previous slides.
- Present **next set of data**.
- Continue until convergence (i.e. squared error does not decrease much), or until the pre-set number of epochs are achieved.

# Contents

- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- **Generalization**
- MATLAB Example
- Deep Neural Networks

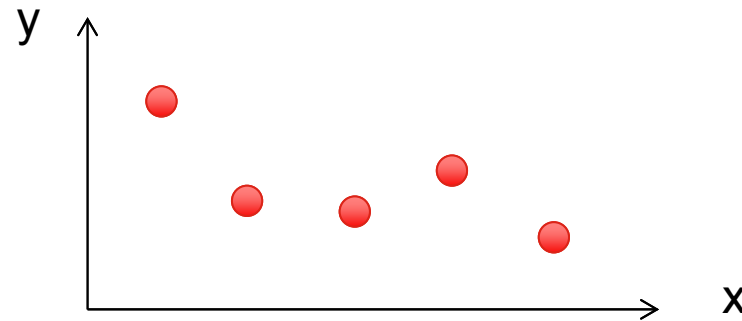


# Generalization

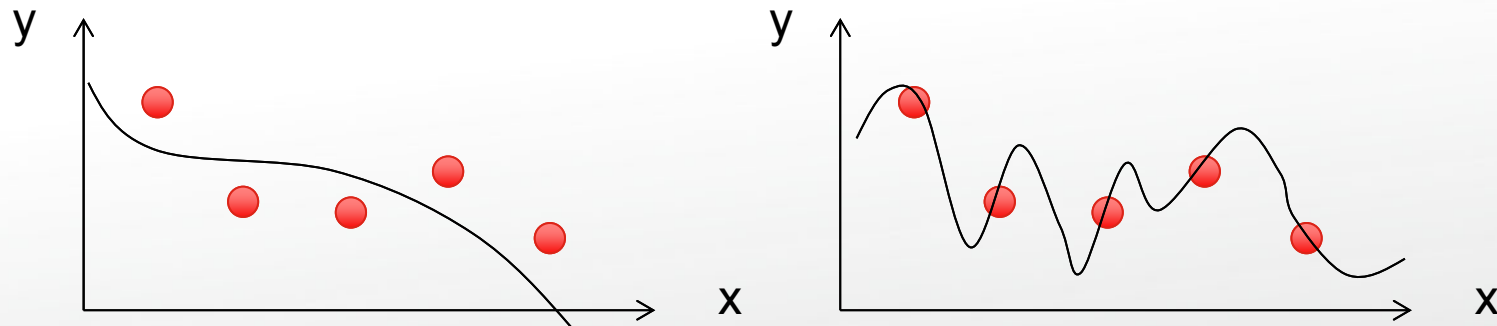
- When updating the weights, we present the network with a set of data, for which **we know what the desired outputs are**.
  - This is called “**supervised training**”, and the data set is called “**training set**”.
- After training, we would like to use the network for any new data coming in.
  - This “new” data set is called “**test set**”.
- For e.g. a network has been **trained** to predict the wind speed of an area, based on historical data of temperature, pressure, previous day wind speed etc.
- Then the network is used to **predict \*tomorrow’s\*** wind speed based on today’s temperature, pressure and yesterday’s wind speed.
  - Because today’s weather pattern is different from any other historical data, it will test the network’s ability to **generalize**.

# Generalization

- Another example: Assume we have some **training data**, which are represented by the red dots.

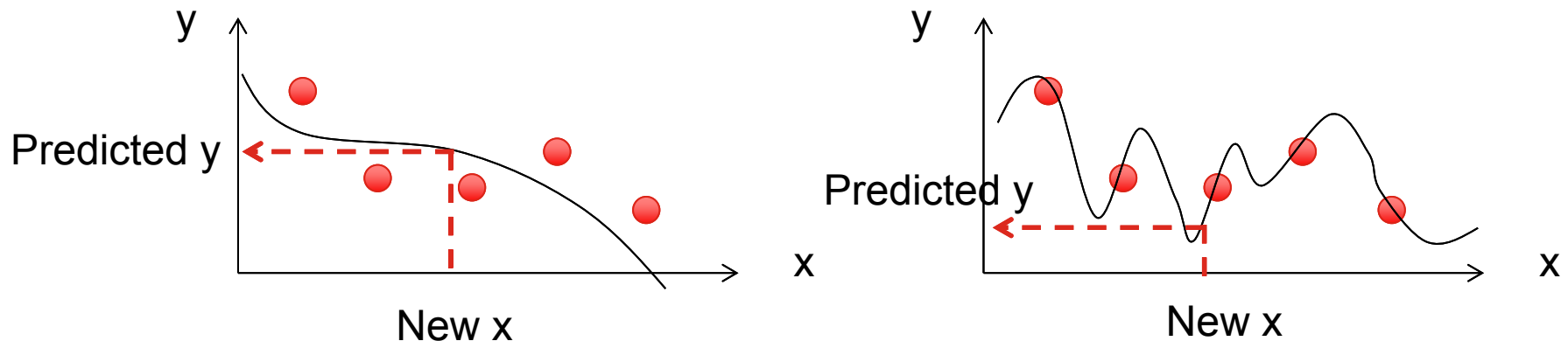


- The network could “fit” the data in the following ways:



# Generalization

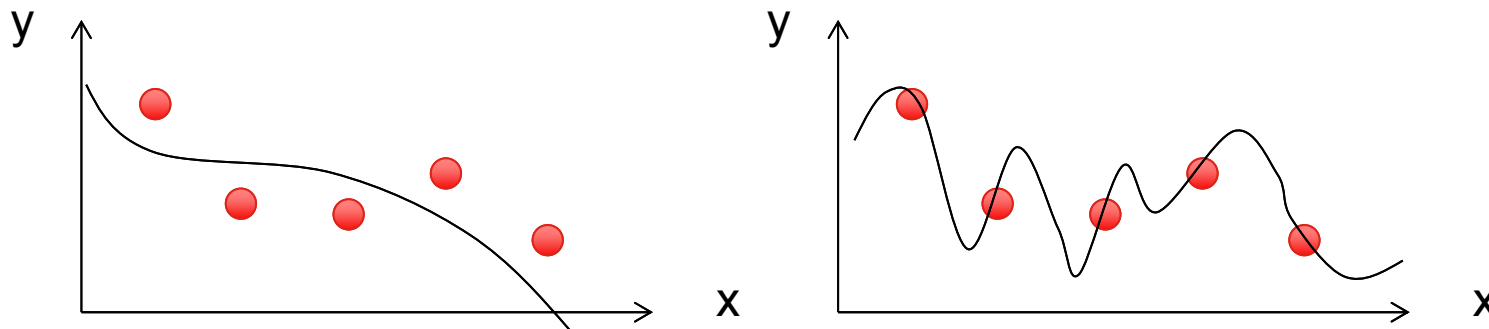
- Now we **test the network** with some new (unseen before) input data.



