


Faculty of SET / School of Computer and Mathematical Sciences

# COMP SCI 3007/7059/7659
# Artificial Intelligence
## Neural Networks

# Acknowledgement of Country

We acknowledge and pay our respects to the Kaurna people, the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to the country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

# Neural Networks

AIMA C18.7

# Outline

**Perceptron**

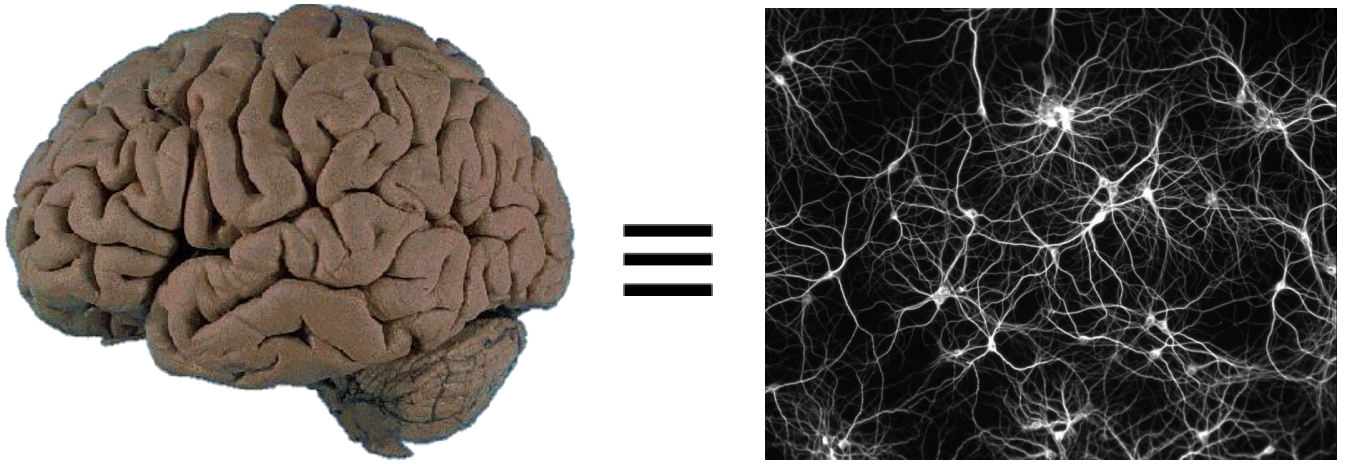
**Backpropagation**

# Artificial Neural Networks

- Artificial Neural Networks are a type of **biologically inspired** information processing systems.
- It is one of the earliest methods of AI, which attracted a lof of attention between the 1950's and 1980's due to their potential to automatically develop ways of solving problems, given appropriate training data.
- Neural networks have gone through several phases of decline and resurgence. Nonetheless it is still an interesting introductory AI topic to study.

# Biology Inspired

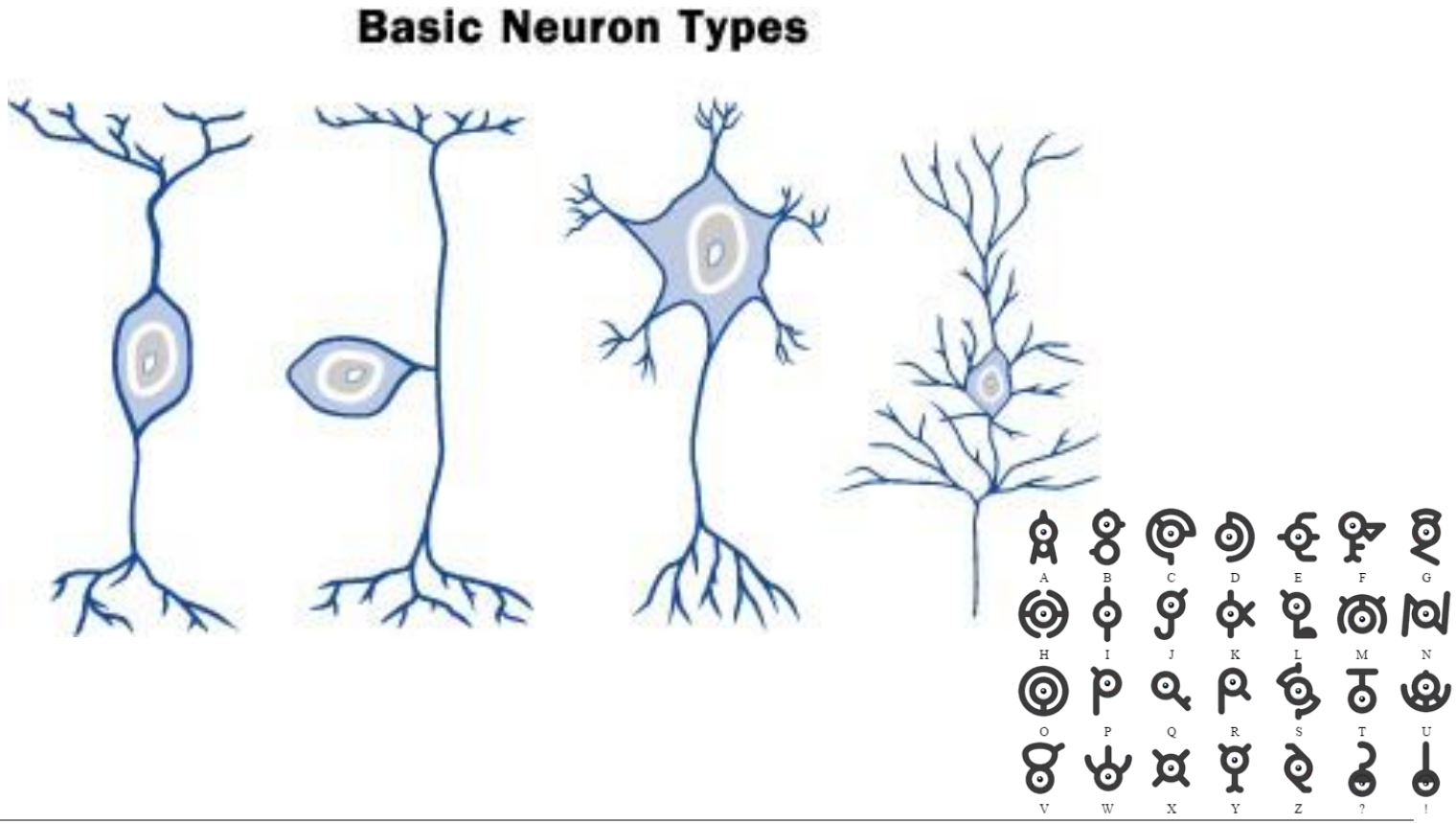
The inspiration behind Artificial Neural Networks is the brain. The brain is composed of cells called **neurons** which are interconnected to form a massive network:



It is thought that the brain's information processing capacity arise due to this network of neurons.

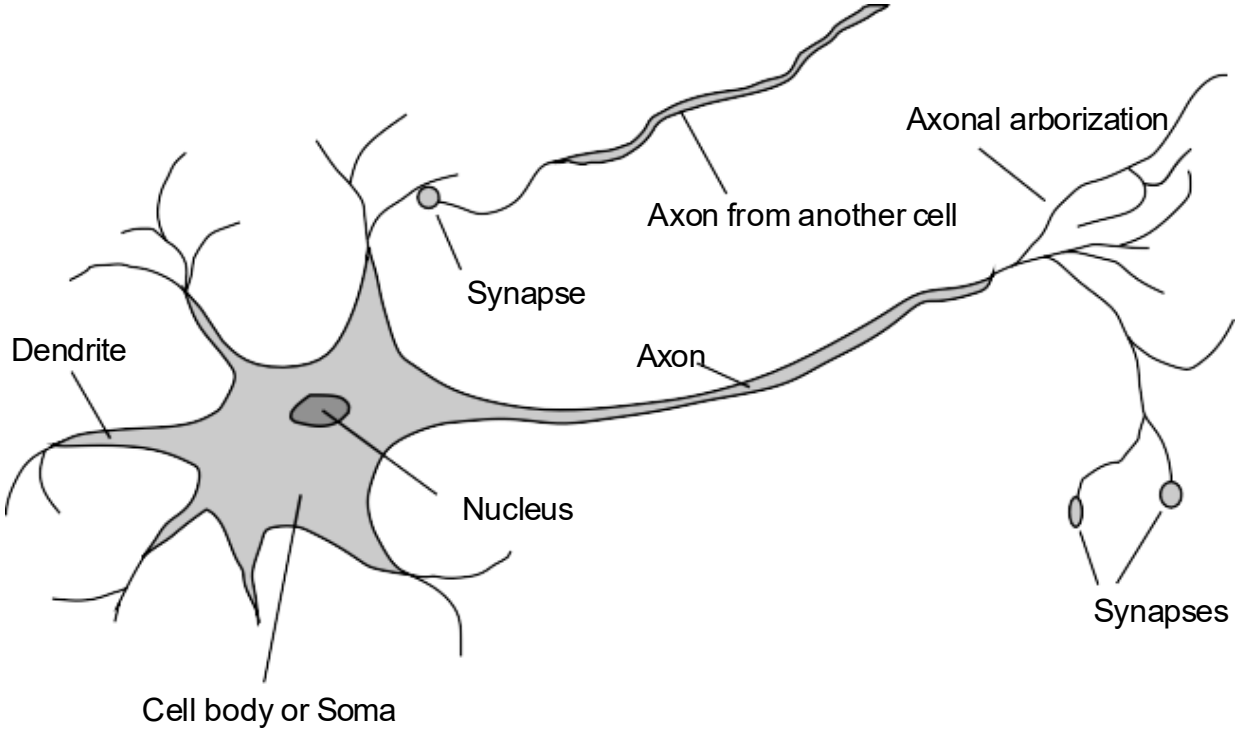
# Neurons

A neuron's principal function is the **collection, processing and dissemination of signals**. There are many types of neurons...



# Neurons (cont.)

… but neurons are more or less composed of the same standard parts:




**Synapses** connect one neuron to another, and signals are transferred through the synapses by an **electro-chemical** process.

New connections can be created by growing new synapses, while old connections can also be destroyed.

