

# **Multilayer Neural Network**

Dr. Emad Natsheh

# Neural Network

There is different types of NNs among them

## ❖ **Supervised learning**

- ❑ Feedforward neural network: Data mining (They are used to make predictions from a large dataset)
  - Convolutional neural network
- ❑ Feedback neural network (memory) forecasting (time series)
  - Recurrent neural network

## ❖ **Unsupervised learning**

- ❖ Self Organizing Map

# Neural Networks

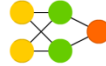
©2016 Fjodor van Veen - asimovinstitute.org

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool

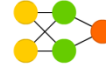
Perceptron (P)



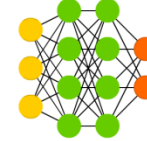
Feed Forward (FF)



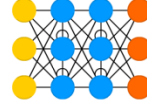
Radial Basis Network (RBF)



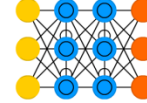
Deep Feed Forward (DFF)



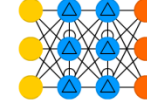
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)



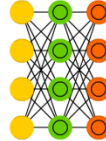
Gated Recurrent Unit (GRU)



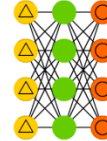
Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)



Sparse AE (SAE)



Markov Chain (MC)



Hopfield Network (HN)



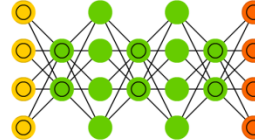
Boltzmann Machine (BM)



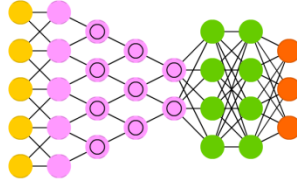
Restricted BM (RBM)



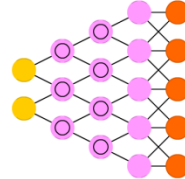
Deep Belief Network (DBN)



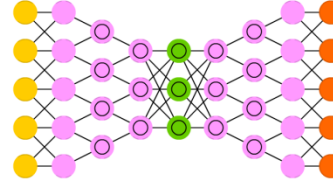
Deep Convolutional Network (DCN)



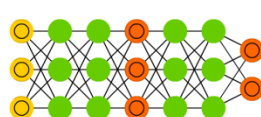
Deconvolutional Network (DN)



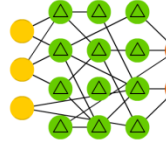
Deep Convolutional Inverse Graphics Network (DCIGN)



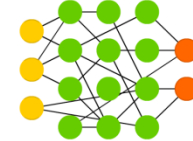
Generative Adversarial Network (GAN)



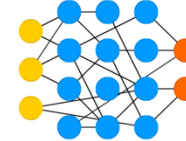
Liquid State Machine (LSM)



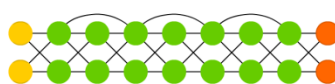
Extreme Learning Machine (ELM)



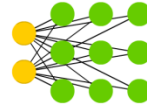
Echo State Network (ESN)



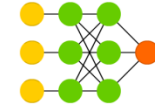
Deep Residual Network (DRN)



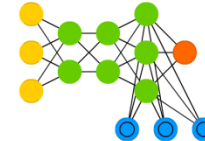
Kohonen Network (KN)



Support Vector Machine (SVM)

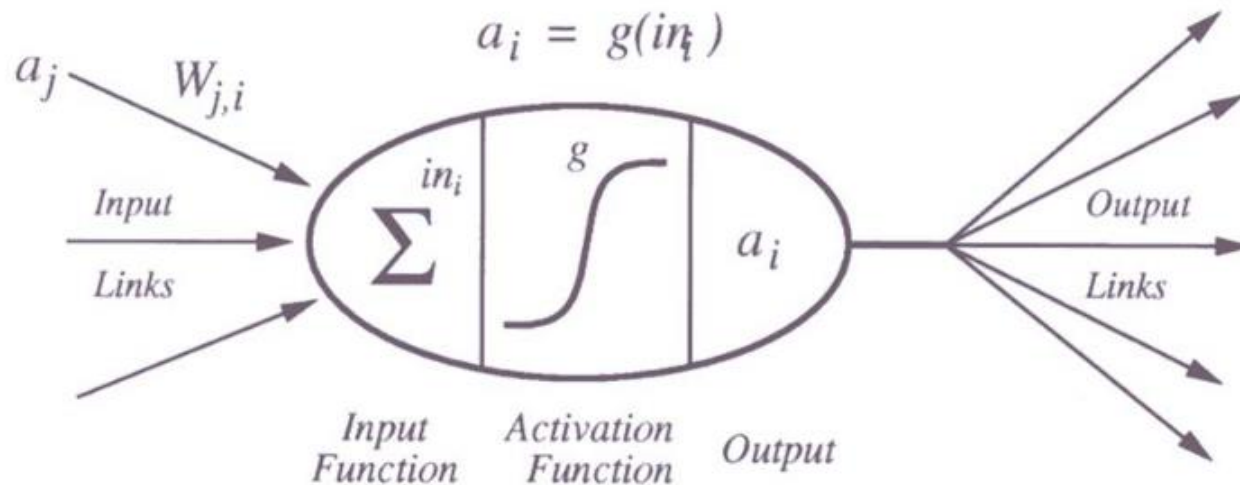


Neural Turing Machine (NTM)



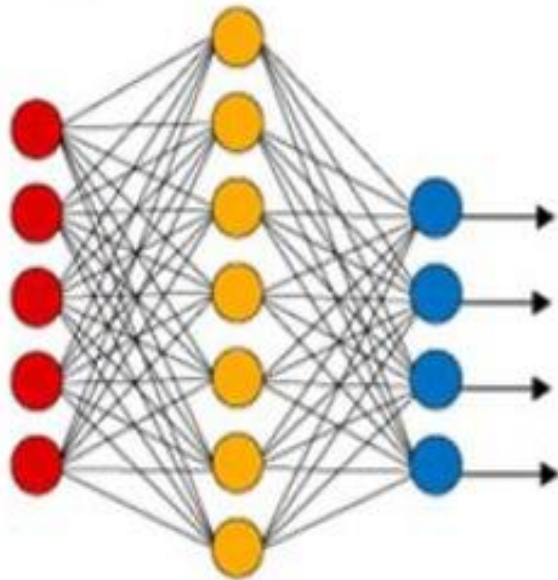
# Feedforward neural network

- Neurons are connected by weight links
- Each neuron can receive different inputs but produce only one output.
- Input signals are propagated from the input to the output (layer-by-layer feed-forward manner)

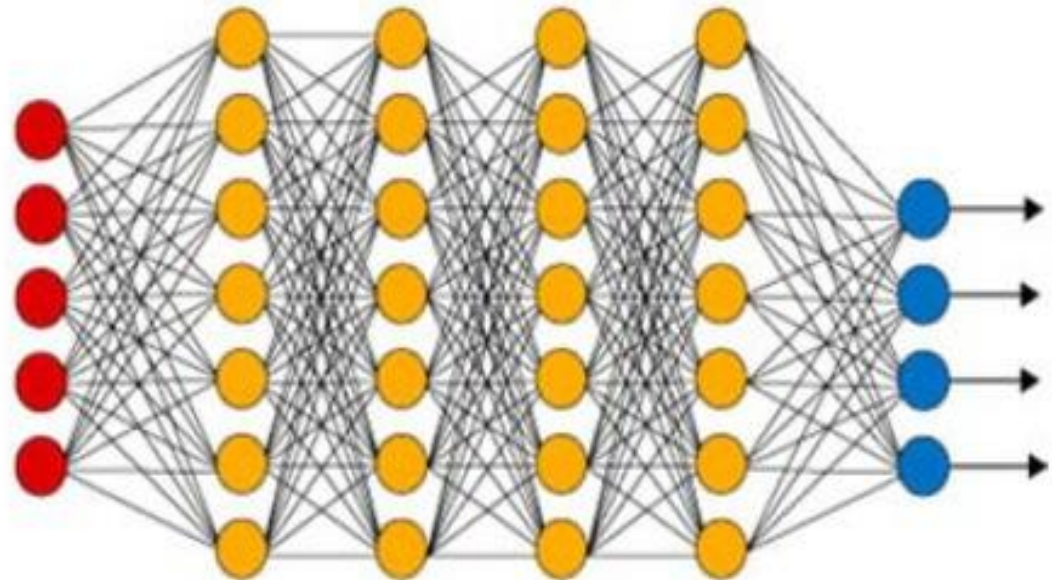


# Structure of Feedforward neural network

Simple Neural Network



Deep Learning Neural Network



● Input Layer

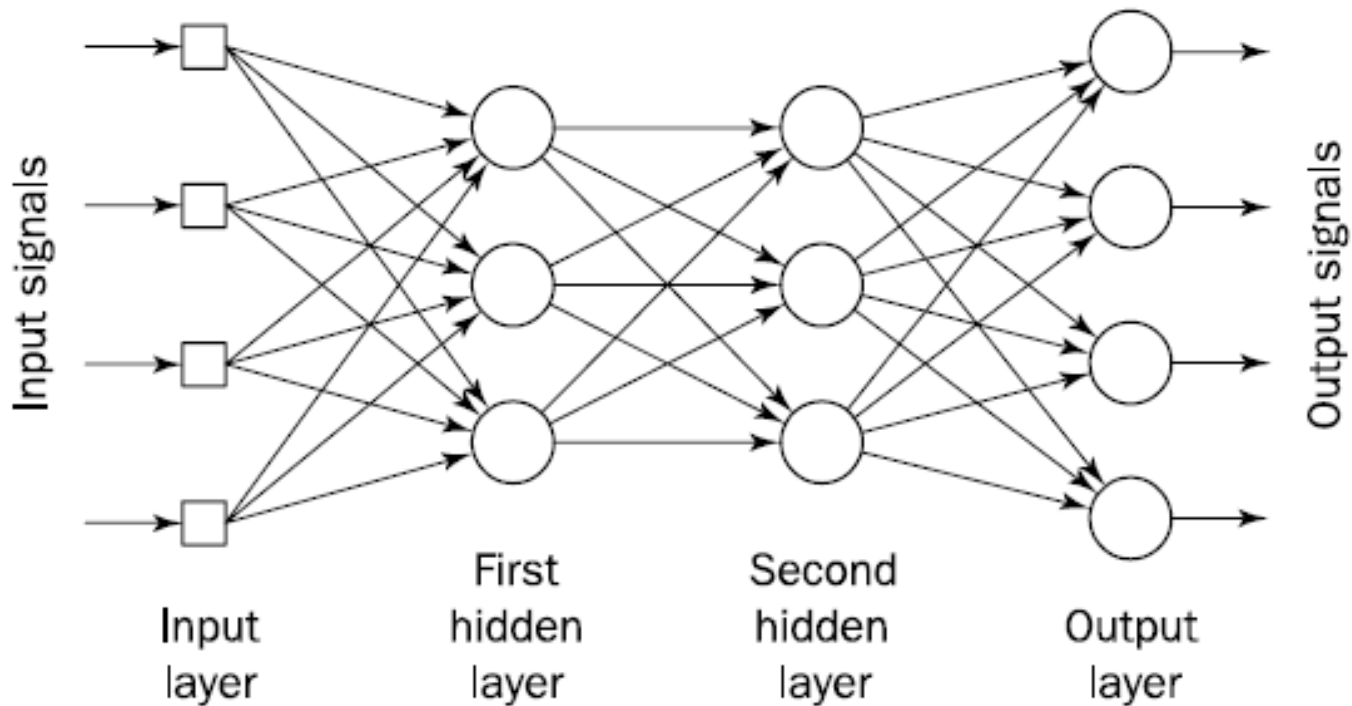
● Hidden Layer

● Output Layer

# Feedforward network Layers

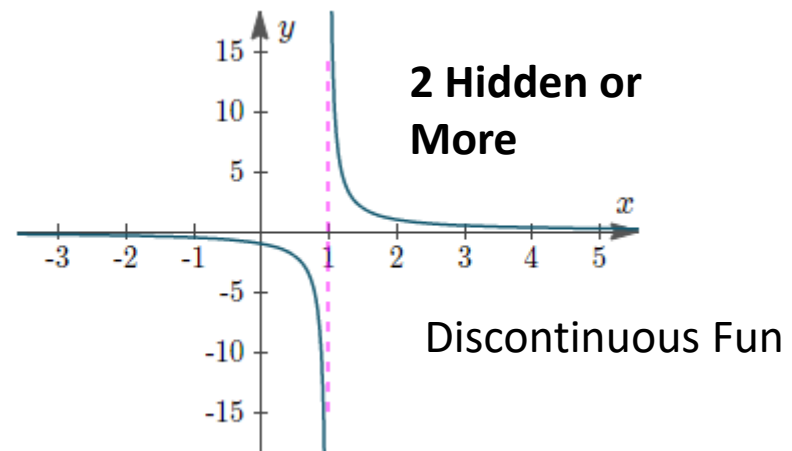
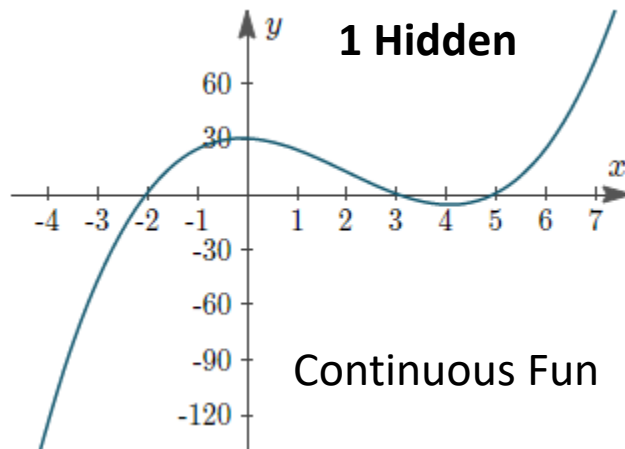
- **An input layer** (source neurons) : The input layer accepts input signals from the outside world and then send these signals to all neurons in the hidden layer
- **One or more hidden layers** (computational neurons): detect the features hidden in the input patterns.
- **An output layer** (computational neurons): The output layer accepts output signals from the hidden layer and generate the output pattern of the entire network.

# Feedforward network Layers



# Feedforward network Layers

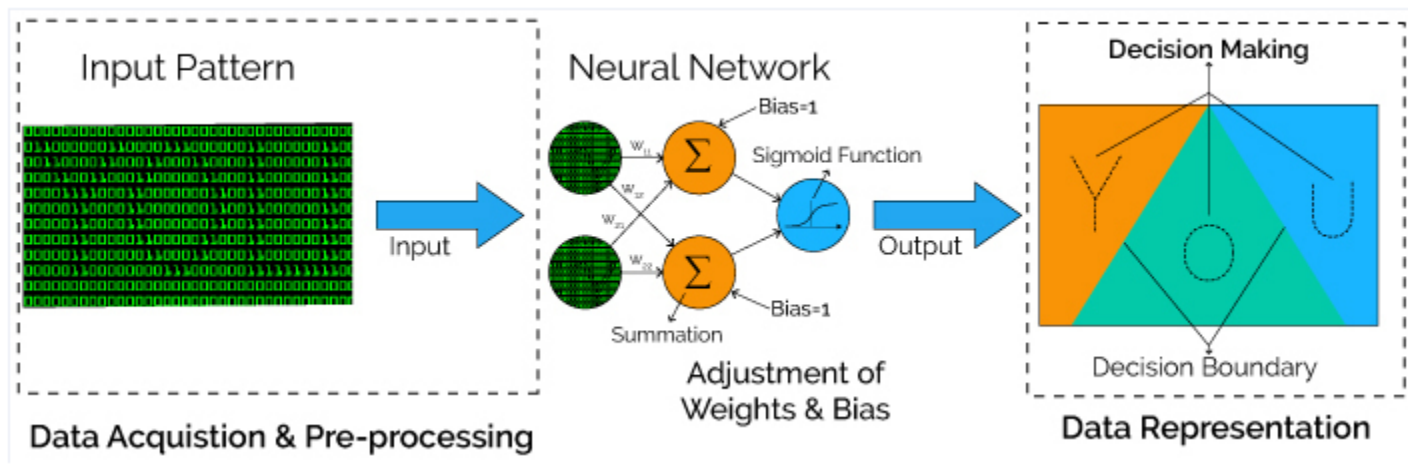
- Each layer in a FFNN has its own specific function.
- With one hidden layer, we can represent any continuous function of the input signals, and with two hidden layers even discontinuous functions can be represented.





