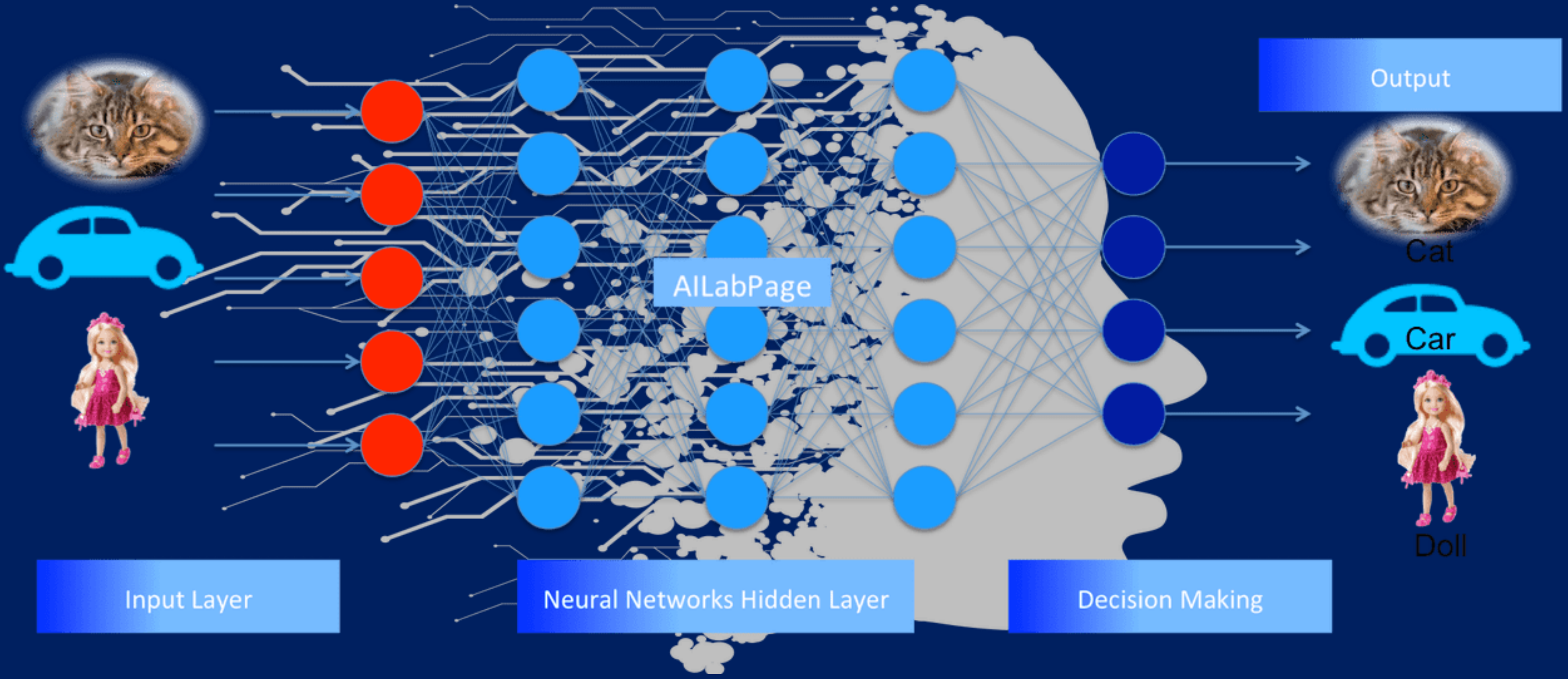
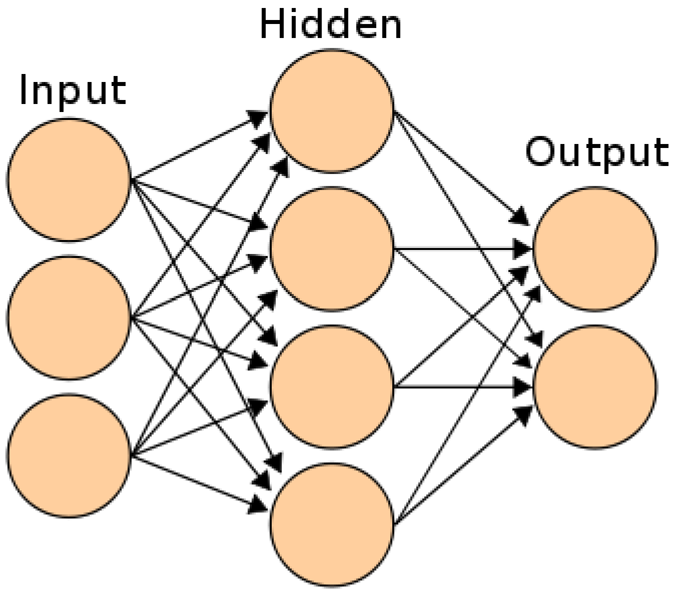
BCSE0106: Machine Learning And Its Applications



Artificial Neural Network

Background and Motivation

- Growth has been explosive since 1987
  - Education institutions, industry, military
  - 500 books on subject
  - 20 journals dedicated to ANNs
  - numerous popular, industry, academic articles
- Truly inter-disciplinary area of study



History of Artificial Neural Networks

- **Creation:**

  1890: William James - defined a neuronal process of learning

- **Promising Technology:**

  1943: McCulloch and Pitts - earliest mathematical models

  1954: Donald Hebb and IBM research group - earliest simulations

  1958: Frank Rosenblatt - The Perceptron

- **Disenchantment:**

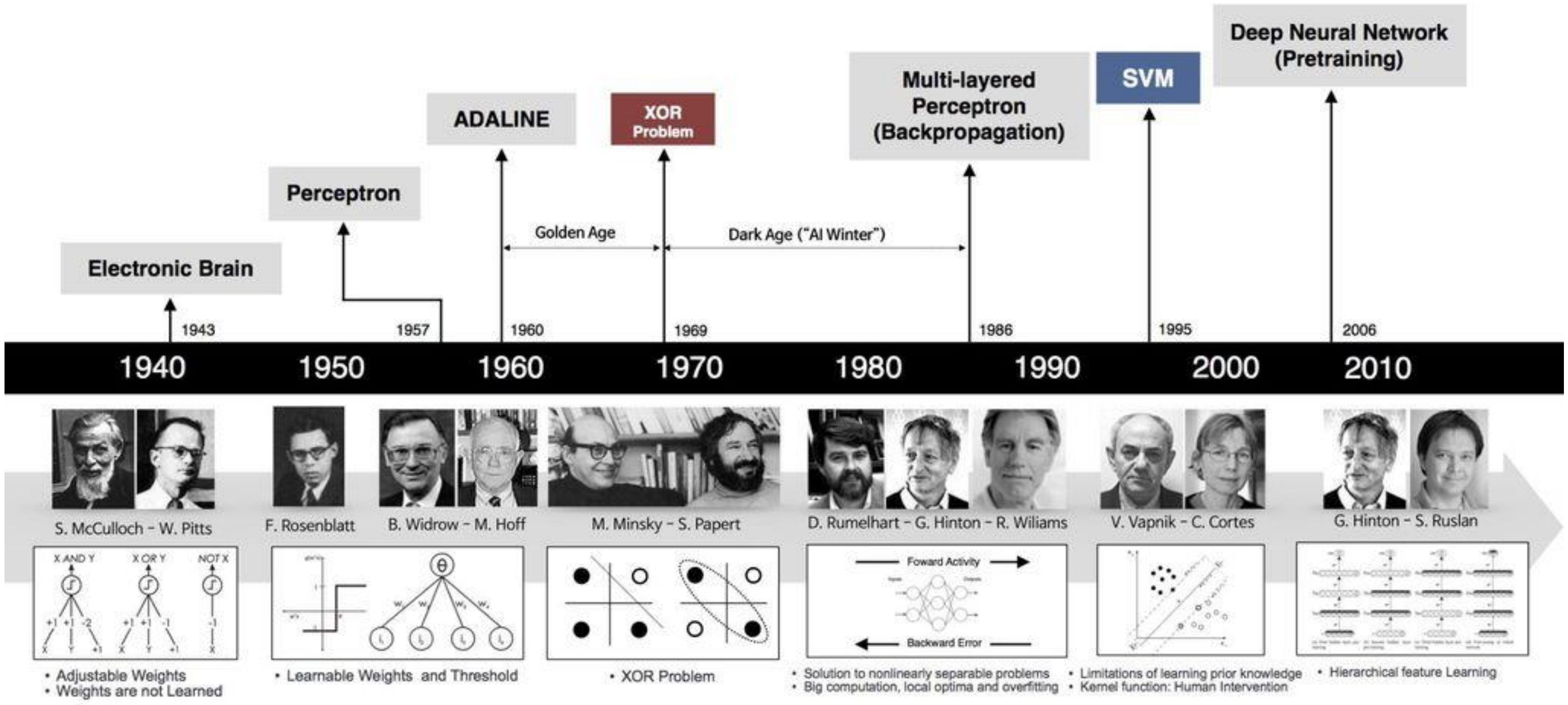
  1969: Minsky and Papert - perceptrons have severe limitations

- **Re-emergence:**

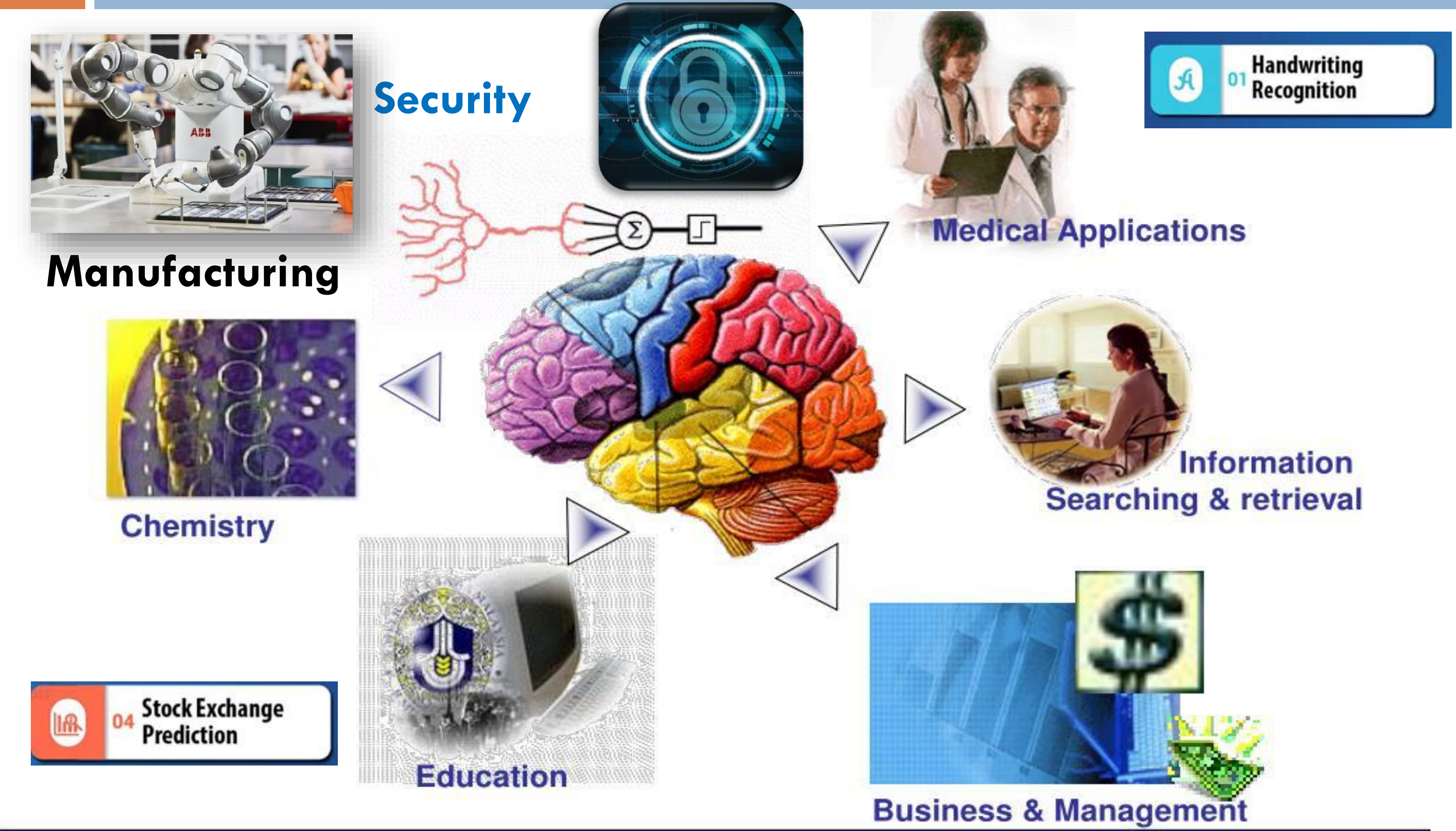
  1985: Multi-layer nets that use back-propagation

  1986: PDP Research Group - multi-disciplined approach

History of Artificial Neural Networks …


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

ANN Application Areas



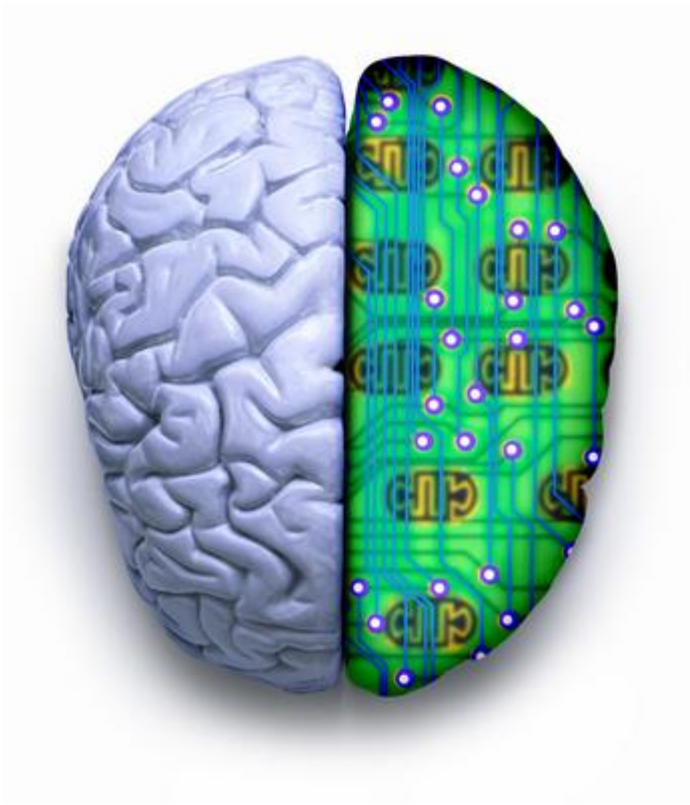
Computers and the Brain: A Contrast

Arithmetic:

1 brain = 1/10 pocket calculator

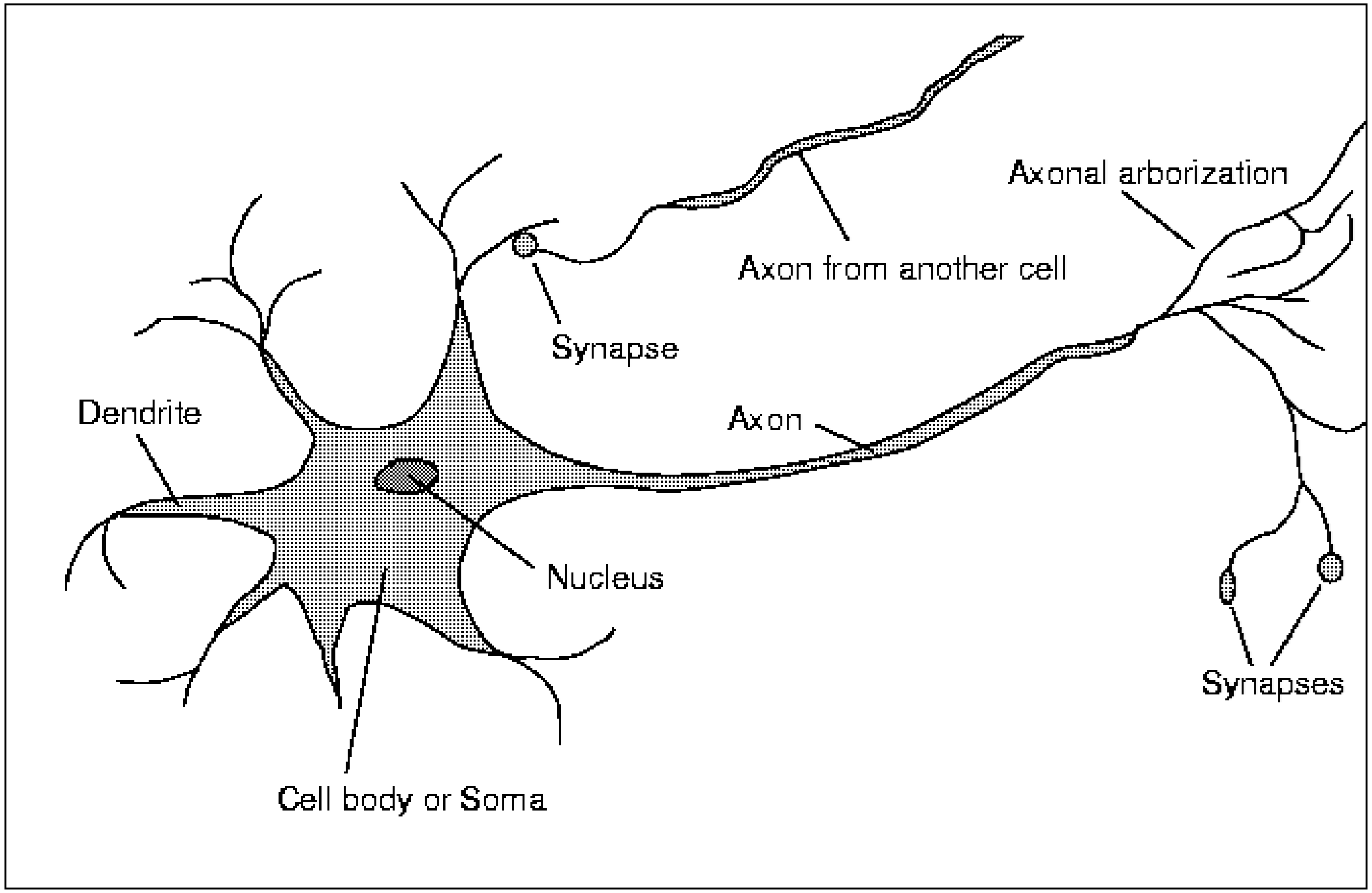
Vision:

- 1 brain = 1000 super computers
- Memory of arbitrary details:
  computer wins
- Memory of real-world facts:
  brain wins
- A computer must be programmed explicitly
- The brain can learn by experiencing the world



Biological Neurons

- The human brain is made up of billions of simple processing units – neurons.