# Comparing the Brain and Computer

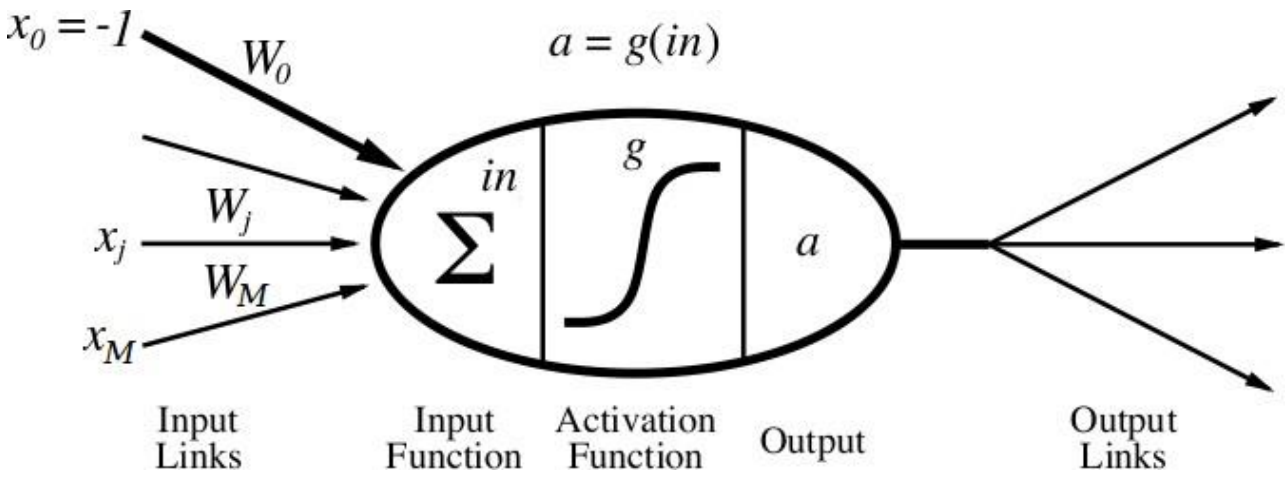
In the human brain, there are $10^{11}$ neurons of more than 20 types and $10^{14}$ synapses with 1ms – 10ms cycle time. Here's a comparison of the human brain and a modern computer:

| | Computer | Human Brain |
|---|---|---|
| Computation units | 1 CPU, $10^8$ gates | $10^{11}$ neurons |
| Storage units | $10^{10}$ bits RAM | $10^{11}$ neurons |
| | $10^{11}$ bits disk | $10^{14}$ synapses |
| Cycle time | $10^{-9}$ sec | $10^{-3}$ sec |
| Bandwidth | $10^{10}$ bits/sec | $10^{14}$ bits/sec |
| Memory updates/sec | $10^9$ | $10^{14}$ |

The human brain has not changed for the last 10,000 years. Moore's Law (doubling of transistors per area every 2 years) will ensure that computers will catch up soon.

# Artificial Neuron

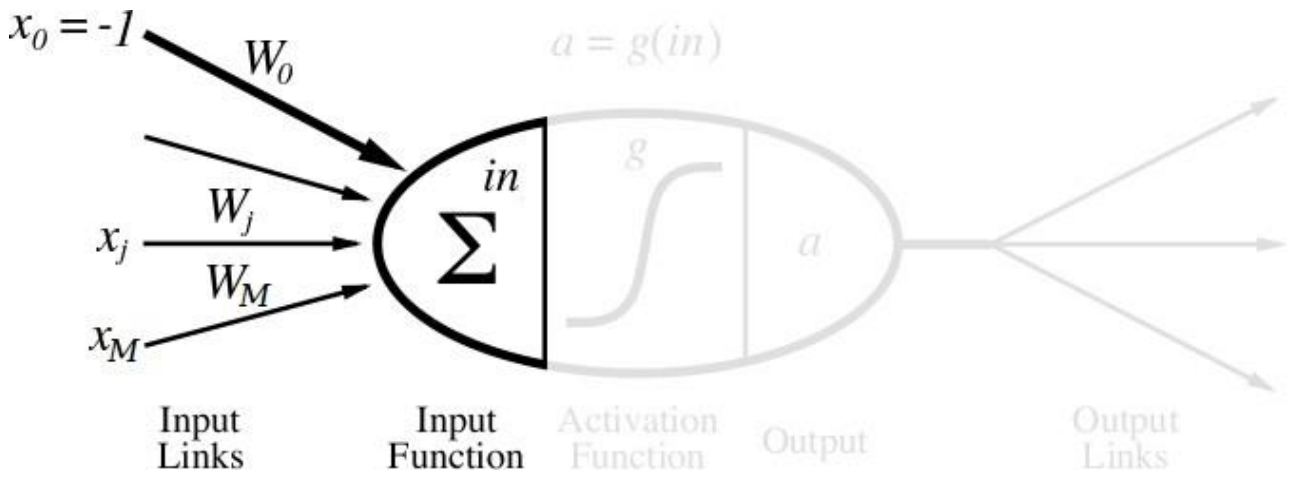
A mathematical model of the neuron due to McCulloch and Pitts (1943):




It is an oversimplification of real neurons, but the goal was to investigate what networks of simple units can do.

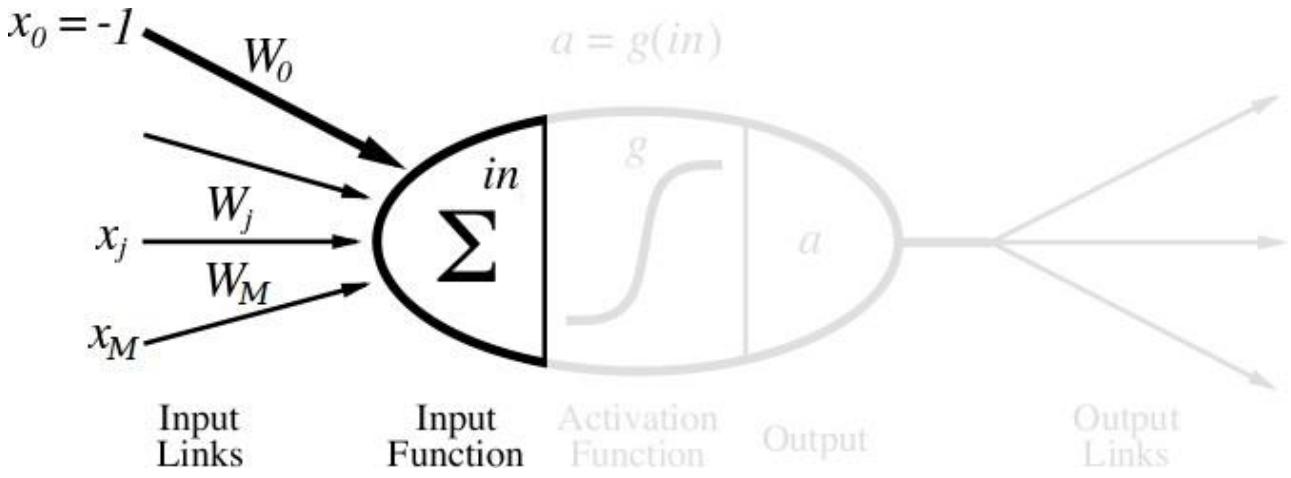
What it does in a nutshell: It fires when a **linear combination of its inputs exceeds some threshold**.

# Inputs




A neuron has $M + 1$ inputs $x_0, x_1, \ldots, x_M$ connected to it. Each link has a numeric weight $W_j$ which determines the strength of the connection. A weight of zero indicates the absence of connection.

# Inputs

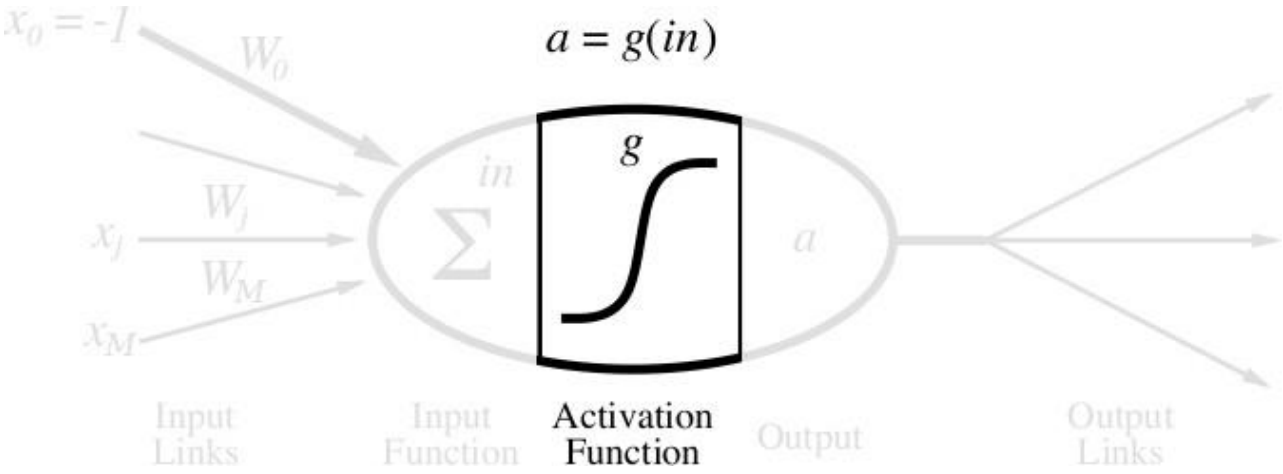



The inputs are multiplied with their corresponding connection weights and summed at the neuron:

$$in = \sum_{j=0}^{M} W_j x_j$$

Note that $x_0$ always has the value of −1. This is the **bias input** and the corresponding weight $W_0$ is the **bias weight**.

# Activation Function

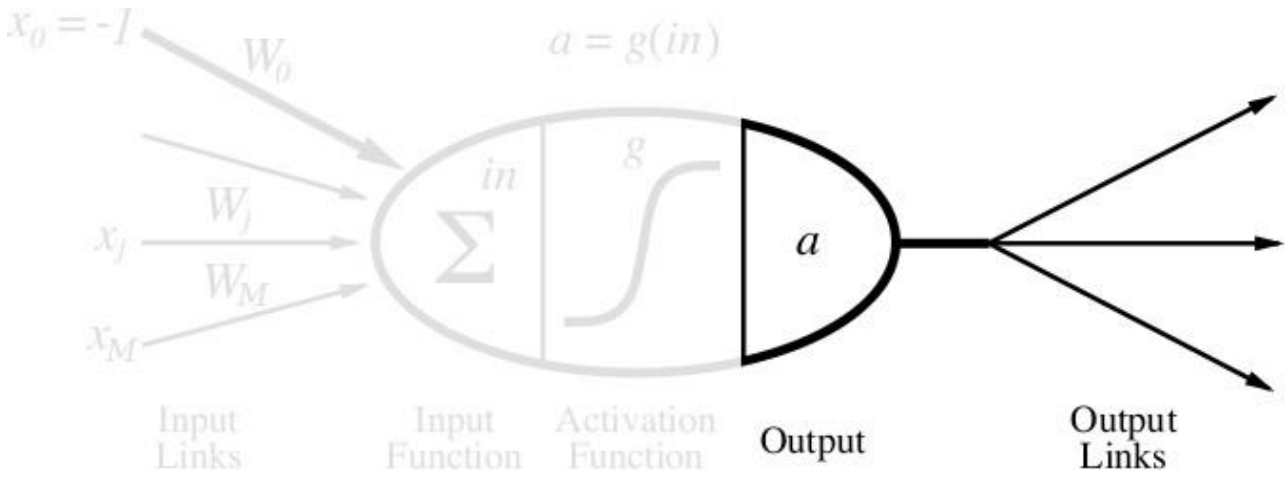



An **activation function** $g$ is then applied to the weighted-sum inputs to yield the output:

$$a = g(in) = g(\sum_{j=0}^{M} W_j x_j)$$

Example activation functions include the threshold function and the sigmoid function— More on this later.