- It can be seen that the left network gives a **better prediction**, in face of new data.
  - The left network can **generalize better** than the right network.

# Generalization

- How do we **improve the generalization** of a network?
- Firstly, observe the two curve fitting again:



- On the left, the curve do not really pass through the training points. So the training error is not really that small, or in other words, the **training accuracy is high but not too high**.
- On the right, the curve passes through almost all the training points. So the training error is very small, or in other words, the **training accuracy is extremely high**.

# Generalization

- It is interesting to observe that:
  - Training accuracy high but not too high → Test accuracy good
  - Training accuracy extremely high → Test accuracy deteriorates!
- This is called “**overfitting**”: The network learns too well such that it even learns noise or random fluctuations.
- So the way to improve generalization is to make sure that the network doesn't learn **too** well.
  - **Limit** the number of hidden neurons
  - **Limit** the size of the weights
  - **Stop** the learning before it has time to overfit

# Contents

- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- Generalization
- **MATLAB Example**
- Deep Neural Networks

# Example 1: Regression

```
clear all;
close all;
clc

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% sampling 41 points in the range of [-1,1] %
% the data is -1, -0.95, -0.9, ..., 0.95, 1 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x=-1:0.05:1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% generating training data, the desired outputs %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

y=0.8*x.^3 + 0.3 * x.^2 -0.4 * x;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% specify the structure and learning algorithm for MLP %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

net=newff(minmax(x), [3,1], {'tansig', 'purelin'}, 'trainlm');
net.trainparam.show=2000; % epochs between display
net.trainparam.lr=0.01; % learning rate
net.trainparam.epochs=10000; % maximum epochs to train
net.trainparam.goal=1e-4; % performance goal, training will stop if this is reached
```

# Example 1: Regression

```
#####
% Train the MLP %
#####

[net,tr]=train(net,x,y);

#####
% Test the MLP, net_output is the output of the MLP, ytest is the desired output. %
#####

xtest=-0.97:0.1:0.93; % Data is -0.97, -0.87, ..., 0.83, 0.93 (never seen before)
ytest=0.8*xtest.^3 + 0.3 * xtest.^2 -0.4 * xtest;
net_output=sim(net,xtest);

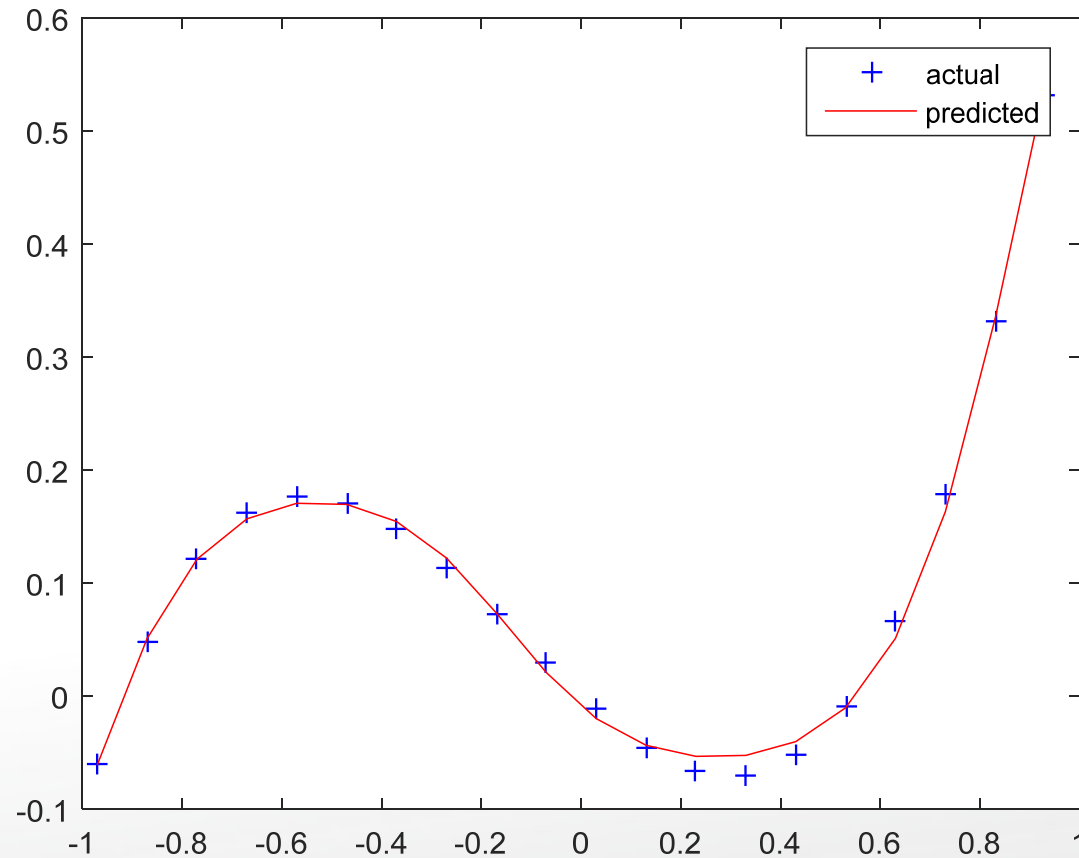
#####
% Plot out the test results %
#####

plot(xtest,ytest,'b+');
hold on;
plot(xtest,net_output,'r-');
hold off
legend('actual','predicted')
```



# Example 1: Regression

- Results:



# About the Code

Two values mean  
one hidden layer and  
one output layer

Levenberg-  
Marquardt  
method

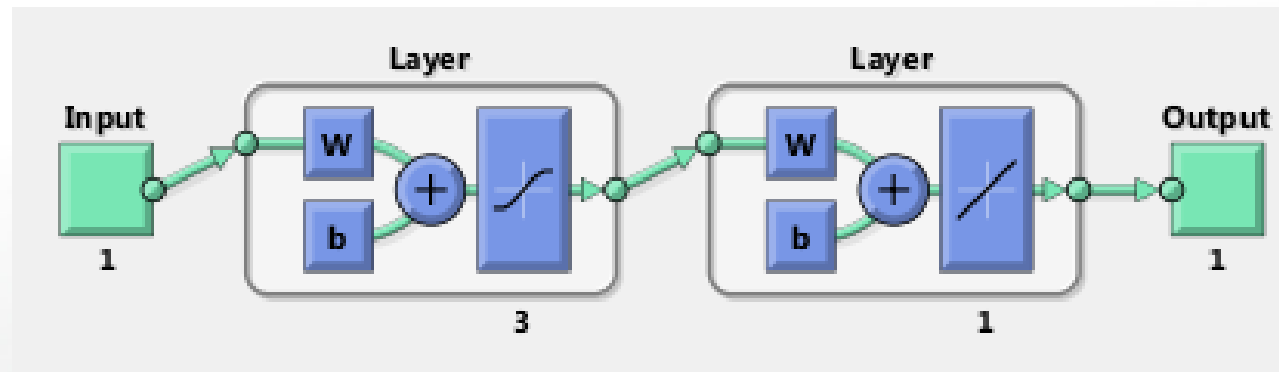
```
net=newff(minmax(x),[3,1],{'tansig','purelin'},'trainlm');
```

3 nodes  
in hidden  
layer

1 node  
in output  
layer

Hidden  
layer  
uses tanh

Output  
layer uses  
linear



# About the Code

You are welcome to try other parameters:

Three values mean two hidden layer and one output layer

Bayesian Regulation method

```
net=newff(minmax(x),[5,3,1],{'logsig','tansig','purelin'},'trainbr');
```

5 nodes in 1<sup>st</sup> hidden layer

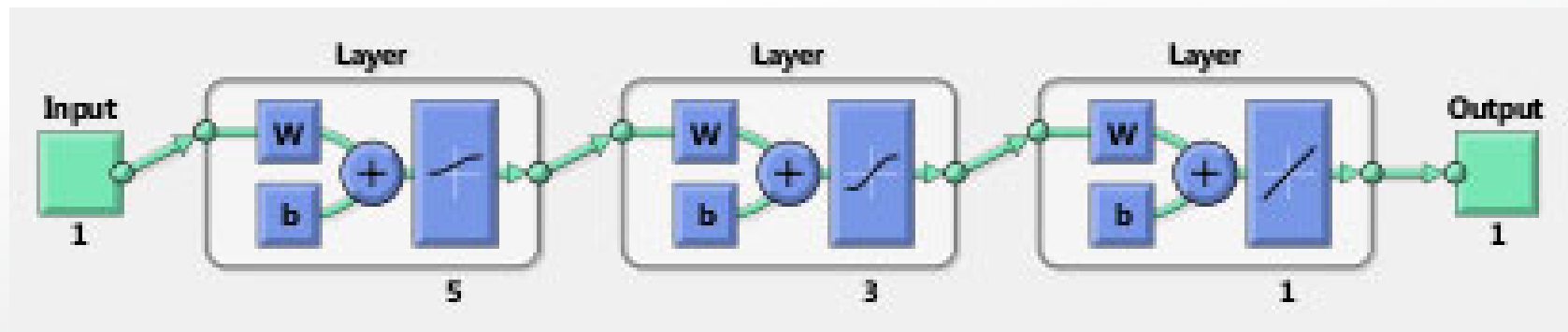
3 nodes in 2<sup>nd</sup> hidden layer

1 node in output layer

1<sup>st</sup> hidden layer uses logistic

2<sup>nd</sup> hidden layer uses tanh

Output layer uses linear



# Training Methods

- We used “trainlm” and “trainbr” in the previous two slides.
  - “Trainlm” is good for small number of weights.
    - Memory requirement is proportional to the square of the number of weights.
  - “Trainbr” produces a network which **generalizes well**.
- There are several other methods:
  - “Traincgf” is conjugate-gradient method, which is suitable for **large number of weights**.
    - Memory requirement is proportional to the number of weights.
    - If you run into memory problems when using “trainlm”, then you can try changing to “traincgf”.

# Example 2: Classification

- Sample datasets can be found in University of California at Irvine's Machine Learning Repository: <https://archive.ics.uci.edu/ml/index.php>
- E.g. Iris Dataset.

- Based on:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm



[https://en.wikipedia.org/wiki/Iris\\_setosa](https://en.wikipedia.org/wiki/Iris_setosa)

- Classify the iris into different types:

- Iris Setosa
- Iris Versicolour
- Iris Virginica



[https://en.wikipedia.org/wiki/Iris\\_virginica](https://en.wikipedia.org/wiki/Iris_virginica)

# Example 2: Classification

- Preparation:
  - The original data are shown on the left.
  - We replace the **three types** of iris using **three numbers** representing the “probability” of the classes, i.e.
    - [0.8, 0.2, 0.2] for Iris setosa
    - [0.2, 0.8, 0.2] for Iris versicolor
    - [0.2, 0.2, 0.8] for Iris virginica.

Avoid 1 and 0 because these numbers can only be reached asymptotically by logistic function. The weights are driven to be larger and larger.

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
⋮
```

```
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
⋮
```

```
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
6.3,2.9,5.6,1.8,Iris-virginica
⋮
```



```
5.1,3.5,1.4,0.2,0.8,0.2,0.2;
4.9,3.0,1.4,0.2,0.8,0.2,0.2;
4.7,3.2,1.3,0.2,0.8,0.2,0.2;
4.6,3.1,1.5,0.2,0.8,0.2,0.2;
⋮
```

```
7.0,3.2,4.7,1.4,0.2,0.8,0.2;
6.4,3.2,4.5,1.5,0.2,0.8,0.2;
6.9,3.1,4.9,1.5,0.2,0.8,0.2;
5.5,2.3,4.0,1.3,0.2,0.8,0.2;
⋮
```

