Neural Networks

3007/7059 Artificial Intelligence

School of Computer Science
The University of Adelaide

Introduction

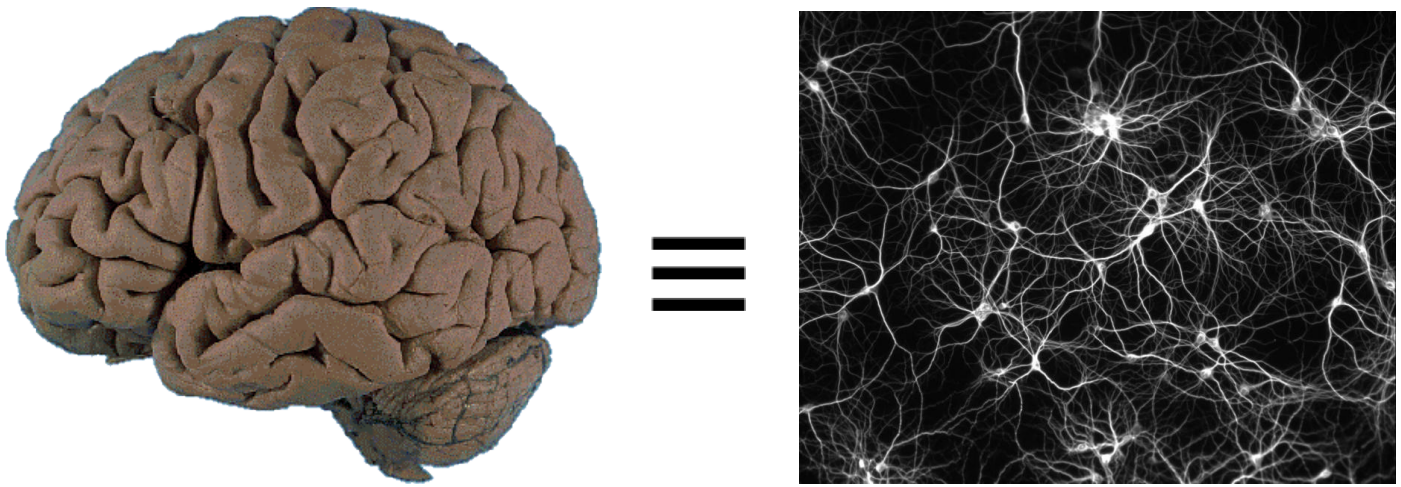
Artificial Neural Networks are a type of biologically inspired information processing systems.

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.

Brains

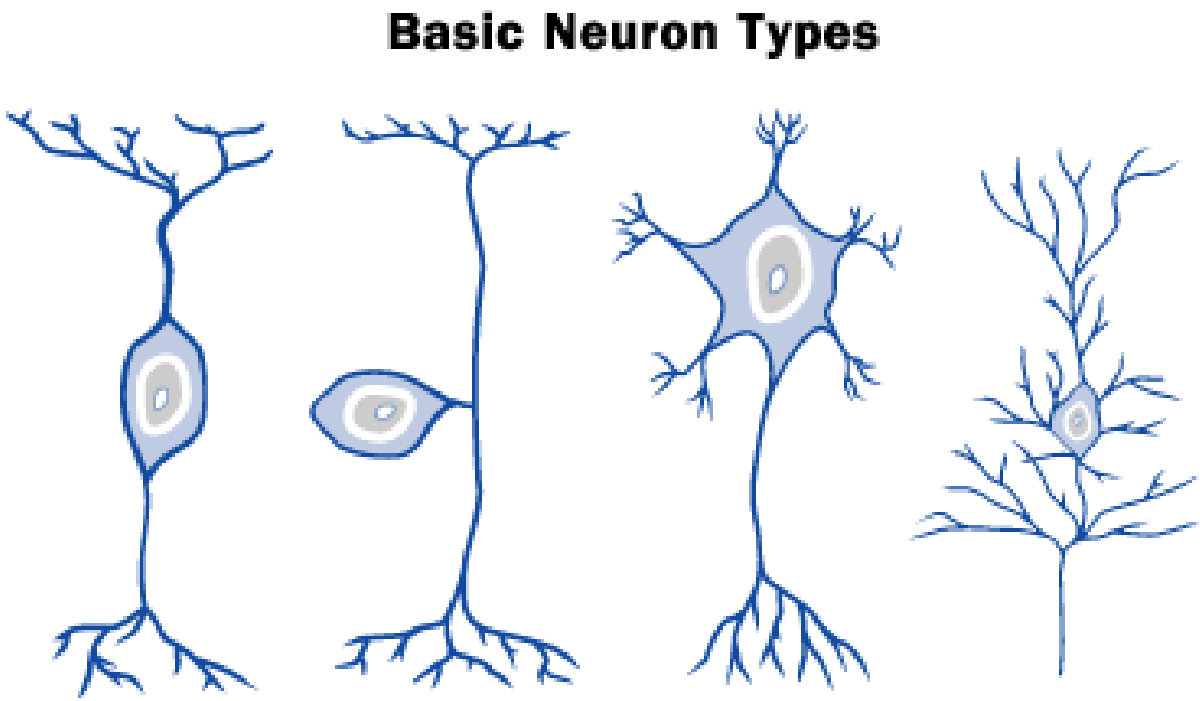
The inspiration behind Artificial Neural Networks is the brain. The brain is composed of cells called **neurons** which are interconnected to form a massive network:



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

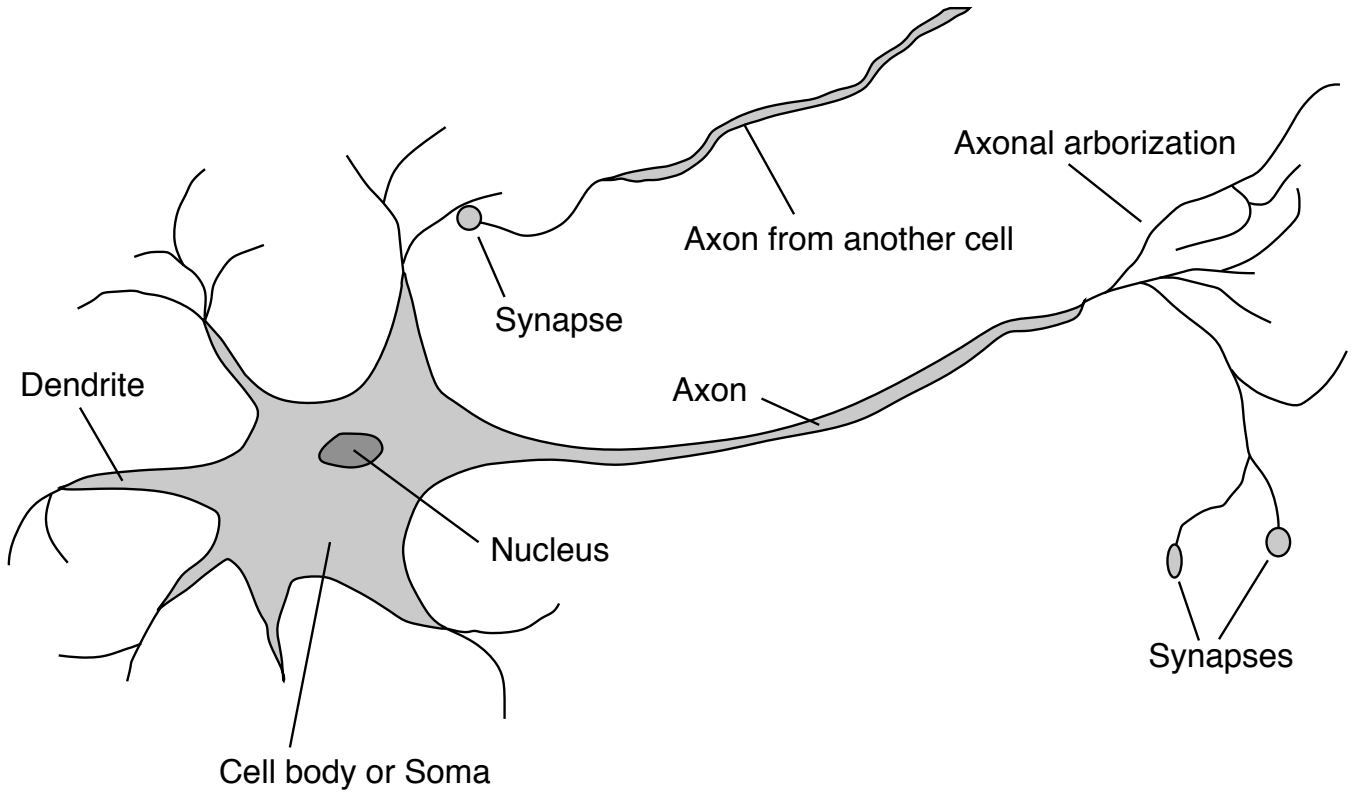
Neurons

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.