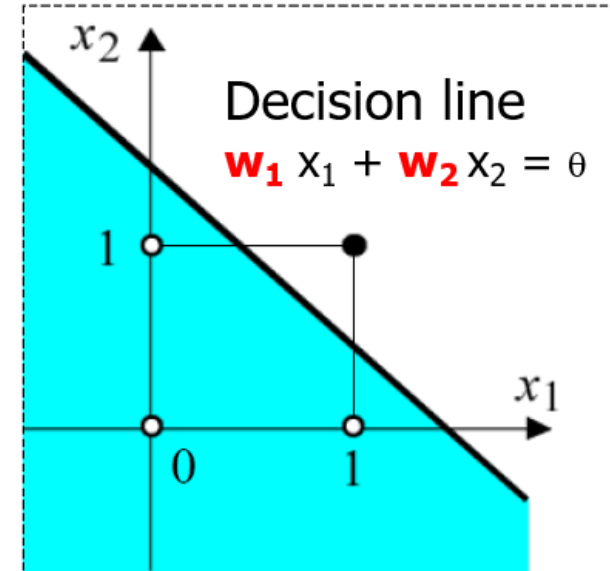
# How to Build a Feedforward network

- 1) **Collect data** and then separate it into **training and testing sets** (some time validation set also)
- 2) **Define the network structure**
  - ❖ how many neurons are to be used (hidden ??)
  - ❖ **What activation function to use**
- 3) chose the **learning algorithm**.
- 4) And finally we **train** the neural network

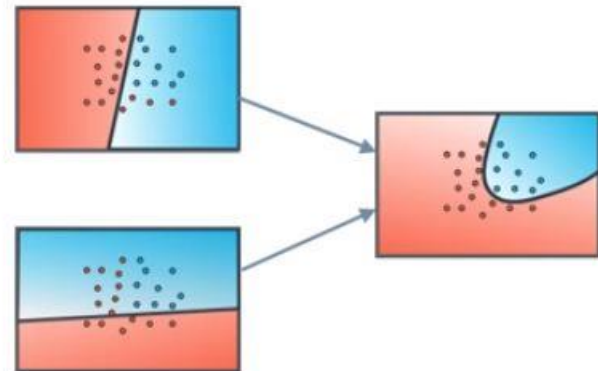
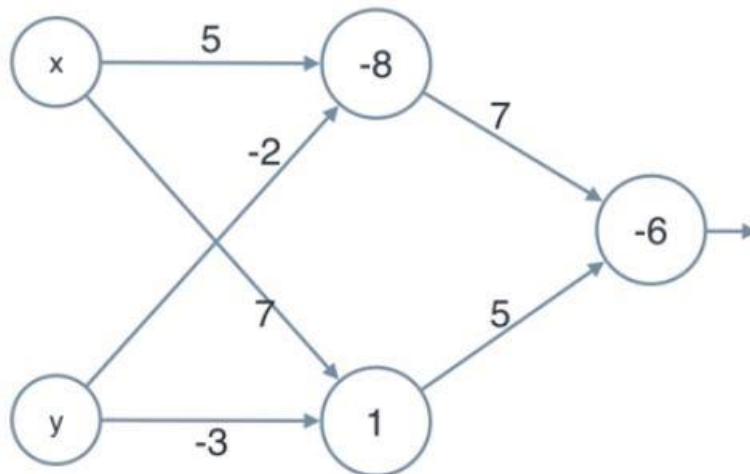


# How many neurons

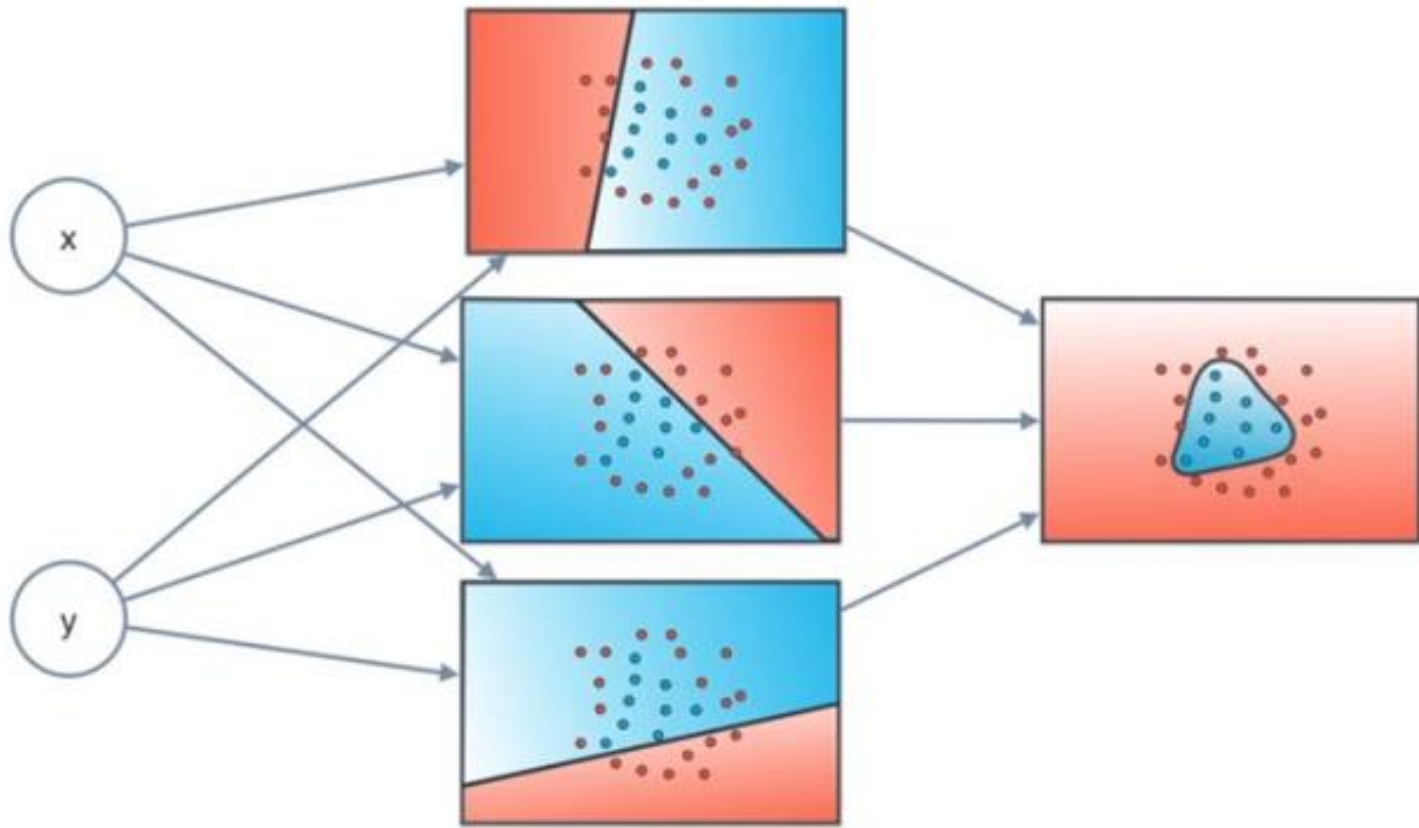
Perceptron



Neural Network

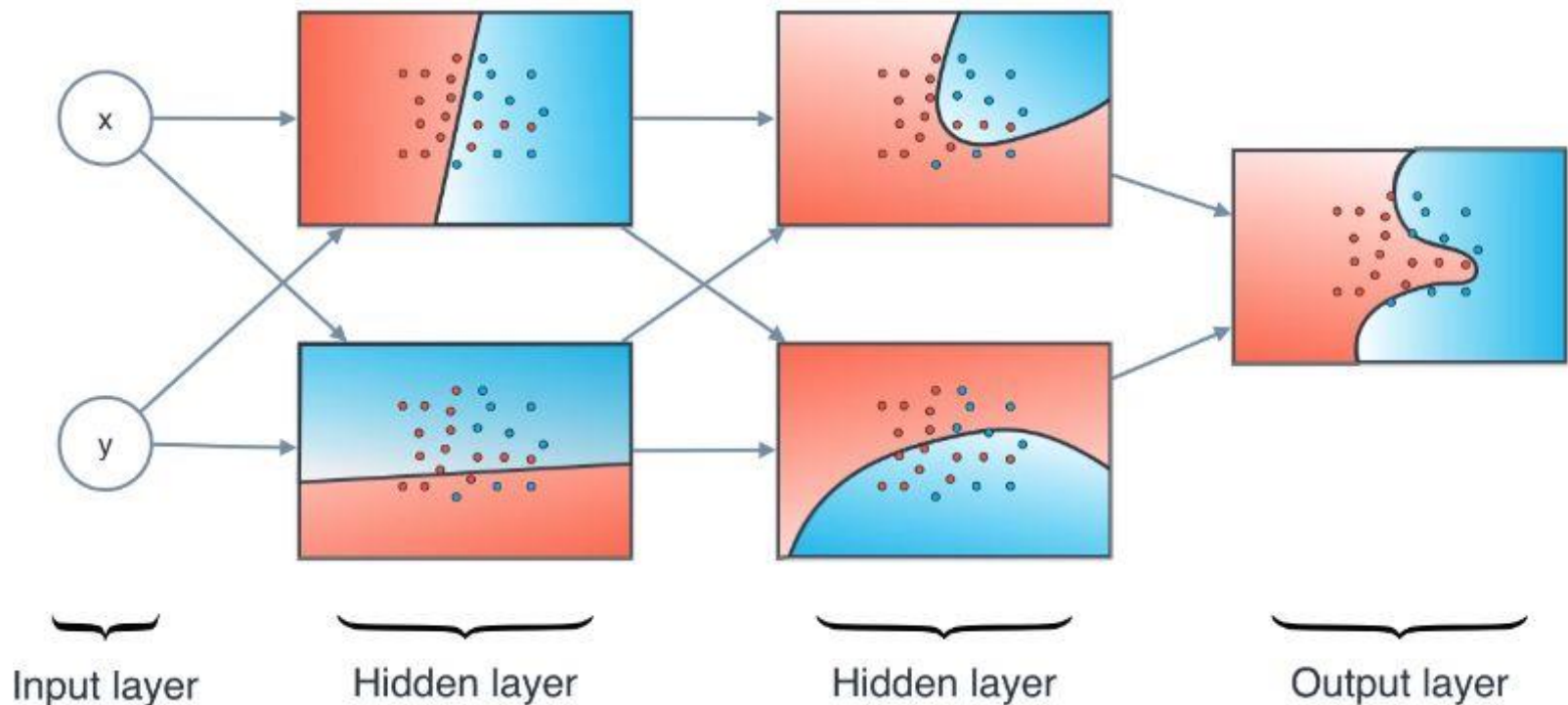


# How many neurons



# How many neurons

## Deep Neural Network

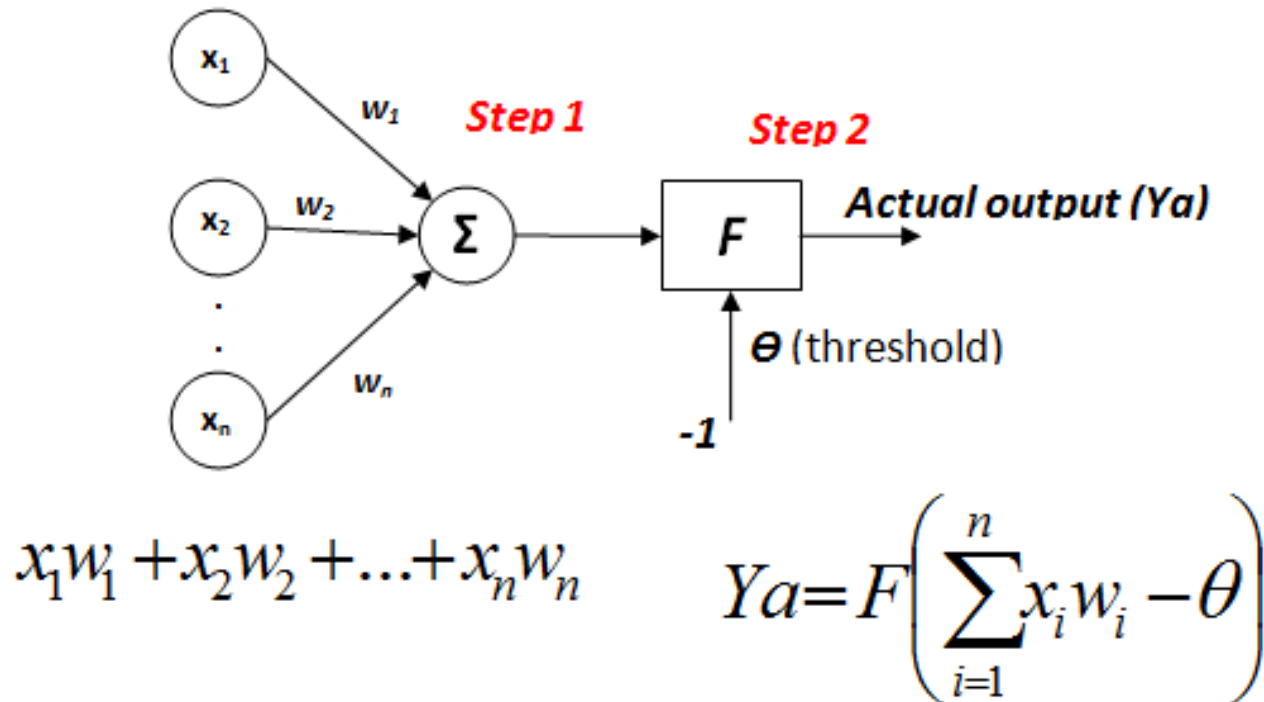


# Examples

- [A Neural Network Playground - TensorFlow](#)

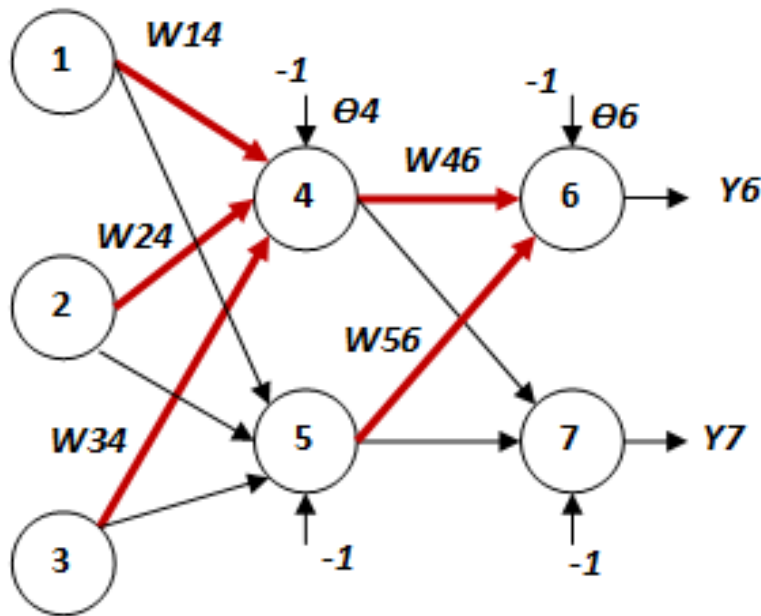
- ❖ Why is a middle layer in a multilayer network called a hidden' layer?
  - A hidden layer 'hides' its desired output.