```
6.3,3.3,6.0,2.5,0.2,0.2,0.8;
5.8,2.7,5.1,1.9,0.2,0.2,0.8;
7.1,3.0,5.9,2.1,0.2,0.2,0.8;
6.3,2.9,5.6,1.8,0.2,0.2,0.8;
⋮
```

# Example 2: Classification

- Randomise Order:
  - The original data was ordered based on the types of iris.
  - We **jumble up the data** to achieve a random order.

	1	2	3	4	5	6	7
1	5.1000	3.5000	1.4000	0.2000	0.8000	0.2000	0.2000
2	4.9000	3	1.4000	0.2000	0.8000	0.2000	0.2000
3	4.7000	3.2000	1.3000	0.2000	0.8000	0.2000	0.2000
4	4.6000	3.1000	1.5000	0.2000	0.8000	0.2000	0.2000
5	5	3.6000	1.4000	0.2000	0.8000	0.2000	0.2000



	1	2	3	4	5	6	7
1	5.4000	3	4.5000	1.5000	0.2000	0.8000	0.2000
2	7.2000	3	5.8000	1.6000	0.2000	0.2000	0.8000
3	7.7000	3	6.1000	2.3000	0.2000	0.2000	0.8000
4	5.2000	2.7000	3.9000	1.4000	0.2000	0.8000	0.2000
5	5.5000	2.3000	4	1.3000	0.2000	0.8000	0.2000

- It is better to use random order, based on experience.

# Example 2: Classification

- Finally, we choose the first 70% of the randomised data as **training set**,
- And the remaining 30% as **test set** (Not to be used during training!)
- The codes up to here are:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Import Data and Randomise Order %
% Choose 70% as training set      %
% And remaining 30% as test set   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IrisData;

[n,m] = size(IrisAttributesAndTypes);
i = randperm(n);
IrisDataJumbled = IrisAttributesAndTypes(i,:);

SeventyPercent = round(0.7*n,0);
IrisDataTrain = IrisDataJumbled(1:SeventyPercent,:);
IrisDataTest = IrisDataJumbled(SeventyPercent+1:n,:);

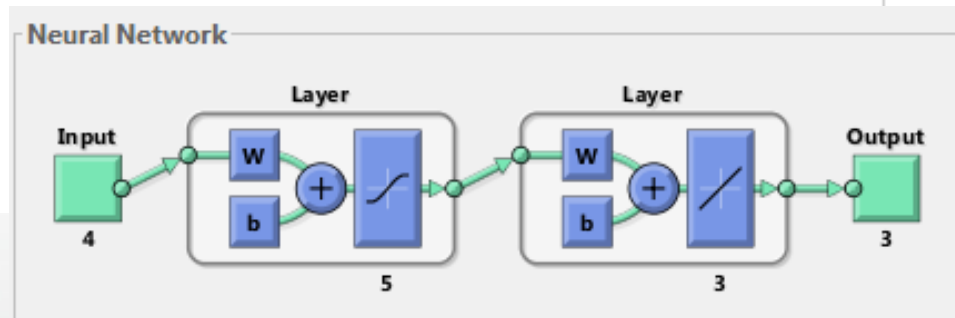
x = IrisDataTrain(:,1:4)';
y = IrisDataTrain(:,5:7)';
```



# Example 2: Classification

- The next part is to set up the network architecture and train the network:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% specify the structure and learning algorithm for MLP %  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
net=newff(minmax(x),[5,3],{'tansig','purelin'},'trainlm');  
net.trainparam.show=2000; % epochs between display  
net.trainparam.lr=0.01; % learning rate  
net.trainparam.epochs=10000; % maximum epochs to train  
net.trainparam.goal=1e-4; % performance goal, training will stop if this is reached  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Train the MLP %  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
[net,tr]=train(net,x,y);
```



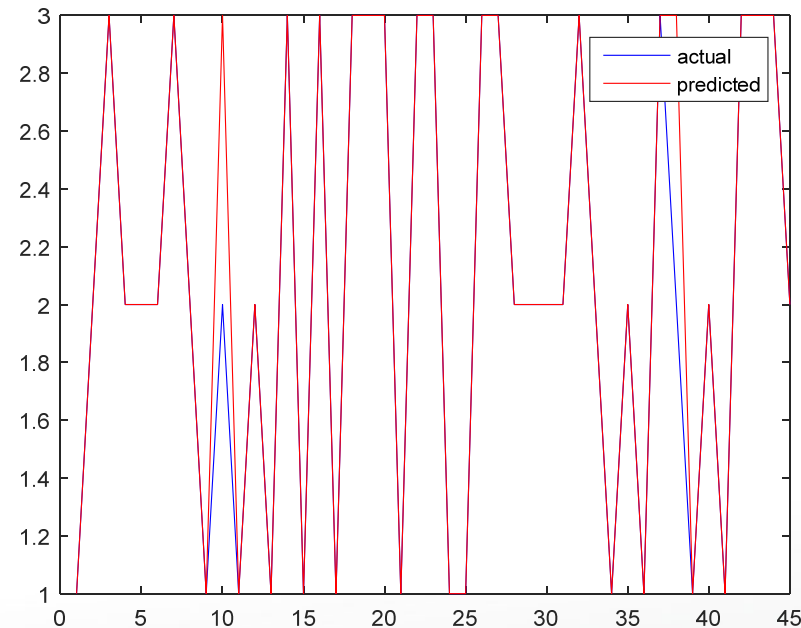
# Example 2: Classification

- Finally, we test the network using the test data.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Test the MLP, net_output is the output of the MLP, ytest is the desired output. %  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
xtest = IrisDataTest(:,1:4)';  
ytest = IrisDataTest(:,5:7)';  
net_output=sim(net,xtest);  
  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
% Plot out the test results %  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
  
[p,q] = size(ytest);  
for j = 1:q  
    [val_Actual(j),idx_Actual(j)] = max(ytest(:,j)); % Switch back from 3 numbers to specific class  
    [val_Predicted(j),idx_Predicted(j)] = max(net_output(:,j));  
end  
  
plot(idx_Actual,'b-');  
hold on;  
plot(idx_Predicted,'r-');  
hold off  
legend('actual','predicted')
```

## Example 2: Classification

- The test results are **mostly correct**, with only two misclassifications.