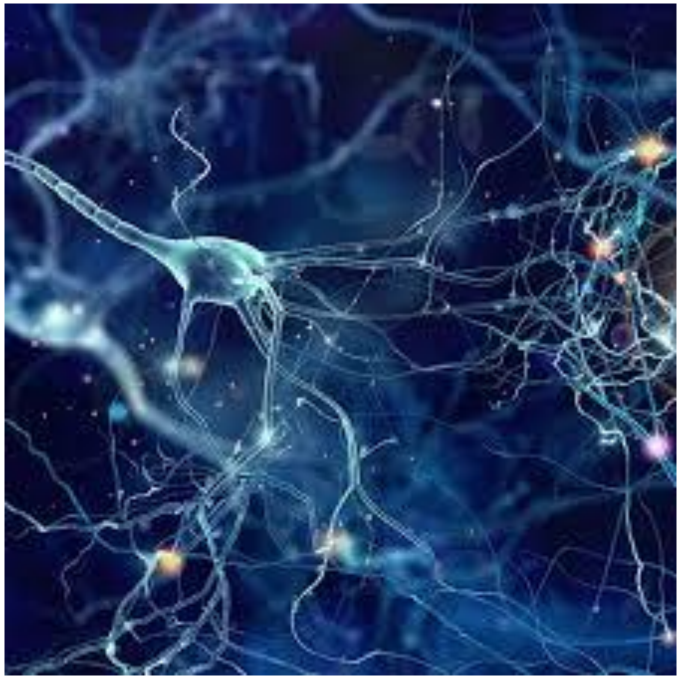


Biological Neurons …

The Neuron - A Biological Information Processor

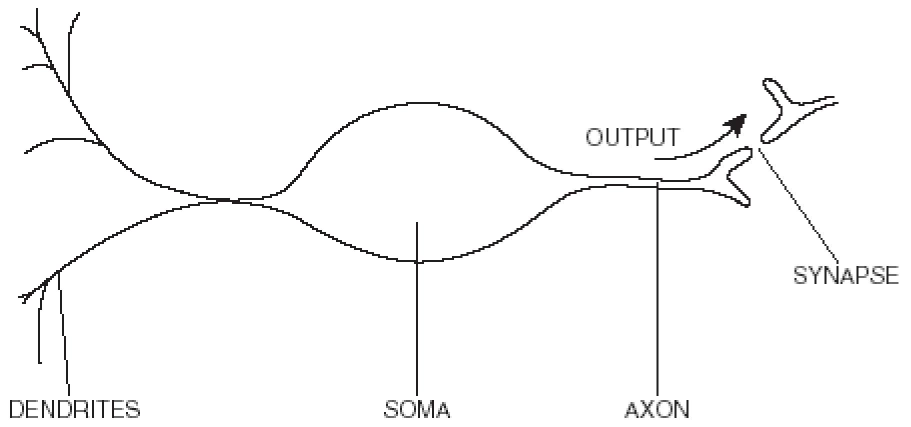
- *dentrites* - the receivers
- *soma* - neuron cell body (sums input signals)
- *axon* - the transmitter
- *synapse* - point of transmission

Neuron activates after a certain *threshold* is met Learning occurs via electro-chemical changes in effectiveness of *synaptic junction*.

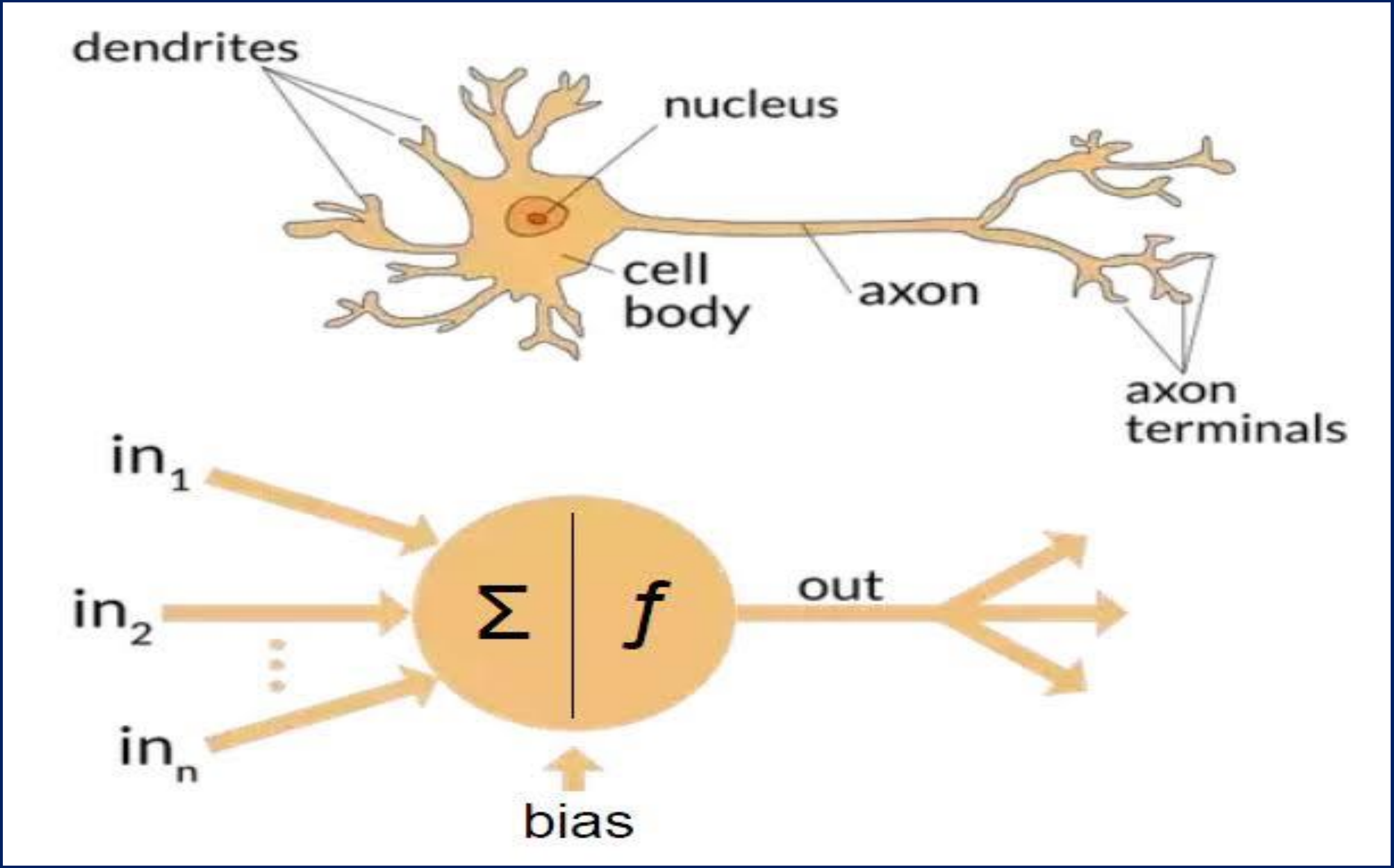


Neuron Model

- Neuron collects signals from *dendrites*
- Sends out spikes of electrical activity through an *axon*, which splits into thousands of branches.
- At end of each branch, a *synapses* converts activity into either exciting or inhibiting activity of a dendrite at another neuron.
- Neuron *fires* when exciting activity surpasses inhibitory activity
- Learning changes the effectiveness of the synapses



Artificial Neural Network



From Biological to Artificial Neurons

An Artificial Neuron - The Perceptron …

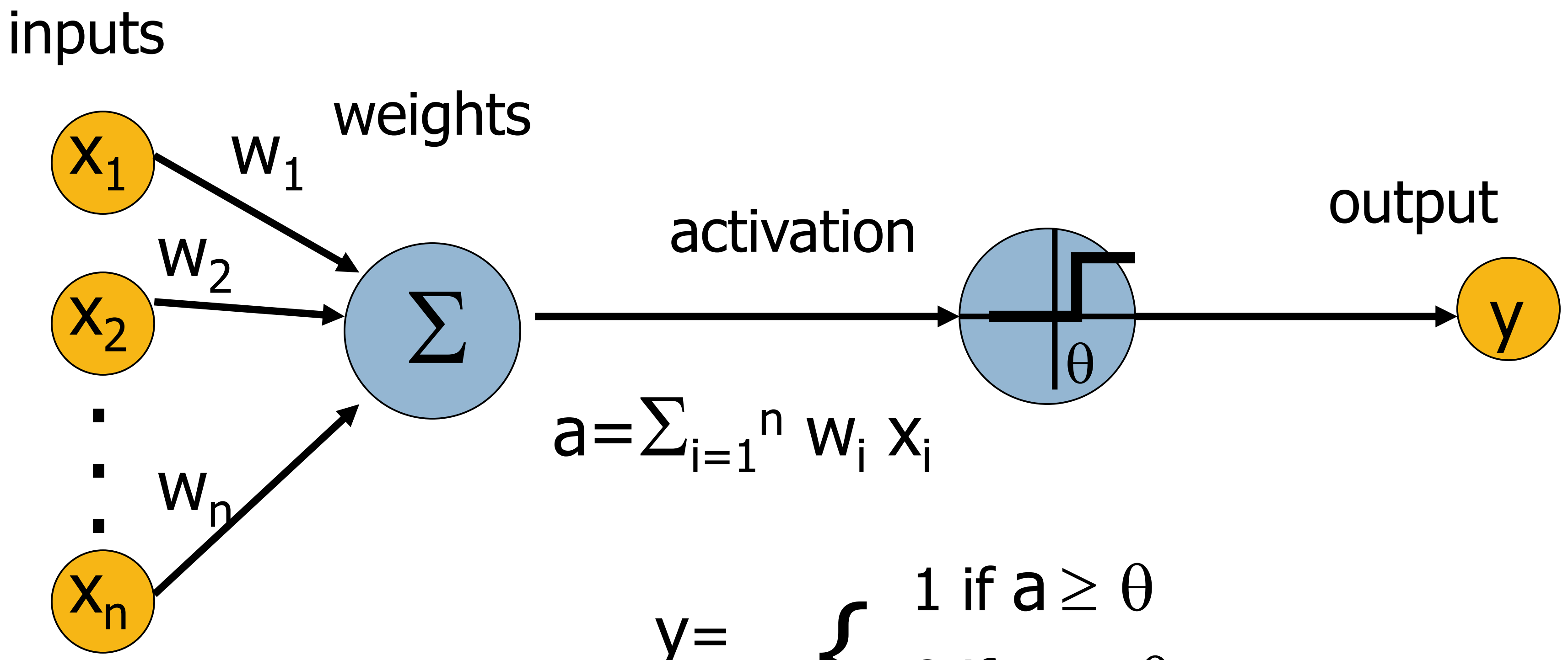
- Basic function of neuron is to sum inputs, and produce output given sum is greater than threshold
- ANN node produces an output as follows:

1. Multiplies each component of the input pattern by the weight of its connection
2. Sums all weighted inputs and subtracts the threshold value => *total weighted input*
3. Transforms the total weighted input into the output using the activation function

An Artificial Neuron - The Perceptron …

- The **Perceptron** is one of the simplest ANN architectures, **invented in 1957 by Frank Rosenblatt**
- A perceptron is a single neuron that classifies a set of inputs into one of two categories (usually 1 or -1).
- The perceptron usually uses a step function, which returns 1 if the weighted sum of inputs exceeds a threshold, and 0 otherwise.

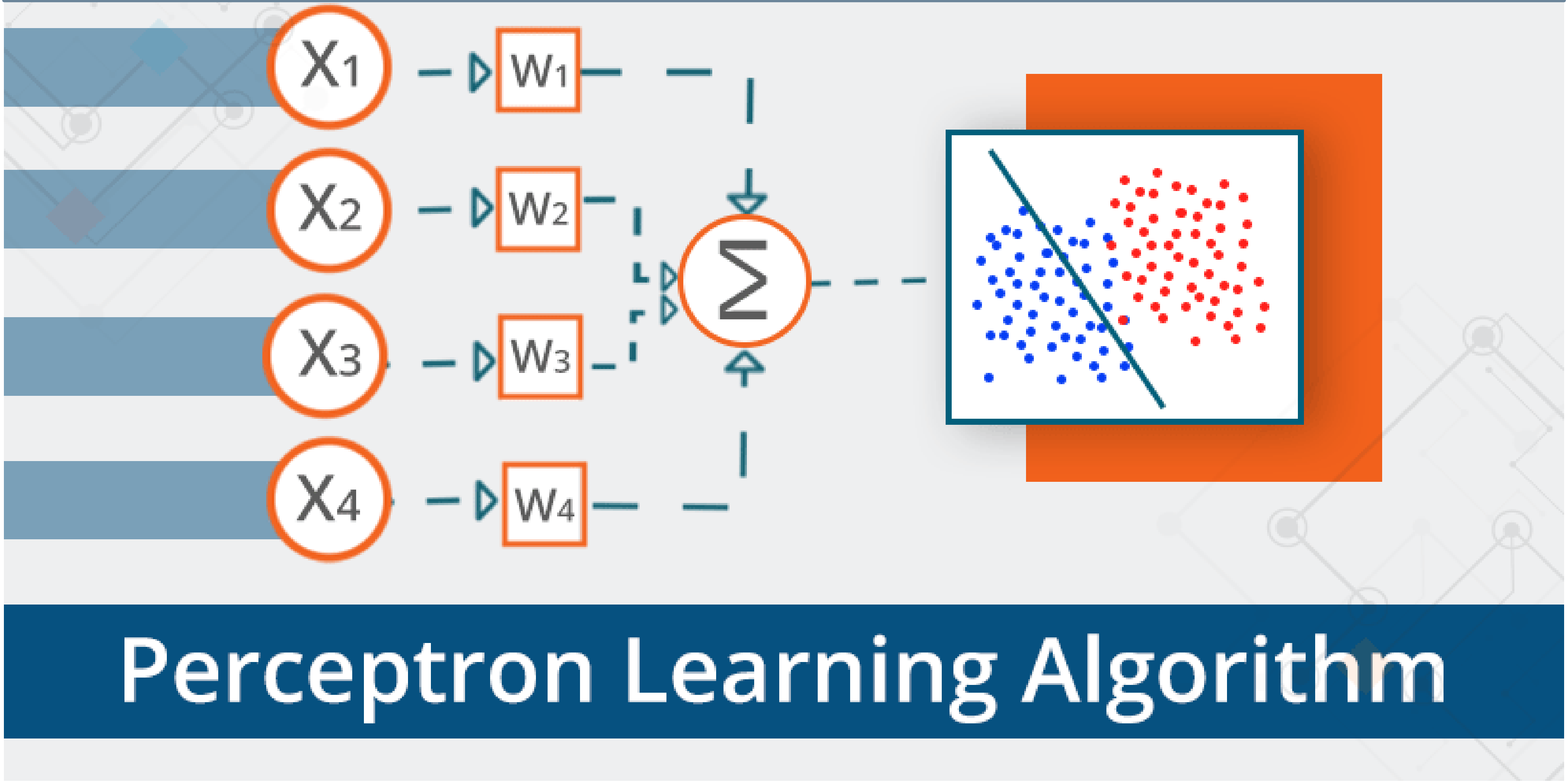
Threshold Logic Unit (TLU)



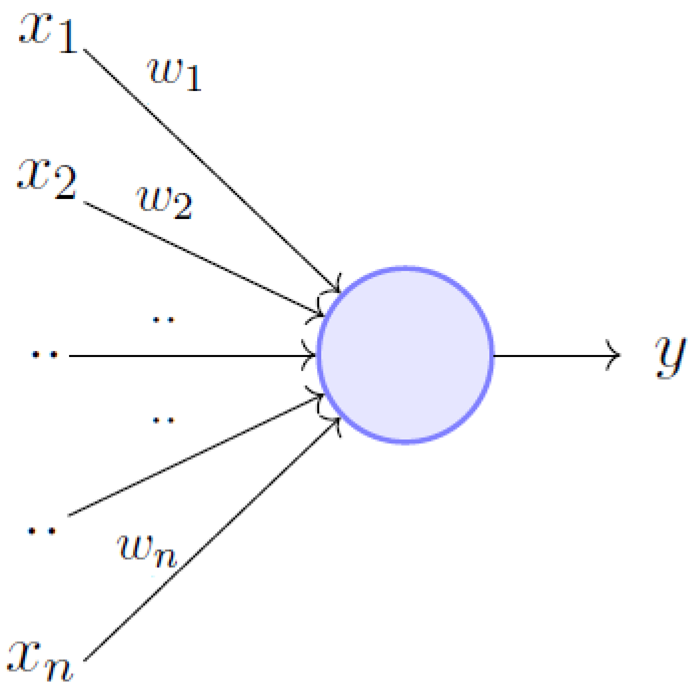
$$a=\sum_{i=1}^{n} w_i\, x_i$$

$$y=\begin{cases} 1 \text{ if } a \geq \theta \\ 0 \text{ if } a < \theta \end{cases}$$