# How to find the actual output of the network



Apply that for each node in the Hidden and Output layers

# How to find the actual output of the network



Hidden Layer

$$Y_4 = f_H((x_1 w_{14} + x_2 w_{24} + x_3 w_{34}) - \theta_4)$$

$$Y_5 = f_H((x_1 w_{15} + x_2 w_{25} + x_3 w_{35}) - \theta_5)$$

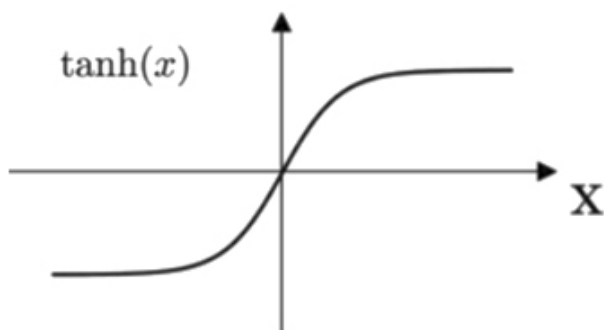
Output Layer (Actual output of the network)

$$Y_6 = f_O((Y_4 w_{46} + Y_5 w_{56}) - \theta_6)$$

$$Y_7 = f_O((Y_4 w_{47} + Y_5 w_{57}) - \theta_7)$$

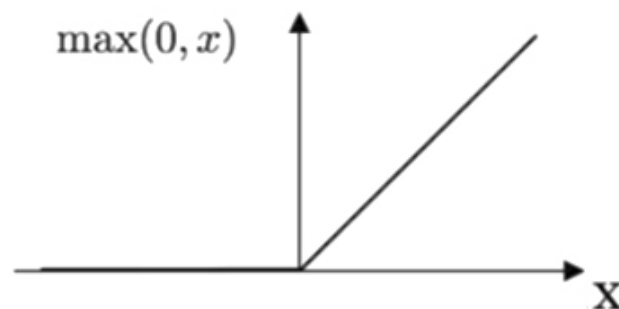
# Activation Functions (1)

$$\text{Tanh} = \frac{2}{1 + e^{-2X}} - 1$$

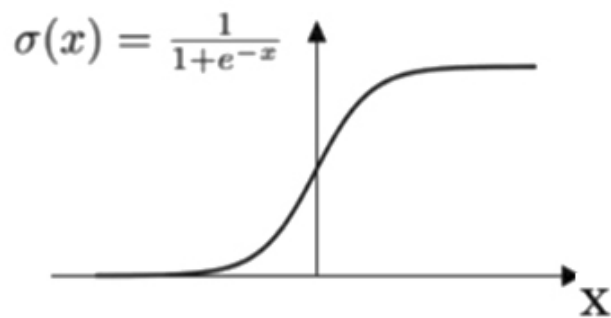


**ReLU**

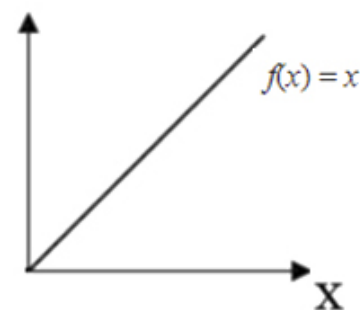
$$\text{ReLU} = \begin{cases} 0 & X < 0 \\ X & X \geq 0 \end{cases}$$



**Sigmoid**



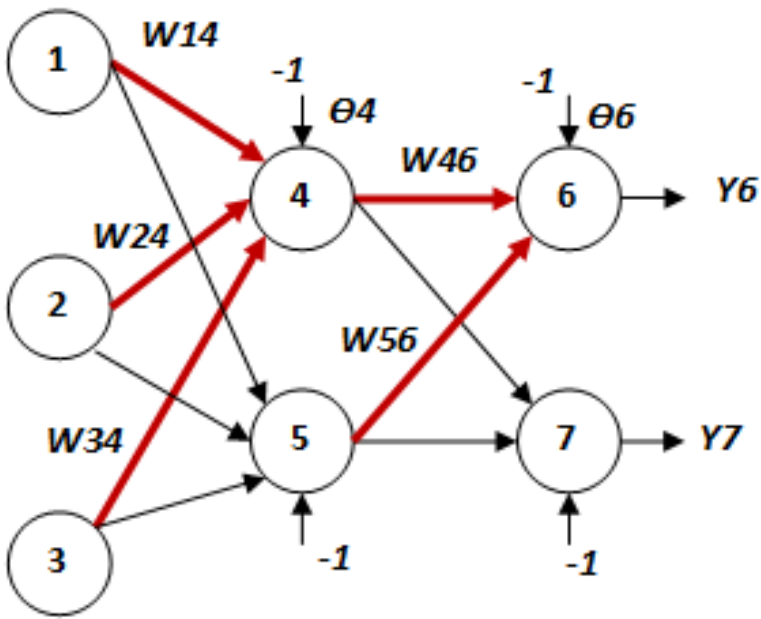
**Linear**





# Exercises

W14	W15	W24	W25	W34	W35	W46	W47	W56	W57	$\theta_4$	$\theta_5$	$\theta_6$	$\theta_7$
-0.5	1.1	-1.4	0.6	-0.4	0.2	0.1	-1.2	1.5	-1.3	0.7	0.6	-0.3	0.2



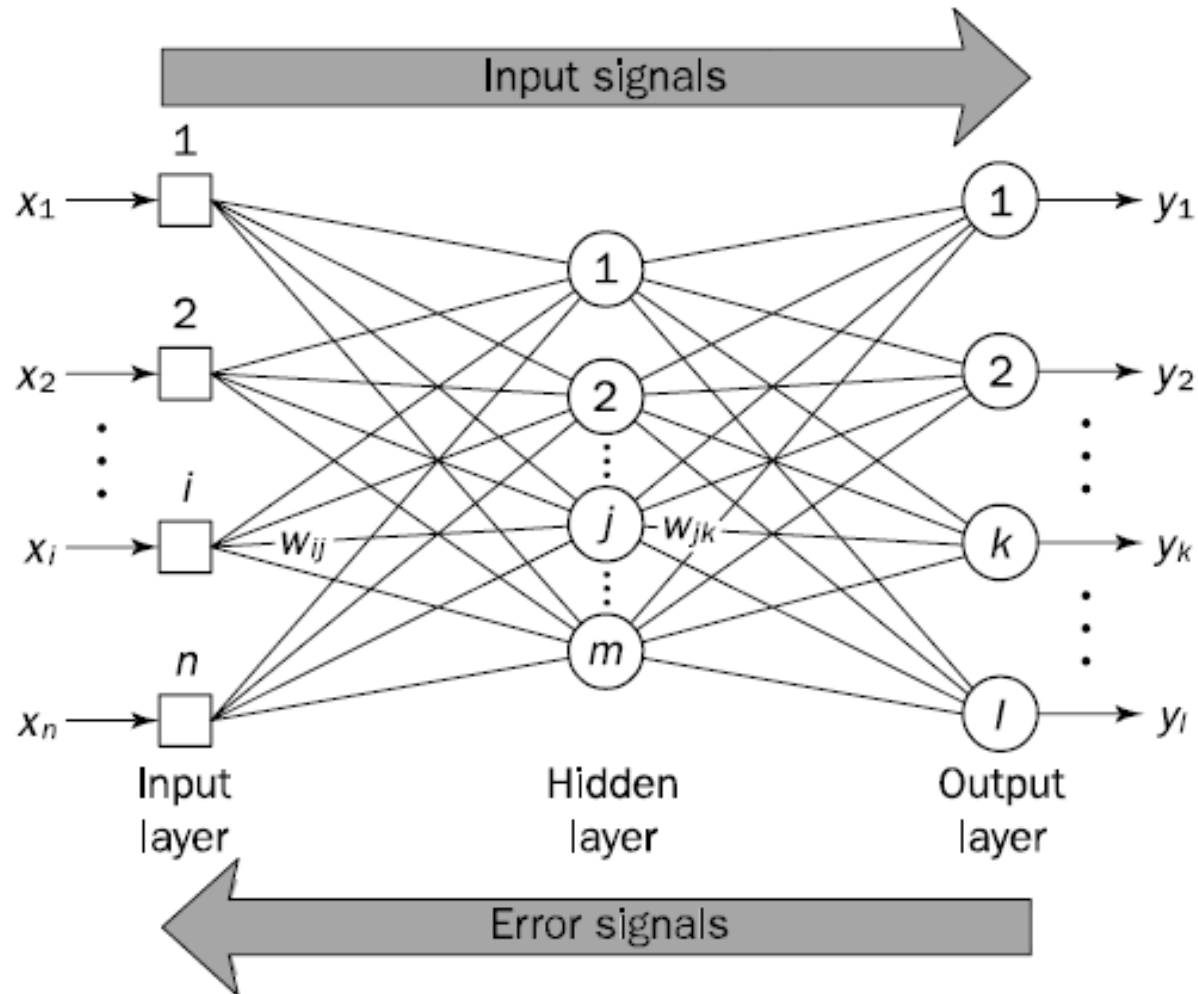
Assume activation function for hidden layer is ReLU and output layer is linear

If  $X_1$  is=1,  $X_2$ =0.5,  $X_3$ =0. What is the actual output for  $Y_7$ ?

# How Neural Networks Learn?

- 1) First, a training input pattern is presented to the network input layer.
- 2) Then the network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- 3) If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer.
  - ❖ The weights are modified as the error is propagated.

# How does the Network learn



# Network Learning

## Step 1: *Initialisation*

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range

$$\left(-\frac{2.4}{F_i}, +\frac{2.4}{F_i}\right),$$

where  $F_i$  is the total number of inputs of neuron  $i$  in the network. The weight initialisation is done on a neuron-by-neuron basis.

## Step 2: *Activation*

Activate the back-propagation neural network by applying inputs  $x_1(p), x_2(p), \dots, x_n(p)$  and desired outputs  $y_{d,1}(p), y_{d,2}(p), \dots, y_{d,n}(p)$ .

(a) Calculate the actual outputs of the neurons in the hidden layer:

$$y_j(p) = \text{sigmoid} \left[ \sum_{i=1}^n x_i(p) \times w_{ij}(p) - \theta_j \right],$$

where  $n$  is the number of inputs of neuron  $j$  in the hidden layer, and *sigmoid* is the sigmoid activation function.

# Network Learning

(b) Calculate the actual outputs of the neurons in the output layer:

$$y_k(p) = \text{sigmoid} \left[ \sum_{j=1}^m x_{jk}(p) \times w_{jk}(p) - \theta_k \right],$$

where  $m$  is the number of inputs of neuron  $k$  in the output layer.

## Step 3: *Weight training*

Update the weights in the back-propagation network propagating backward the errors associated with output neurons.

(a) Calculate the error gradient for the neurons in the output layer:

**Derivative  
Sigmoid**

$$\delta_k(p) = y_k(p) \times [1 - y_k(p)] \times e_k(p)$$

where

$$e_k(p) = y_{d,k}(p) - y_k(p)$$

Calculate the weight corrections:

$$\Delta w_{jk}(p) = \alpha \times y_j(p) \times \delta_k(p)$$

Update the weights at the output neurons:

$$w_{jk}(p+1) = w_{jk}(p) + \Delta w_{jk}(p)$$

$$\delta_k(p) = \frac{\partial y_k(p)}{\partial X_k(p)} \times e_k(p),$$

# Network Learning

(b) Calculate the error gradient for the neurons in the hidden layer:

$$\delta_j(p) = y_j(p) \times [1 - y_j(p)] \times \sum_{k=1}^l \delta_k(p) \times w_{jk}(p)$$

Calculate the weight corrections:

$$\Delta w_{ij}(p) = \alpha \times x_i(p) \times \delta_j(p)$$

Update the weights at the hidden neurons:

$$w_{ij}(p+1) = w_{ij}(p) + \Delta w_{ij}(p)$$

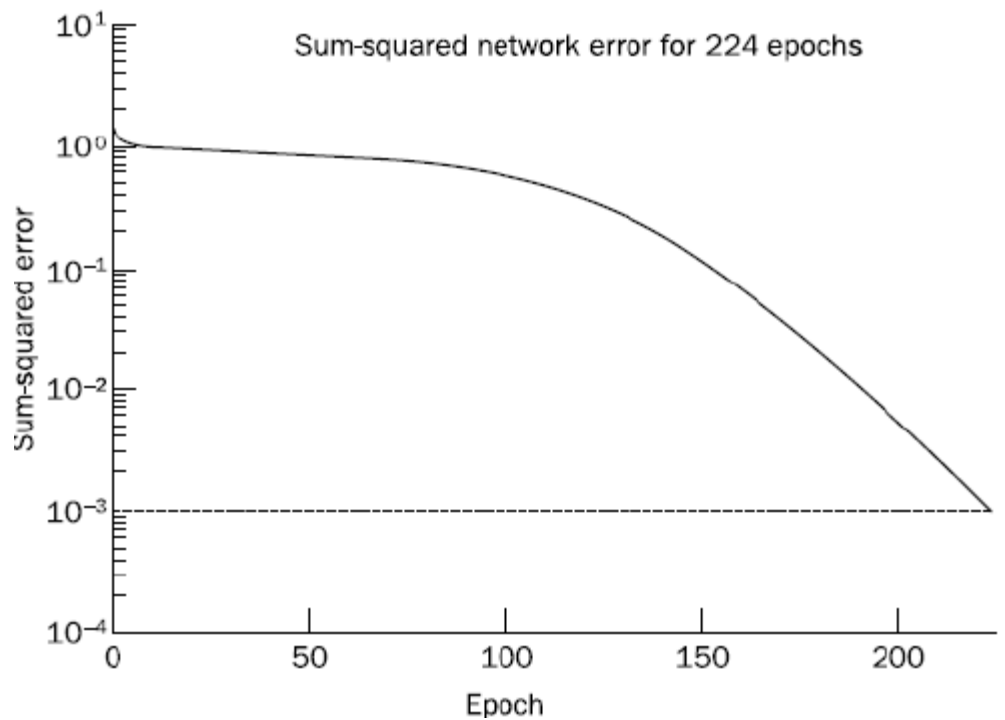
## Step 4: *Iteration*

Increase iteration  $p$  by one, go back to Step 2 and repeat the process until the selected error criterion is satisfied.



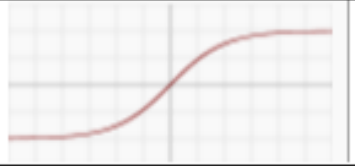

# SSE and MSE

$$SSE = \sum_{i=1}^n (Yd_i - Ya_i)^2$$

$$MSE = \frac{1}{n} \sum_{i=1}^n (Yd_i - Ya_i)^2$$



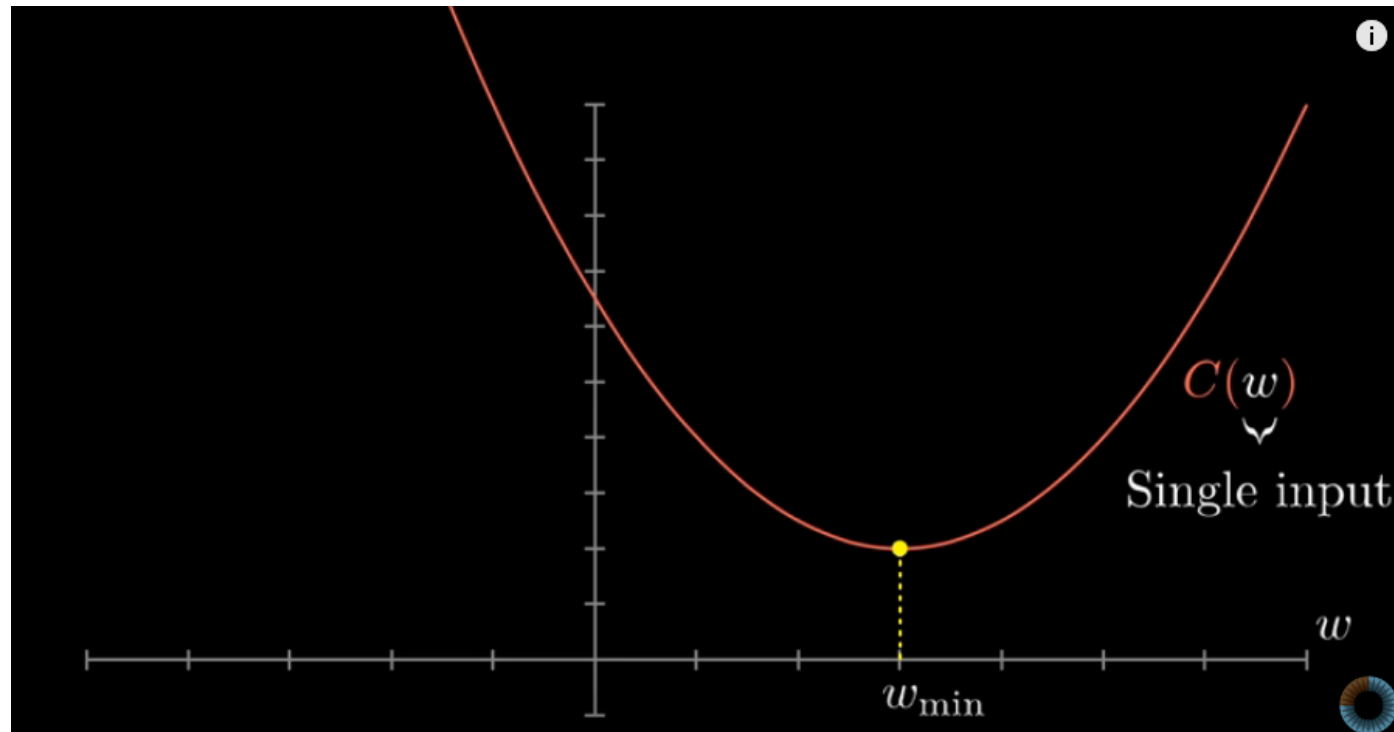
# Function Derivative

Plot	$F(x)$	$F'(x)$
	$f(x) = X$	$f'(x) = 1$
	$f(x) = \frac{1}{1 + e^{-X}}$	$f'(x) = f(x)(1 - f(x))$
	$f(x) = \frac{2}{1 + e^{-2X}} - 1$	$f'(x) = 1 - f(x)^2$
	$f(x) = \begin{cases} 0 & X < 0 \\ X & X \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & X < 0 \\ 1 & X \geq 0 \end{cases}$