# Output

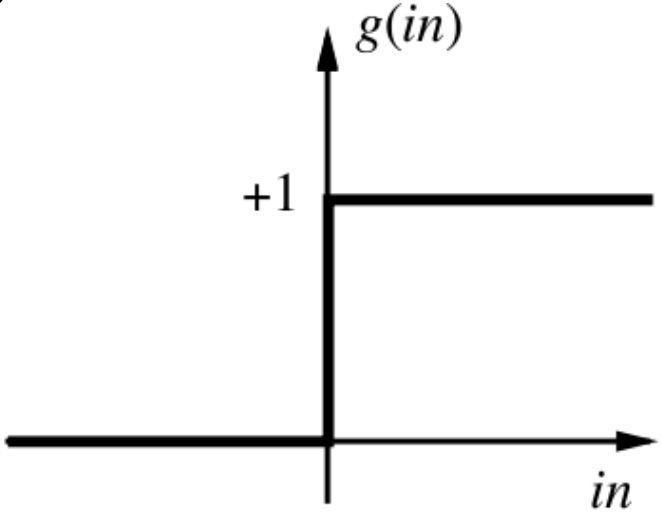


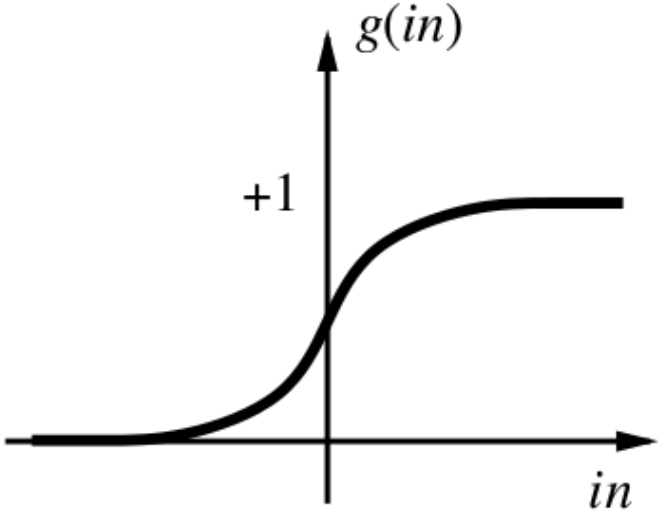

The output of the neuron can be taken as the overall output of the system or be transmitted to other neurons for further processing.

# Activation Function

The activation function **determines whether the neuron fires** given the inputs. The neuron fires when its output is close to +1, otherwise its output is close to 0.

Frequently used activation functions are the **threshold** function and the **sigmoid** function:



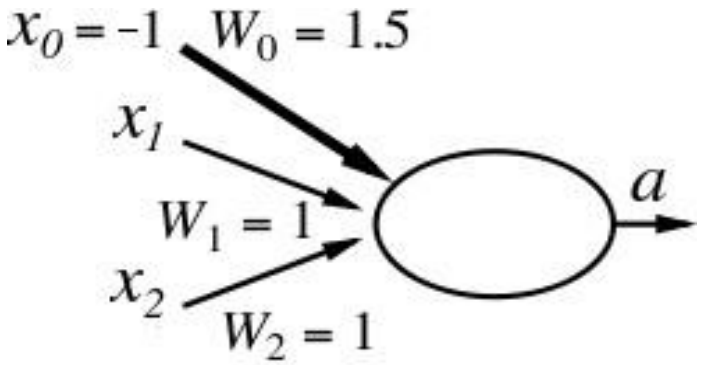




$$g_{th}(in) = \begin{cases} 1 & \text{if } in \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$g_{sig}(in) = \frac{1}{1 + e^{-in}}$$

# Activation Function (cont.)

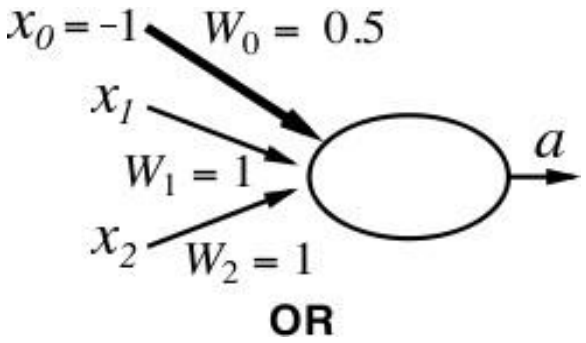
Example: Implementing Boolean functions with a neuron (of two different activation functions).




**AND**

| $x_1$ | $x_2$ | $in$ | $a = g_{th}(in)$ | $a = g_{sig}(in)$ |
|---|---|---|---|---|
| 0 | 0 | $-1.5$ | 0 | 0.1824 |
| 0 | 1 | $-0.5$ | 0 | 0.3775 |
| 1 | 0 | $-0.5$ | 0 | 0.3775 |
| 1 | 1 | 0.5 | 1 | 0.6225 |

# Activation Function (cont.)




| $x_1$ | $x_2$ | $in$ | $a = g_{th}(in)$ | $a = g_{sig}(in)$ |
|---|---|---|---|---|
| 0 | 0 | $-0.5$ | 0 | 0.3775 |
| 0 | 1 | 0.5 | 1 | 0.6225 |
| 1 | 0 | 0.5 | 1 | 0.6225 |
| 1 | 1 | 1.5 | 1 | 0.8176 |

Observe that the sigmoid function provides a **smoothed version** of the Boolean outputs, and the output can be considered as the **degree of activation** of the neuron— output close to 0 $\Longrightarrow$ weakly activated, output close to 1 $\Longrightarrow$ strongly activated.

# Activation Function (cont.)

- Rectified Linear Unit (ReLU)

$$g_{ReLU} = \max(0, in)$$

PReLU/LReLU: $\max(0, in) + a * \min(0, in)$

- Softmax function (softargmax):

$$\text{softmax}(v_i) = \frac{\exp(v_i)}{\sum_j \exp(v_j)},$$

Softmax has a temperature parameter

$$\text{for } \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix}$$

$$\text{v} = [1,2,3],$$
$$\exp(\text{v}) = [2.72.\ 7{,}39,\ 20.09],$$
$$\text{softmax}(\text{v}) = [0.09, 0.24, 0.67]$$

# Activation Function (cont.)

- Softmax function:

$$\text{softmax}(v_i) = \frac{\exp(v_i)}{\sum_j \exp(v_j)},$$

$$\text{for } \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix}$$

- Recall sigmoid function:

$$\text{sigmoid}(u) = \frac{1}{1+\exp(-u)} = \frac{\exp(0)}{\exp(0)+\exp(-u)} = \text{softmax}\left(\begin{bmatrix} 0, \\ -u \end{bmatrix}\right), \text{ or,}$$

$$\text{sigmoid}(u) = \frac{\exp(\frac{1}{2}u)}{\exp\left(\frac{1}{2}u\right)+\exp(-\frac{1}{2}u)} = \text{softmax}\left(\begin{bmatrix} \frac{1}{2}u, \\ -\frac{1}{2}u \end{bmatrix}\right)$$

# The Bias Weight

Recalling that $x_0$ is always fixed at $-1$, we can re-express the weighted-sum of inputs as

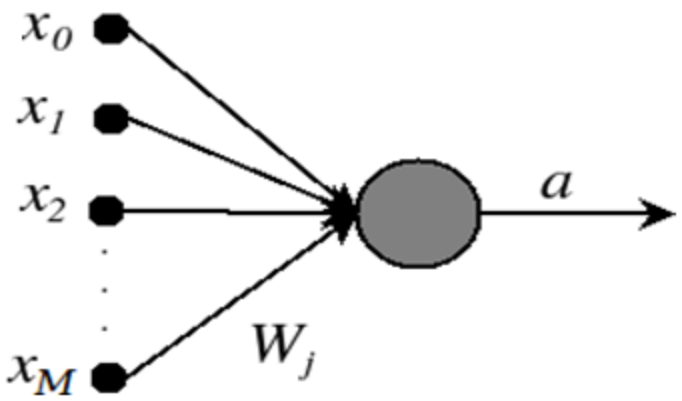
$$in = \sum_{j=0}^{M} W_j x_j = \sum_{j=1}^{M} W_j x_j - W_0$$

Observe that $in$ is positive only if $\sum_{j=1}^{M} W_j x_j$ is more than $W_0$ and negative otherwise.

Since the activation functions approach $+1$ when $in$ is positive, the bias weight $W_0$ acts as the **actual threshold** of the neuron that the weighted-sum of the other inputs $x_1, x_2, \ldots, x_M$ must **surpass in order to activate the neuron**.

# Perceptron

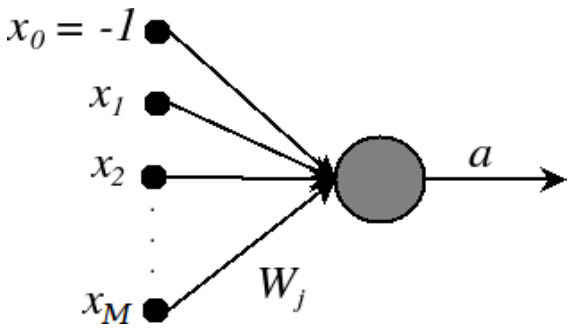
**Perceptron** (Rosenblatt, 1957): A **neural network** with only a single neuron. It is used for supervised learning.



# Perceptron Training

**Learning the weights.**

We motivate with a network with **a single neuron**:




We need a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ of input-output pairs, where

- Each $\mathbf{x}_i$ is a vector of length $M + 1$ feature values

$$\mathbf{x}_i = [x_{i,0} \; x_{i,1} \; \ldots \; x_{i,M}]^T$$

and we assume $x_{i,0}$ is always set to $-1$ (the bias input).

- $y_i$ is the target output of the $i$-th sample, where $y_i \in \{0, 1\}$ (we are trying to solve a **binary** classification problem).