actual

1	2	3	4	5	6	7	8	9	10	11
1	2	3	2	2	2	3	2	1	2	1

## Predicted

1	2	3	4	5	6	7	8	9	10	11
1	2	3	2	2	2	3	2	1	3	1

# Exercise

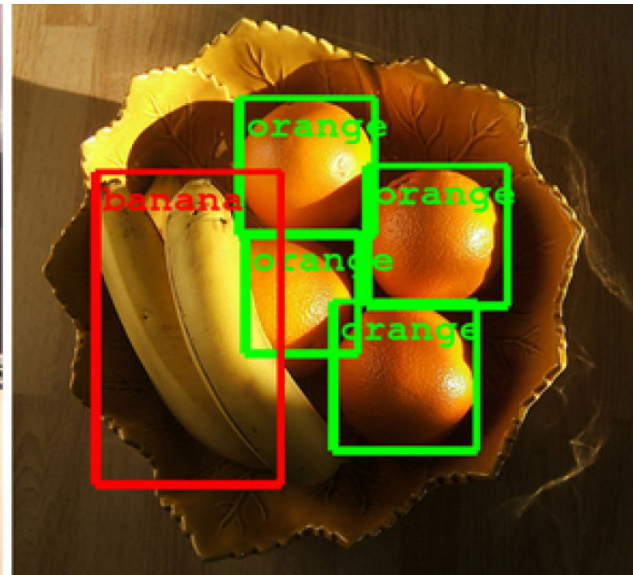
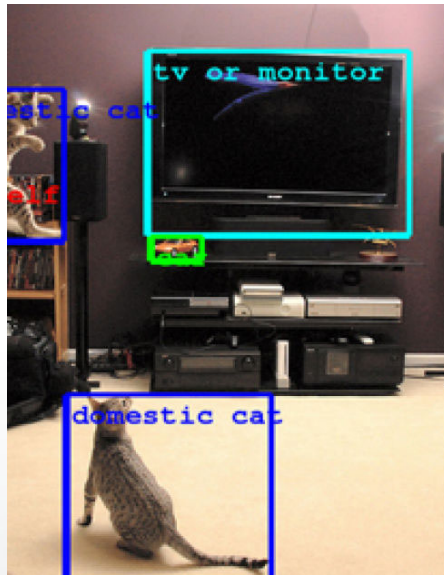
- Download other datasets from University of California at Irvine's Machine Learning Repository: <https://archive.ics.uci.edu/ml/index.php>
- Use MATLAB's neural network toolbox to perform data classification on the data.

# Contents

- Introduction
- The Biological Neuron
- The Artificial Neuron
- Network Architectures
- Perceptron
- Multilayer Perceptrons
- Generalization
- MATLAB Example
- **Deep Neural Networks**

# Deep Neural Networks

- In the iris example just now, we had to **extract some features manually**, and put them into numerical numbers (lengths and widths in cm), before feeding into the neural networks.
- On the other hand, you might have seen some computer images, where **items are recognised automatically** without manual intervention.



<https://www.engadget.com/2014/09/08/google-details-object-recognition-tech/>

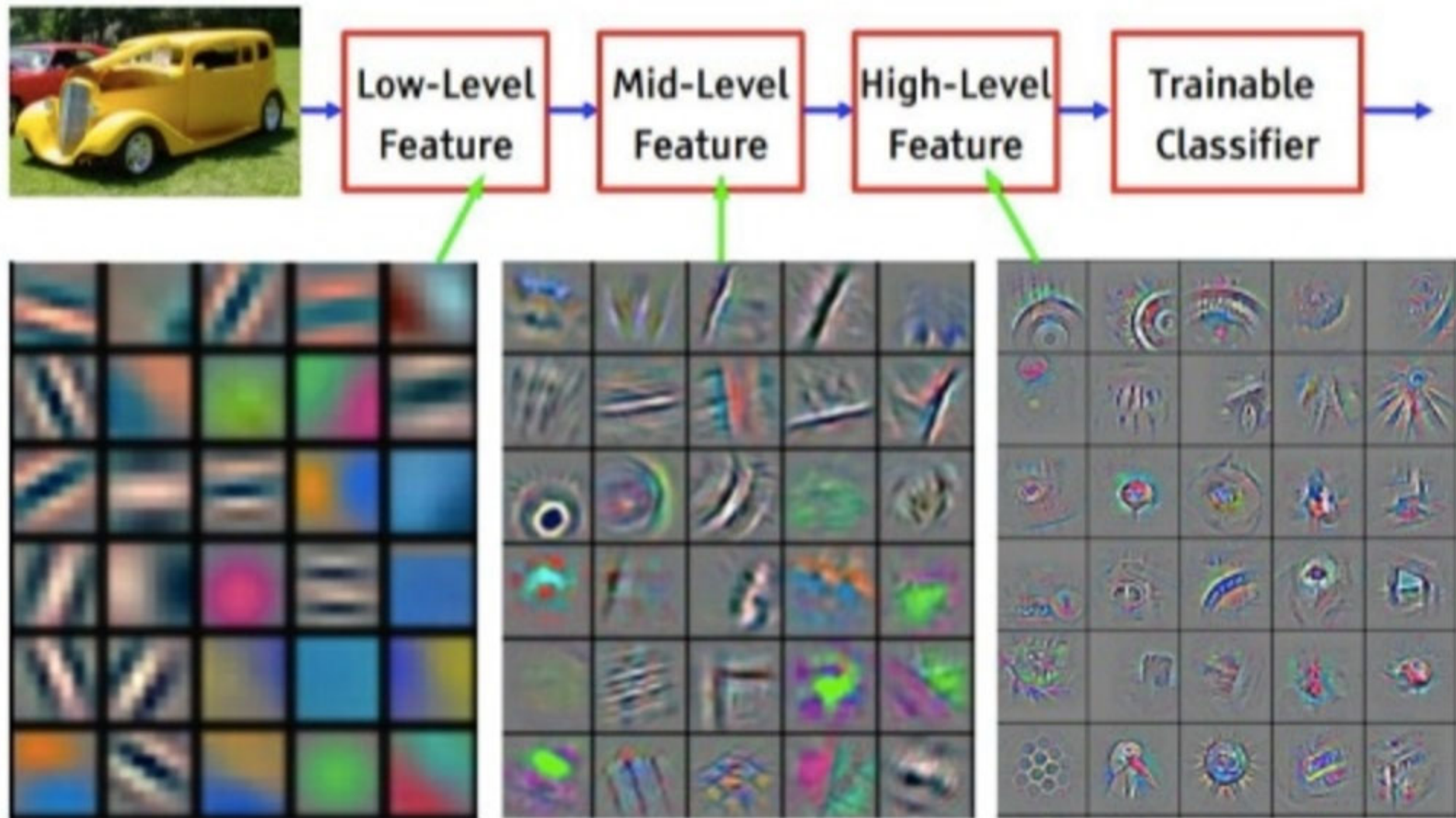
- How do they do it?