An Artificial Neuron - The Perceptron …



An Artificial Neuron - The Perceptron …

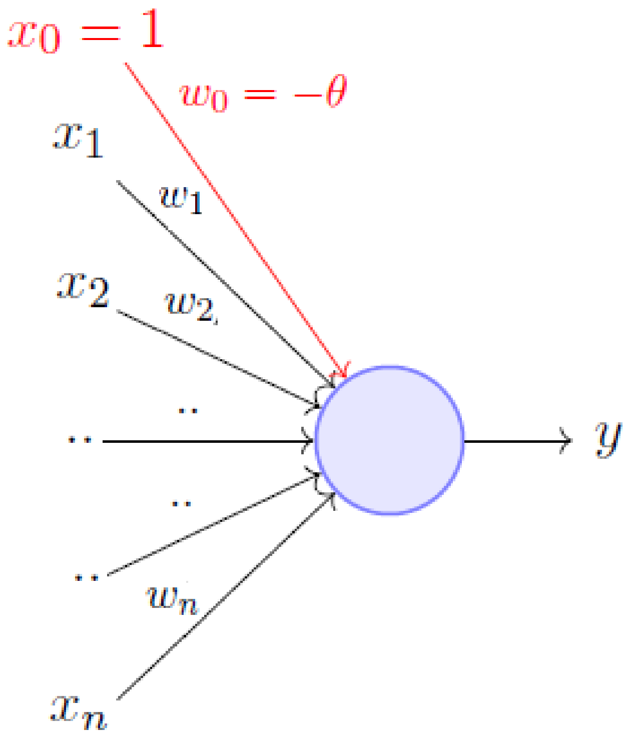


$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

An Artificial Neuron - The Perceptron …



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

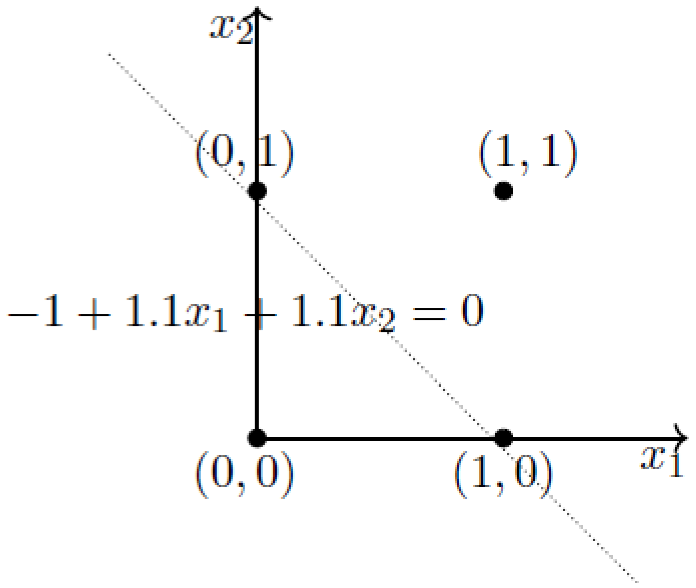
An Artificial Neuron - The Perceptron …

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is

$$w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1$$



Perceptron Training Rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

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

Where:

- $t = c(\vec{x})$ is target value
- o is perceptron output
- η is small constant (e.g., .1) called *learning rate*

Can prove it will converge

- If training data is linearly separable
- and learning rate sufficiently small

Perceptron Training

- Learning involves choosing values for the weights
- The perceptron is trained as follows:
  - First, inputs are given random weights (usually between –0.5 and 0.5).
  - An item of training data is presented. If the perceptron mis-classifies it, the weights are modified according to the following:

$$w_i \leftarrow w_i + \left(\alpha \times x_i \times (t - o)\right)$$

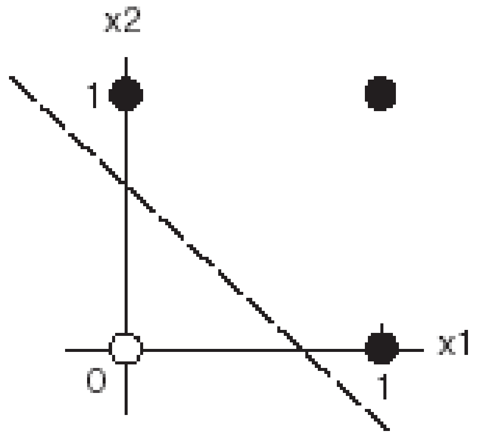
    - where *t* is the target output for the training example, o is the output generated by the preceptron and α is the learning rate, between 0 and 1 (usually small such as 0.1)
- Cycle through training examples until successfully classify all examples
  - Each cycle known as an **epoch**

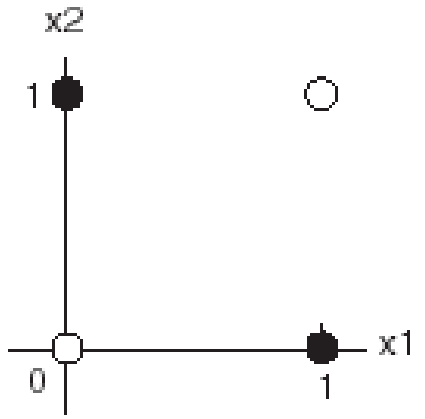
An Artificial Neuron - The Perceptron …

```
Input :
  • Training set $S = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m))$;
  • Leanring rate $\eta > 0$;
  • Maximum number of iterations $T$;
Initialisation :
  • Initialize weights $\boldsymbol{w}^{(0)} = (\bar{\boldsymbol{w}}^{(0)}, w_0^{(0)})$;
  • $t \leftarrow 0$; // Generally $\boldsymbol{w}^{(0)} = \mathbf{0}$
while $t \leqslant T$ do
    Choose an example at random $(\mathbf{x}, y) \in S$;
    if $y \times \left(\left\langle \bar{\boldsymbol{w}}^{(t)}, \mathbf{x} \right\rangle + w_0^{(t)}\right) \leqslant 0$ then
        $w_0^{(t+1)} \leftarrow w_0^{(t)} + \eta \times y$;
        $\bar{\boldsymbol{w}}^{(t+1)} \leftarrow \bar{\boldsymbol{w}}^{(t)} + \eta \times y \times \mathbf{x}$;
    else
        $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)}$;
    $t \leftarrow t + 1$;
Output :     Parameters of the linear model $\boldsymbol{w}^{(t)}$
```

An Artificial Neuron - The Perceptron …

- Perceptrons can only classify linearly separable functions.
- Consider the following Graph:





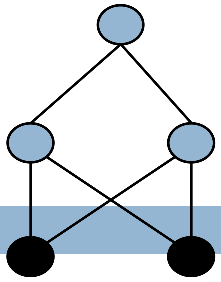
- The first graphs shows a linearly separable function (OR).
- The second is not linearly separable (Exclusive-OR).

The Limitations of Perceptrons (Minsky and Papert, 1969)

- Able to form only linear discriminate functions; i.e. classes which can be divided by a line or hyper-plane
- Most functions are more complex; i.e. they are non-linear or not linearly separable
- This crippled research in neural net theory for 15 years ....
- Contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they just make predictions based on a hard threshold. This is one of the good reasons to prefer Logistic Regression over Perceptrons.

EXAMPLE

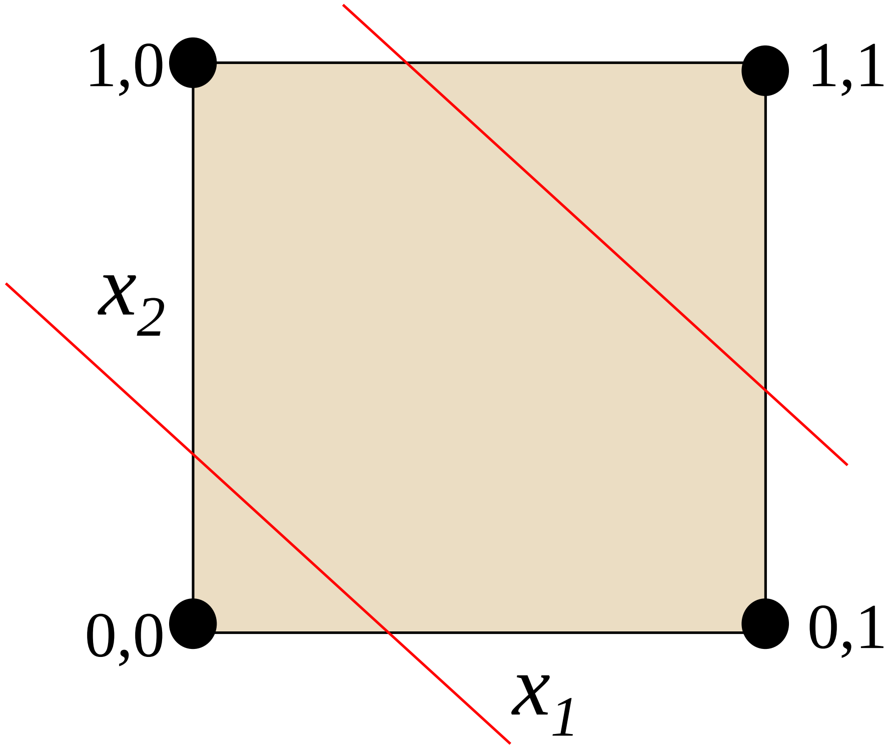
Multi-layer
Neural Network



Hidden layer
of neurons

Logical XOR
Function

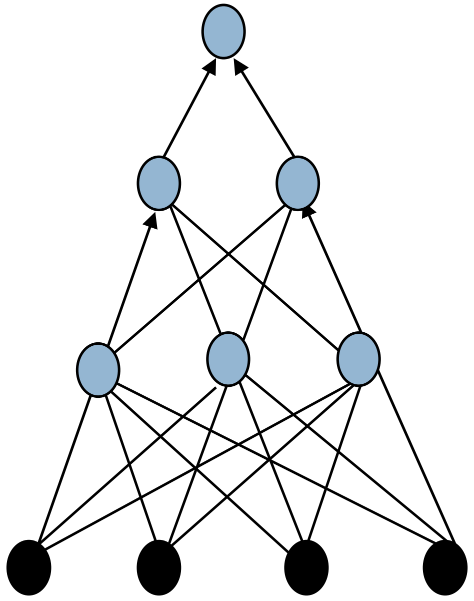
x1	*x2*	*y*
0	0	0
0	1	1
1	0	1
1	1	0

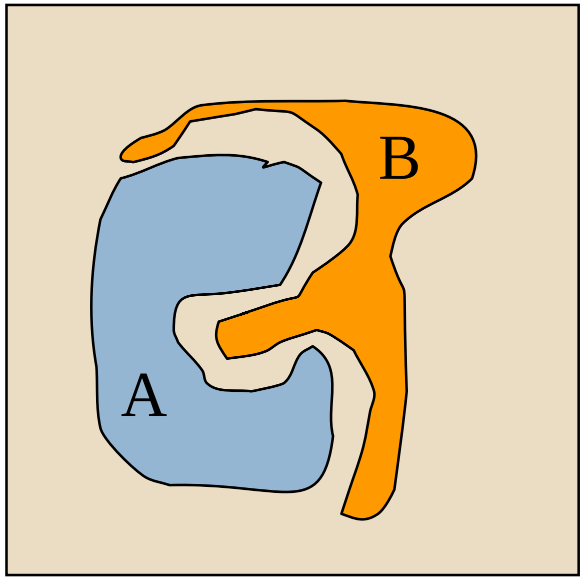


Two neurons are need! Their combined results can produce good classification.

The Limitations of Perceptrons

EXAMPLE





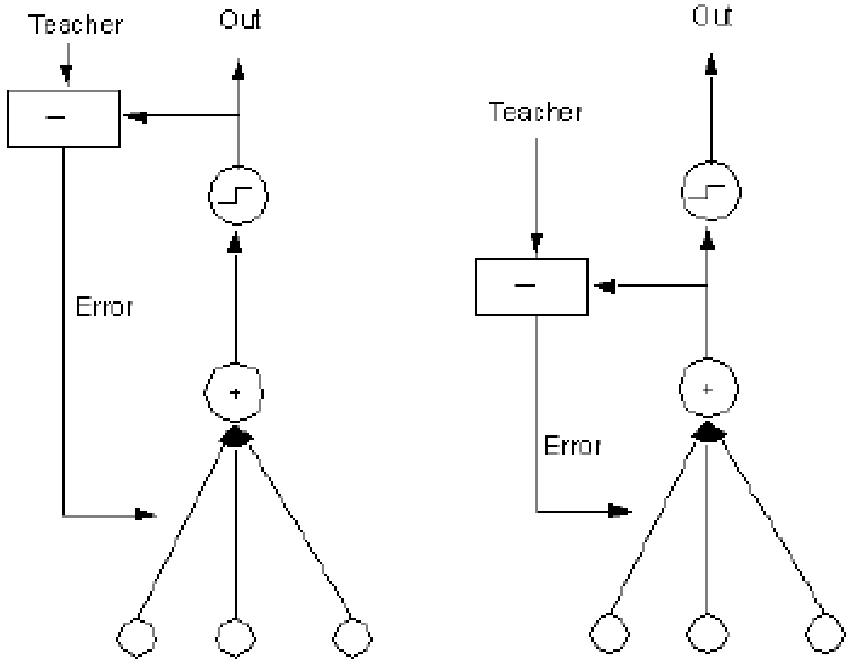
More complex multi-layer networks are needed to solve more difficult problems.

Gradient Descent and the Delta Rule

- Developed by Widrow & Hoff, aka Least Mean Square (LMS)
- Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.
- The **Delta rule**, is designed to overcome this difficulty.
- **The key idea of delta rule:** to use gradient descent to search the space of possible weight vector to find the weights that best fit the training examples.
- The delta learning rule is only valid for continuous activation functions and in the supervised training mode.

Linear Units

- Linear units are like perceptrons, but the output is used directly (not thresholded to 1 o 1)
- A linear unit can be thought of as an unthresholded perceptron



Perceptron Learning

Delta Rule

Gradient Descent …

To understand, consider simpler *linear unit*, where

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

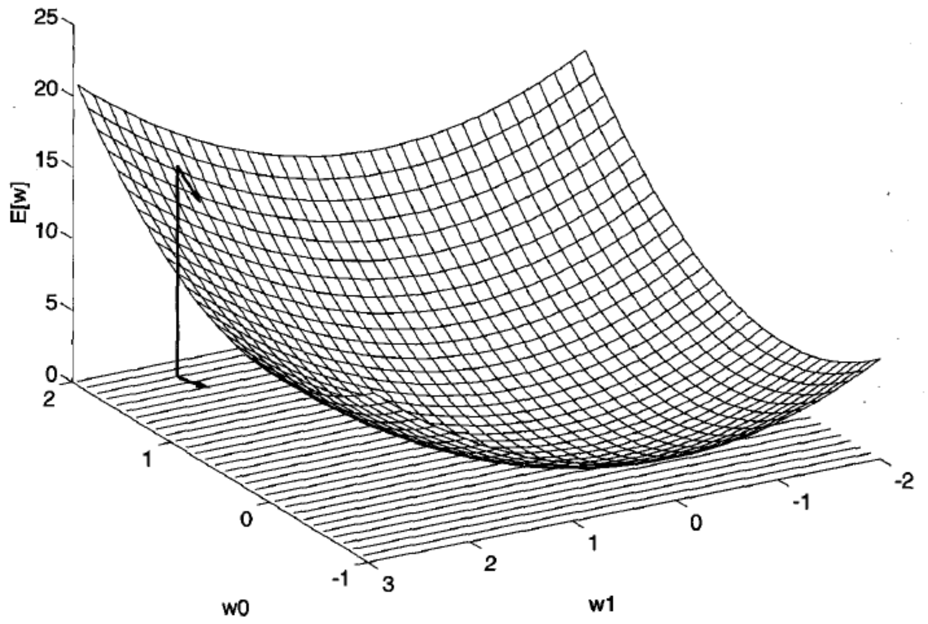
Let's learn w_i's that minimize the squared error

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Where D is set of training examples

This learning rule can be derived from the condition of least squared error between t_d and o_d.

Gradient Descent …



Here the axes w_o and w_1 represent possible values for the two weights of a simple linear unit. The w_o, w_1 plane therefore represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure. This process continues until the global minimum error is reached.

Derivation Of The Gradient Descent Rule

The direction of steepest descent can be found by computing the derivative of E with respect to each component of the vector w

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease.

The training rule for gradient descent is: $\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$

Where

$$\Delta \vec{w} = -\eta \nabla E(\vec{w}) \Longrightarrow \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

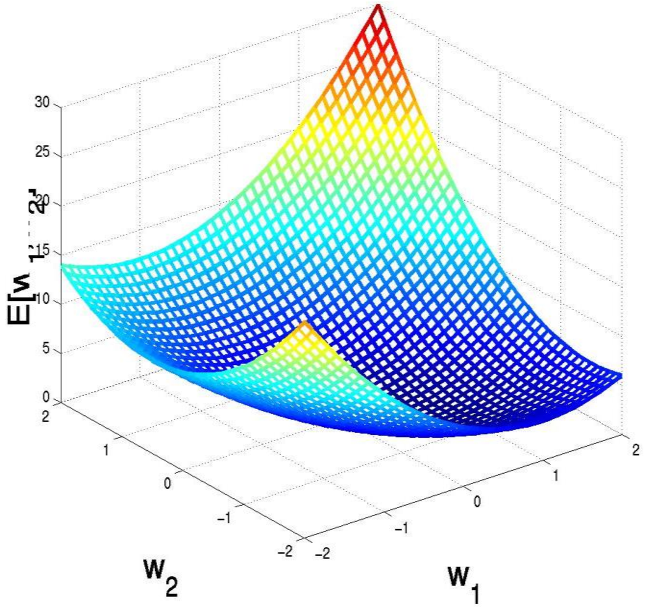
Derivation Of The Gradient Descent Rule …

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d)$$

$$= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d)$$

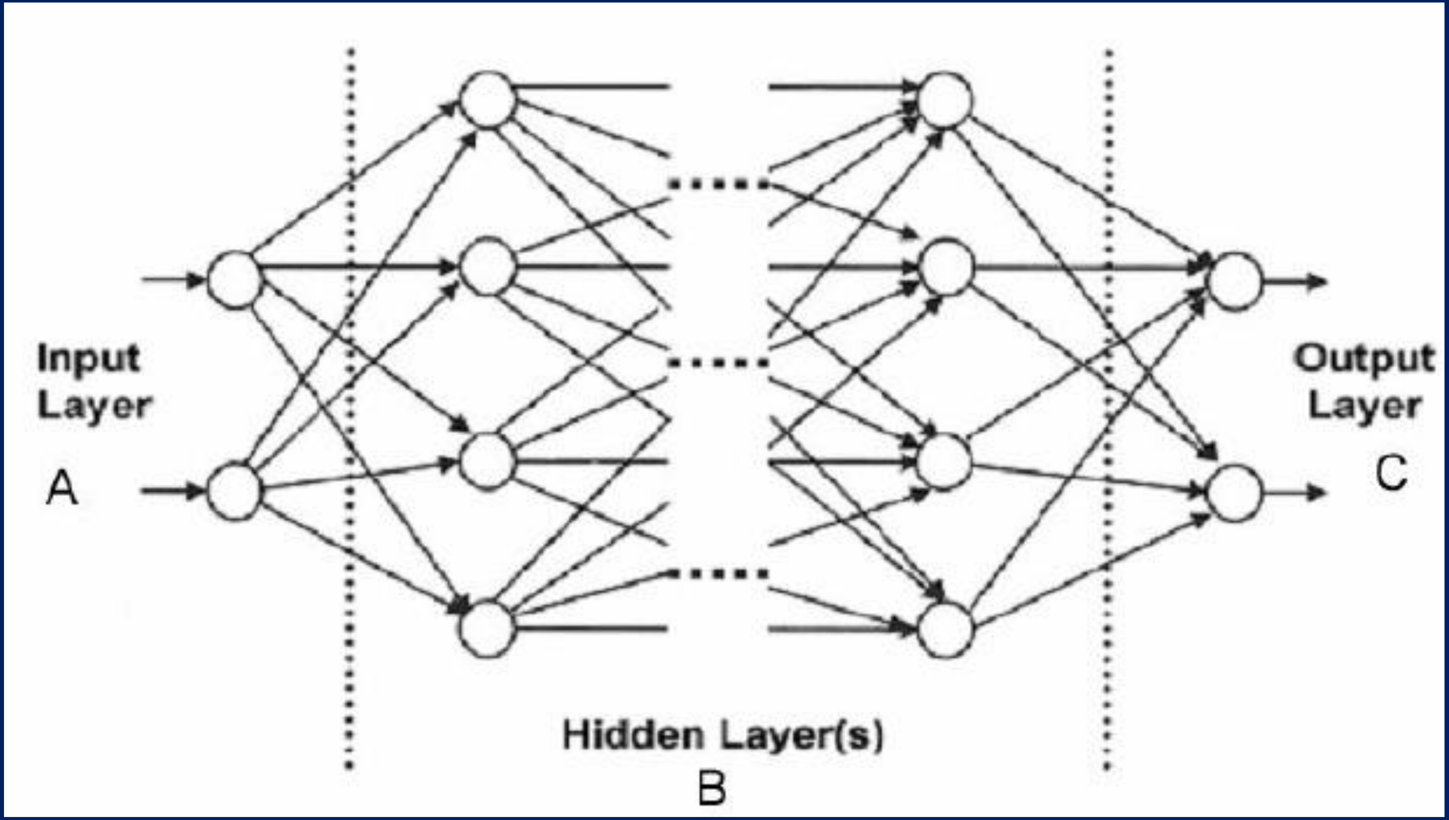


$$\frac{\partial E}{\partial w_i} = \sum_{d\in D}(t_d - o_d)(-x_{id}) \Longrightarrow \Delta w_i = \eta\sum_{d\in D}(t_d - o_d)\, x_{id}$$

Learning Perceptron Weights

- Two similar algorithms for iteratively learning perceptron weights.
- The key difference between these algorithms is that the perceptron training rule updates weights based on the error in the thresholded perceptron output, whereas the delta rule updates weights based on the error in the unthresholded linear combination of inputs.
- These two training rules have **different convergence properties**.
  - ✓ The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable.
  - ✓ The delta rule converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable.