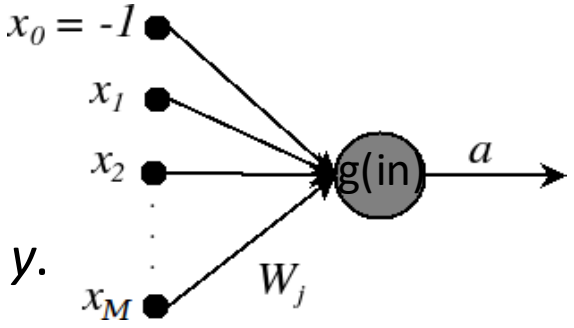
# Perceptron Training

**Perceptron Algorithm**: Supervised learning for Perceptrons- **learning the weights.**

Principle: Start with randomly initialised weights,
Then adjusting the weights to minimise error.

- How to measure the error? Squared error
- How to adjust the weight? Gradient descent

We motivate with a network with a single neuron:




Squared error

Let (**x**, *y*) be a single training sample with its **true** output *y*.

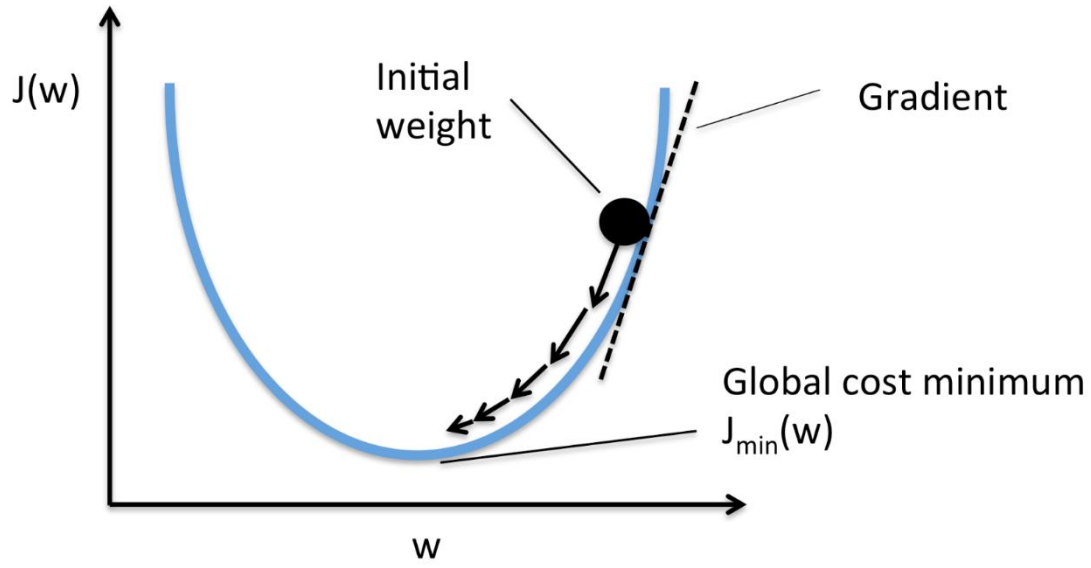
The squared error is given by:

$$E = \frac{1}{2} Err^2$$

$$= \frac{1}{2}(y - a)^2 \quad = \frac{1}{2}\left(y - g(in)\right)^2 = \frac{1}{2}\left(y - g\left(\sum_{j=0}^{M} W_j x_j\right)\right)^2$$

# Perceptron Training

- How to adjust the weight? Gradient descent
  - ✓ Minimize the loss function (i.e., error measurement)
  - ✓ The quickest weight update is along the opposite direction of gradient at the point

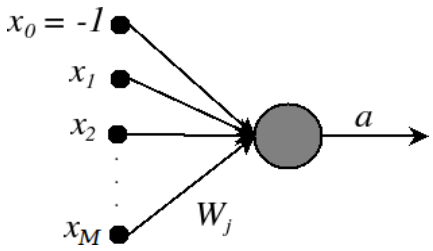


# Perceptron Training

$$E = \frac{1}{2} Err^2 = \frac{1}{2} \left( y - g\left( \sum_{j=0}^{M} W_j x_j \right) \right)^2$$

Calculating the partial derivative of the error against a particular weight, we have:

$$\begin{aligned} \frac{\partial E}{\partial W_j} &= \text{Err} \times \frac{\partial Err}{\partial W_j} \\ &= Err \times \frac{\partial}{\partial W_j} \left( y - g\left( \sum_{j=0}^{M} W_j x_j \right) \right) \\ &= -\text{Err} \times g' \left( \sum_{j=0}^{M} W_j x_j \right) \times x_j \end{aligned}$$




where *g'* is the derivative of the activation function *g*.

# Perceptron Training

- How to adjust the weight? Gradient descent
  - ✓ Minimize the loss function (i.e., error measurement)
  - ✓ The quickest weight update is along the opposite direction of gradient at the point

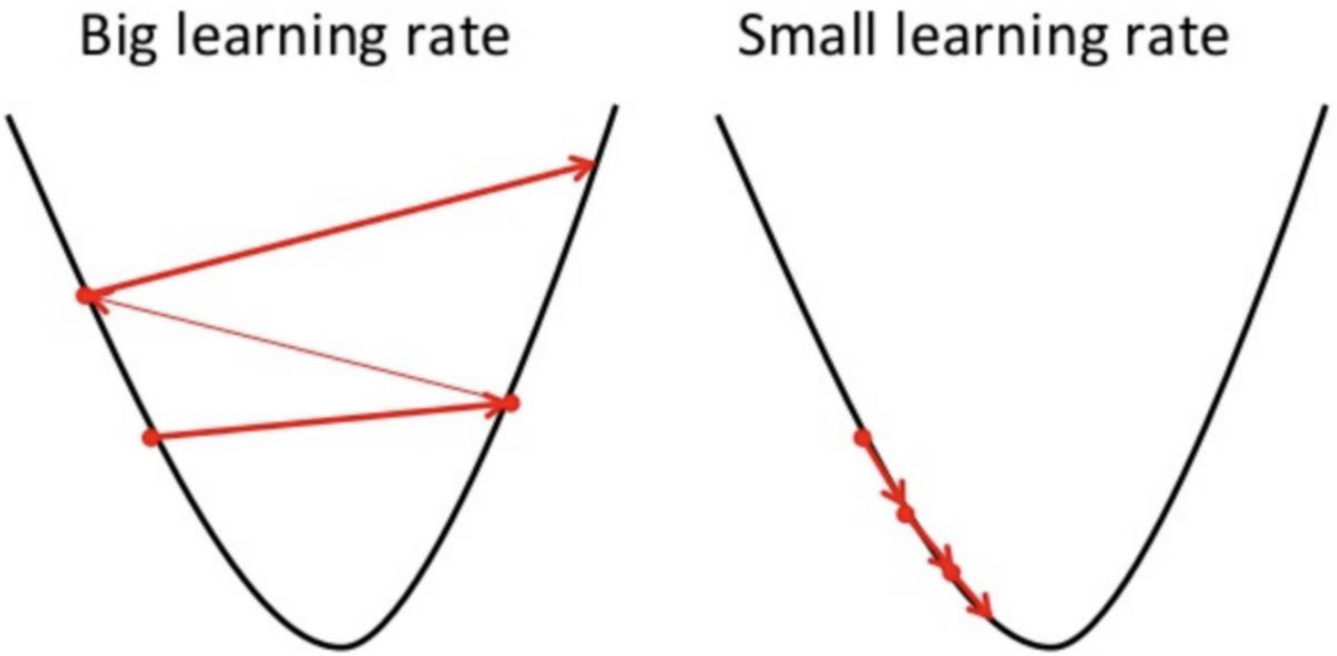
Using the gradient descent algorithm, if we want to reduce *E*, we update the weight as follows:

$$W_j \leftarrow Wj - \alpha E'$$

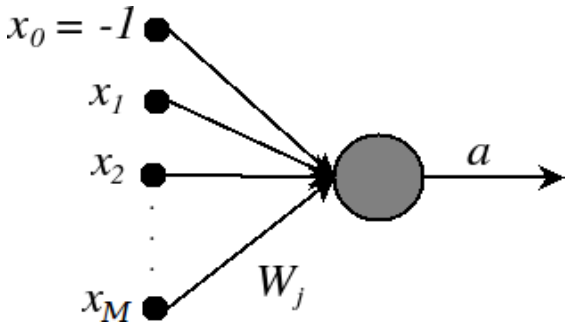
$$W_j \leftarrow Wj + \alpha * Err * g'(\sum_{j=0}^{M} W_j x_j) \times x_j$$

Where $\alpha$ is the learning rate, which controls how fast the algorithm converges. Setting the correct $\alpha$ is crucial: Too low ⟹ training takes too long. Too high ⟹ algorithm behaves erratically with significant transients.

# Learning Rate



# Perceptron Training Summary




Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ of input-output pairs,

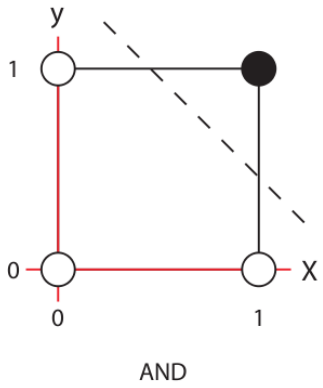
- Start with randomly initialised weights,
- Then adjusting the weights to minimise error by gradient descent:

$$W_j \leftarrow W_j + \alpha * Err * g'(\sum_{j=0}^{M} W_j x_j) \times x_j$$

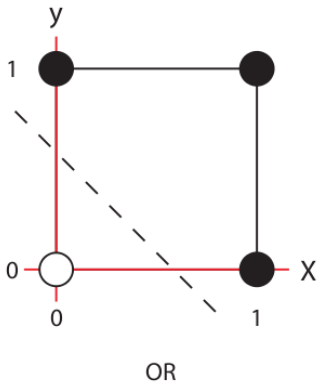
- We present the samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ **one by one** to the network, and adjust the weights $W_j$ according to how the network is currently performing with each sample.
- Each iteration through all the samples once is called an **epoch**.

# Convergence of Perceptron Algorithm

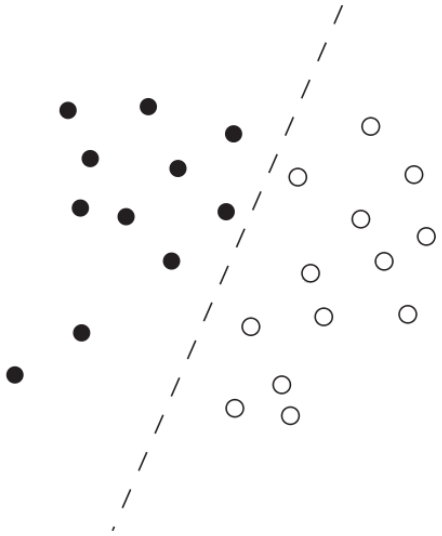
Data that can be separated by a linear hyperplane can be solved using Perceptrons. Such classification problems are called **linearly separable** problems.