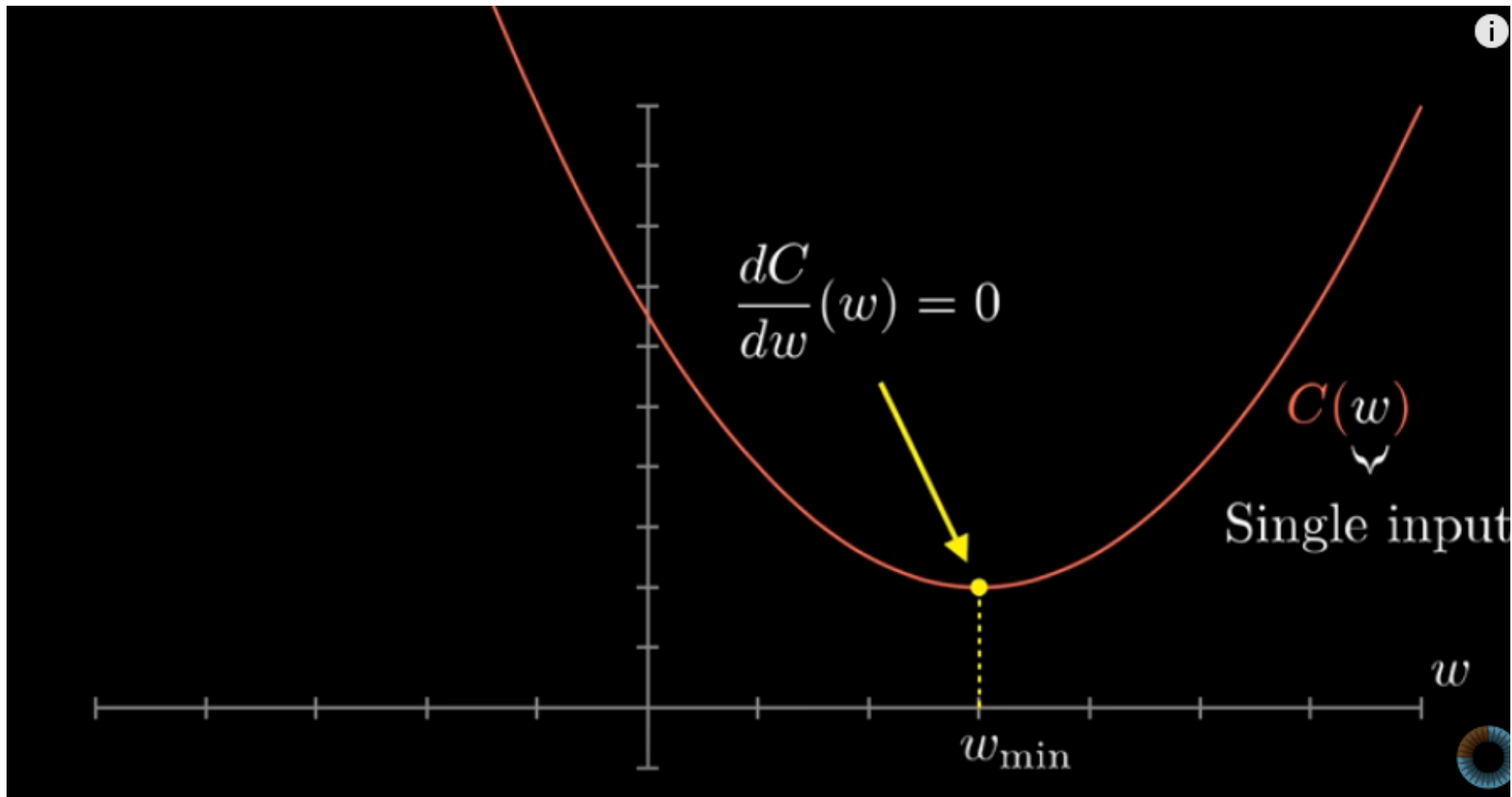


# Gradient descent

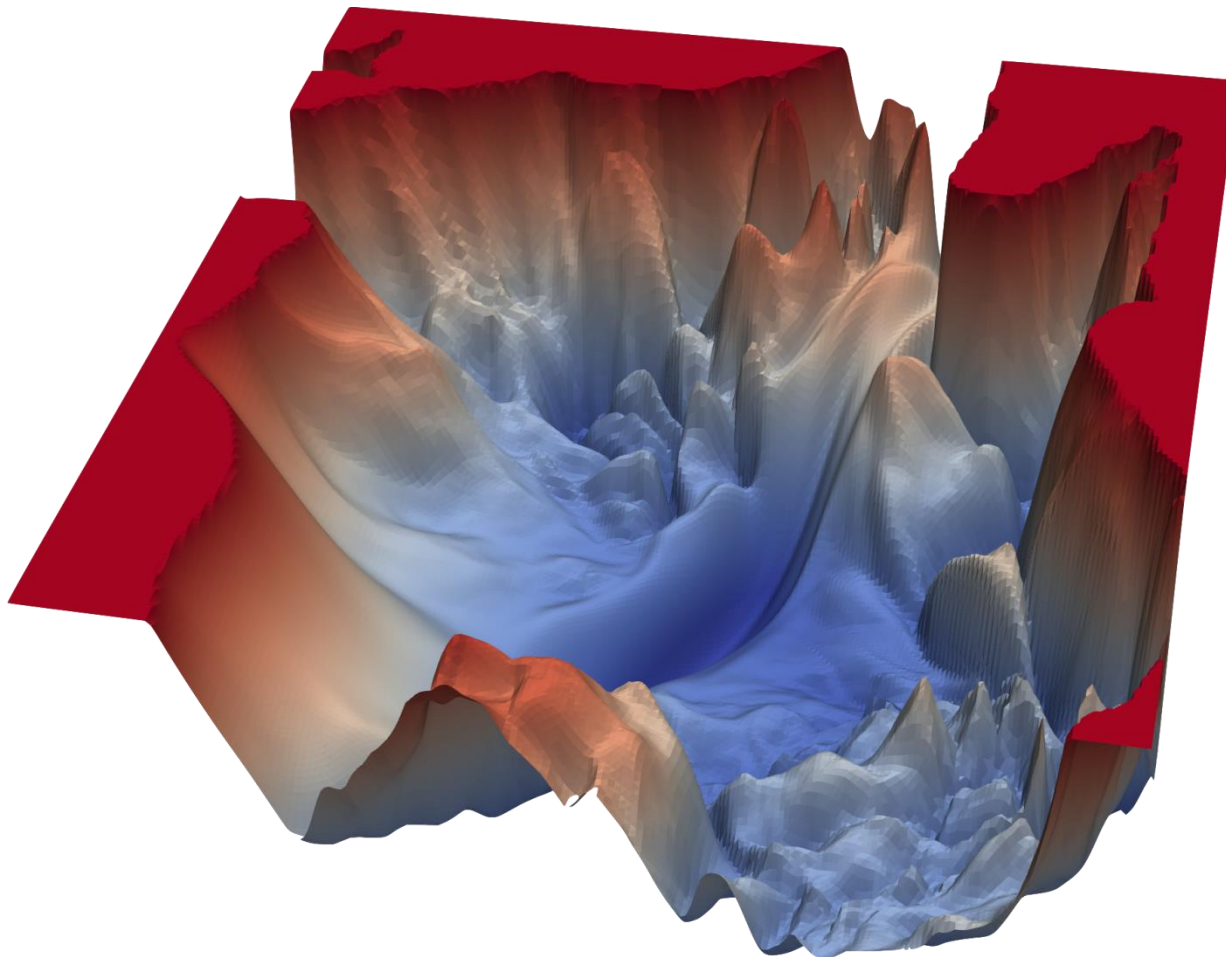
- **Gradient Descent** is the idea of using partial derivatives of function iteratively to get to its local minimum.