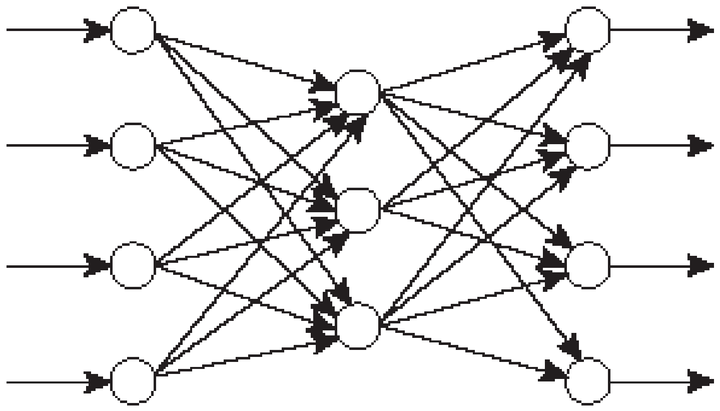
Artificial Neural Network



Multi-layer Feed-forward ANNs

Multi-layer Feed-forward ANNs

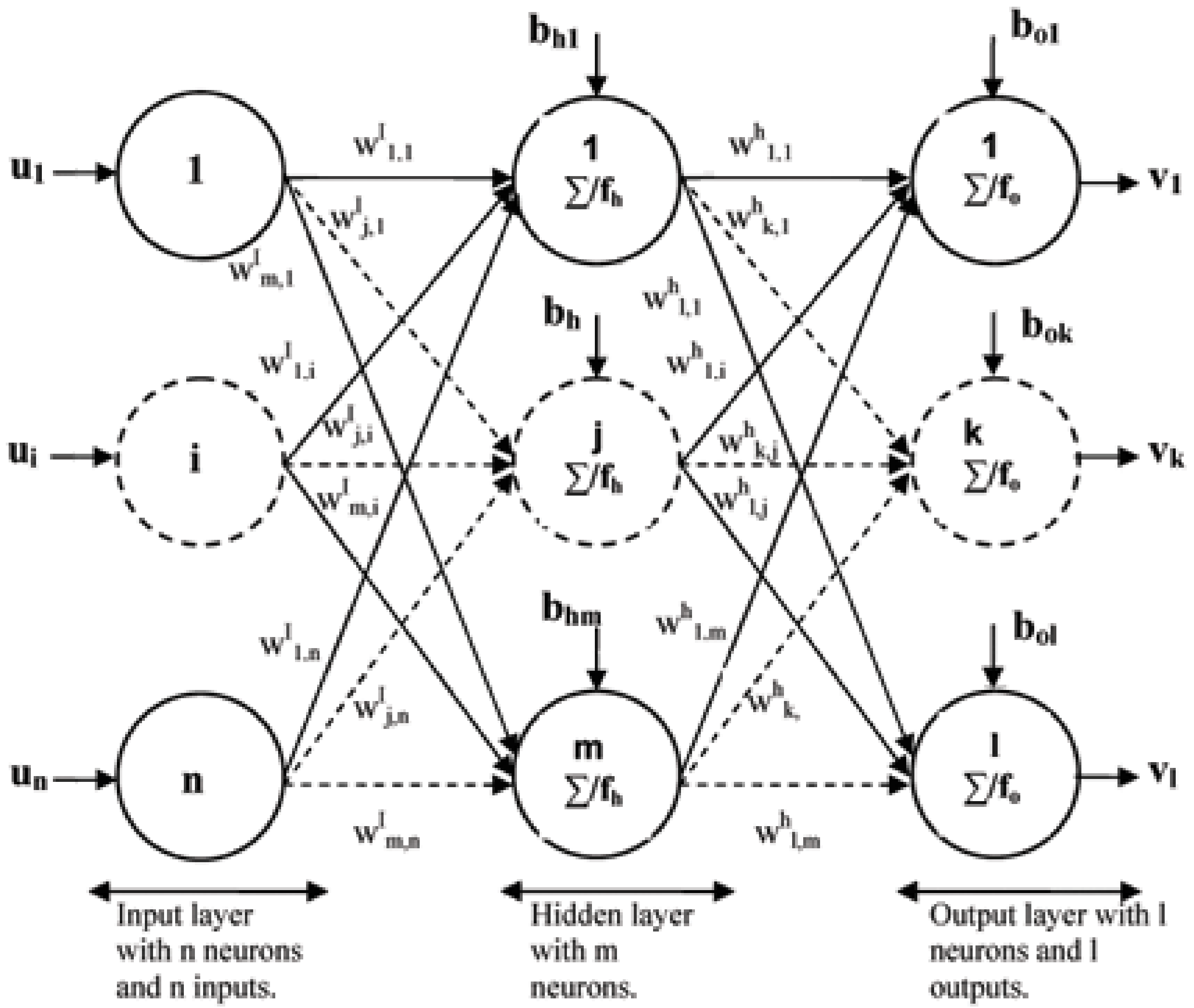
- **Multilayer neural networks can classify a range of functions, including non linearly separable ones.**
- Each input layer neuron connects to all neurons in the hidden layer.
- The neurons in the hidden layer connect to all neurons in the output layer.



A feed-forward network

Multi-layer Feed-forward ANNs …

Figure: an example of feedforward neural network

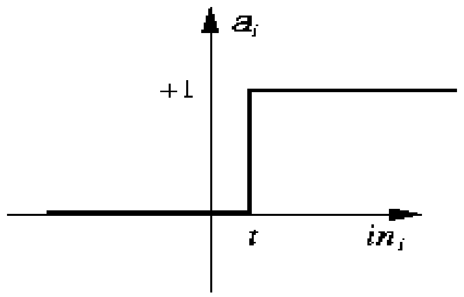


Types of Layers

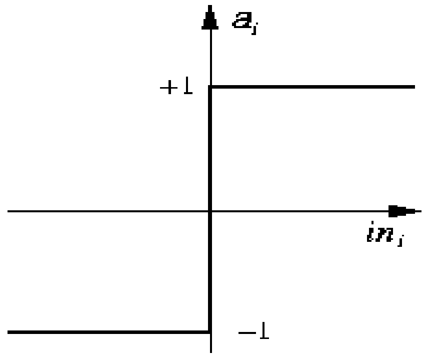
- **The input layer**
  - Introduces input values into the network.
  - No activation function or other processing.
- **The hidden layer(s).**
  - Perform classification of features
  - Two hidden layers are sufficient to solve any problem
- **The output layer.**
  - Functionally just like the hidden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

Activation Functions

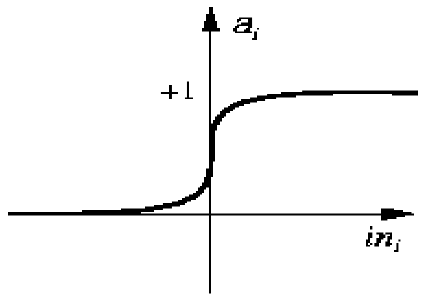
- Transforms neuron's input into output.
- Features of activation functions:
  - A squashing effect is required
    - Prevents accelerating growth of activation levels through the network.
  - Simple and easy to calculate



(a) Step function



(b) Sign function



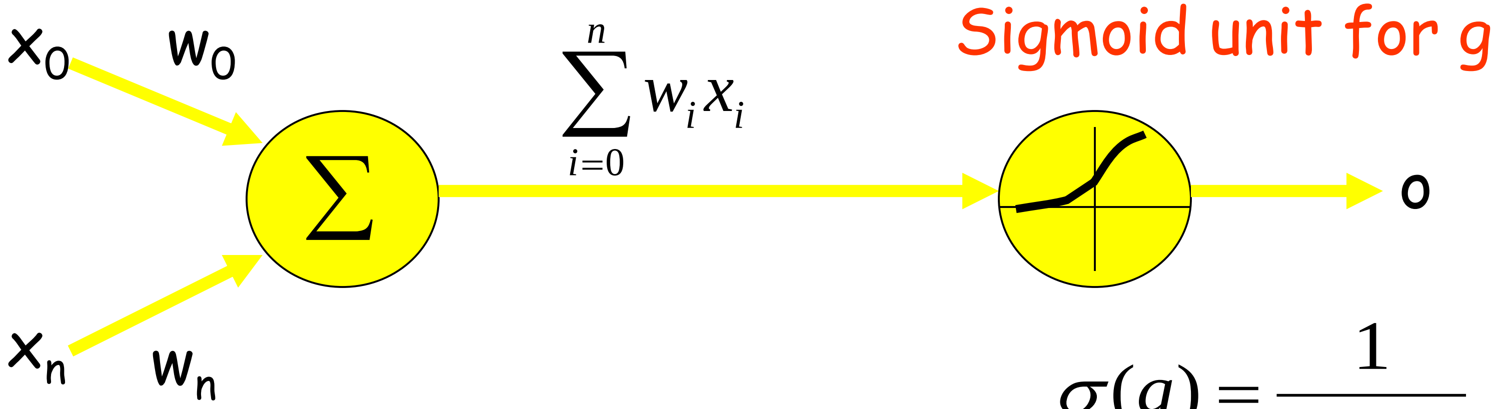
(c) Sigmoid function

Standard Activation Functions

We need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.

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

Sigmoid Units



$$\sigma(a) = \frac{1}{1+e^{-a}}$$

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1-\sigma(a))$$

This is g' (the basis for gradient descent)

What is the derivative of the logistic sigmoid function?

$$
\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\frac{1}{1+e^{-x}} \\
&= \frac{d}{dx}\left(1+e^{-x}\right)^{-1} && \text{[apply chain rule]} \\
&= -(1+e^{-x})^{-2} \cdot \frac{d}{dx}(1+e^{-x}) && \text{[apply sum rule]} \\
&= -(1+e^{-x})^{-2} \cdot \left(\frac{d}{dx}1 + \frac{d}{dx}e^{-x}\right) \\
&= -(1+e^{-x})^{-2} \cdot \frac{d}{dx}e^{-x} && \text{[apply chain rule]} \\
&= -(1+e^{-x})^{-2} \cdot e^{-x}\frac{d}{dx}(-x) \\
&= -(1+e^{-x})^{-2} \cdot \left(-e^{-x}\right) \\
&= \frac{1}{(1+e^{-x})^2} \cdot e^{-x} \\
&= \frac{e^{-x}}{(1+e^{-x})^2}
\end{aligned}
$$

What is the derivative of the logistic sigmoid function? …

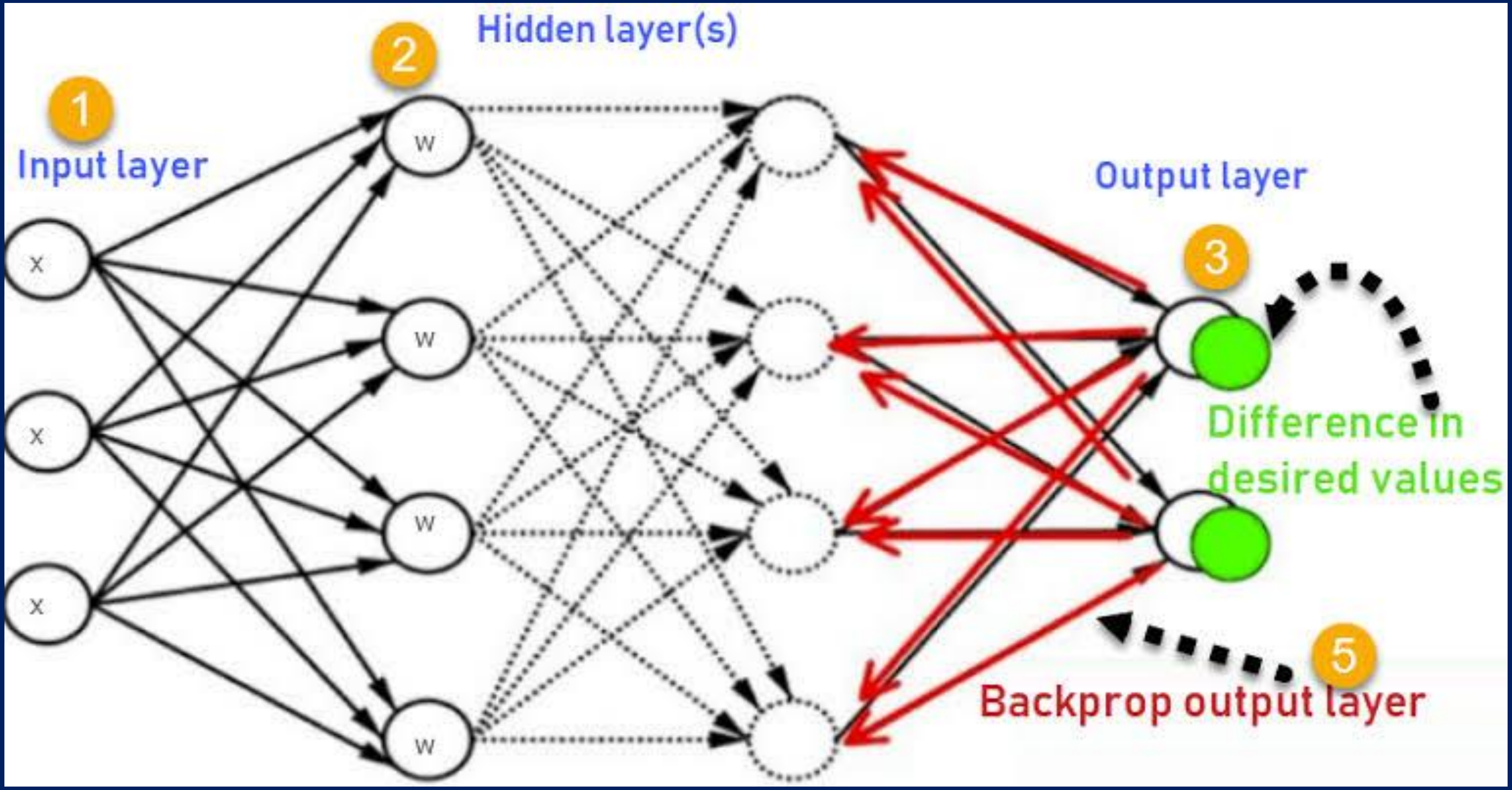
We can further simplify the derivative to the expression $\sigma(x)(1-\sigma(x))$:

$$\begin{aligned}
\frac{e^{-x}}{(1+e^{-x})^2} &= \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1}{1+e^{-x}} \\
&= \frac{-1+1+e^{-x}}{1+e^{-x}} \cdot \frac{1}{1+e^{-x}} \\
&= \left(\frac{-1}{1+e^{-x}} + \frac{1+e^{-x}}{1+e^{-x}}\right) \cdot \frac{1}{1+e^{-x}} \\
&= \left(\frac{-1}{1+e^{-x}} + 1\right) \cdot \frac{1}{1+e^{-x}} \\
&= \left(1 - \frac{1}{1+e^{-x}}\right) \cdot \frac{1}{1+e^{-x}} \\
&= (1-\sigma(x)) \cdot \sigma(x)
\end{aligned}$$

Multi-layer Feed-forward ANNs …

- **Basic Idea:** Adjust neural network weights to map inputs to outputs.
- Use a set of sample patterns where the desired output (given the inputs presented) is known.
- Non-linear activation functions displayed properties closer to real neurons
- However ... there was no learning algorithm to adjust the weights of a multi-layer network - weights had to be set by hand.

Artificial Neural Network



Back-propagation Algorithm

What is Backpropagation?

- 1986: the solution to multi-layer ANN weight update rediscovered
- Conceptually simple - the global error is backward propagated to network nodes, weights are modified proportional to their contribution
- Most important ANN learning algorithm
- Become known as back-propagation because the error is send back through the network to correct all weights

What is Backpropagation? …

- Back-propagation is the essence of neural net training.
- It is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch (i.e., iteration).
- Proper tuning of the weights allows to reduce error rates and to make the model reliable by increasing its generalization.
- It is a standard method of training artificial neural networks.
- This method helps to calculate the gradient of a loss function with respects to all the weights in the network.

Why We Need Backpropagation?