AND



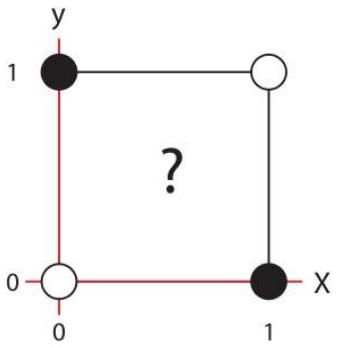

OR



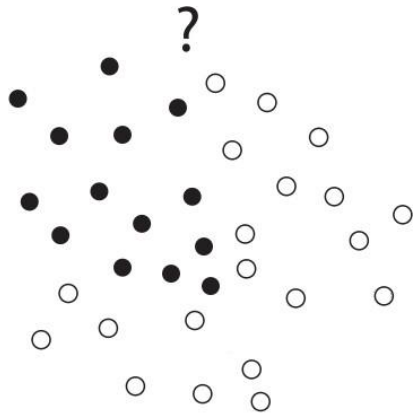
**If the problem is linearly separable, the Perceptron Algorithm is guaranteed to converge in a finite number of iterations** (Novikoff, 1962).

# Limitation of Perceptron

What if the data is not linearly separable, e.g., the simple XOR Boolean function?



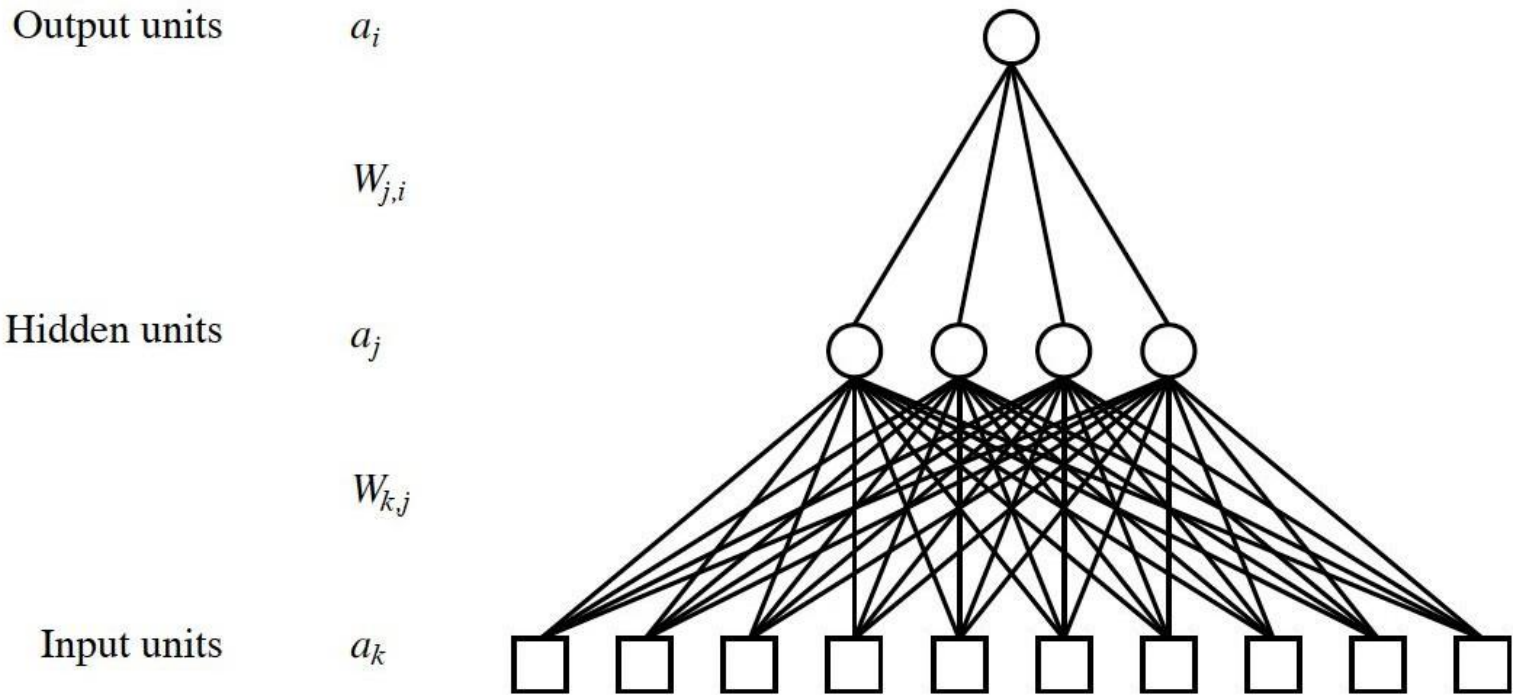

XOR




- The Perceptron Algorithm will not converge for such **linearly non-separable** problems.

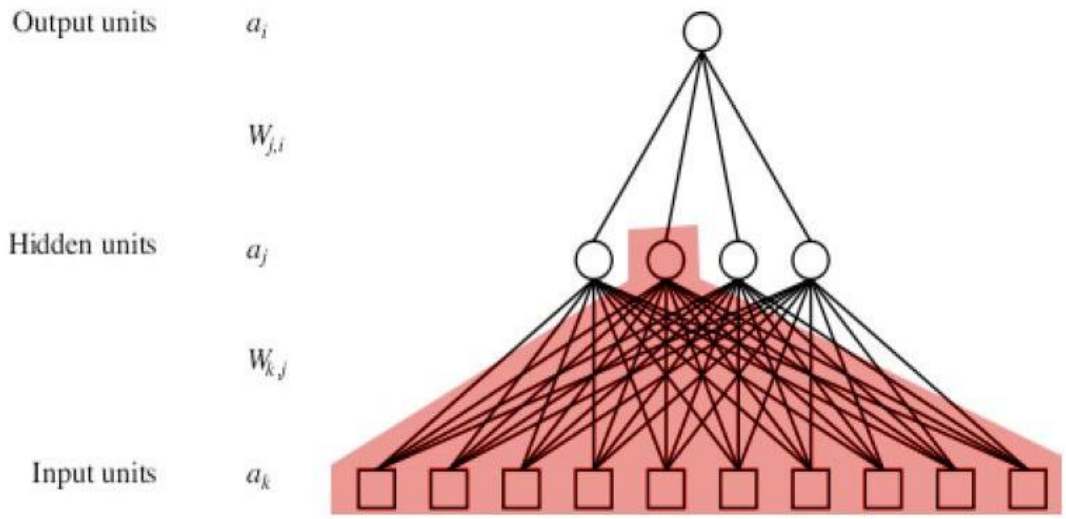
# Multi-layer Perceptron - A Non-linear Classifier

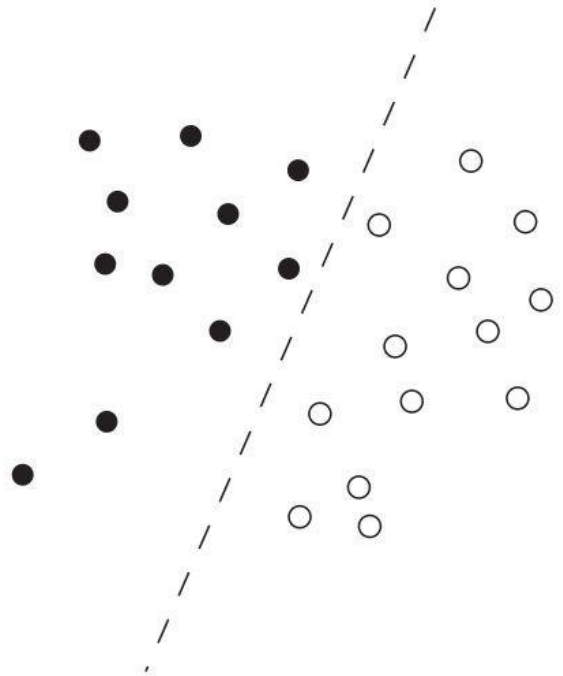
The simplest case involves a single *hidden layer*.



# Multi-layer Perceptron - A Non-linear Classifier

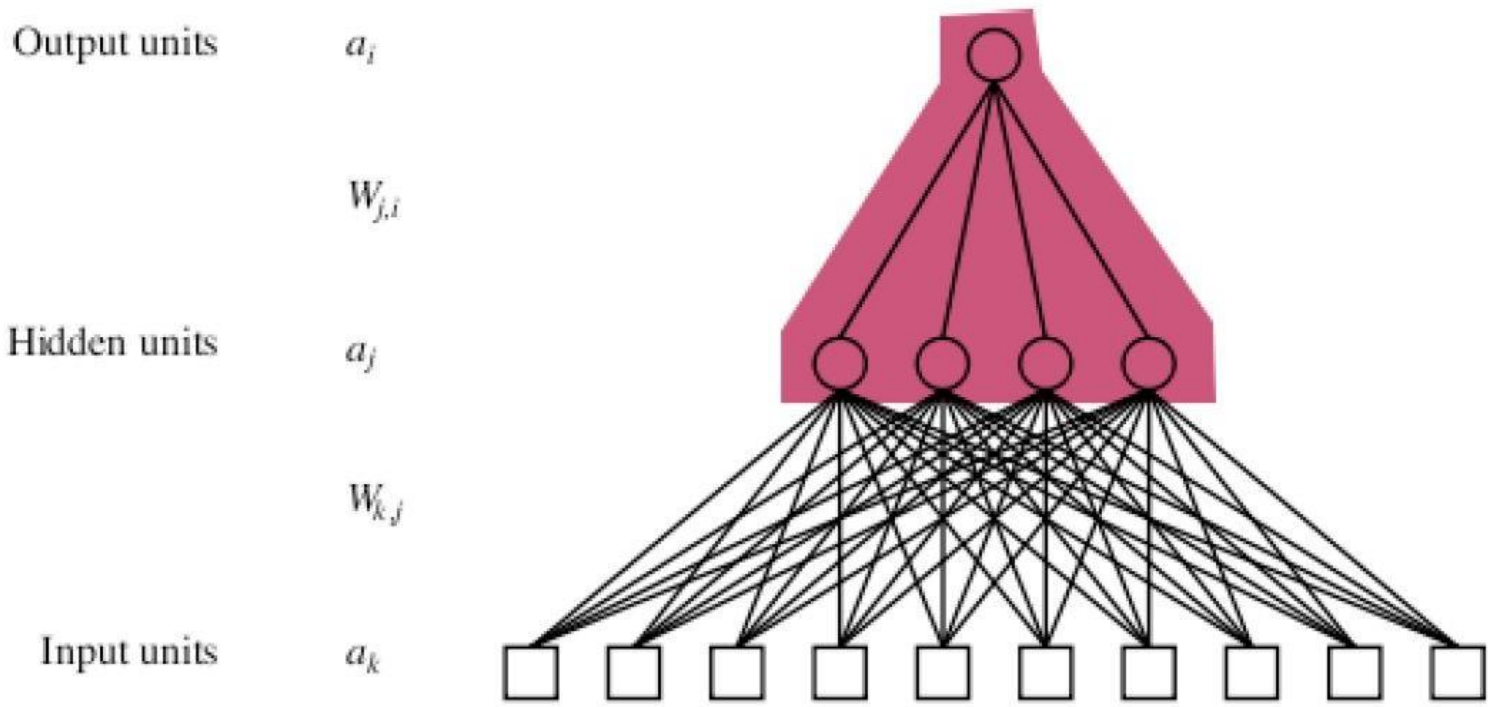
Each *hidden unit* can be considered as single output perceptron network, which is capable of separating the training examples linearly.