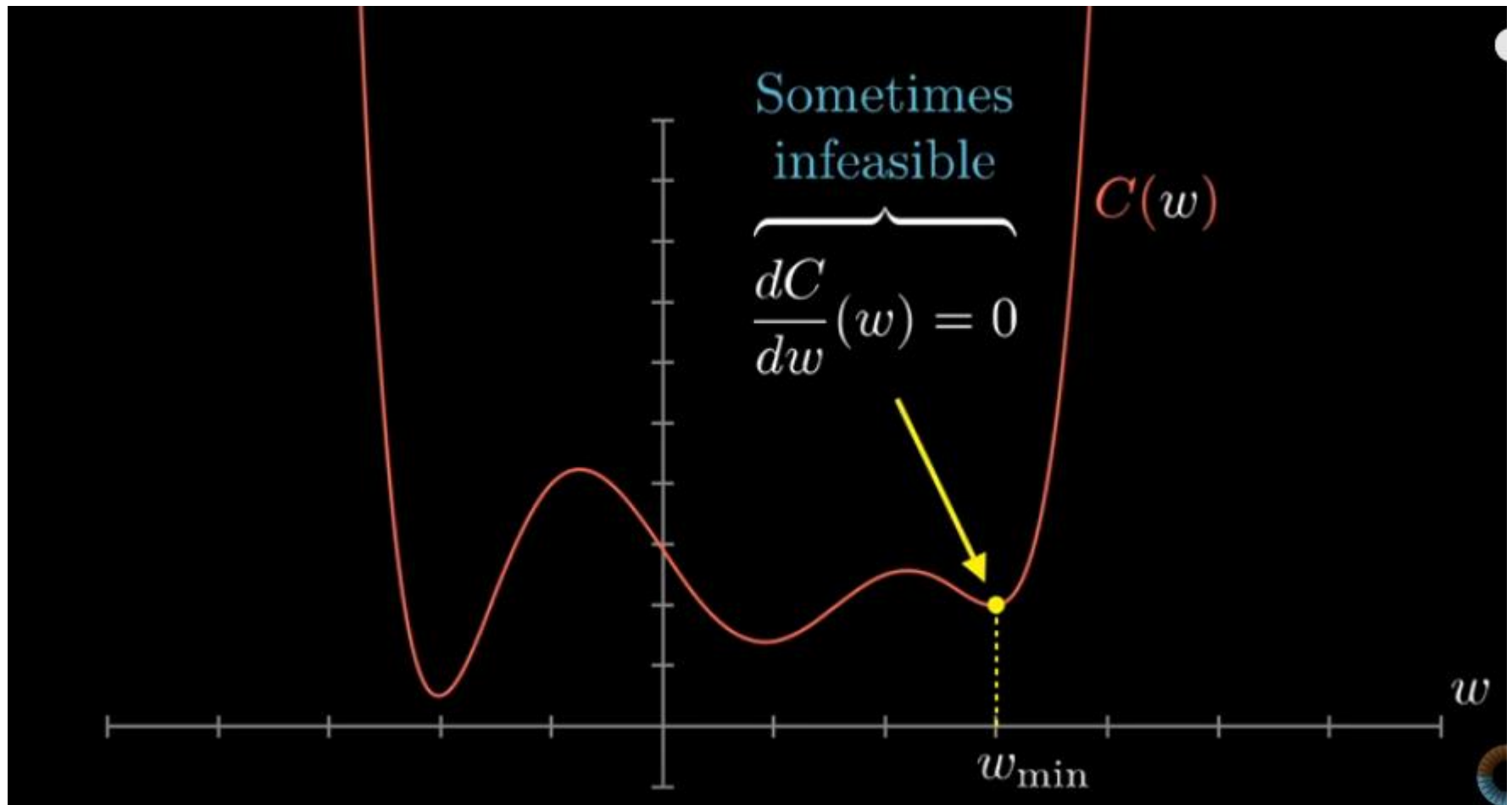
# Gradient descent



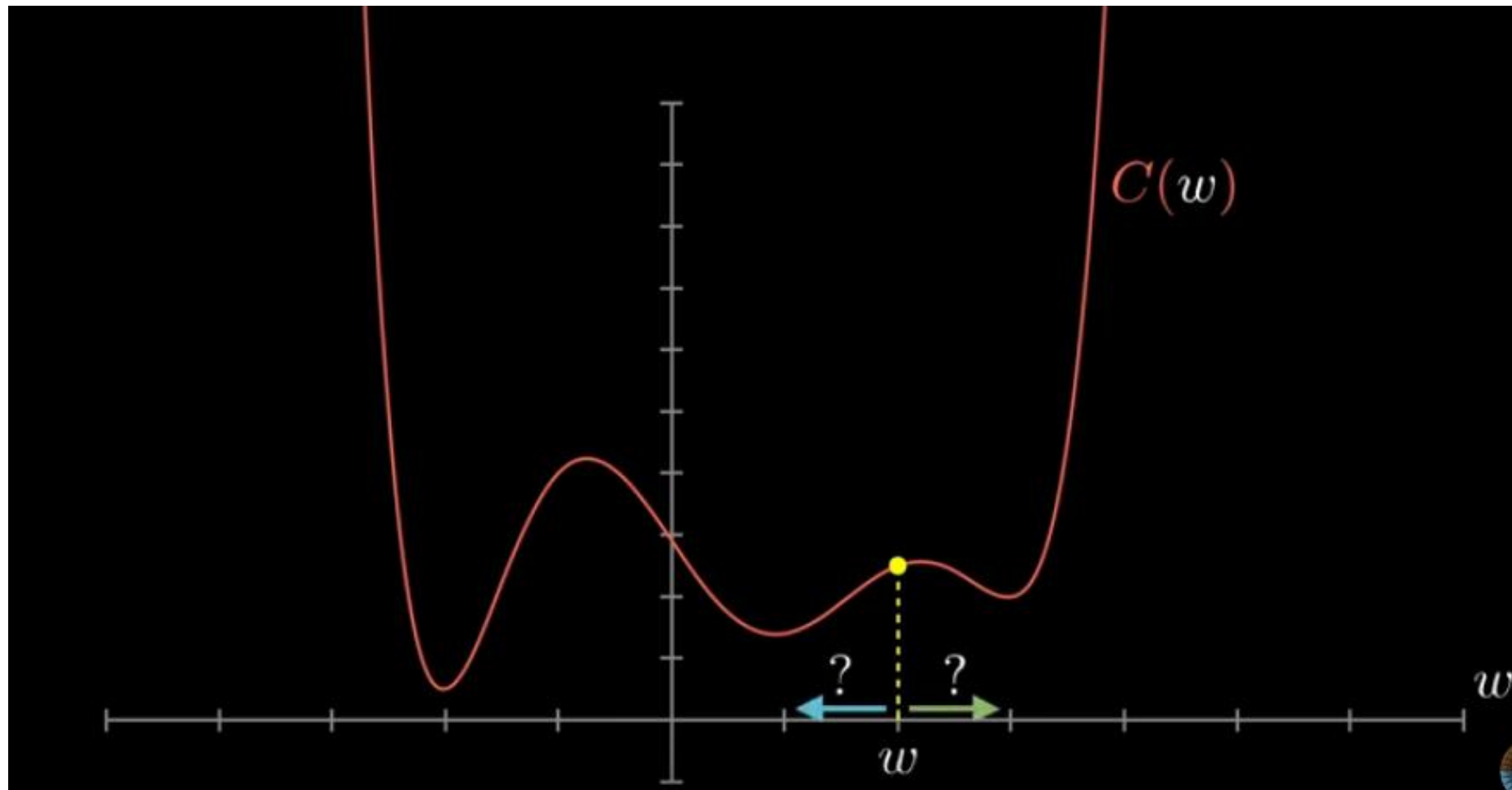
# Gradient descent



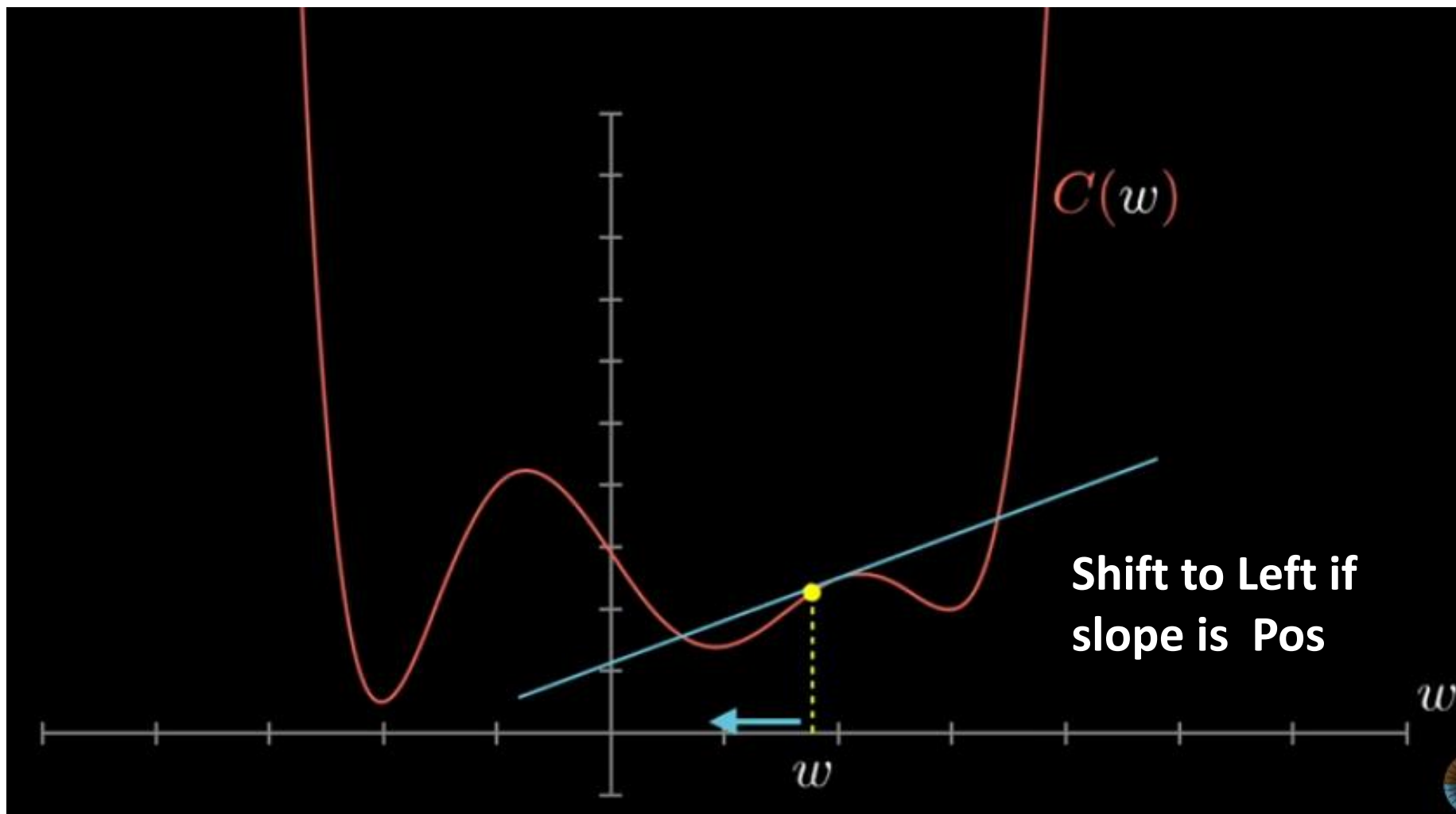
# Gradient descent



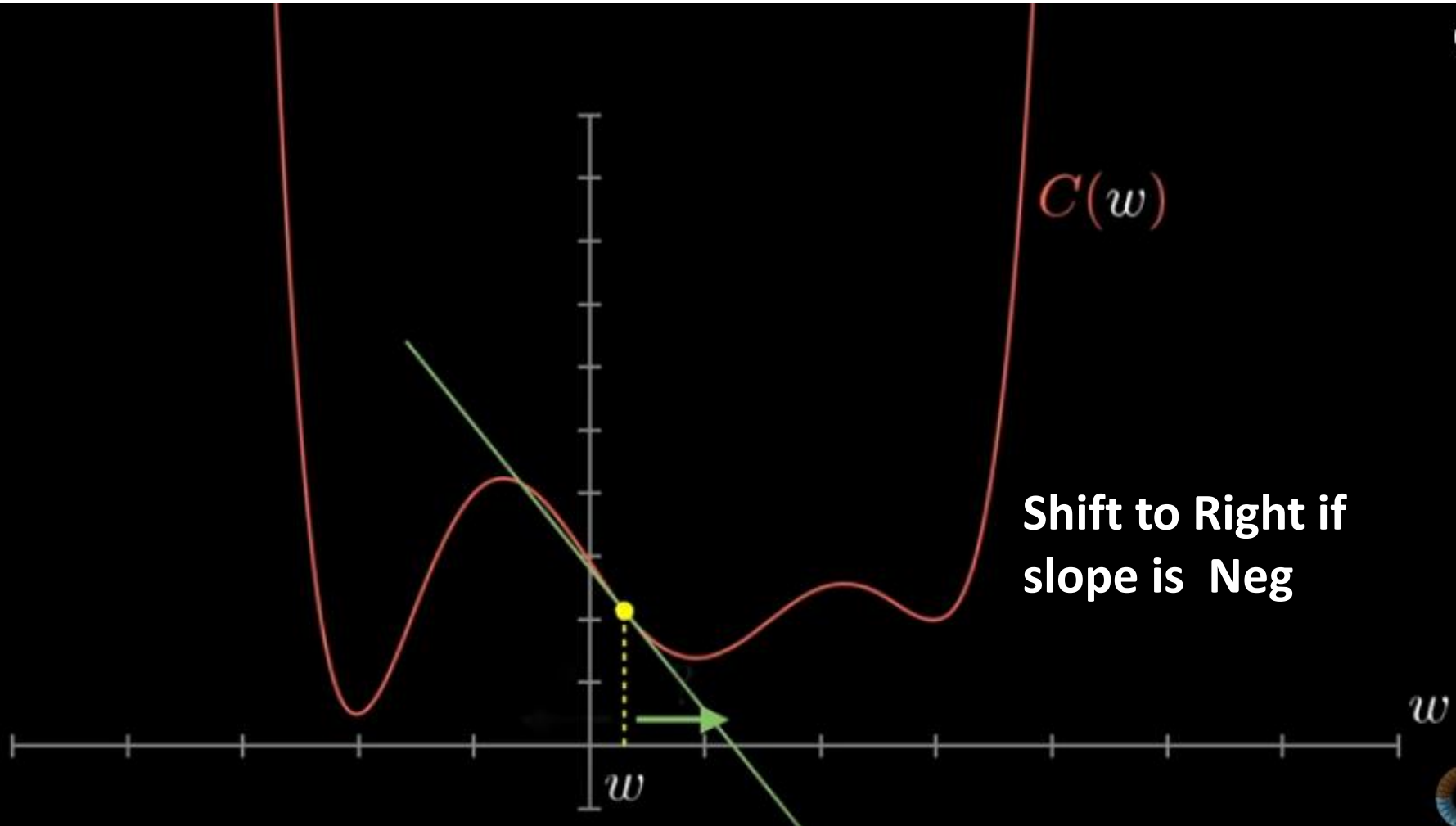
# Gradient descent



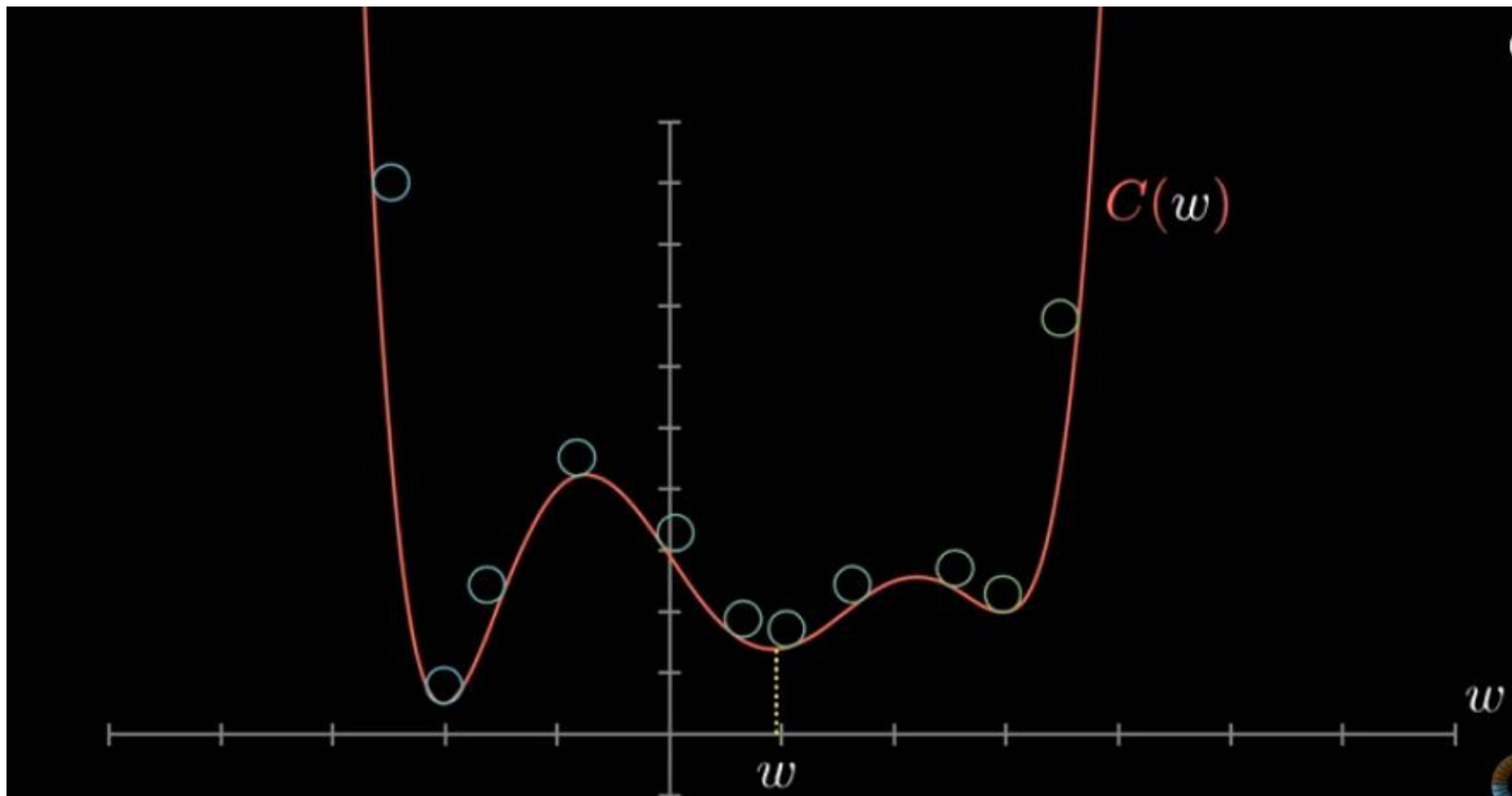
# Gradient descent



# Gradient descent



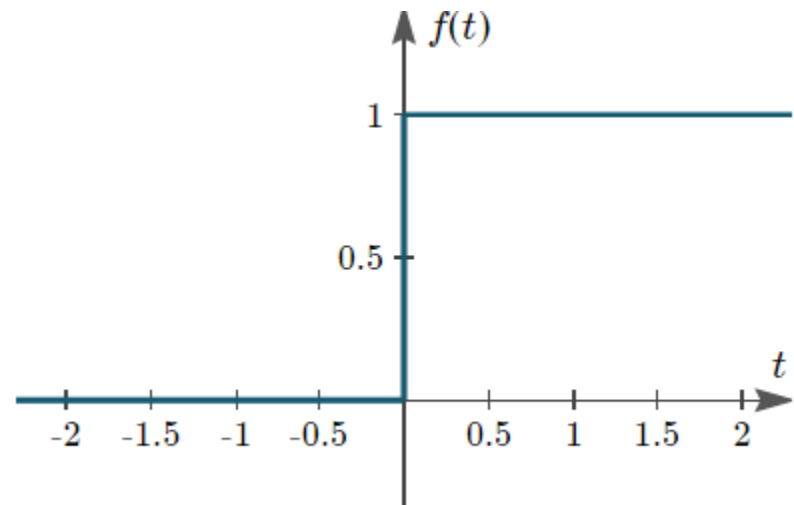
# Gradient descent





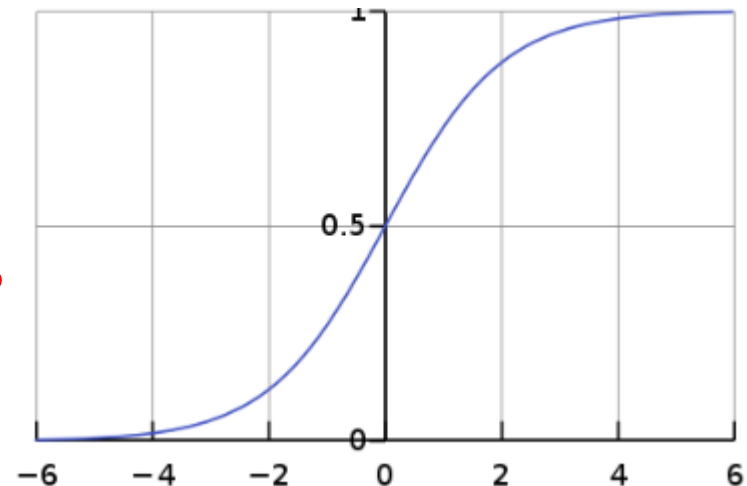
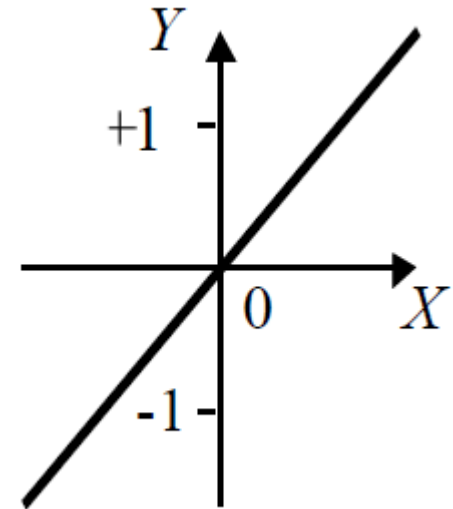
# Activation Functions (2)

- **Activation functions** are mathematical equations that determine the **output of a neural network**. The **function** is attached to each neuron in the network, and determines whether it should be **activated or not**.
- Step Function
  - Binary classification
  - **Derivative is 0 !!!!**



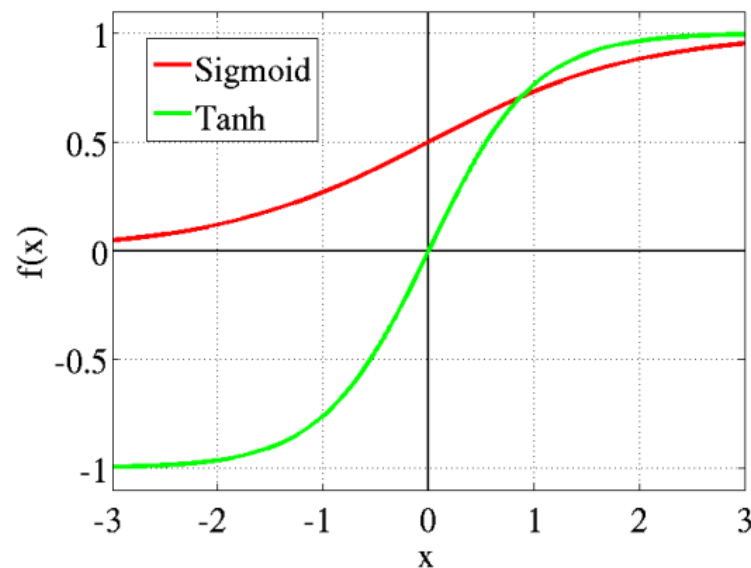
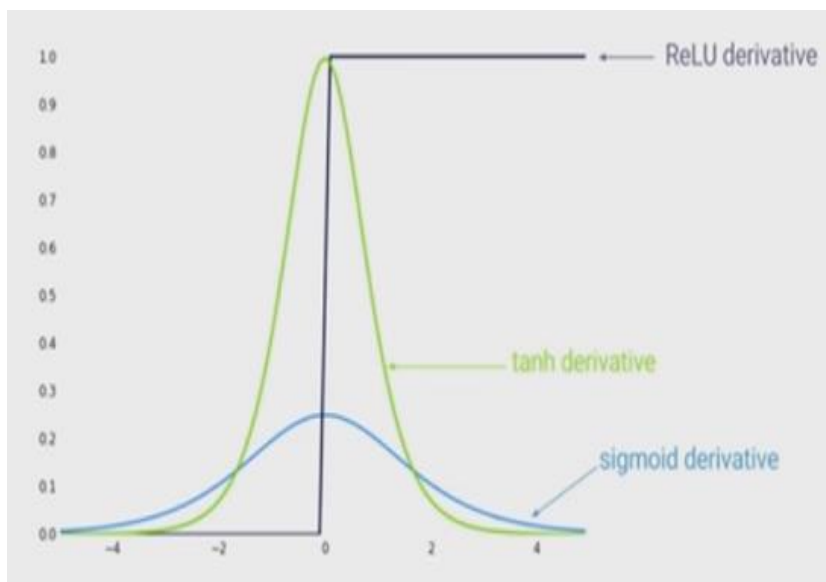
# Activation Functions (2)

- Linear
  - Range of activations (MAX)
  - Derivative is a constant. That means, the gradient has no relationship with  $X$
- Sigmoid
  - It's good for a classifier.
  - Saturates and kills gradients



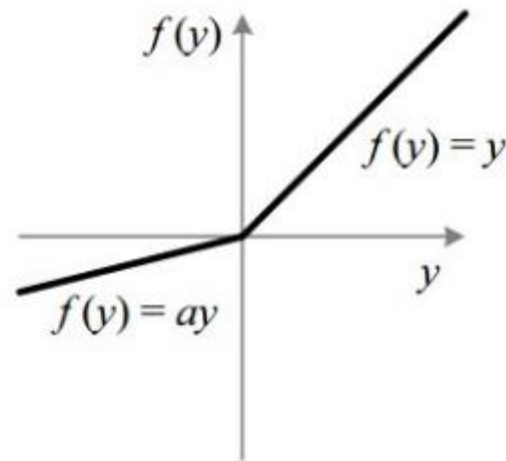
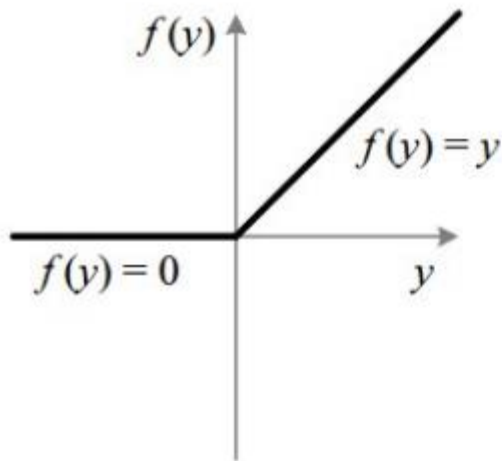
# Activation Functions (2)

- Tanh
  - It's good for a classifier.
  - The gradient is stronger for tanh than sigmoid (optimization converges faster)
  - **Vanishing gradients**