# Deep Neural Networks

- The state-of-the-art method to do this is **Deep Neural Networks**.
- It gives very good accuracy and has won many competitions / challenges in object recognition.
- Why is it called **deep**?
  - Shallow networks, e.g. MLP's usually have **only a few layers**. Adding more layers to MLP's does not increase its capability to approximate functions.
  - Deep neural networks, on the other hand, has **many layers (can be several tens or hundreds)** where each layer has its **own specific task**, e.g. in convolutional neural network:
    - First layer finds the edges within an image
    - Second layer combines the edges to get corners etc.
    - Third layer combines the corners (etc.) to get nose, eyes etc.
    - Fourth layer finally recognises that it is a face.



# Example

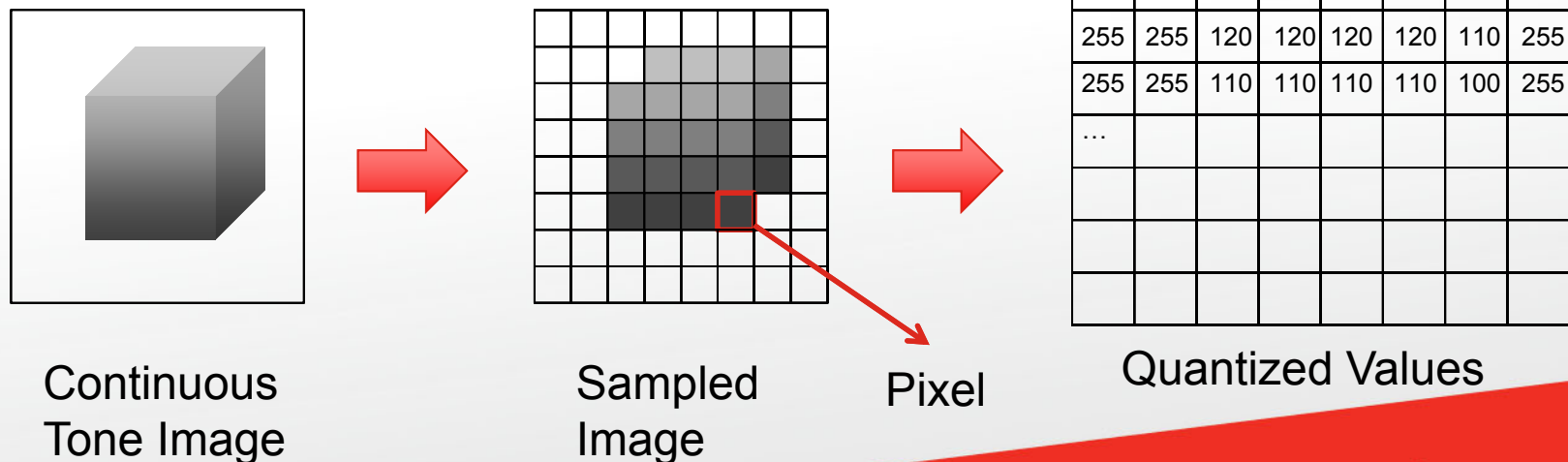


<https://towardsdatascience.com/identifying-traffic-signs-with-deep-learning-5151eece09cb>