# Multi-layer Perceptron - A Non-linear Classifier

The output unit of the multi-layer network can then be considered a soft-thresholded linear combination of the hidden units:



# Multi-layer Perceptron - The Good and the Bad

- Universal Approximation Theorem

  *With a single, sufficient large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy.*

- Unfortunately, for any particular network, it is harder to characterize exactly which functions can be represented and which ones cannot.
- As a consequence, given a particular learning algorithm it is unknown how to choose the *right number of hidden units* in advance.
- One usually reports to cross validation, but this can be computationally expensive for large networks.

# Training MLP
# - Backpropagation

# Backpropagation
# Muti-output Multi-layer Perceptron

- We need to consider multiple output units for multi-layer networks. Let $(\mathbf{x}, \mathbf{y})$ be a single sample with its desired output labels $\mathbf{y} = \{y_1, \ldots, y_i, \ldots, y_M\}$.
- The error at the output units is just $\mathbf{y} - h_{\mathbf{W}}(\mathbf{x})$, and we can use this to adjust the weights between the hidden layer and the output layer.
- The above steps produces a term equivalent to the error at the hidden layer, i.e. the error at the output layer is back-propagated to the hidden later.
- This is subsequently used to update the weights between the input units and the hidden layer.

# Backpropagation
# Muti-output Multi-layer Perceptron

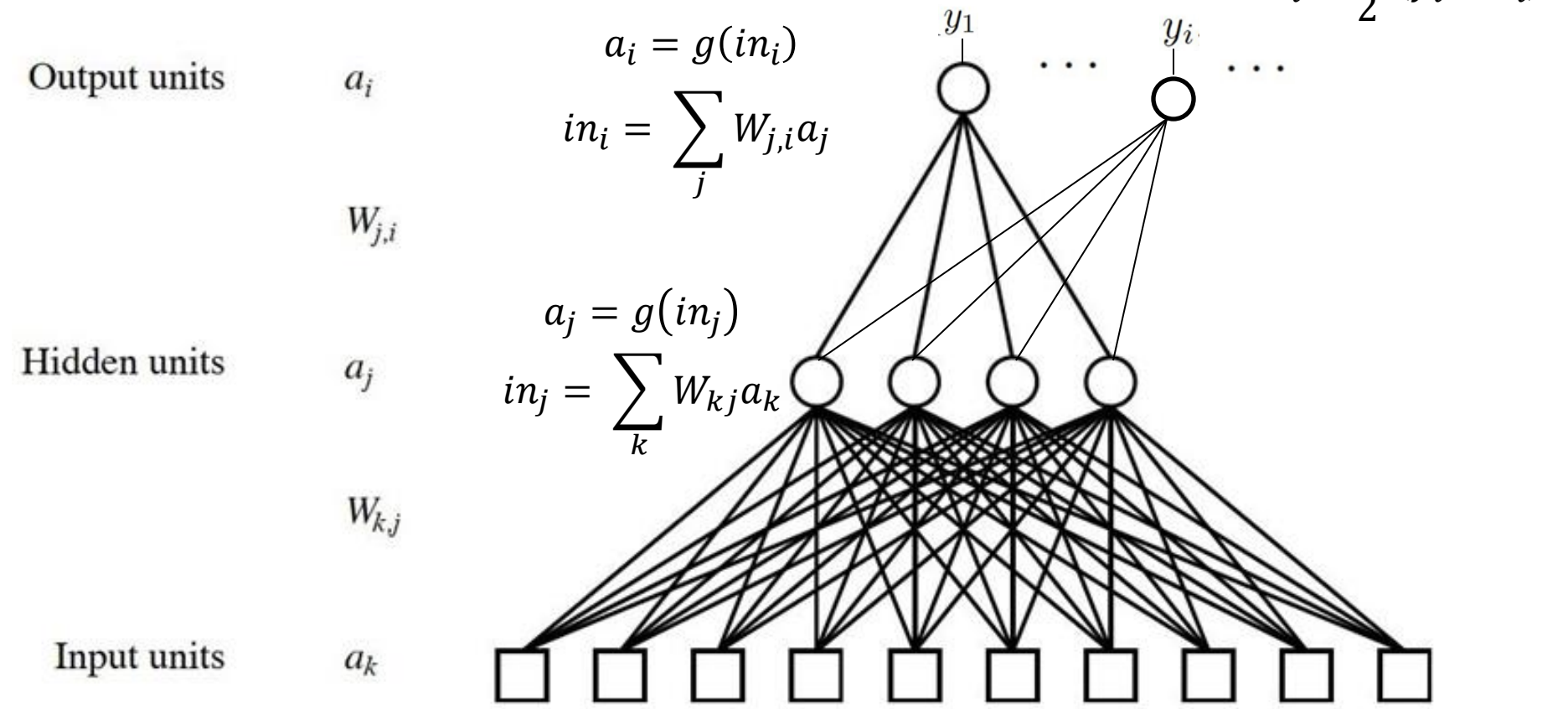
$$Err_i = \frac{1}{2}(y_i - a_i)^2$$

$$a_i = g(in_i)$$

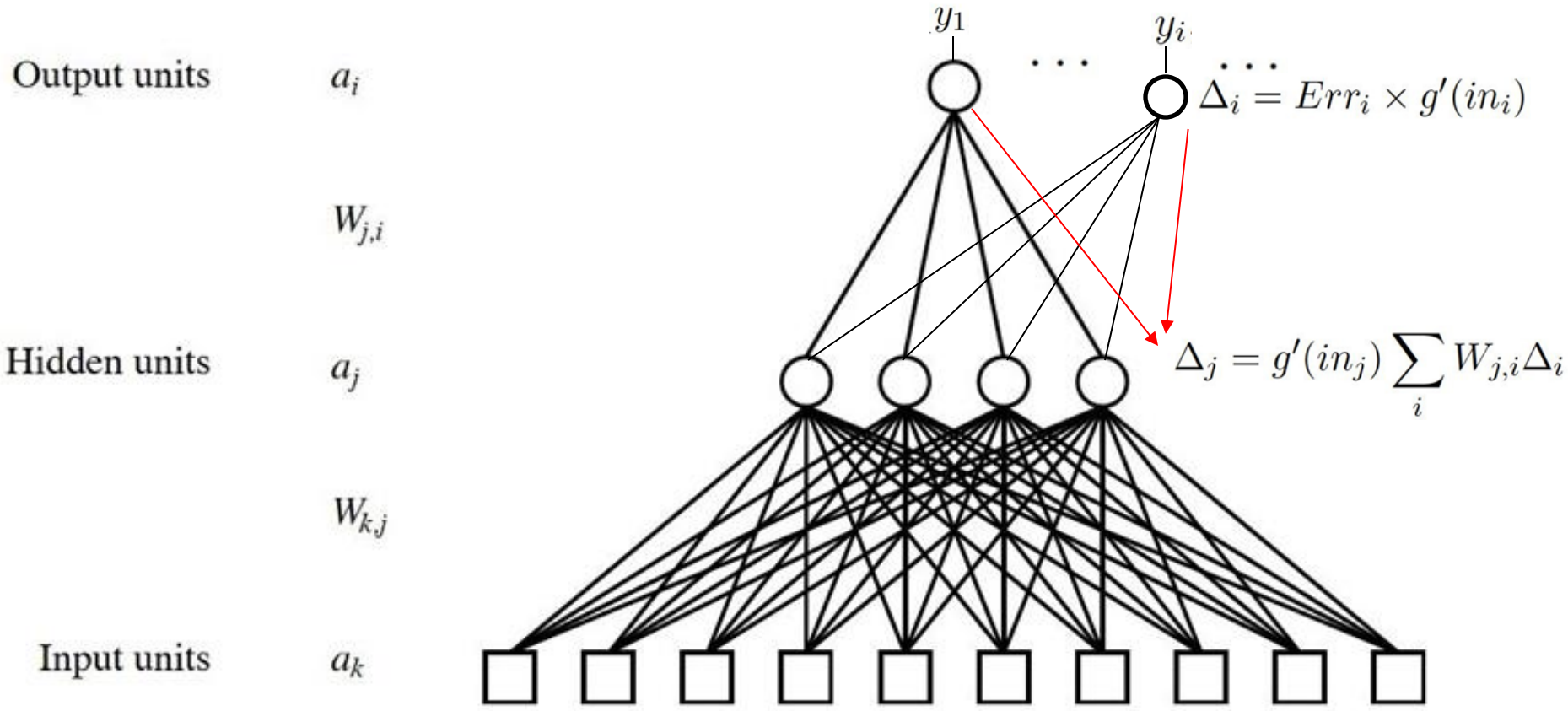
$$in_i = \sum_j W_{j,i} a_j$$

$$a_j = g(in_j)$$

$$in_j = \sum_k W_{kj} a_k$$



# Backpropagation
# Muti-output Multi-layer Perceptron



# Backpropagation

**Step 1: Update the weights between the hidden and output layers.**

- Let $Err_i$ be the $i$-th component of the error vector $\mathbf{y} - h_{\mathbf{W}}(\mathbf{x})$.
- Define $\Delta_i = Err_i \times g'(in_i)$.
- The weight-update rule becomes

$$W_{j,i} \longleftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

This is similar to weight-updates for Perceptrons!

# Backpropagation

Step 2: Back-propagate the error to the hidden layer.

- The idea is that the hidden node $j$ is "responsible" for some fraction of the error $\Delta_i$ in each of the output nodes to which it connects.
- Thus the $\Delta_i$ values are divided according to the strength (weight) of the connection between the hidden node and the output node:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$

# Backpropagation

Step 3: Update the weights between the input units and the hidden layer.

- Again, this is similar to weight-updates in Perceptrons:

$$W_{k,j} \longrightarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

# Backpropagation

For the general case of *multiple hidden* layers:

1. Compute the $\Delta$ values for the output units, using the observed error.
2. Starting with the output layer, repeat the following for each layer in the network, until the earliest hidden later is reached:
   - Propagate the $\Delta$ values back to the previous layer.
   - Update the weights between the two layers.
3. Repeat Steps 1 to 2 for all training samples.