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

The Back-propagation Algorithm…

- Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd}-o_{kd})^2$$

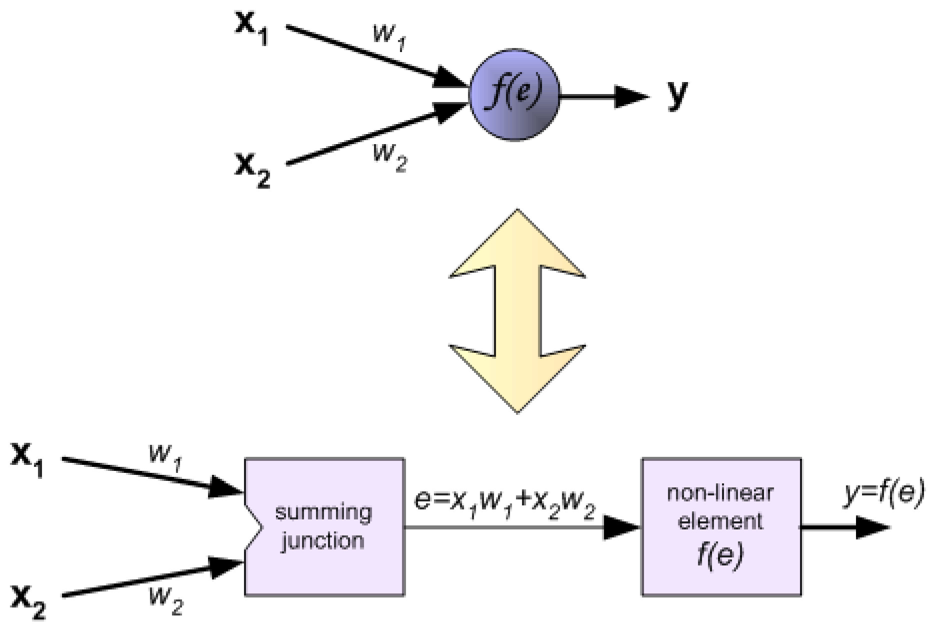
- However in BP it is the rate of change of the error which is the important feedback through the network

generalized delta rule $\Delta w_{ij} = -\eta \frac{\delta E}{\delta w_{ij}}$

- Relies on the sigmoid activation function for communication

The Back-propagation Algorithm… Example

Each neuron is composed of two units. First unit adds products of weights coefficients and input signals. The second unit realise nonlinear function, called neuron transfer (activation) function. Signal *e* is adder output signal, and *y = f(e)* is output signal of nonlinear element. Signal *y* is also output signal of neuron.



The Back-propagation Algorithm… Example

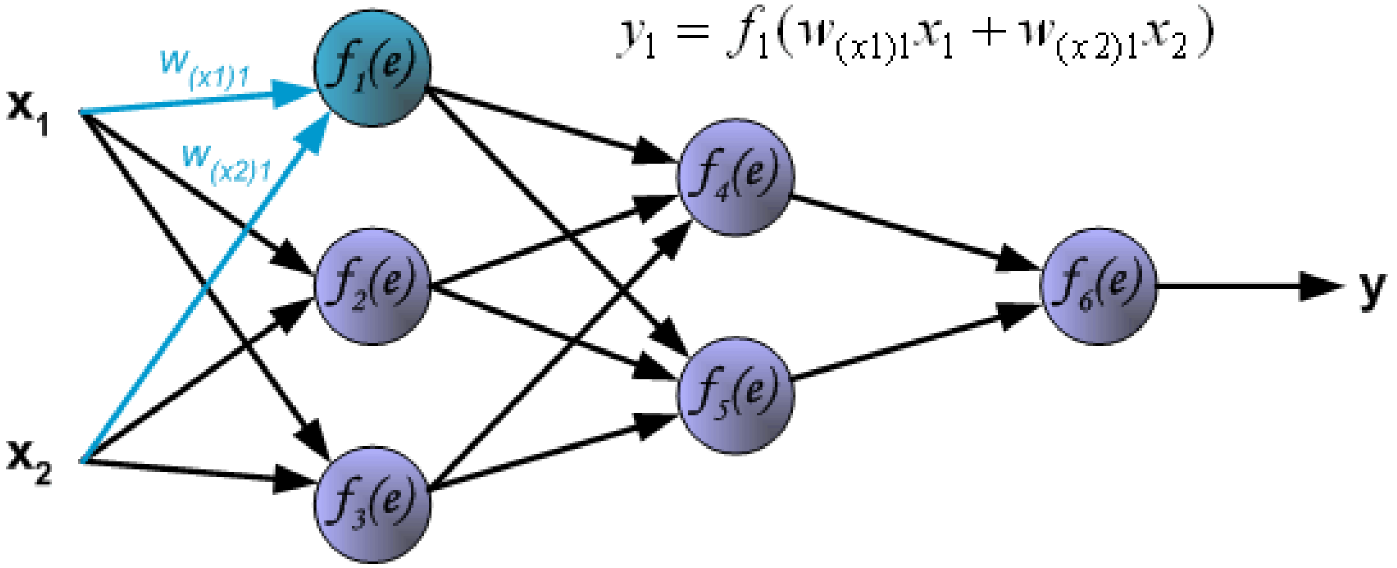
- To teach the neural network we need training data set. The training data set consists of input signals (x_1 and x_2) assigned with corresponding target (desired output) z.

- The network training is an iterative process. In each iteration weights coefficients of nodes are modified using new data from training data set. Modification is calculated using algorithm described below:

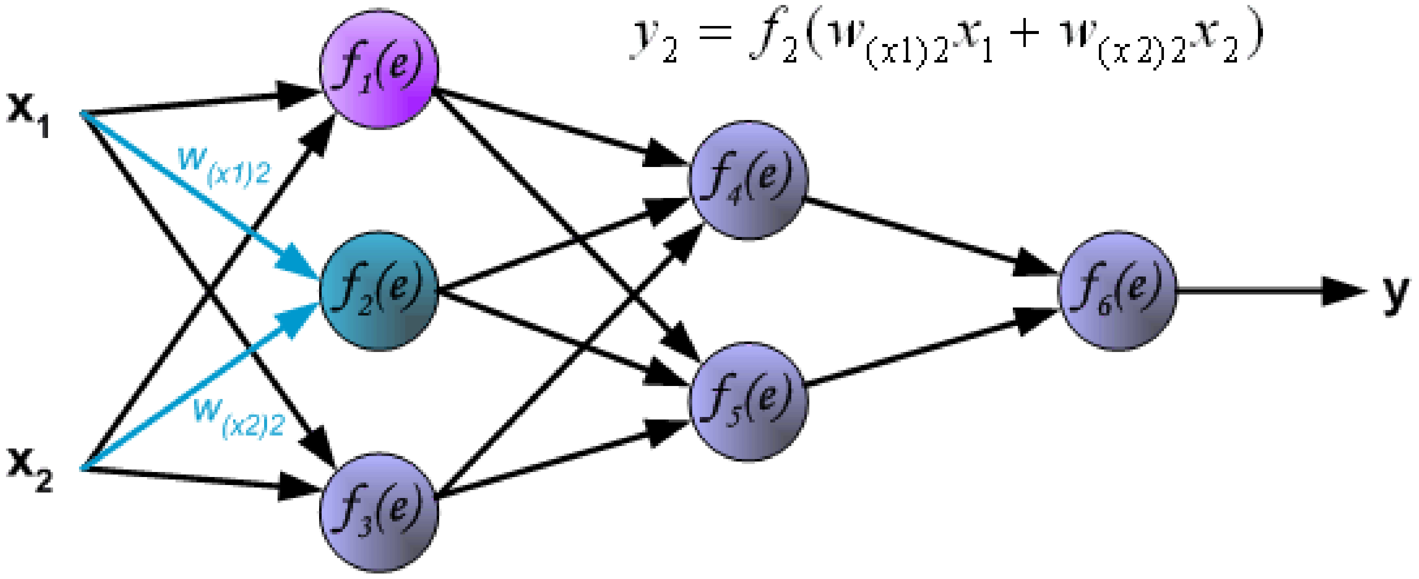
- Each teaching step starts with forcing both input signals from training set. After this stage we can determine output signals values for each neuron in each network layer.

The Back-propagation Algorithm… Example

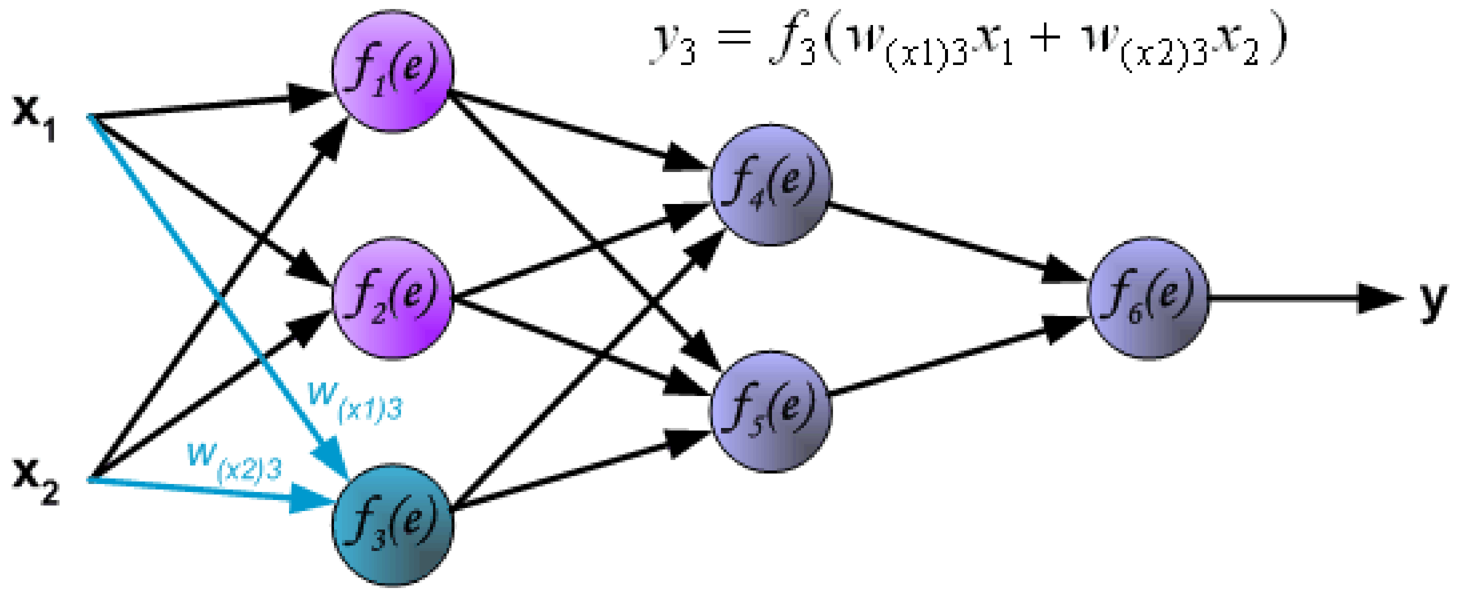
Pictures below illustrate how signal is propagating through the network, Symbols $w_{(xm)n}$ represent weights of connections between network input x_m and neuron n in input layer. Symbols y_n represents output signal of neuron n.



The Back-propagation Algorithm…
Example

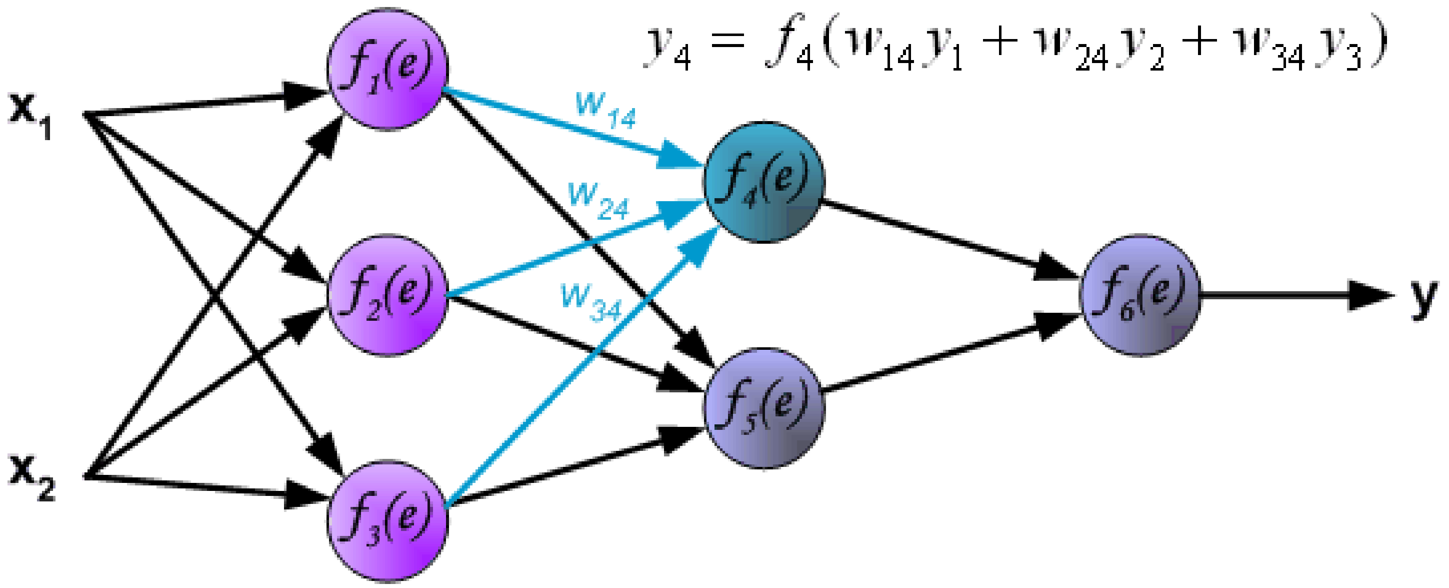


The Back-propagation Algorithm… Example



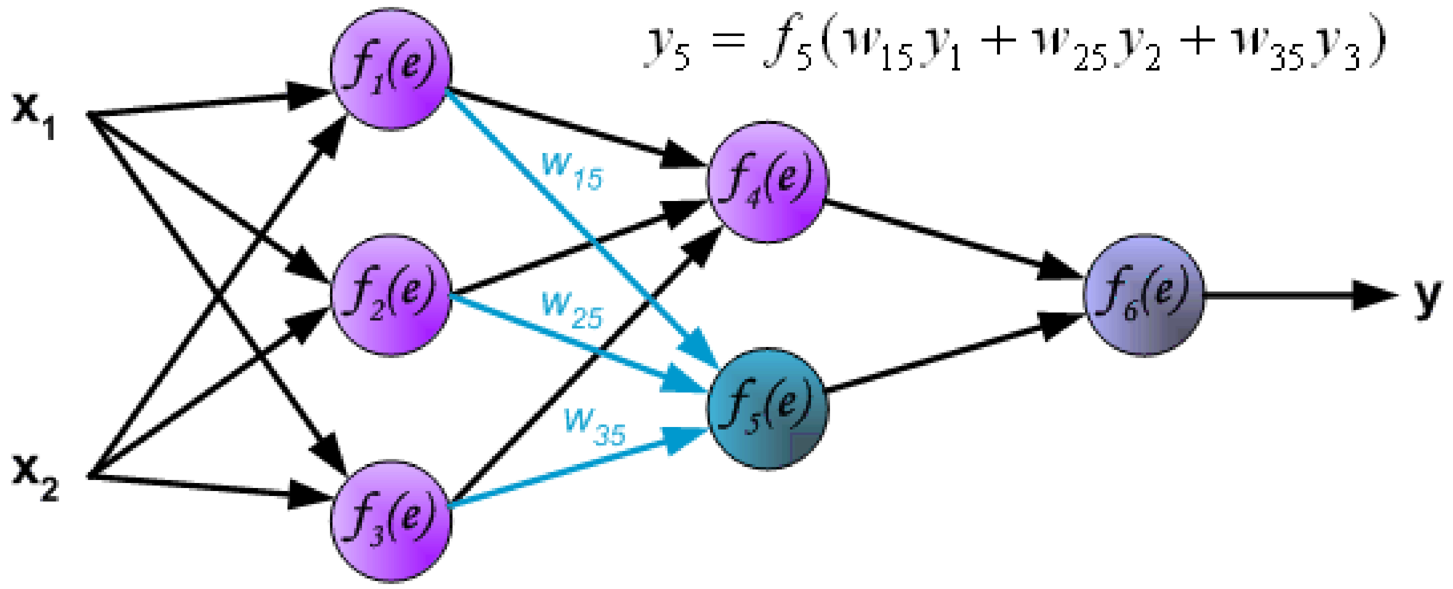
The Back-propagation Algorithm… Example

Propagation of signals through the hidden layer.
Symbols w_{mn} represent weights of connections between output of neuron *m* and input of neuron *n* in the next layer.