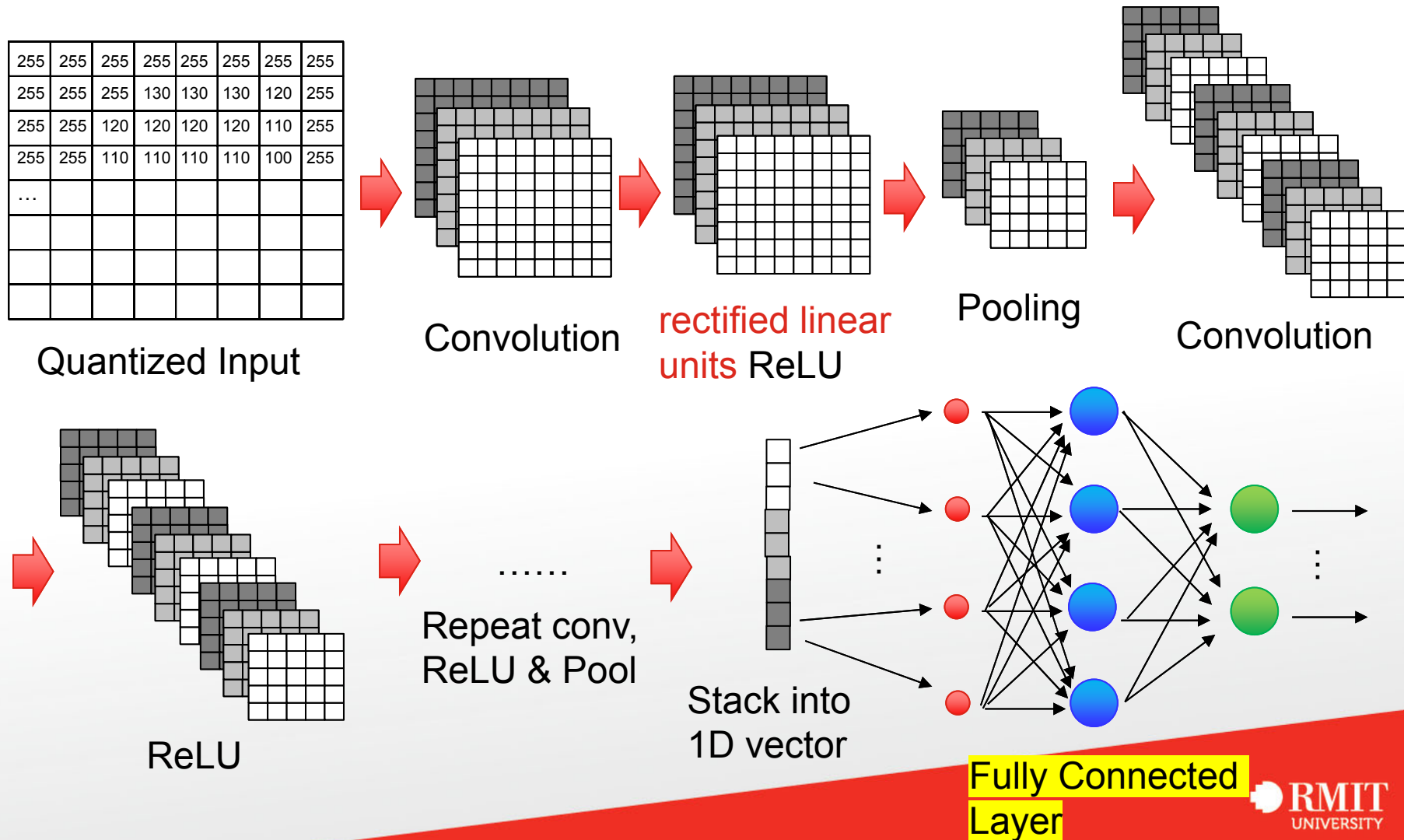
# Convolutional Neural Networks

- Here, we will learn about **Convolutional Neural Networks (CNN)**, which is a type of deep network, commonly applied to analyse visual images.
- Let's talk about the **input** first:
  - A real world image varies continuously in shades and colours.
  - When we use a **digital camera** to take the picture, the continuous image is divided into **individual points of brightness**.
  - This is followed by describing each point of brightness using a **digital data value** → 255 for pure white, 0 for pure black.



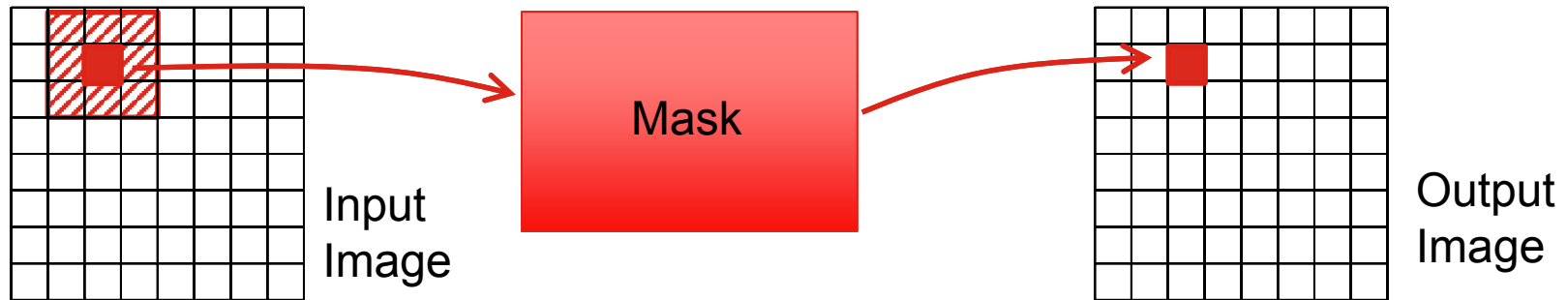
# Convolutional Neural Networks

- The quantized input is now passed through a series of layers:



# Convolution Layer

- An area (e.g. 3 x 3, or 5 x 5 etc.) of pixels are “**convolved**” with a mask (**weights**), and the output would be a single number at the center.



- The **convolution process** is very simple:

$I(x-1, y-1)$	$I(x, y-1)$	$I(x+1, y-1)$
$I(x-1, y)$	$I(x, y)$	$I(x+1, y)$
$I(x-1, y+1)$	$I(x, y+1)$	$I(x+1, y+1)$

Intensity of pixels surrounding the center pixel (x, y)

**X**  
(element wise)

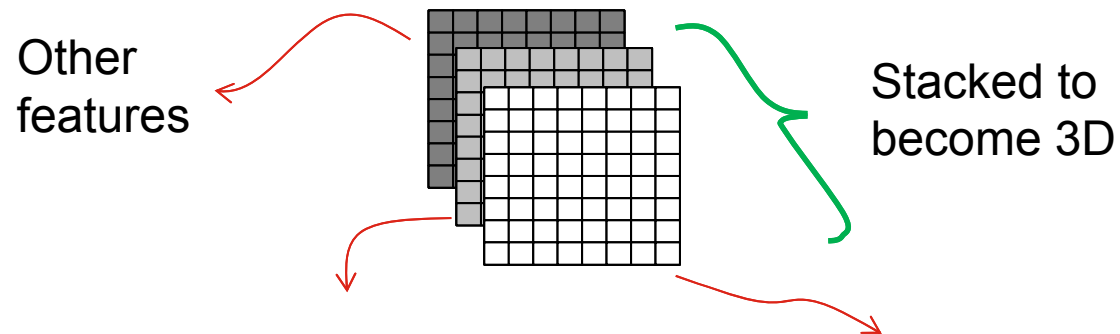
$W_{11}$	$W_{12}$	$W_{13}$
$W_{21}$	$W_{22}$	$W_{23}$
$W_{31}$	$W_{32}$	$W_{33}$

Weights for each pixel  
(a.k.a. convolution mask)

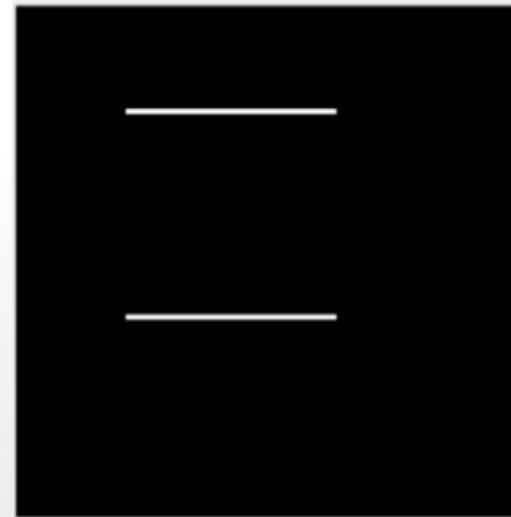
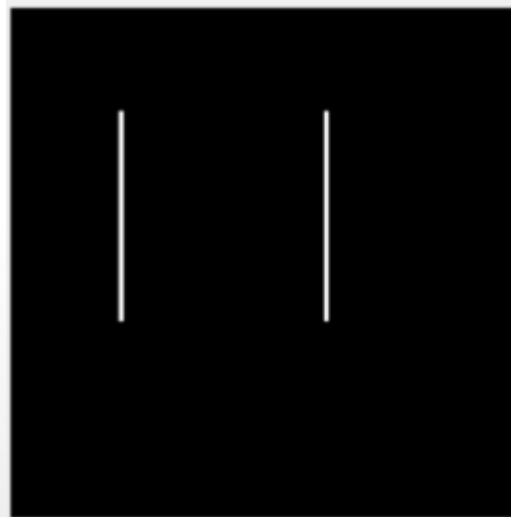
→ Then **sum up** all the elements.