# Backpropagation

**function** BACK-PROP-LEARNING($examples$, $network$) **returns** a neural network
**inputs**: $examples$, a set of examples, each with input vector **x** and output vector **y**
$network$, a multilayer network with $M$ layers, weights $W_{j,i}$, activation function $g$

**repeat**
**for each** $e$ **in** $examples$ **do**
**for each** node $j$ in the input layer **do** $a_j \leftarrow x_j[e]$ Initialize
**for** $\ell = 2$ **to** $M$ **do** forward
$in_i \leftarrow \sum_j W_{j,i}\, a_j$
$a_i \leftarrow g(in_i)$
**for each** node $i$ in the output layer **do** backward
$\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
**for** $\ell = M - 1$ **to** 1 **do**
**for each** node $j$ in layer $\ell$ **do**
$\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i}\, \Delta_i$
**for each** node $i$ in layer $\ell + 1$ **do**
$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
**until** some stopping criterion is satisfied
**return** NEURAL-NET-HYPOTHESIS($network$)

# Backpropagation

- Chain rule:

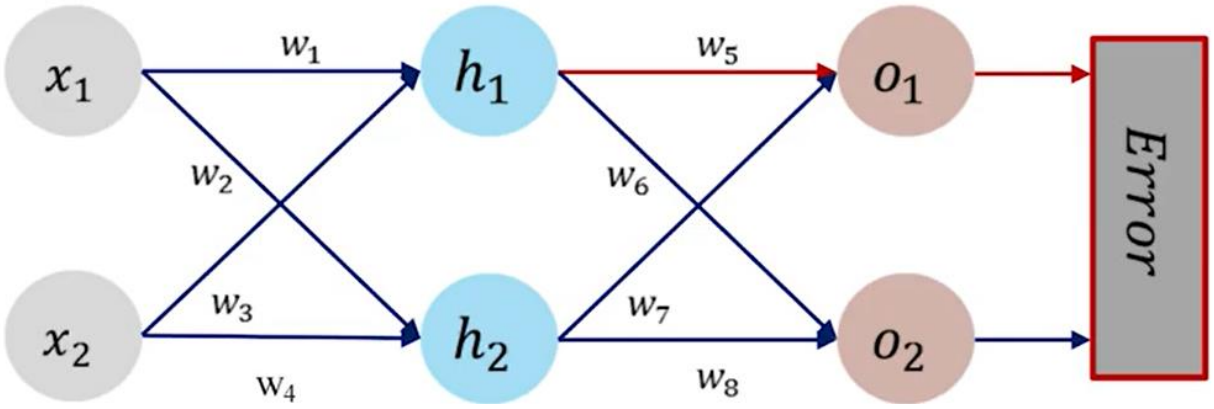
Given $f(x) = u(v(x))$, the derivative of $f(x)$ respect to x is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

Given $f(x) = u(v(w(x)))$, the derivative of $f(x)$ respect to x is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

# An Example




$$In_{h_1} = w_1 * x_1 + w_3 * x_2$$

$$h_1 = Out_{h_1} = Sigmoid(In_{h_1})$$

$$In_{o_1} = w_5 * h_1 + w_7 * h_2$$

$$o_1 = Out_{o_1} = Sigmoid(In_{o_1})$$

$$In_{h_2} = w_2 * x_1 + w_4 * x_2$$

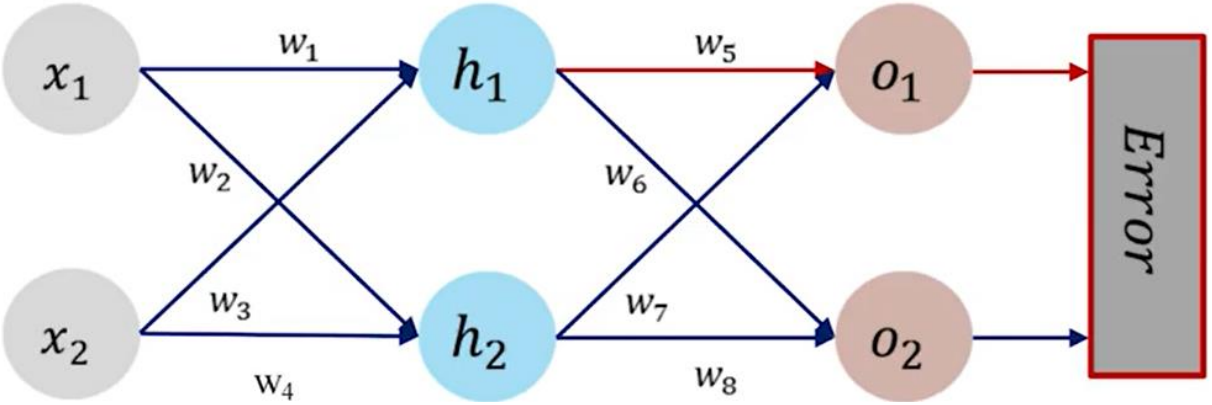
$$h_2 = Out_{h_2} = Sigmoid(In_{h_2})$$

$$In_{o_2} = w_6 * h_1 + w_8 * h_2$$

$$o_2 = Out_{o_2} = Sigmoid(In_{o_2})$$

$$Error = \frac{1}{2}\sum_{i=1}^{2}(o_i - y_i)^2$$

# An Example




$$\frac{\partial \text{ Error}}{\partial w_5} = \frac{\partial \text{ Error}}{\partial o_1} * \frac{\partial o_1}{\partial \text{In } o_{o_1}} * \frac{\partial \text{In } o_{o_1}}{\partial w_5}$$

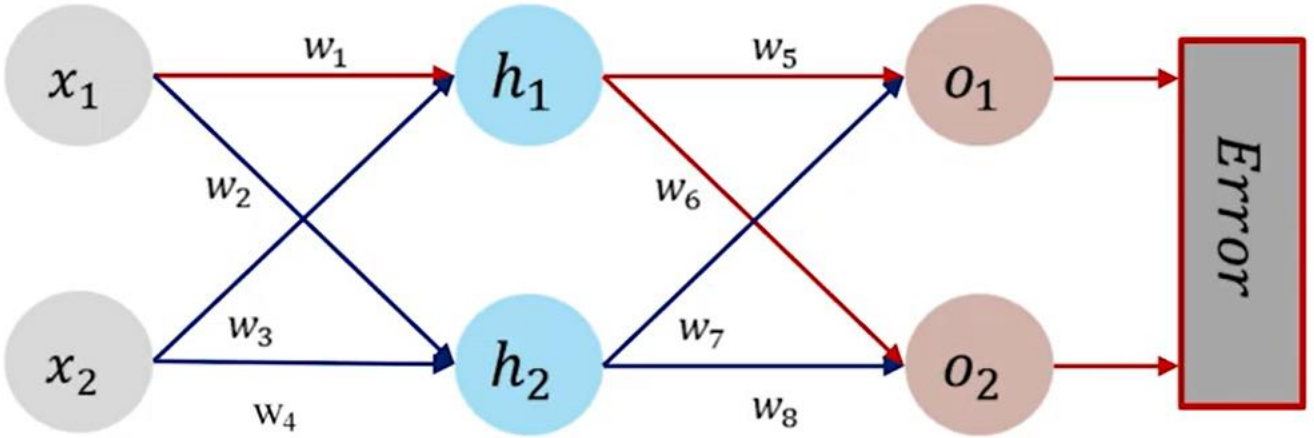
$$\triangle_5 = \frac{\partial \text{ Error}}{\partial o_1} * \frac{\partial o_1}{\partial In_{o_1}}$$

$$\frac{\partial \text{ Error}}{\partial o_1} = o_1 - y_1 \quad \Leftarrow \quad \text{Error} = \frac{1}{2}\sum_{i=1}^{2}(o_i - y_i)^2$$

$$\frac{\partial o_1}{\partial In_{o_1}} = In_{o_1} * (1 - In_{o_1}) \quad \Leftarrow \quad o_1 = \text{Sigmoid}\,(\text{In}_{o_1})$$

$$\frac{\partial In_{o_1}}{\partial w_5} = h_1 \quad \Leftarrow \quad In_{o_1} = w_5 * h_1 + w_7 * h_2$$

# An Example



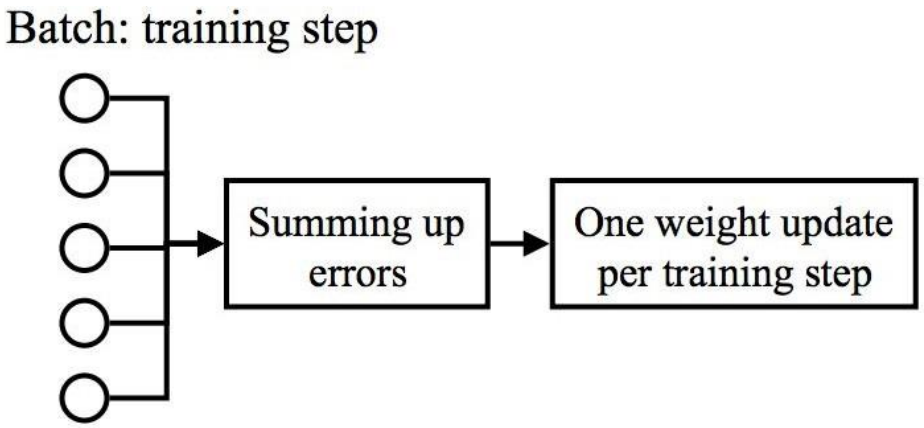

$$\frac{\partial \text{ Error}}{\partial w_1} = \frac{\partial \text{ Error}}{\partial o_1} * \frac{\partial o_1}{\partial w_1} + \frac{\partial Error}{\partial o_2} * \frac{\partial o_2}{\partial w_1}$$

$$= \frac{\partial \text{ Error}}{\partial o_1} * \frac{\partial o_1}{\partial \text{In}_{o_1}} * \frac{\partial \text{In}_{o_1}}{\partial h_1} * \frac{\partial h_1}{\partial \text{In}_{h_1}} * \frac{\partial \text{In}_{h_1}}{\partial w_1} + \frac{\partial \text{ Error}}{\partial o_2} * \frac{\partial o_2}{\partial \text{In}_{o_2}} * \frac{\partial \text{In}_{o_2}}{\partial h_1} * \frac{\partial h_1}{\partial \text{In}_{h_1}} * \frac{\partial \text{In}_{h_1}}{\partial w_1}$$

$$= \left( \frac{\partial \text{ Error}}{\partial o_1} * \frac{\partial o_1}{\partial In_{o_1}} * \frac{\partial In_{o_1}}{\partial h_1} + \frac{\partial Error}{\partial o_2} * \frac{\partial o_2}{\partial In_{o_2}} * \frac{\partial In_{o_2}}{\partial h_1} \right) * \frac{\partial h_1}{\partial In_{h_1}} * \frac{\partial \text{In}_{h_1}}{\partial w_1}$$