# Activation Functions (2)

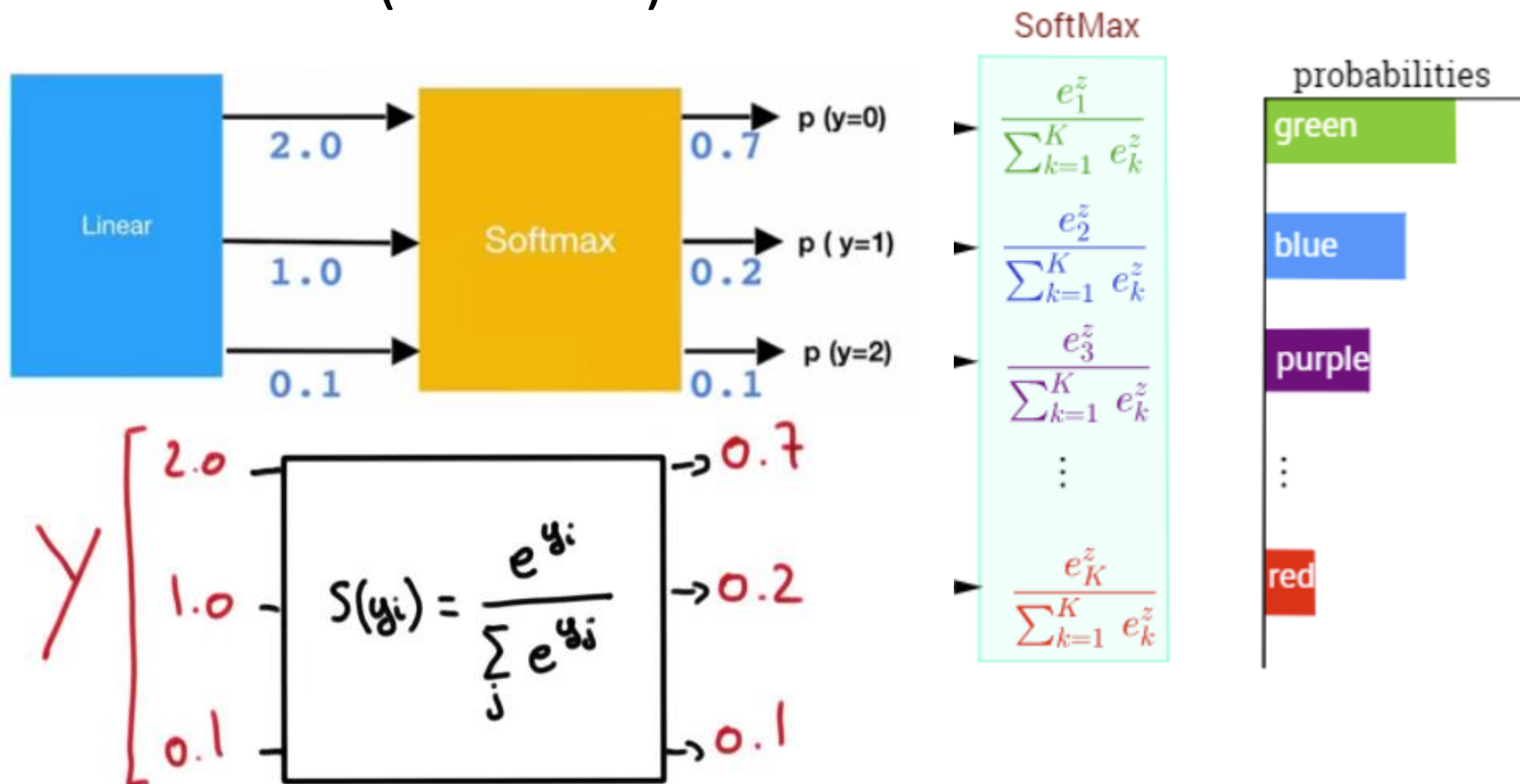
- ReLU
  - Simple and efficient  $[0, \text{Inf}]$
  - Dying neuron



Leaky ReLU

# Activation Functions (2)

- Softmax (**OUTPUT**)



# Softmax and Cross Entropy



Dog

0.9

Cat

0.1

$$H(p, q) = - \sum_x p(x) \log q(x)$$

1

0

# Softmax and Cross Entropy

# NN1

# NN2

Dog

1

0.9

0.6

Cat

0

0.1

0.4

Dog

0

0.1

0.3

Cat

1

0.9

0.7

Dog

1

0.4

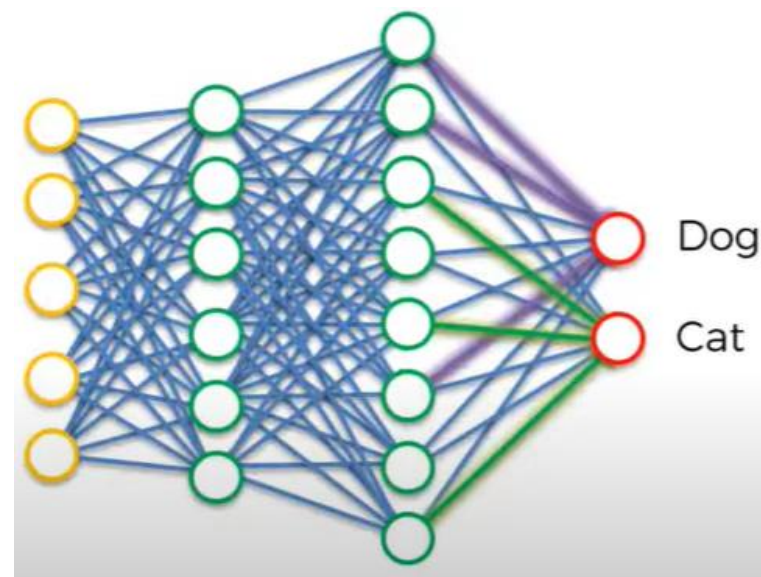
0.1

Cat

0

0.6

0.9



# Softmax and Cross Entropy

NN1

Row	Dog^	Cat^	Dog	Cat
#1	0.9	0.1	1	0
#2	0.1	0.9	0	1
#3	0.4	0.6	1	0

Classification Error

$$1/3 = 0.33$$

Mean Squared Error

$$0.25$$

Cross-Entropy

$$0.38$$

NN2

Row	Dog^	Cat^	Dog	Cat
#1	0.6	0.4	1	0
#2	0.3	0.7	0	1
#3	0.1	0.9	1	0

$$1/3 = 0.33$$

$$0.71$$

$$1.06$$

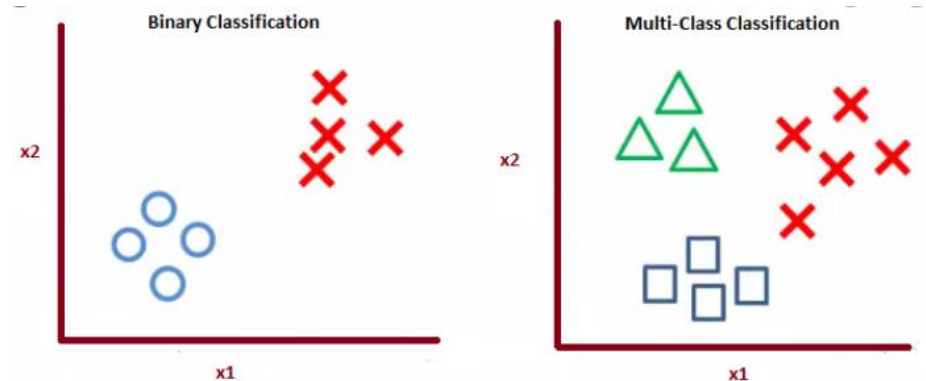
$$\text{cross-entropy} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^k t_{i,j} \log(p_{i,j})$$



# Supervised Learning

## Output Layer

- **Regression**
  - Linear
- **Classification**
  - Binary classification **Sigmoid**, or **Tanh**
  - Multi-class classification **Softmax**

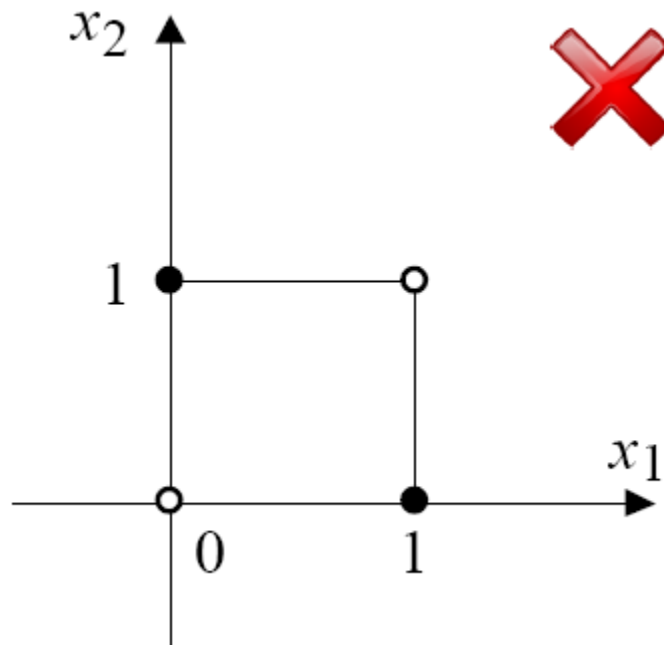


## Hidden Layer

- **Tanh**, or **ReLU**

# XOR Example

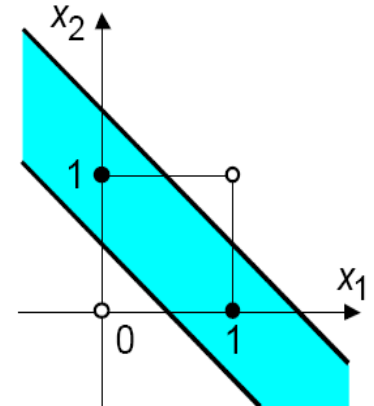
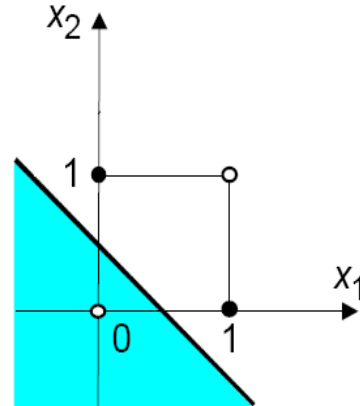
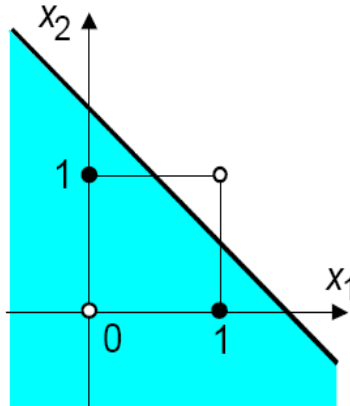
- Can we train a perceptron to recognize logical XOR??



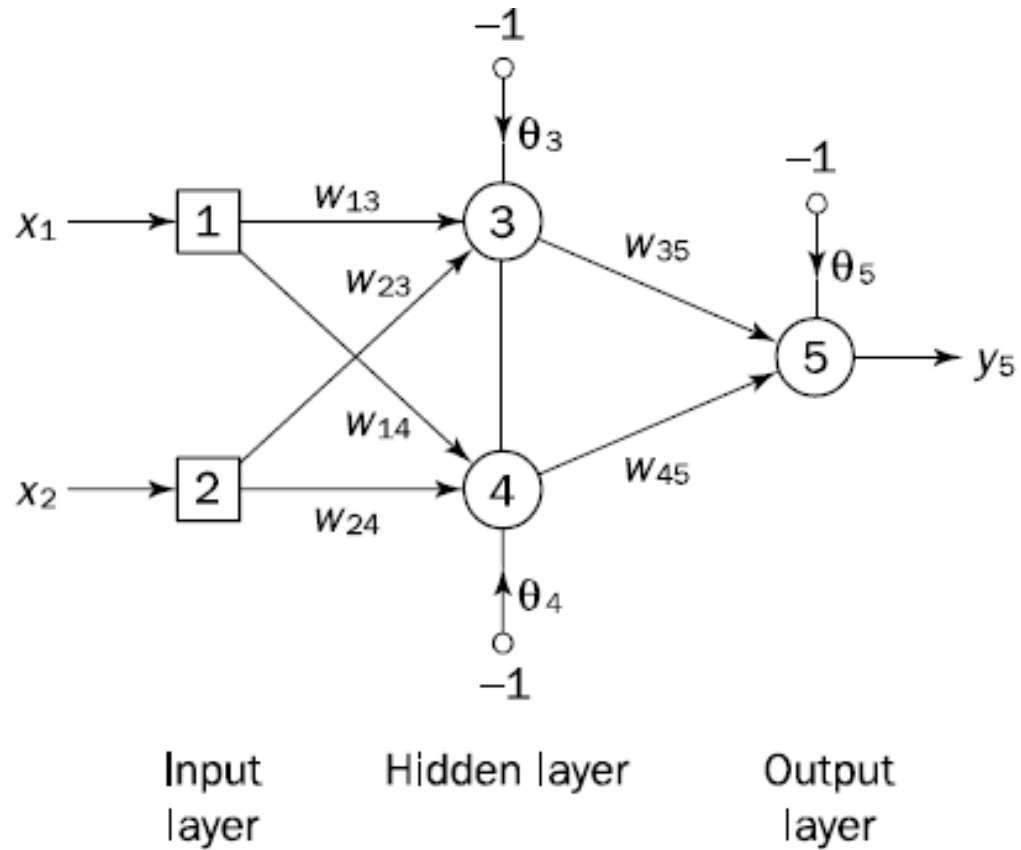
# XOR Example

- Train a neural network to recognize logical XOR

A	B	AXORB
1	1	0
1	0	1
0	1	1
0	0	0



# XOR Example



# XOR Example

## 1. Initialization

$w_{13} = 0.5, w_{14} = 0.9, w_{23} = 0.4, w_{24} = 1.0, w_{35} = -1.2, w_{45} = 1.1,$   
 $\theta_3 = 0.8, \theta_4 = -0.1$  and  $\theta_5 = 0.3$ .

## 2. Activation (X1=1, X2=1, Yd=0)

a) Hidden

$$y_3 = \text{sigmoid}(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1/[1 + e^{-(1 \times 0.5 + 1 \times 0.4 - 1 \times 0.8)}] = 0.5250$$

$$y_4 = \text{sigmoid}(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1/[1 + e^{-(1 \times 0.9 + 1 \times 1.0 + 1 \times 0.1)}] = 0.8808$$

# XOR Example

## b) Output

$$y_5 = \text{sigmoid}(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1/[1 + e^{-( -0.5250 \times 1.2 + 0.8808 \times 1.1 - 1 \times 0.3 )}] = 0.5097$$

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

## 3. Training

### a) Output

$$\delta_5 = y_5(1 - y_5)e = 0.5097 \times (1 - 0.5097) \times (-0.5097) = -0.1274$$

# XOR Example

- Assume  $\alpha=0.1$

$$\Delta w_{35} = \alpha \times y_3 \times \delta_5 = 0.1 \times 0.5250 \times (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \times y_4 \times \delta_5 = 0.1 \times 0.8808 \times (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \times (-1) \times \delta_5 = 0.1 \times (-1) \times (-0.1274) = 0.0127$$

## b) Hidden

$$\delta_3 = y_3(1 - y_3) \times \delta_5 \times w_{35} = 0.5250 \times (1 - 0.5250) \times (-0.1274) \times (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \times \delta_5 \times w_{45} = 0.8808 \times (1 - 0.8808) \times (-0.1274) \times 1.1 = -0.0147$$

# XOR Example

$$\Delta w_{13} = \alpha \times x_1 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \times x_2 \times \delta_3 = 0.1 \times 1 \times 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \times (-1) \times \delta_3 = 0.1 \times (-1) \times 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \times x_1 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \times x_2 \times \delta_4 = 0.1 \times 1 \times (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \times (-1) \times \delta_4 = 0.1 \times (-1) \times (-0.0147) = 0.0015$$



# XOR Example

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta \theta_3 = 0.8 - 0.0038 = 0.7962$$

$$\theta_4 = \theta_4 + \Delta \theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta \theta_5 = 0.3 + 0.0127 = 0.3127$$