# Convolution Layer

- By using suitable masks (weights), different **features** can be extracted from an image, e.g. vertical edges, horizontal edges etc.



$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



$$M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

# ReLU Layer

- ReLU is the abbreviation for **rectified linear units**.
- This function is simple – It just **truncates any negative number** (output from convolution layer) to zero. Positive numbers are not affected.
  - This allows for faster and more effective training.
  - E.g.

3	2	1
-2	1	4
0	-1	3

Output from convolution layer



3	2	1
0	1	4
0	0	3

Output from ReLU Layer

# Pooling Layer

- An image might have **millions of pixels**.
- It will be **hard for a neural network** to learn from all of them, some of which carry very little information anyway (e.g. low intensity).
- The pooling layer is used to perform “**down-sampling**”, by choosing the **maximum value (most information)** in a certain region. E.g.

255	255	255	255	255	255	255	255
255	255	255	130	130	130	120	255
255	255	120	120	120	120	110	255
255	255	110	110	110	110	100	255
...							

Output from  
ReLU Layer

Max-  
Pooling

255	255	

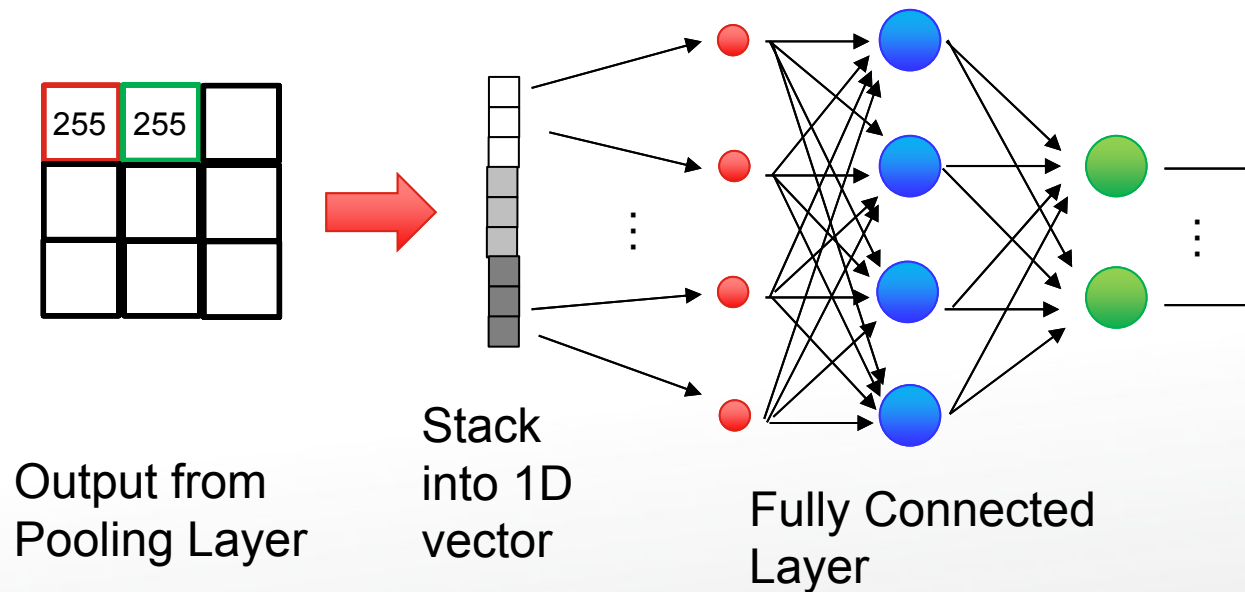
Output from  
Pooling Layer

# Pooling Layer

- Note that the pooling layer **does not need to be included every time** in between ReLU and Conv, e.g.
  - Conv → ReLU → **Pool** → Conv → ReLU → **Pool** → Conv → ReLU → **Pool** → Conv → ReLU → **Pool** → 1D Vector
  - Conv → ReLU → Conv → ReLU → Conv → ReLU → **Pool** → Conv → ReLU → **Pool** → 1D Vector

# 1D Vector & Fully Connected Layer

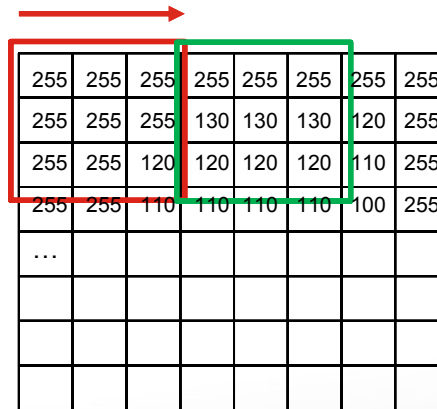
- Finally, we **stack the pixel values** into a 1D vector, and feed this through a **fully connected layer** (e.g. MLP) to perform classification.





# Terminologies

- Here are some terminologies which you need to know:
- **Stride**: It means **how many pixels** you step across each time.
  - In this example, the stride is 3 pixels.



255	255	255	255	255	255	255	255
255	255	255	130	130	130	120	255
255	255	120	120	120	120	110	255
255	255	110	110	110	110	100	255
...							

- **Padding**: It means **adding extra pixels** outside of the image.
  - Zero padding means the extra pixels have zero values.
  - This **preserves the size** of the original image. Otherwise each time after convolution, the image size will decrease.

# MATLAB Example

- Download the following files from Canvas:
  - helperCIFAR10Data.m
  - Week11\_CNN\_Classification.m
- And run the second m-file.



# Reference

- Lecture Notes on “Neural Networks”, Tan Kay Chen, National University of Singapore. (2007)
- Lecture Notes on “Neural Networks”, Xiang Cheng, National University of Singapore. (2010)

Thank you,  
Questions