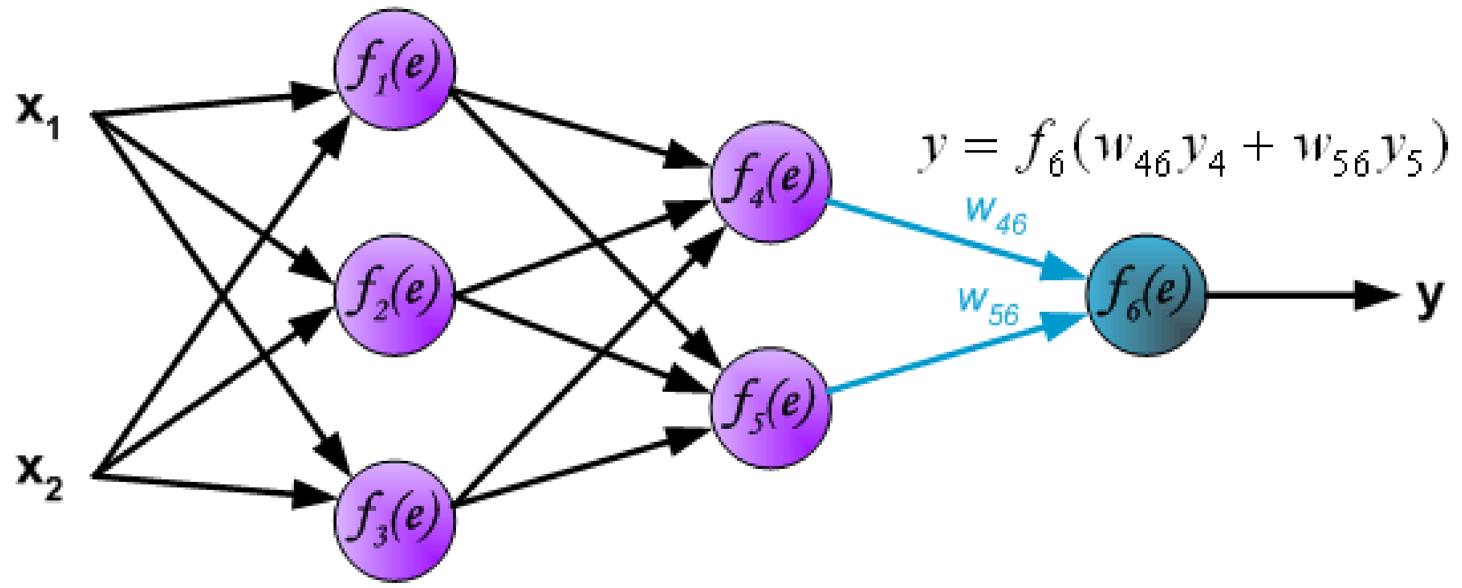
The Back-propagation Algorithm… Example

$$y_5 = f_5(w_{15}y_1 + w_{25}y_2 + w_{35}y_3)$$



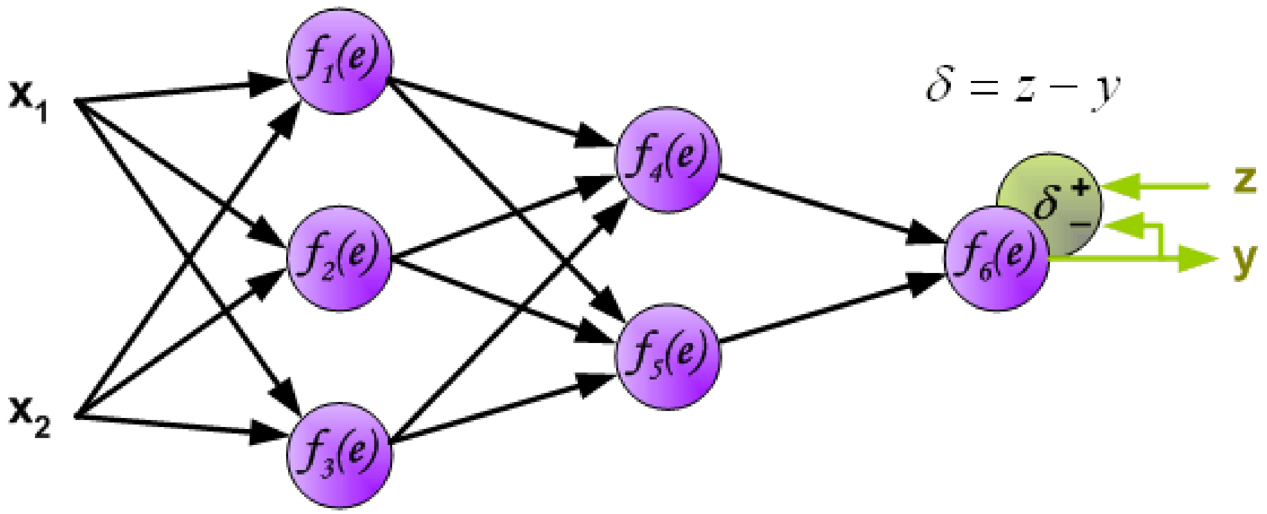
The Back-propagation Algorithm… Example

Propagation of signals through the output layer.



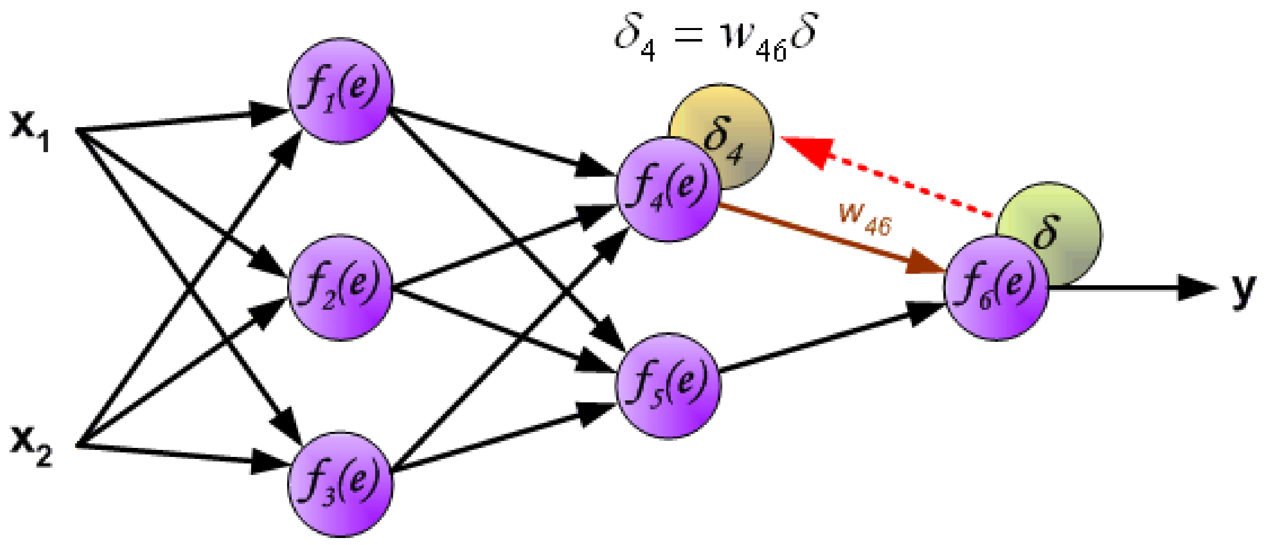
The Back-propagation Algorithm… Example

In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron



The Back-propagation Algorithm… Example

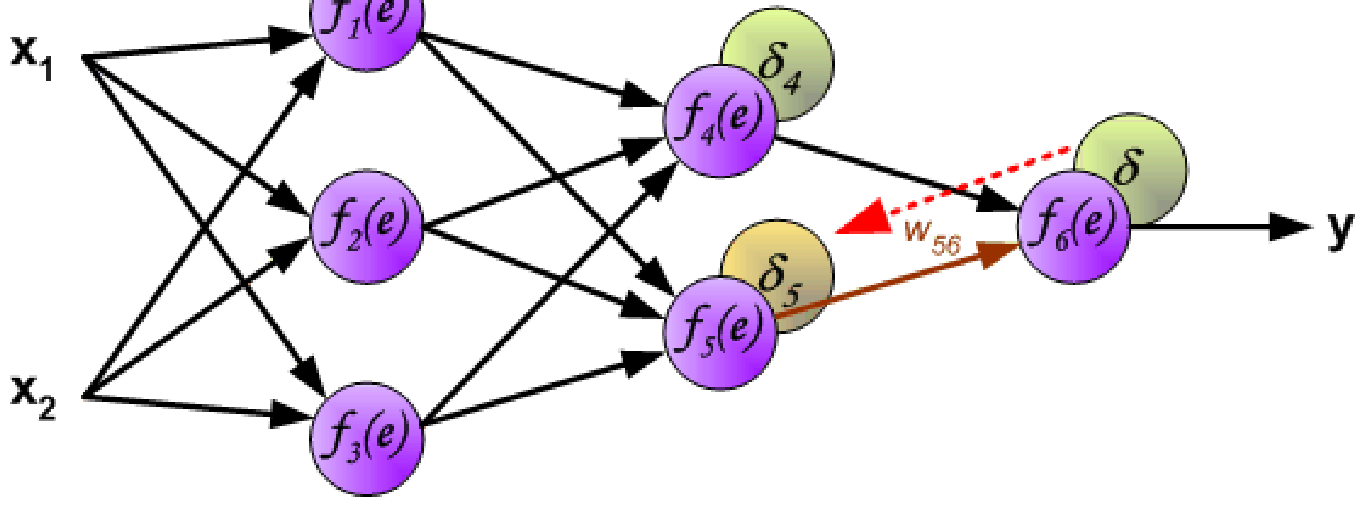
The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.



The Back-propagation Algorithm… Example

The idea is to propagate error signal *d* (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

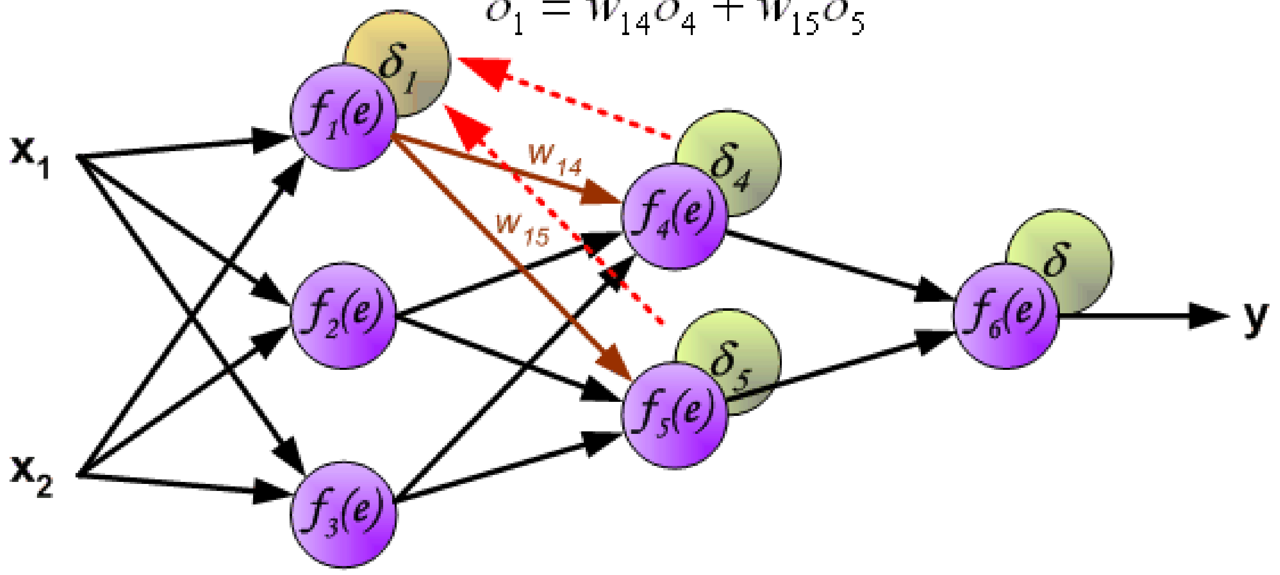
$$\delta_5 = w_{56}\delta$$



The Back-propagation Algorithm… Example

The weights' coefficients w_{mn} used to propagate errors back are equal to this used during computing output value. Only the direction of data flow is changed (signals are propagated from output to inputs one after the other). This technique is used for all network layers. If propagated errors came from few neurons they are added. The illustration is below:

$$\delta_1 = w_{14}\delta_4 + w_{15}\delta_5$$

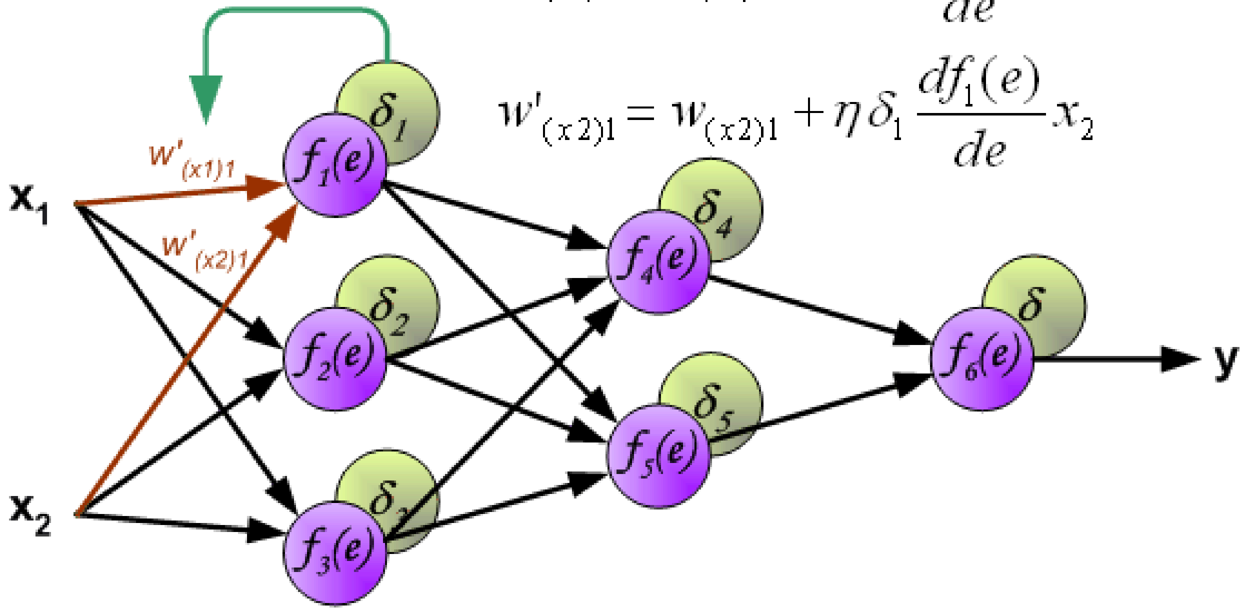


The Back-propagation Algorithm… Example

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$

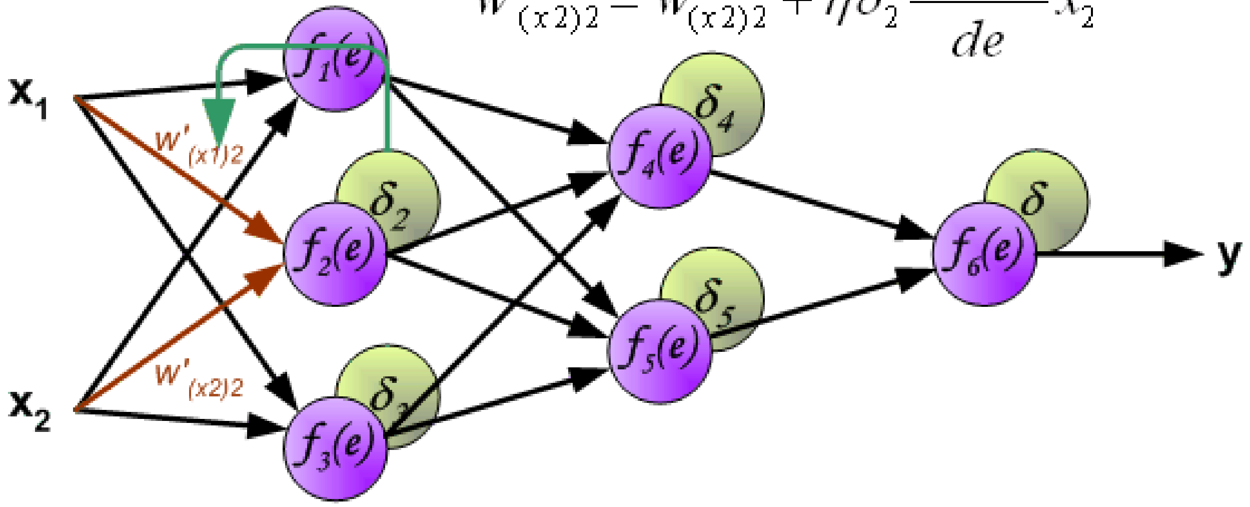


The Back-propagation Algorithm… Example

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$

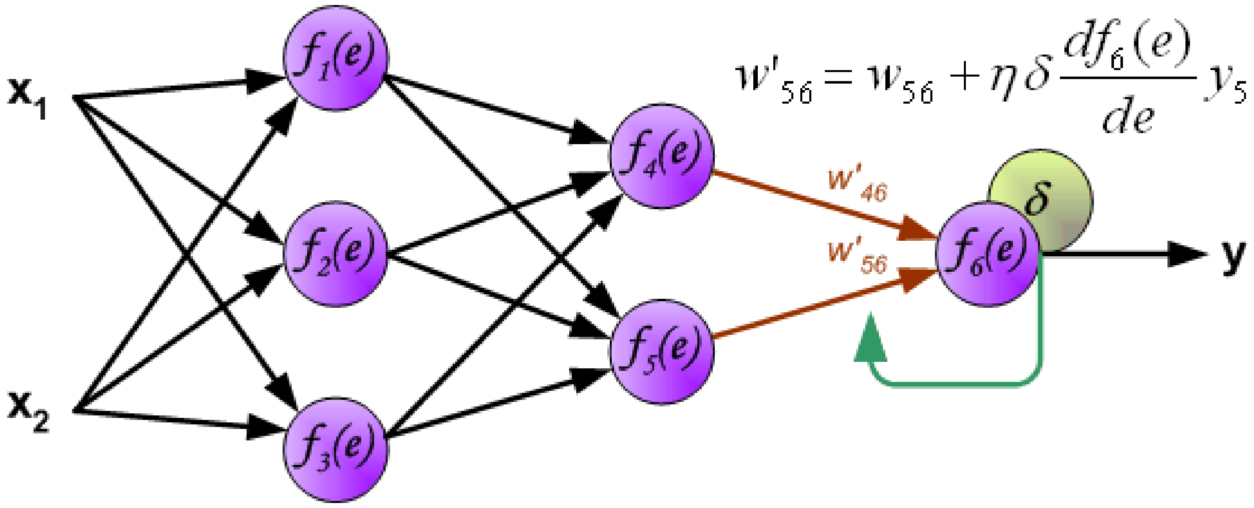


The Back-propagation Algorithm… Example

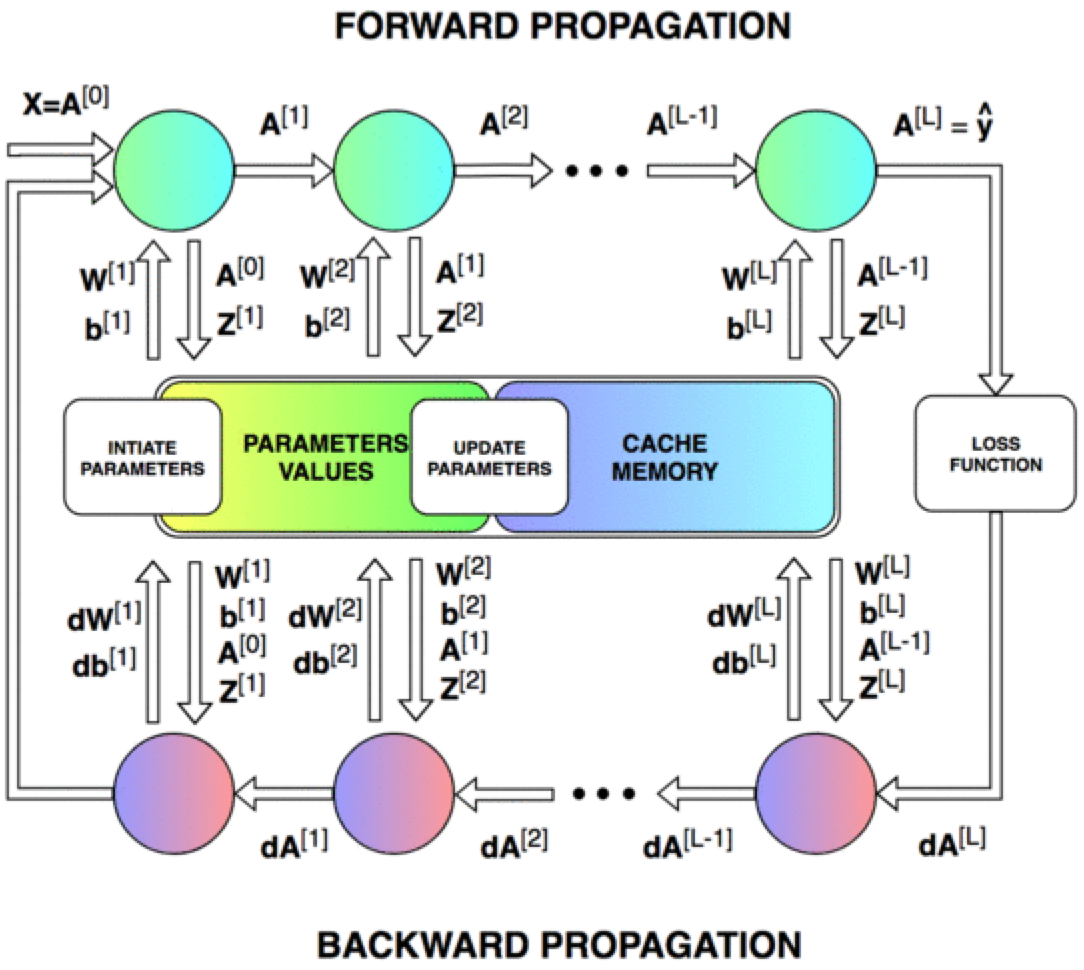
When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{46} = w_{46} + \eta\delta \frac{df_6(e)}{de} y_4$$