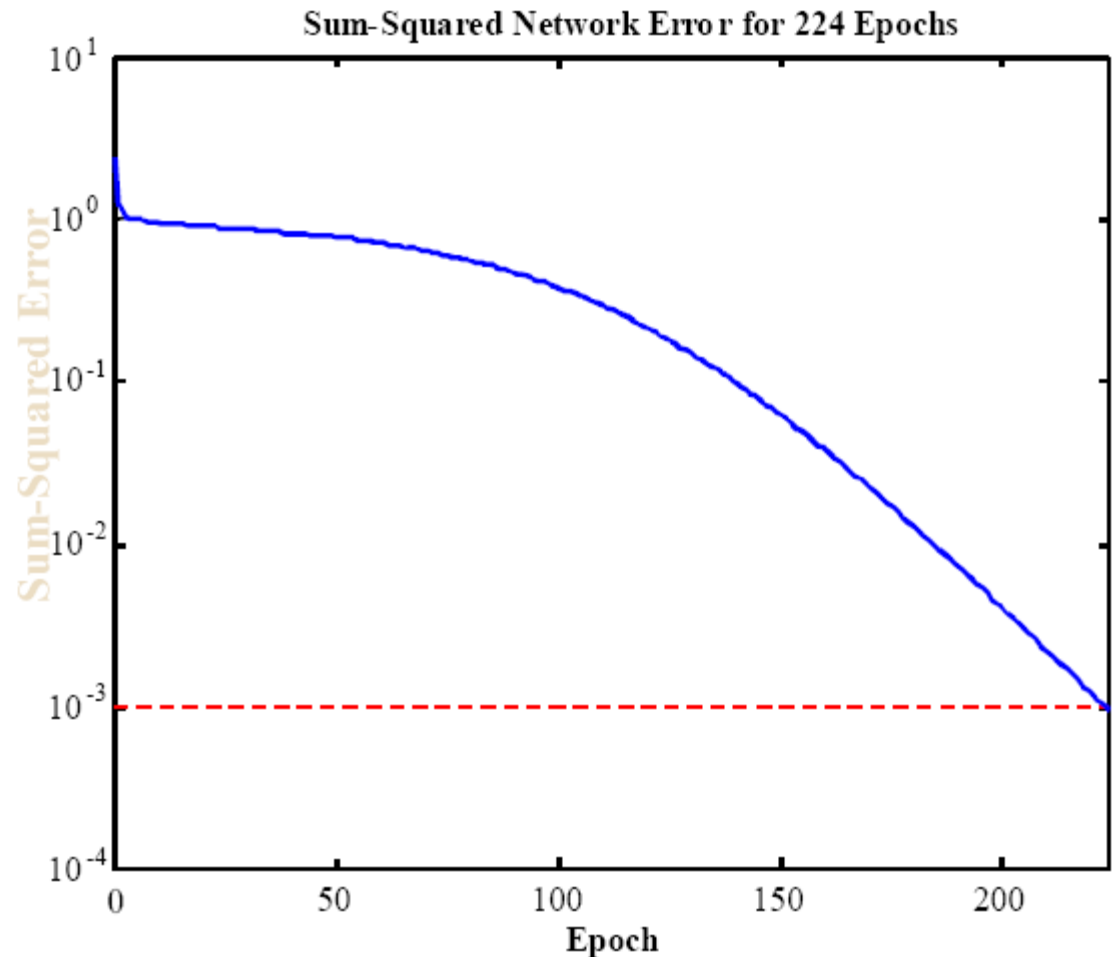
## 4. Iteration

(go to next iteration)

The training process is repeated until the sum of squared errors is **less than 0.001**.

# Final results of three-layer network learning

- Start with *random* weights
  - ▣ Repeat until the *sum of the squared errors* is below 0.001
  - ▣ Depending on initial weights, final converged results may vary



# Final results of three-layer network learning

- It took 224 epochs or 896 iterations to train our network to perform the Exclusive-OR operation.
- The following set of final weights and threshold levels satisfied the chosen error criterion:

$$w_{13} = 4.7621, w_{14} = 6.3917, w_{23} = 4.7618, w_{24} = 6.3917, w_{35} = -10.3788, \\ w_{45} = 9.7691, \theta_3 = 7.3061, \theta_4 = 2.8441 \text{ and } \theta_5 = 4.5589.$$

- The network has solved the problem! We may now test our network by presenting all training sets and calculating the network's output.

# Final results of three-layer network learning

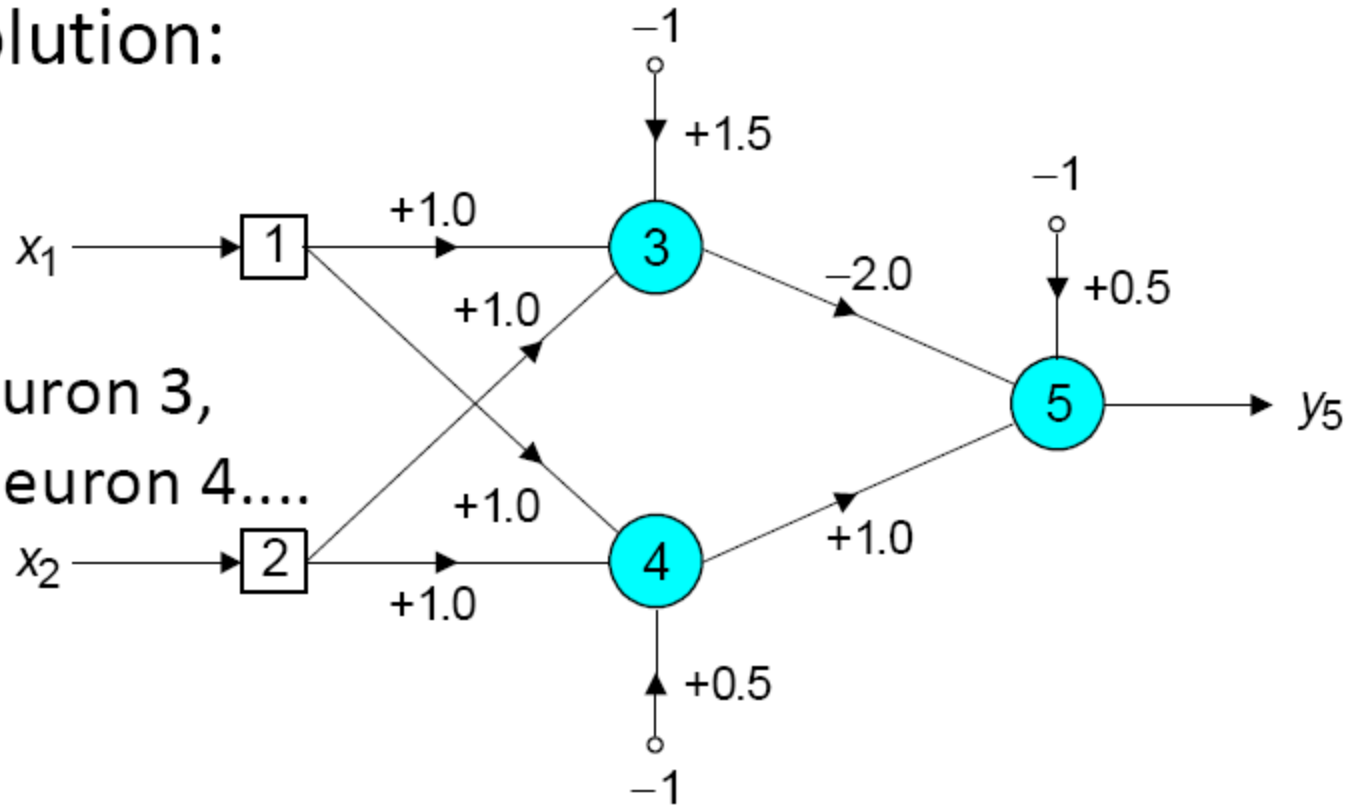
- After 224 epochs (896 individual iterations), the neural network has been trained

Inputs		Desired output $y_d$	Actual output $y_5$	Error $e$	Sum of squared errors
$x_1$	$x_2$				
1	1	0	0.0155	-0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	-0.0175	

# Another Solution !!

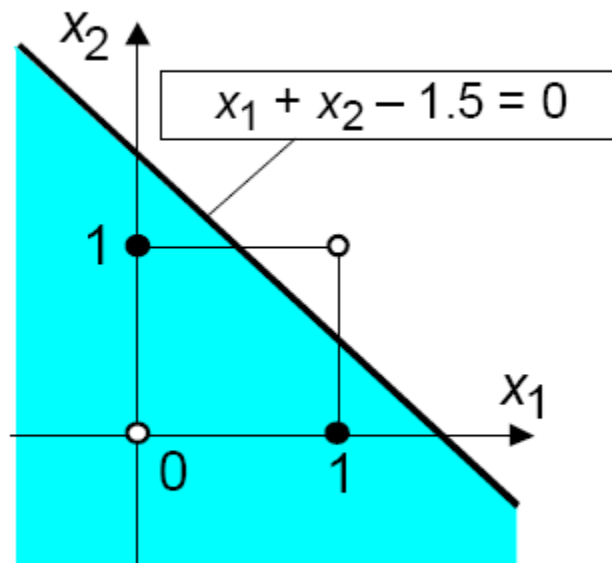
- No longer limited to *linearly separable functions*
- Another solution:

▣ Isolate neuron 3,  
then neuron 4....

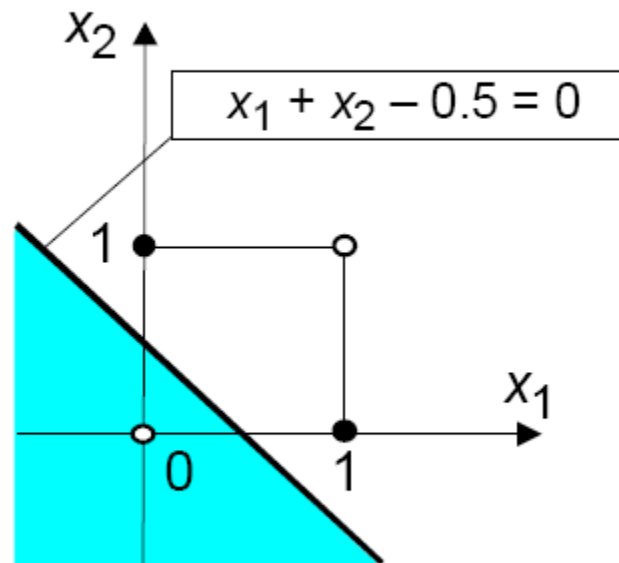


# Another Solution !!

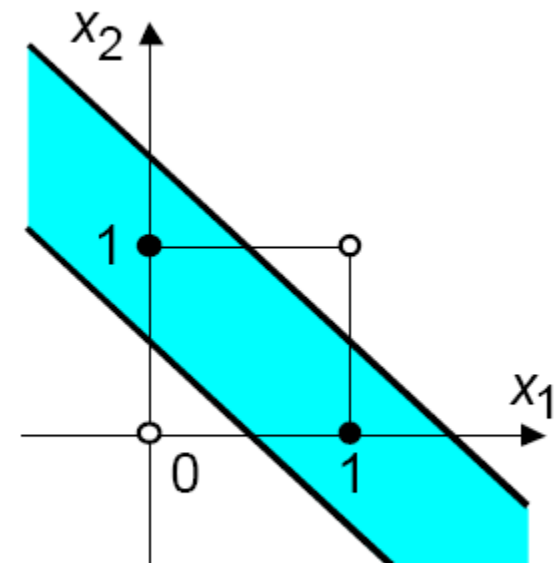
- Combine *linearly separable functions* of neurons 3 and



(a)



(b)



(c)

# Accelerating the learning in the FFNN

## 1. Hyperbolic Tangent

A multilayer network, in general, learns much faster when the sigmoidal activation function is represented by a **hyperbolic tangent**,

$$y^{tanh} = \frac{2a}{1 + e^{-bx}} - a, \quad f(x) = \frac{2}{1 + e^{-2x}} - 1$$

where  $a$  and  $b$  are constants.

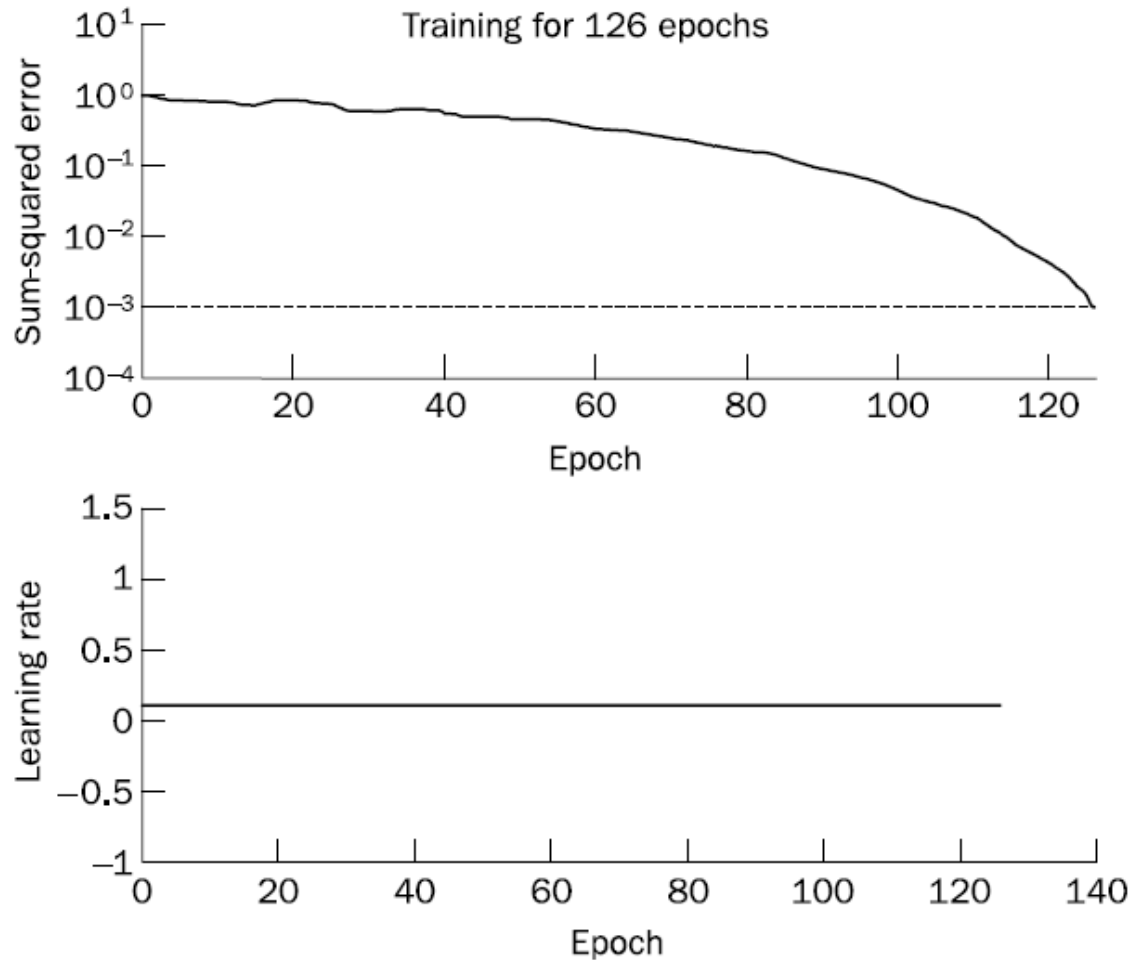
## 2. Momentum Term

We also can accelerate training by including a **momentum term** in the delta rule

$$\Delta w_{jk}(p) = \beta \times \Delta w_{jk}(p-1) + \alpha \times y_j(p) \times \delta_k(p),$$

where  $\beta$  is a positive number ( $0 \leq \beta < 1$ ) called the momentum constant. Typically, the momentum constant is set to 0.95.