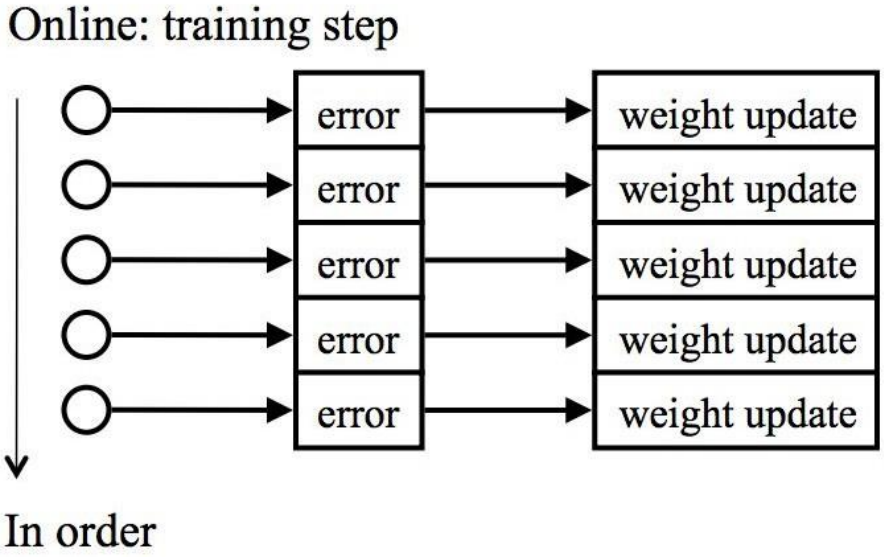
$$= (\triangle_5 * w_5 + \triangle_6 * w_6) * \frac{\partial h_1}{\partial In_{h_1}} * \frac{\partial In_{h_1}}{\partial w_1}$$

# Backpropagation with SGD




**Gradient Decent**

**Batch**: training over all given samples once.

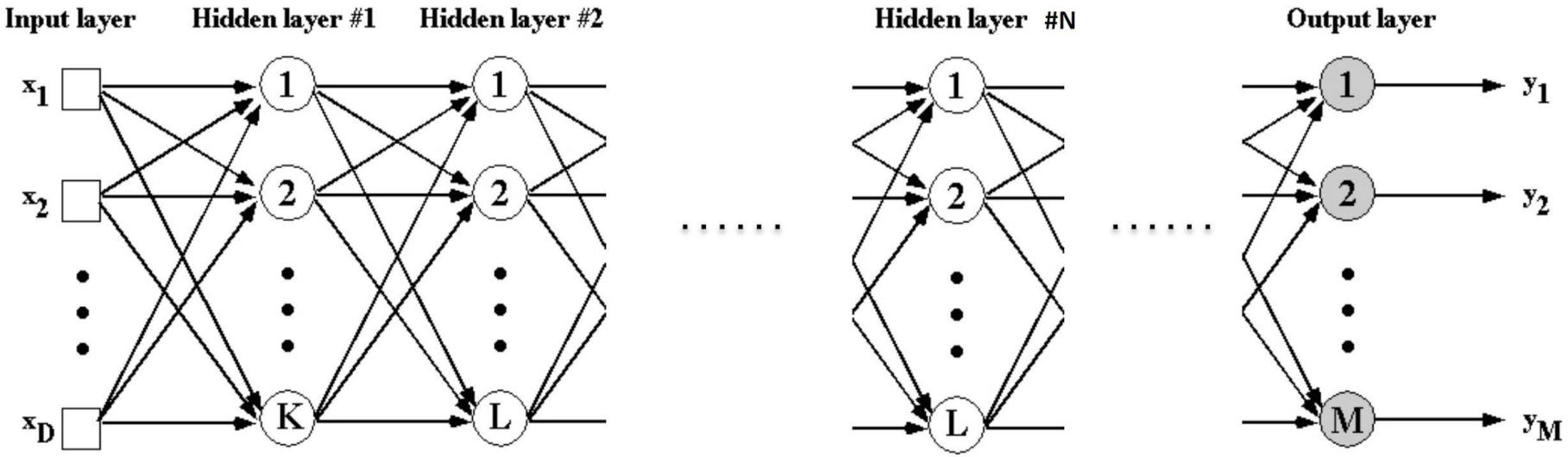



**Stochastic Gradient Decent**

- Randomly choose *m* samples
- Compute the gradients
- Do backpropogation
- Repeat

# Deep Multi-layer Perceptron

- We can learn anything !!!
- More than one hidden layer ⇒ **deep.**
- Higher level representations ⇒ Better at high-level tasks.
- Visual Classification / Speech Recoginition / Scene understanding / Visual question answering ….



# Not so fast... ☹

- Backpropagation [late 80s, early 90s]
  - Goal was to train nets with large number of layers, so that features could be learned directly from input data, but it didn't quite work
  - Notable exception: convolutional neural net by Y. LeCun (large # layers, but small # parameters)

- Issues with MLP training via backpropagation
  - Very slow convergence, particularly in large nets and large databases
  - Slow computers of the 80s and 90s
  - Local minima (how to initialize SGD)
  - Net structure (cross validation)
  - Overfitting

# MLP Based Deep Learning

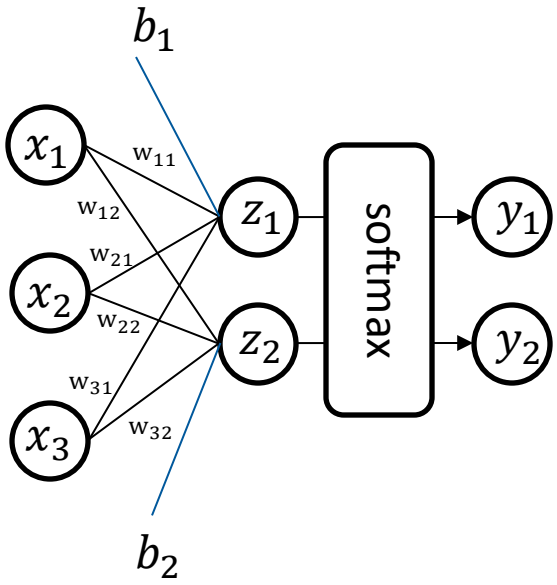
- Unmanageable number of parameters
  - For example, let's take MNIST problem with 28*28=784 input images
  - 1 hidden layer with 1000 nodes and an output layer with 10 nodes: #weights = 784*1000+1000*10=79400 weights
  - 2 hidden layers with 1000 nodes and an output layer with 10 nodes: #weights = 784*1000+1000*1000+100*10=1794000weights
- Gradient descent didn't work beyond a couple of hidden layers
  - Magnitude kept reducing as the gradient flowed back to the input layer (vanishing gradient problem [Hochreiter91])
  - Convergence issues

# MLP Based Deep Learning

- Unmanageable number of parameters
  - For example, let's take MNIST problem with 28*28=784 input images
  - 1 hidden layer with 1000 nodes and an output layer with 10 nodes: #weights = 784*1000+1000*10=79400 weights
  - 2 hidden layers with 1000 nodes and an output layer with 10 nodes: #weights = 784*1000+1000*1000+100*10=1794000weights

- Gradient descent didn't work beyond a couple of hidden layers
  - Magnitude kept reducing as the gradient flowed back to the input layer (vanishing gradient problem [Hochreiter91])
  - Convergence issues

# Softmax activated network



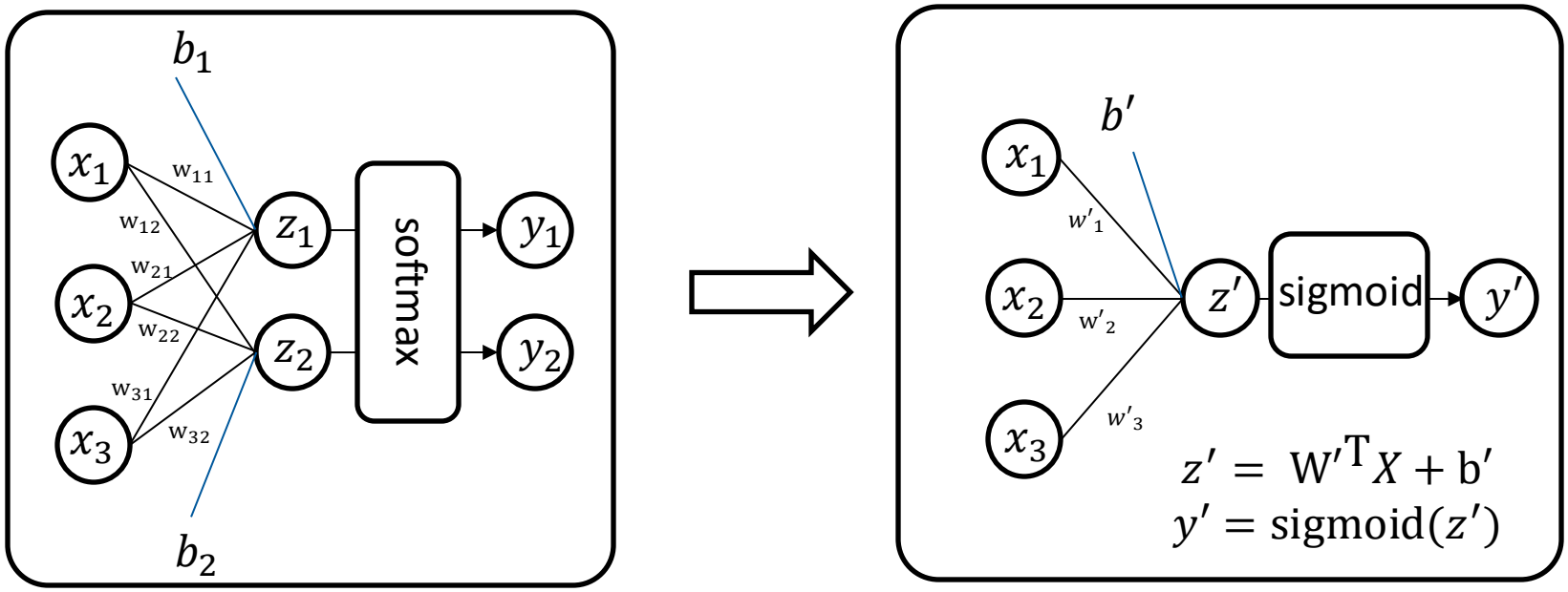

$$\mathrm{X} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \mathrm{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \mathrm{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$Z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \; Y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

$$Z = \mathrm{W}^{\mathrm{T}} X + \mathrm{b}$$
$$Y = \mathrm{softmax}(Z)$$

# Binary softmax->sigmoid activated network

Q: Can the previous binary softmax network be transformed into a sigmoid network?




$$\text{softmax}(z_1) = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2)} = \frac{1}{1 + \exp(z_2 - z_1)} = \text{sigmoid}(z_1 - z_2),$$

$z' = z_1 - z_2$; $x_1, x_2, x_3$ are the same

Then W′= ? b′ = ?