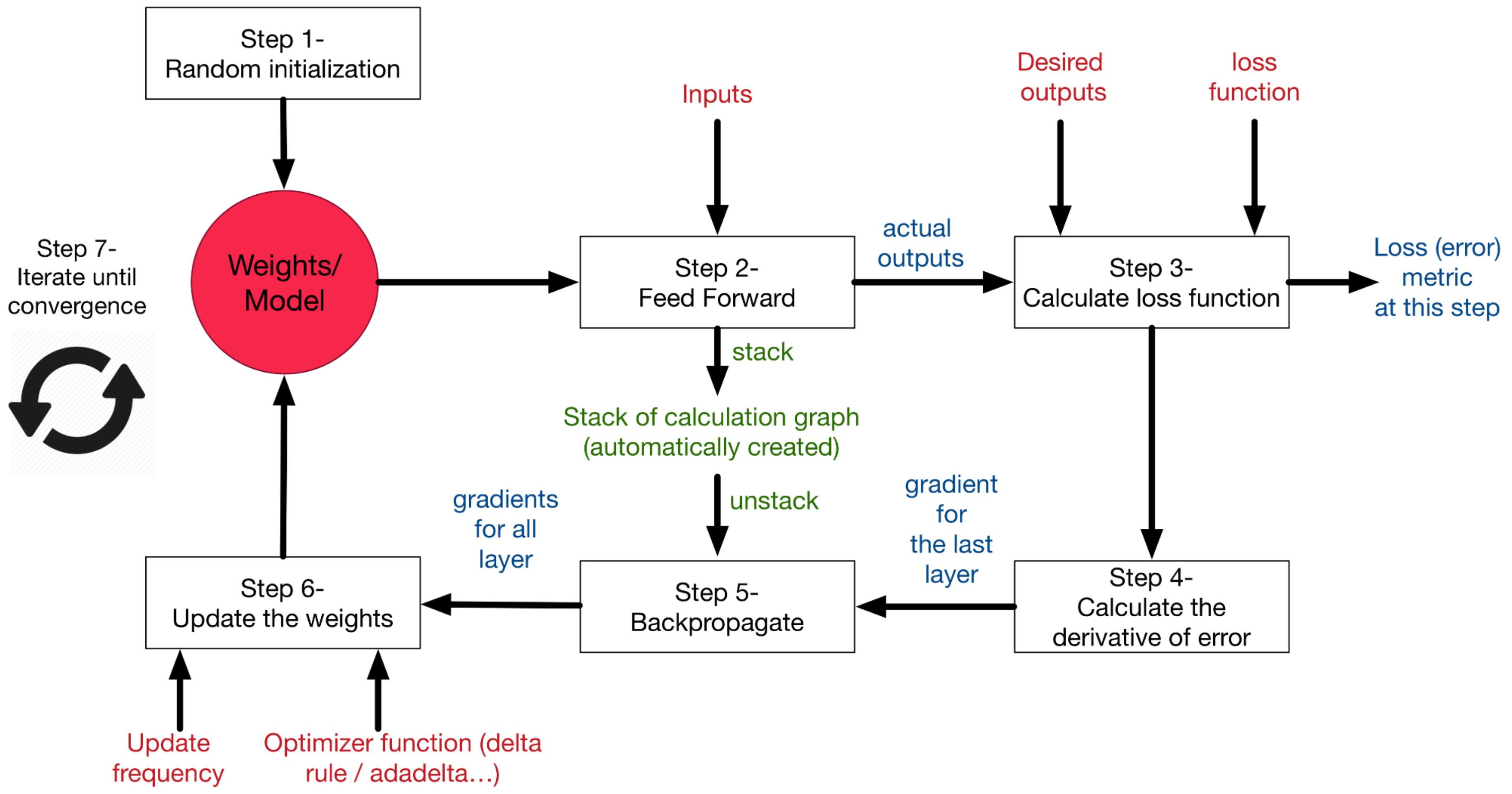
$$w'_{56} = w_{56} + \eta\delta \frac{df_6(e)}{de} y_5$$



The Back-propagation Algorithm…



The Back-propagation Algorithm…



Neural networks step-by-step

Backpropagation Algorithm

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers
- Until termination condition is met, Do
  - For each $<x,t>$ in training examples, Do

Propagate the input forward through the network:

1. Input the instance x to the network and compute the output o_u of every unit u in the network

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k)$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \alpha \delta_j x_{ji}$$

Derivation of the Backpropagation Rule

For each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

where E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here outputs is the set of output units in the network, t_k is the target value of unit k for training example d, and o_k is the output of unit k given training example d.

Derivation of the Backpropagation Rule …

Follow the notation shown below

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

Weight w_{ji} can influence the rest of the network only through net_j. Therefore, we can use the chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} x_{ji}$$

Case 1: Training Rule for Output Unit Weights {case where unit j is an output unit for the network}

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1-o_j)$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} = -(t_j - o_j)$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\ o_j(1-o_j)$$

$$\boxed{\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1-o_j)x_{ji}}$$

Case 2: Training Rule for Hidden Unit Weights

In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by *Downstream(j)*. Notice that net_j can influence the network outputs (and therefore E_d) only through the units in *Downstream(j)*.

Therefore, we can write

$$
\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1 - o_j)
\end{aligned}
$$

Case 2: Training Rule for Hidden Unit Weights …

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)$$

δ_i to denote the quantity $-\frac{\partial E_d}{\partial net_i}$ for an arbitrary unit i.

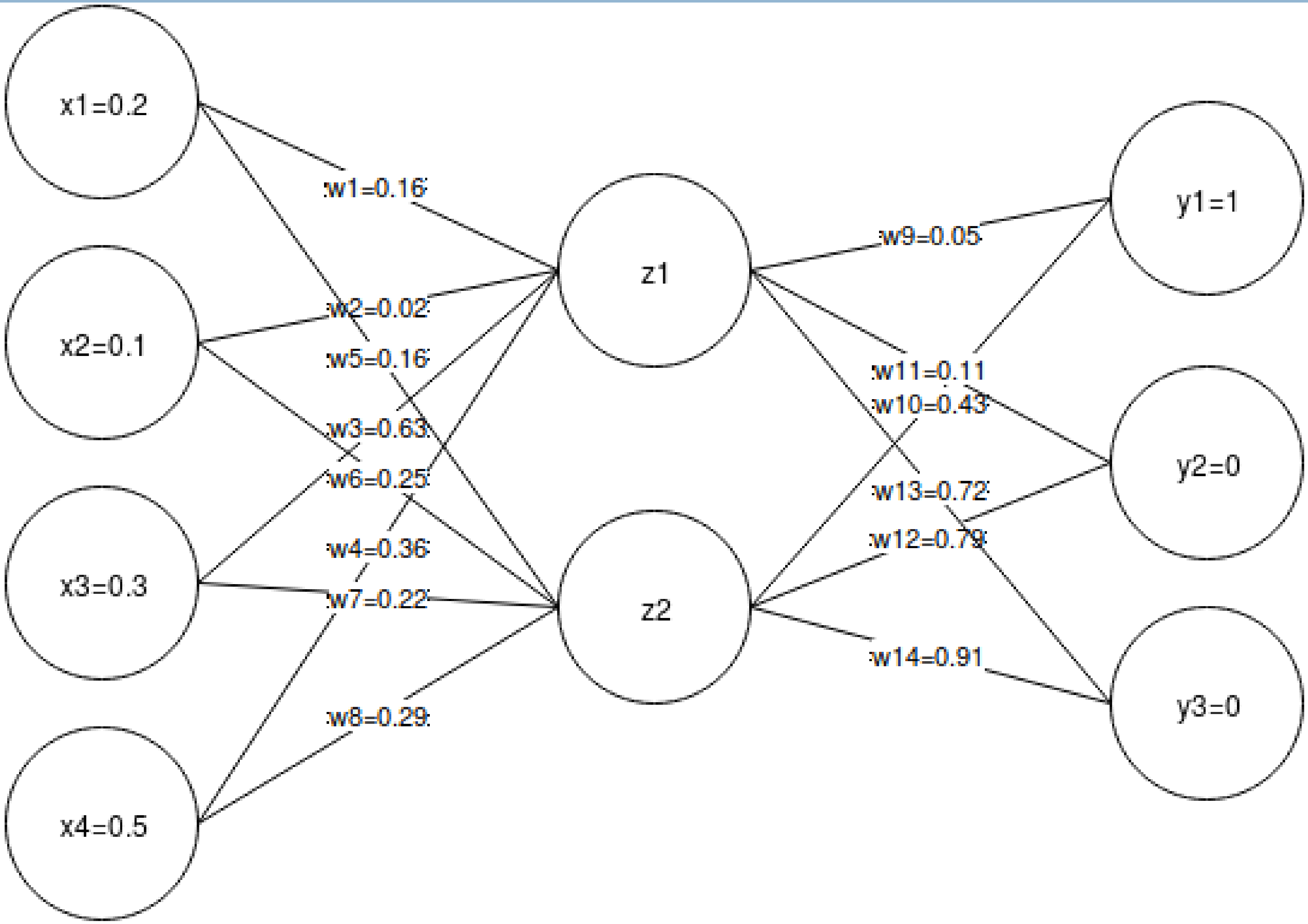
Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

and

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

The Back-propagation Algorithm… Example



Numerical Example …

initial weights

$$W_1 = \begin{bmatrix} 0.16 & 0.16 \\ 0.02 & 0.25 \\ 0.63 & 0.22 \\ 0.36 & 0.29 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.05 & 0.33 & 0.72 \\ 0.43 & 0.79 & 0.91 \end{bmatrix}$$

Activation function such as the **sigmoid function** given by:

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

Numerical Example …

Mathematically, we can then represent information flow from X to Z:

$$Z = a((X)(W_1))$$

$$Z = \begin{bmatrix} 0.2 & 0.1 & 0.3 & 0.5 \end{bmatrix} \begin{bmatrix} 0.16 & 0.16 \\ 0.02 & 0.25 \\ 0.63 & 0.22 \\ 0.36 & 0.29 \end{bmatrix}$$

$$Z = \begin{bmatrix} 0.403 & 0.268 \end{bmatrix}$$

$$Z = a(\begin{bmatrix} 0.403 & 0.268 \end{bmatrix})$$

$$Z = \begin{bmatrix} 0.60 & 0.57 \end{bmatrix}$$

$$Y = a((Z_1)(W_2))$$

$$Y = \begin{bmatrix} 0.60 & 0.57 \end{bmatrix} \begin{bmatrix} 0.05 & 0.33 & 0.72 \\ 0.43 & 0.79 & 0.91 \end{bmatrix}$$

$$Y = \begin{bmatrix} 0.28 & 0.65 & 0.95 \end{bmatrix}$$

$$Y = a(\begin{bmatrix} 0.28 & 0.65 & 0.95 \end{bmatrix})$$

$$Y = \begin{bmatrix} 0.57 & 0.66 & 0.72 \end{bmatrix}$$

Loss Function

$$E(Y,\widehat{Y}) = \sum_{i}^{n} \frac{1}{2}(Y_i - \widehat{Y}_i)^2$$

$$E(Y,\widehat{Y}) = \frac{1}{2}(0.57-1)^2 + \frac{1}{2}(0.66-0)^2 + \frac{1}{2}(0.72-0)^2$$

$$e_1 = \frac{1}{2}(0.57-1)^2$$

$$e_2 = \frac{1}{2}(0.66-0)^2$$

$$e_3 = \frac{1}{2}(0.72-0)^2$$

$$E(Y,\widehat{Y}) = e_1 + e_2 + e_3$$

$$E(Y,\widehat{Y}) = 0.09245 + 0.2178 + 0.2592$$

$$E(Y,\widehat{Y}) = 0.56945$$

BP from Output to Last Hidden Layer

Take the **loss function** for each `y_i` for instance

$$e_i = \frac{1}{2}(y_i - \widehat{y_i})^2$$

The first derivative `e'_i` for the **loss function** would be:

$$e_i' = \widehat{y_i} - y_i$$

We also take the derivative for the neurons in the right layer (output layer in this case) which we will refer to as Y'. The activation function we used was:

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

$$y_i' = a(x)(1 - a(x))$$

The derivative of a given output:

$$y_i' = y_i(1 - y_i)$$

BP from Output to Last Hidden Layer …

Compute gradients G_h (with a one to one correspondence to the right layer / output layer in this case):

$$G = \begin{bmatrix} (e'_1)(y'_1) & (e'_2)(y'_2) & (e'_3)(y'_3) \end{bmatrix}$$

Plugging in the necessary values we have:

$$G = \begin{bmatrix} (0.09245)(0.2451) & (0.2178)(0.2244) & (0.2592)(0.2016) \end{bmatrix}$$

$$G = \begin{bmatrix} 0.022679495 & 0.04887432 & 0.052255 \end{bmatrix}$$

Delta weights can be computed by multiplying the transpose of our gradients G with the output of the left layer -- in this case Z. We then transpose the result to have the same shape as W_2.

$$\delta W_2 = (G^T Z)^T$$

Plugging in the values we have:

$$\delta W_2 = (\begin{bmatrix} 0.022679495 \\ 0.04887432 \\ 0.052255 \end{bmatrix} \begin{bmatrix} 0.60 & 0.57 \end{bmatrix})^T$$

$$\delta W_2 = (\begin{bmatrix} 0.013596 & 0.012916 \\ 0.029325 & 0.027858 \\ 0.031353 & 0.029785 \end{bmatrix})^T$$

$$\delta W_2 = \begin{bmatrix} 0.013596 & 0.029325 & 0.031353 \\ 0.012916 & 0.027858 & 0.029785 \end{bmatrix}$$

BP from Output to Last Hidden Layer …

Finally we can update the weights from Z to Y:

$$\widehat{W_2} = W_2 - \delta W_2$$

$$\widehat{W_2} = \begin{bmatrix} 0.05 & 0.33 & 0.72 \\ 0.43 & 0.79 & 0.91 \end{bmatrix} - \begin{bmatrix} 0.013596 & 0.029325 & 0.031353 \\ 0.012916 & 0.027858 & 0.029785 \end{bmatrix}$$

$$\widehat{W_2} = \begin{bmatrix} 0.036404 & 0.300675 & 0.688647 \\ 0.417084 & 0.762142 & 0.880215 \end{bmatrix}$$

BP from Last Hidden Layer

Computing the gradients and updated weights from the last hidden layer Z down to the input layer X is a bit different but generally follows the same process. Probably the major difference in this step of back propagation is to apply the gradients computed in the last operation (gradients corresponding to Y) as part of the computation.

$$G_p = G$$

We then have to solve for a new G which corresponds to the gradients of the right layer in this operation (Z). Aside from taking the previous operation's gradients, we have to also account for the old weights in the previous operation (W).

$$G = (G_p W_p^T) \times Z'$$

BP from Last Hidden Layer …

Let's solve for the derivatives Z' first using the same derivative equation as Y':

$$Z' = \begin{bmatrix} z_1' & z_2' \end{bmatrix} = \begin{bmatrix} (0.60)(1-0.60) & (0.57)(1-0.57) \end{bmatrix} = \begin{bmatrix} 0.24 & 0.2451 \end{bmatrix}$$

Next we solve for GpWp^T:

$$G_p = \begin{bmatrix} 0.022679495 & 0.04887432 & 0.052255 \end{bmatrix}$$

$$W_p^T = \begin{bmatrix} 0.05 & 0.43 \\ 0.33 & 0.79 \\ 0.72 & 0.91 \end{bmatrix}$$

$$G_pW_p^T = \begin{bmatrix} 0.022679495 & 0.04887432 & 0.052255 \end{bmatrix} \begin{bmatrix} 0.05 & 0.43 \\ 0.33 & 0.79 \\ 0.72 & 0.91 \end{bmatrix}$$

$$G_pW_p^T = \begin{bmatrix} 0.054885 & 0.095906 \end{bmatrix}$$

BP from Last Hidden Layer …

Finally putting them all together to solve for G = (GpWp^T) \times Z'$:

$$G = \begin{bmatrix} 0.054885 & 0.095906 \end{bmatrix} \times \begin{bmatrix} 0.24 & 0.2451 \end{bmatrix} = \begin{bmatrix} 0.0131723757 & 0.02350658 \end{bmatrix}$$