# Accelerating the learning in the FFNN





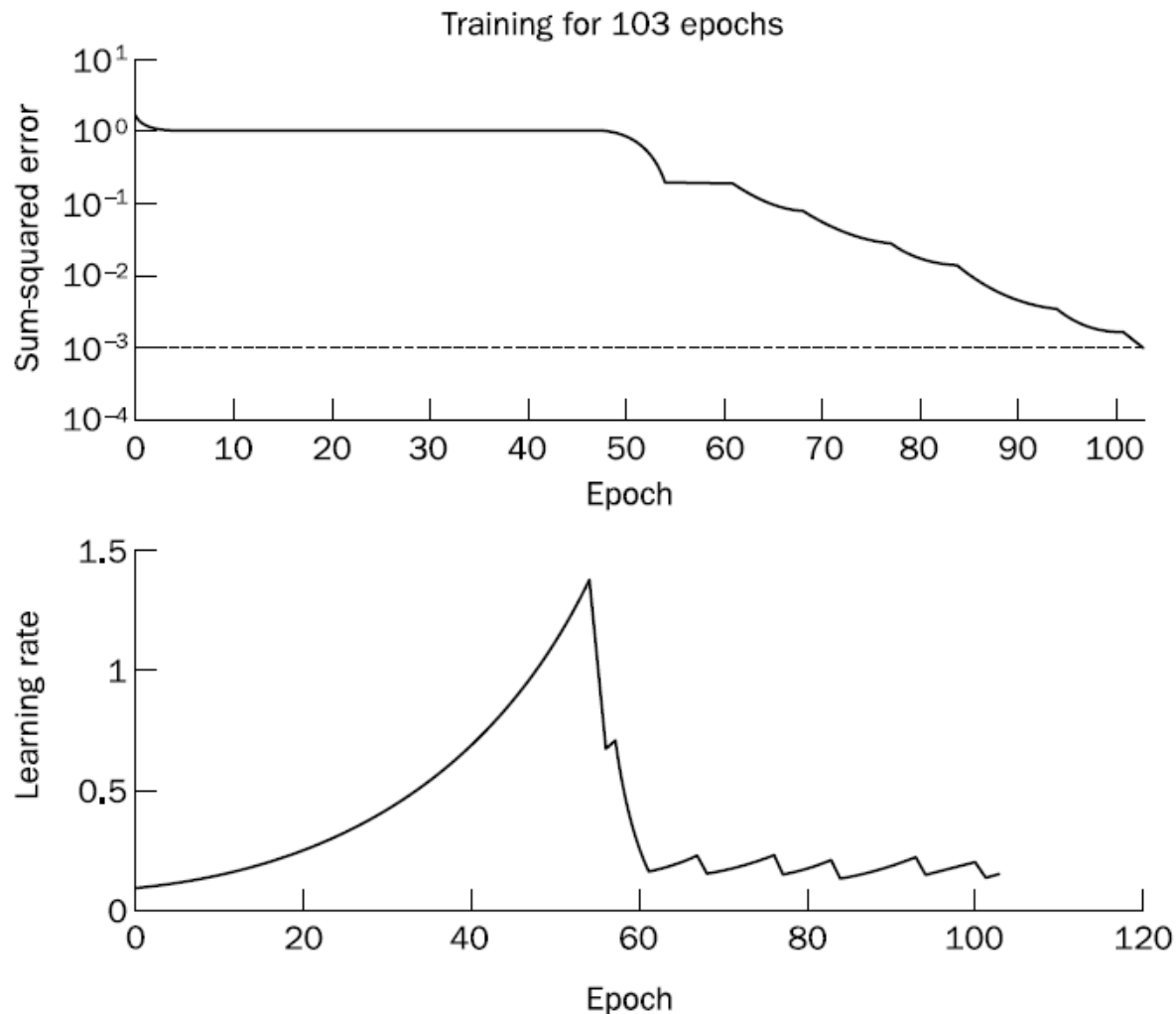
# Accelerating the learning in the FFNN

## 3. Adaptive learning rate

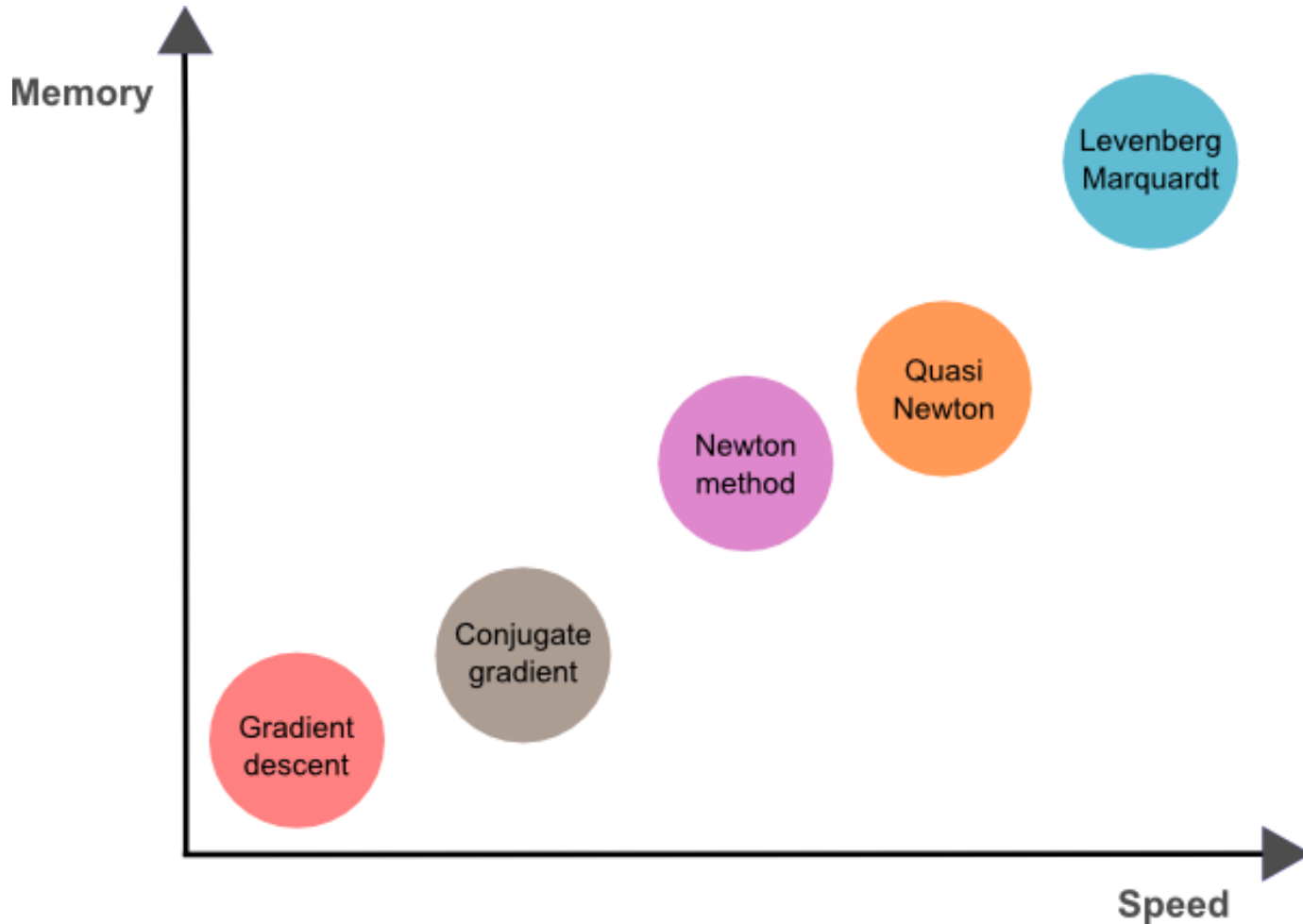
- If the **error** for current epoch is **decreasing** (than the previous one) the **learning rate  $\alpha$** , should be **increased** (Mult by 1.05).
- If the **error** for current epoch is **increasing** or **remaining constant** the **learning rate  $\alpha$** , should be **decreased** (Mult by 0.7).

## 4. Training Algorithm

# Accelerating the learning in the FFNN



# Training Algorithm



**Input:** A training input vector ( $I_N$ ), desired output vector ( $O_N$ )

**Output:** A vector of modified weights and Biases ( $W$ )

**Algorithm:**

Set all the weights and Bias levels of the network to random numbers uniformly distributed inside a small range  $\left[-2.4/f_i, +2.4/f_i\right]$

Compute: 
$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_{vi} - \bar{Y}_{vi})^2,$$

**while** (not stop-criterion) **do**

    Compute: 
$$\Delta W = (JJ^T + \mu I_m)^{-1} J^T e,$$

    Update the network weights ( $W$ ) using  $\Delta W$

    Recalculate **MSE** using the updated weights

**if** **MSE** decrease **then**

$$\mu = 0.1\mu$$

**else**

            Discard the new weights

$$\mu = 10\mu$$

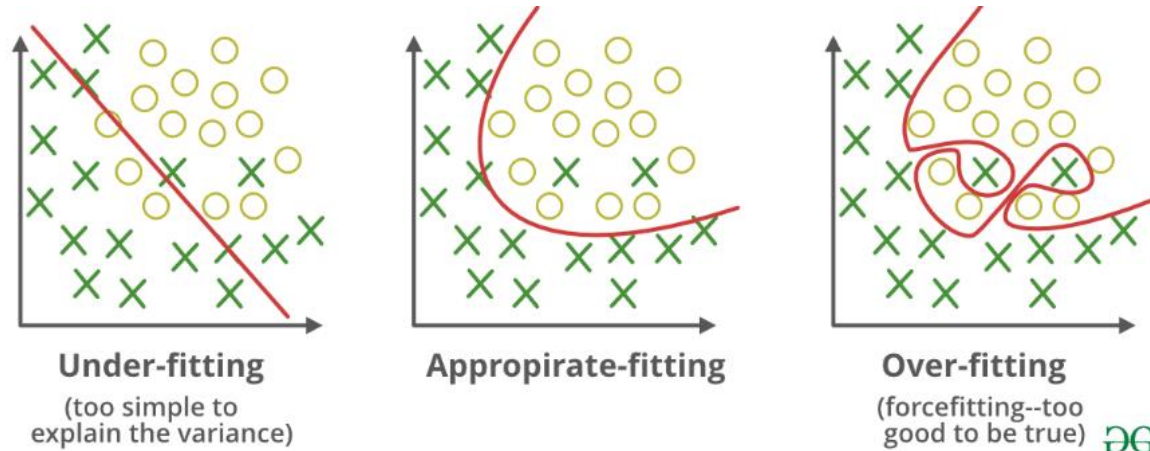
**end if**

**end while**

## Levenberg Training Algorithm

$$J = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \dots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \dots & \frac{\partial e_{1,2}}{\partial w_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \dots & \frac{\partial e_{1,M}}{\partial w_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial e_{P,1}}{\partial w_1} & \frac{\partial e_{P,1}}{\partial w_2} & \dots & \frac{\partial e_{P,1}}{\partial w_N} \\ \frac{\partial e_{P,2}}{\partial w_1} & \frac{\partial e_{P,2}}{\partial w_2} & \dots & \frac{\partial e_{P,2}}{\partial w_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial e_{P,M}}{\partial w_1} & \frac{\partial e_{P,M}}{\partial w_2} & \dots & \frac{\partial e_{P,M}}{\partial w_N} \end{bmatrix}$$

# Overfitting (Early Stopping)



## Training:

These are presented to the network during training, and the network is adjusted according to its error.

## Validation:

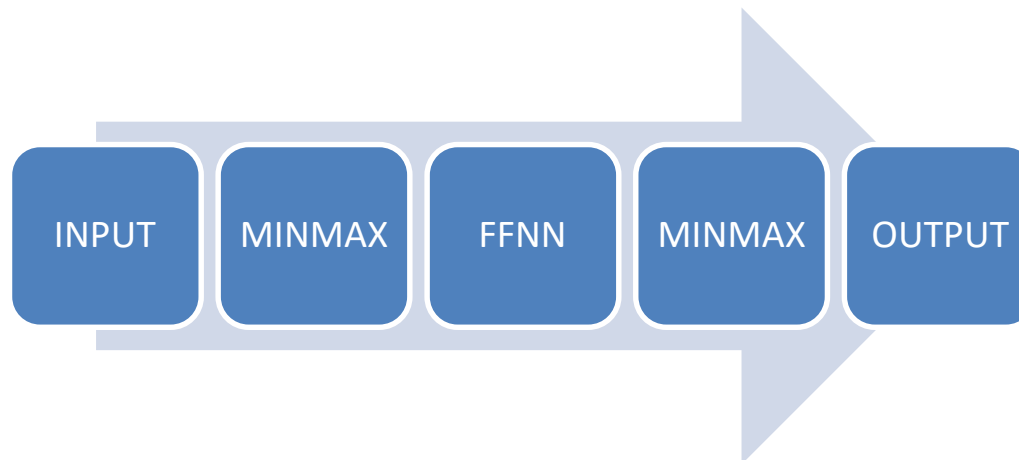
These are used to measure network generalization, and to halt training when generalization stops improving.

## Testing:

These have no effect on training and so provide an independent measure of network performance during and after training.

# Data normalization

- Normalization is a technique often applied as part of data preparation for machine learning.
- Normalizing your data is an essential part of machine learning. You might have an amazing dataset with many great features, but if you forget to normalize, one of those features might completely dominate the others. It's like you're throwing away almost all of your information! Normalizing solves this problem.



# Data normalization

## Min-Max normalization method

Guarantees all features will have the exact same scale

$$I_N = (I - I_{\min}) \left[ \frac{N_{\max} - N_{\min}}{I_{\max} - I_{\min}} \right] + N_{\min} \quad (4)$$

where  $I$  is the non-normalized training input value,  $I_N$  is the normalized training input value,  $I_{\min}$  is the minimum value for the input vector  $I$ , and  $I_{\max}$  is the maximum value for the input vector  $I$ . For this work,  $N_{\max} = 1$  and  $N_{\min} = -1$ . After getting the predicting result, the normalized values must become normal by using the following equation:

$$O = (O_N - N_{\min}) \left[ \frac{O_{\max} - O_{\min}}{N_{\max} - N_{\min}} \right] + O_{\min} \quad (5)$$

where  $O$  is the non-normalized training output value,  $O_N$  is the normalized training output value,  $O_{\min}$  is the minimum value for the output vector  $O$ , and  $O_{\max}$  is the maximum value for the output vector  $O$ .

# R-squared (Regression)

- $R\text{-squared} = 1 - (SS_{\text{res}} / SS_{\text{tot}})$

## R-Squared Example

Here's an example with an imaginary linear regression model  $f(x)$ .

obs	y	avg(y) - y	(avg(y) - y) ^ 2	f(x)	f(x)-y	(f(x)-y) ^ 2
1	4	2	4	5	1	1
2	10	-4	16	9	-1	1
3	2	4	16	4	2	4
4	8	-2	4	8	0	0
5	6	0	0	5	-1	1
SUM	30		40			7
AVG	6					

The table above shows the individual calculations with  $SS_{\text{tot}} = 40$  and  $SS_{\text{res}} = 7$ . The next step is to take that quotient of  $SS_{\text{res}} / SS_{\text{tot}} = 7 / 40 = 0.175$

Now, we subtract that quotient from one.  $1 - 0.175 = 0.835$ . We now know that the model explains 83.5% of the natural variance in the y variable.



# Confusion Matrix (Classification)

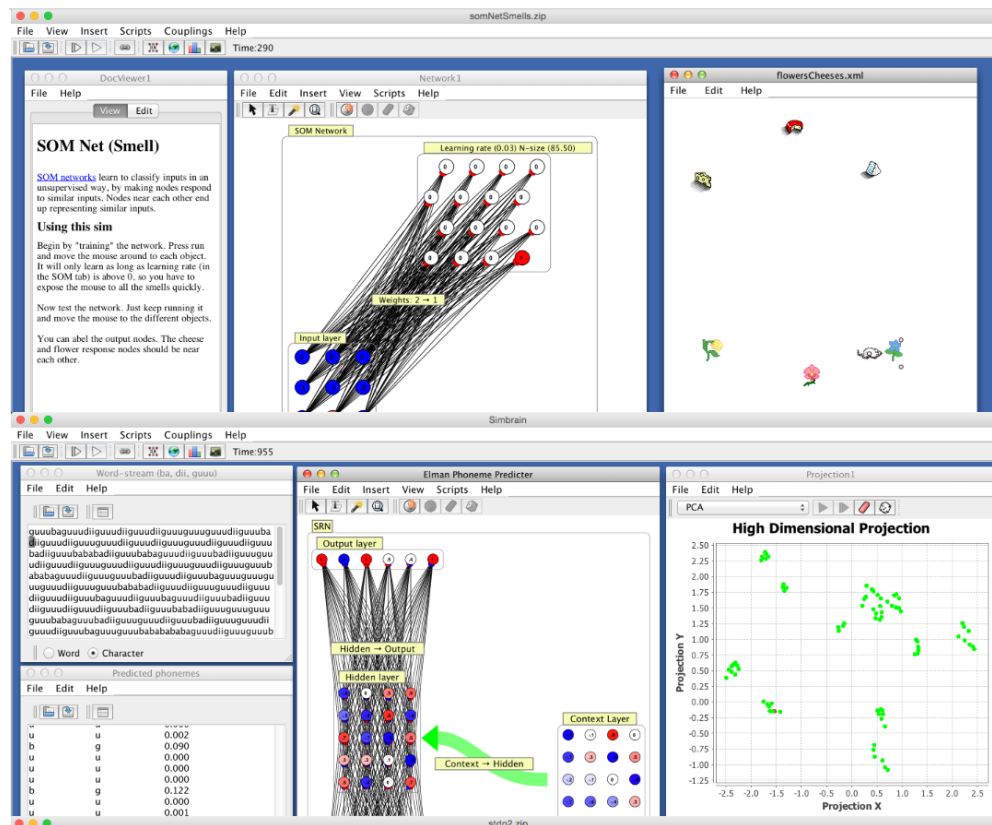
- A confusion matrix is a table that is often used to **describe the performance of a classification model** (or "classifier") on a set of test data for which the true values are known.

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	• Accura

- **Accuracy:** Overall, how often is the classifier correct?
  - $(TP+TN)/total = (100+50)/165 = 0.91$
- **Misclassification Rate:** Overall, how often is it wrong?
  - $(FP+FN)/total = (10+5)/165 = 0.09$
  - equivalent to 1 minus Accuracy
  - also known as "Error Rate"

# Simbrain Tool

- SIMBRAIN is a free tool for building, running, and analysing neural-networks



# MATLAB

- <https://www.mathworks.com/discovery/neural-network.html>
- <https://www.mathworks.com/help/deeplearning/examples.html>
- <https://www.mathworks.com/products/deep-learning.html>
- [https://www.mathworks.com/?s\\_tid=gn\\_logo](https://www.mathworks.com/?s_tid=gn_logo)
- <https://www.mathworks.com/help/deeplearning/gs/fit-data-with-a-neural-network.html>

# Animated math

- <https://www.3blue1brown.com/neural-networks>