We can then extract the delta weights delta{W1} using the following:

$$\delta W_1 = X^T G_h$$

$$\delta W_1 = \begin{bmatrix} 0.2 \\ 0.1 \\ 0.3 \\ 0.5 \end{bmatrix} \begin{bmatrix} 0.0131723757 & 0.02350658 \end{bmatrix} = \begin{bmatrix} 0.00263447514 & 0.004701316573467 \\ 0.00131723757 & 0.002350658286734 \\ 0.00395171271 & 0.007051974860201 \\ 0.00658618785 & 0.011753291433668 \end{bmatrix}$$

$$\widehat{W_1} = W_1 - \delta W_1$$

$$\widehat{W_1} = \begin{bmatrix} 0.16 & 0.16 \\ 0.02 & 0.25 \\ 0.63 & 0.22 \\ 0.36 & 0.29 \end{bmatrix} - \begin{bmatrix} 0.00263447514 & 0.004701316573467 \\ 0.00131723757 & 0.002350658286734 \\ 0.00395171271 & 0.007051974860201 \\ 0.00658618785 & 0.011753291433668 \end{bmatrix} = \begin{bmatrix} 0.15736552486 & 0.155298683426533 \\ 0.01868276243 & 0.247649341713266 \\ 0.62604828729 & 0.212948025139799 \\ 0.35341381215 & 0.278246708566332 \end{bmatrix}$$

We now have the following updated weights for our network:

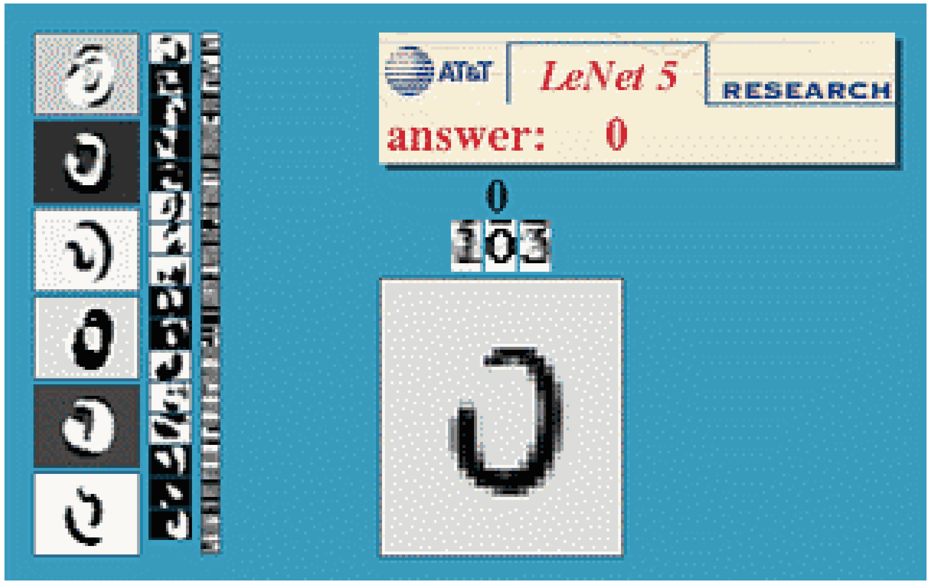
$$W_1 = \begin{bmatrix} 0.15736552486 & 0.155298683426533 \\ 0.01868276243 & 0.247649341713266 \\ 0.62604828729 & 0.212948025139799 \\ 0.35341381215 & 0.278246708566332 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.036404303 & 0.300675408 & 0.688647168 \\ 0.417084089 & 0.762141638 & 0.880214810 \end{bmatrix}$$

To test if we have indeed trained the network, we'll use the same input and perform a feed forward using the updated weights. This will iterated till the convergence of the network

Example Demo

Handwritten Digit Classification



Source: http://yann.lecun.com/

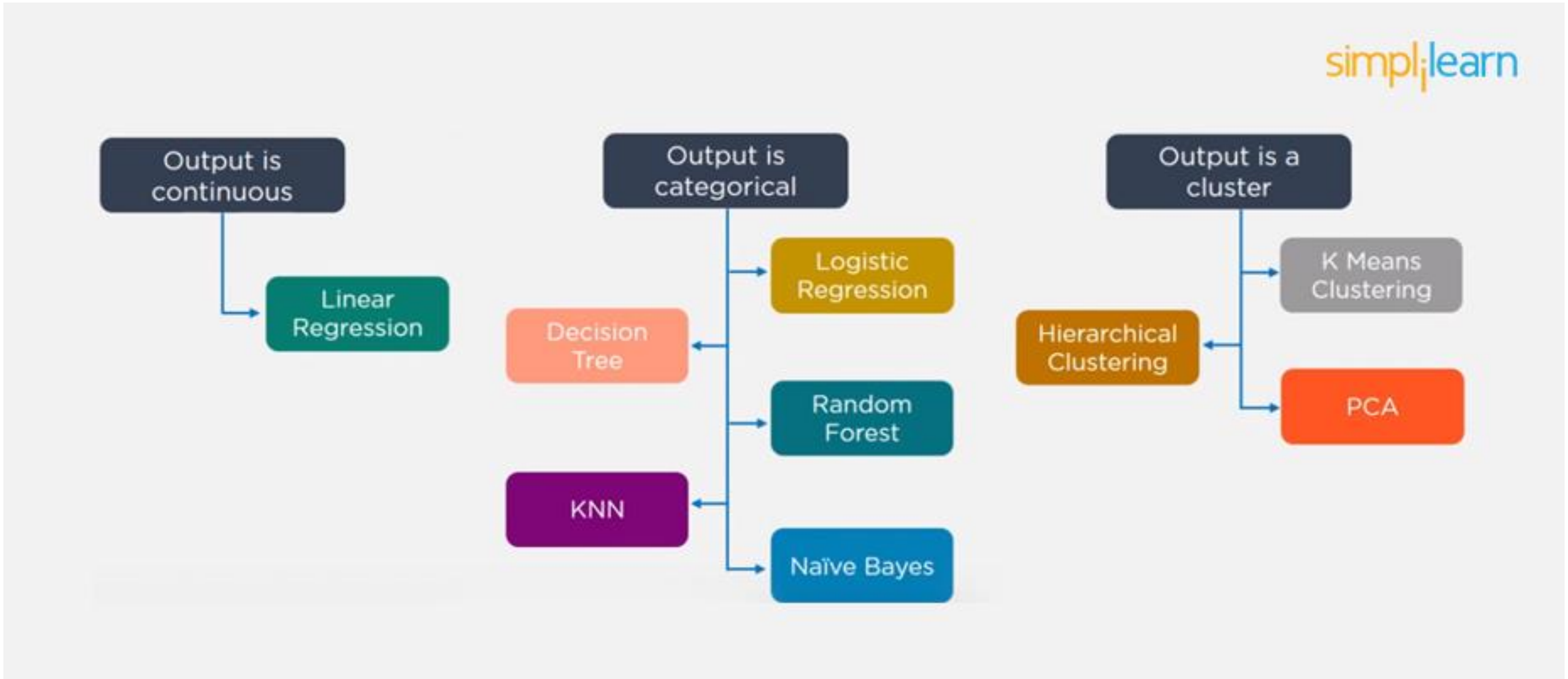
Yann LeCun,

VP and Chief AI Scientist, Facebook

Professor of Computer Science, New York University.

Given a Data Set, How Do You Decide Which One to Use?

Choosing an algorithm depends on the following questions:
•How much data do you have, and is it continuous or categorical?
•Is the problem related to classification, association, clustering, or regression?
•Predefined variables (labeled), unlabeled, or mix?
•What is the goal?
Based on the above questions, the following algorithms can be used:



1. Which of the following statements are true? Check all that apply.

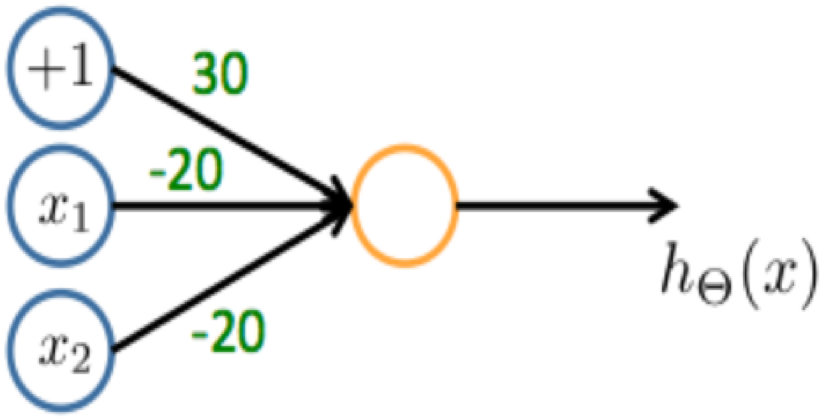
- ☐ A two layer (one input layer, one output layer; no hidden layer) neural network can represent the XOR function.

- ☐ Any logical function over binary-valued (0 or 1) inputs x_1 and x_2 can be (approximately) represented using some neural network.

- ☐ Suppose you have a multi-class classification problem with three classes, trained with a 3 layer network. Let $a_1^{(3)} = (h_\Theta(x))_1$ be the activation of the first output unit, and similarly $a_2^{(3)} = (h_\Theta(x))_2$ and $a_3^{(3)} = (h_\Theta(x))_3$. Then for any input x, it must be the case that $a_1^{(3)} + a_2^{(3)} + a_3^{(3)} = 1$.

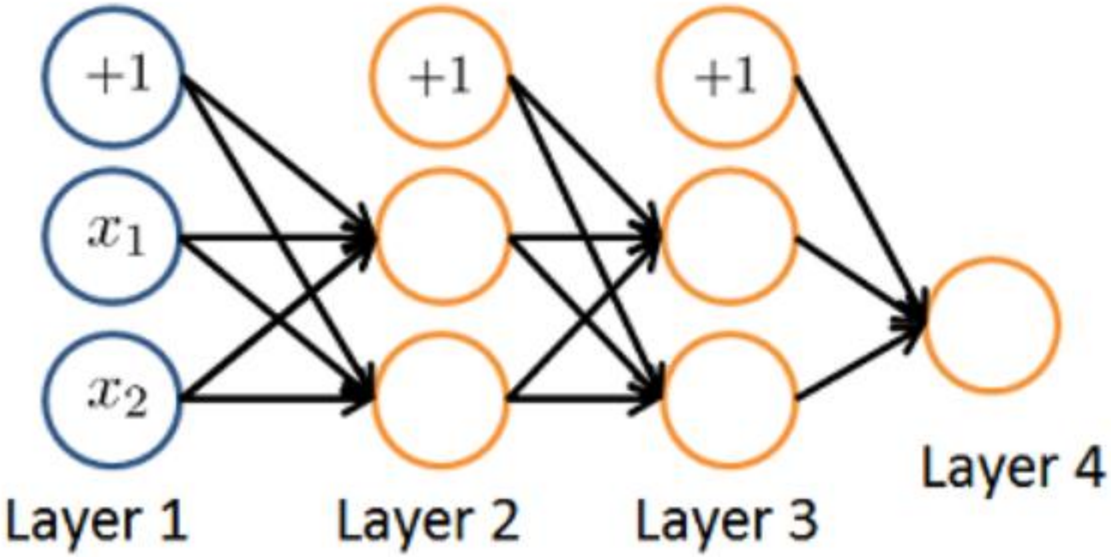
- ☐ The activation values of the hidden units in a neural network, with the sigmoid activation function applied at every layer, are always in the range (0, 1).

Consider the following neural network which takes two binary-valued inputs $x_1, x_2 \in \{0, 1\}$ and outputs $h_\Theta(x)$. Which of the following logical functions does it (approximately) compute?



A) NAND
B) AND
C) OR
D) XOR

Consider the neural network given below. Which of the following equations correctly computes the activation $a_1^{(3)}$? Note: $g(z)$ is the sigmoid activation function.



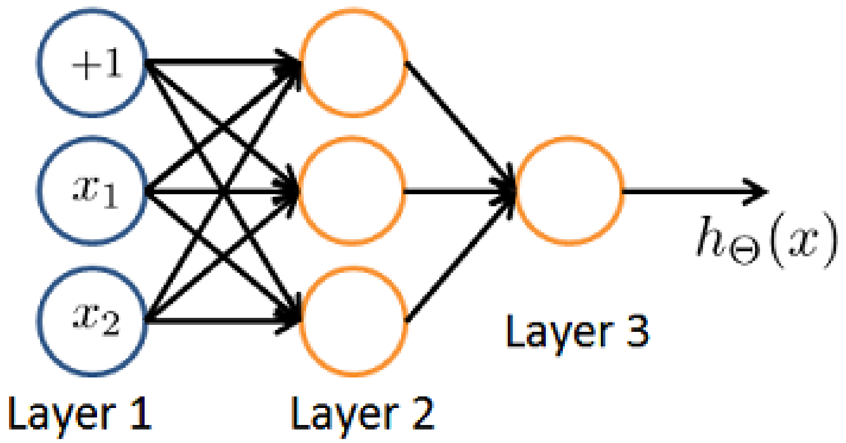
$$a_1^{(3)} = g(\Theta_{1,0}^{(2)} a_0^{(2)} + \Theta_{1,1}^{(2)} a_1^{(2)} + \Theta_{1,2}^{(2)} a_2^{(2)})$$

$$a_1^{(3)} = g(\Theta_{1,0}^{(2)} a_0^{(1)} + \Theta_{1,1}^{(2)} a_1^{(1)} + \Theta_{1,2}^{(2)} a_2^{(1)})$$

$$a_1^{(3)} = g(\Theta_{1,0}^{(1)} a_0^{(2)} + \Theta_{1,1}^{(1)} a_1^{(2)} + \Theta_{1,2}^{(1)} a_2^{(2)})$$

$$a_1^{(3)} = g(\Theta_{2,0}^{(2)} a_0^{(2)} + \Theta_{2,1}^{(2)} a_1^{(2)} + \Theta_{2,2}^{(2)} a_2^{(2)})$$

You have the following neural network



You'd like to compute the activations of the hidden layer $a^{(2)} \in \mathbb{R}^3$. One way to do so is the follow
Octave code:

```
% Theta1 is Theta with superscript "(1)" from lecture
% ie, the matrix of parameters for the mapping from layer 1 (input) to layer 2
% Theta1 has size 3x3
% Assume 'sigmoid' is a built-in function to compute 1 / (1 + exp(-z))

a2 = zeros (3, 1);
for i = 1:3
  for j = 1:3
    a2(i) = a2(i) + x(j) * Theta1(i, j);
  end
  a2(i) = sigmoid (a2(i));
end
```

You want to have a vectorized implementation of this (i.e., one that does not use for loops). Which of the following implementations correctly compute a^{(2)}a ? Check all that apply.

```
z = Theta1 * x; a2 = sigmoid (z);
```

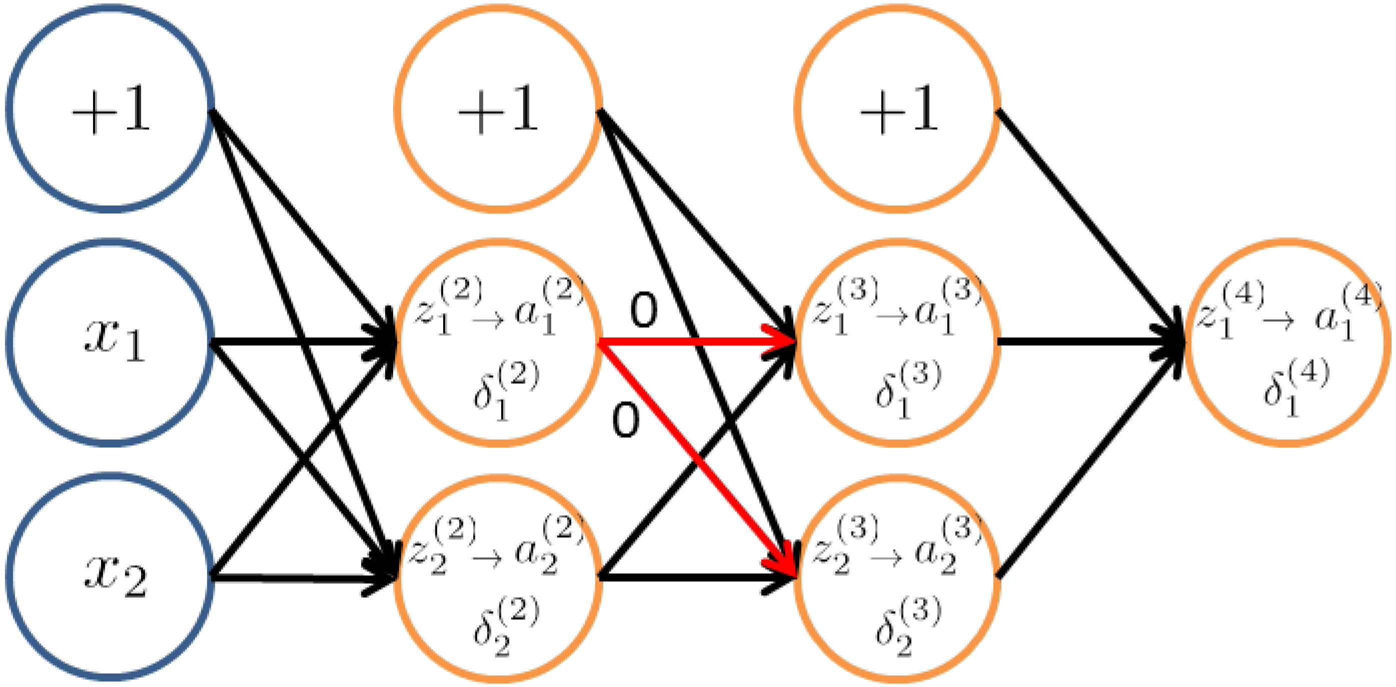
```
a2 = sigmoid (x * Theta1);
```

```
a2 = sigmoid (Theta2 * x);
```

```
z = sigmoid(x); a2 = sigmoid (Theta1 * z);
```

```
a2 = sigmoid (Theta1 * x);
```

Consider the following neural network:



$+1$
x_1
x_2
$+1$
$z_1^{(2)} \to a_1^{(2)}$
$\delta_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$
$\delta_2^{(2)}$
0
0
$+1$
$z_1^{(3)} \to a_1^{(3)}$
$\delta_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$
$\delta_2^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$
$\delta_1^{(4)}$

Suppose both of the weights shown in red ($\Theta_{11}^{(2)}$ and $\Theta_{21}^{(2)}$) are equal to 0. After running backpropagation, what can we say about the value of $\delta_1^{(3)}$?

- ○ $\delta_1^{(3)} > 0$
- ○ $\delta_1^{(3)} = 0$ only if $\delta_1^{(2)} = \delta_2^{(2)} = 0$, but not necessarily otherwise
- ○ $\delta_1^{(3)} \leq 0$ regardless of the values of $\delta_1^{(2)}$ and $\delta_2^{(2)}$
- ○ There is insufficient information to tell

Any Questions ?