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

Comparing the brain and the computer

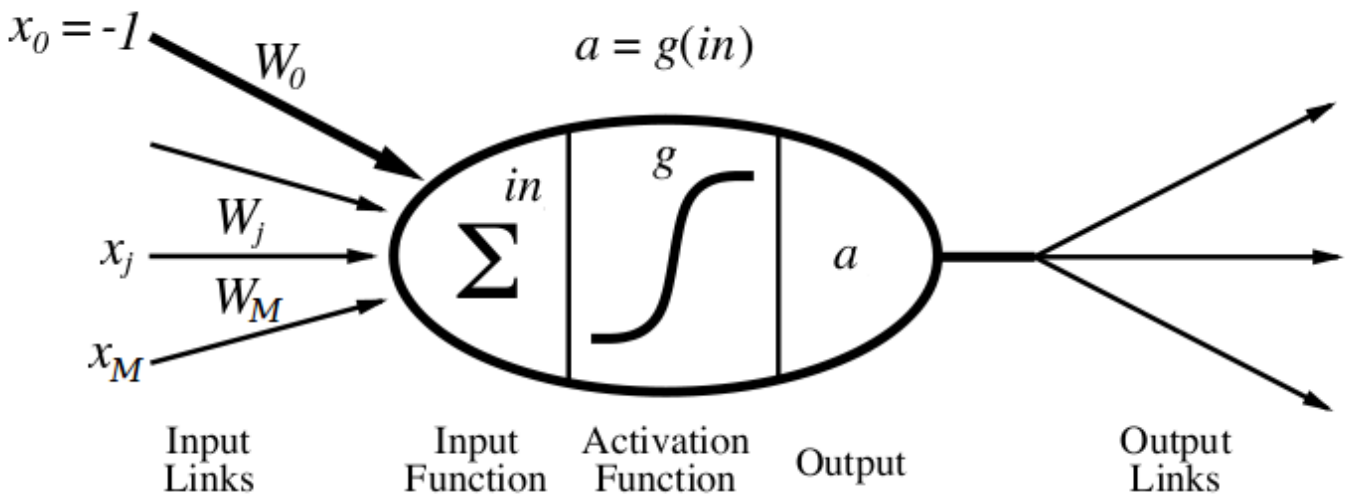
In the human brain, there are 10^{11} neurons of more than 20 types and 10^{14} synapses with 1ms – 10ms cycle time. Here's a comparison of the human brain and a modern computer:

	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

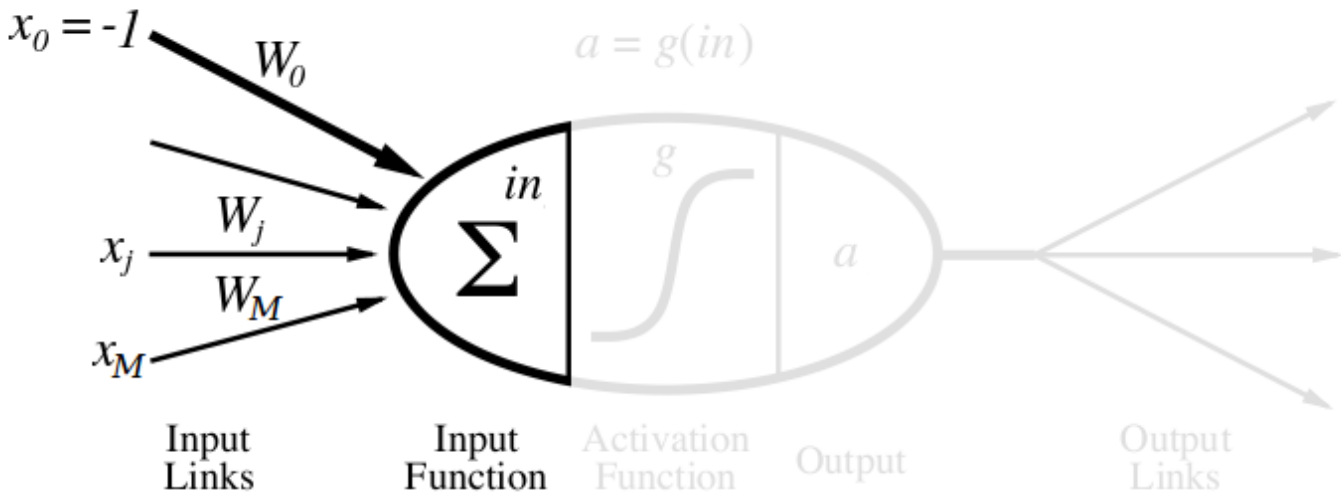
A mathematical model of the neuron due to McCulloch and Pitts:



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

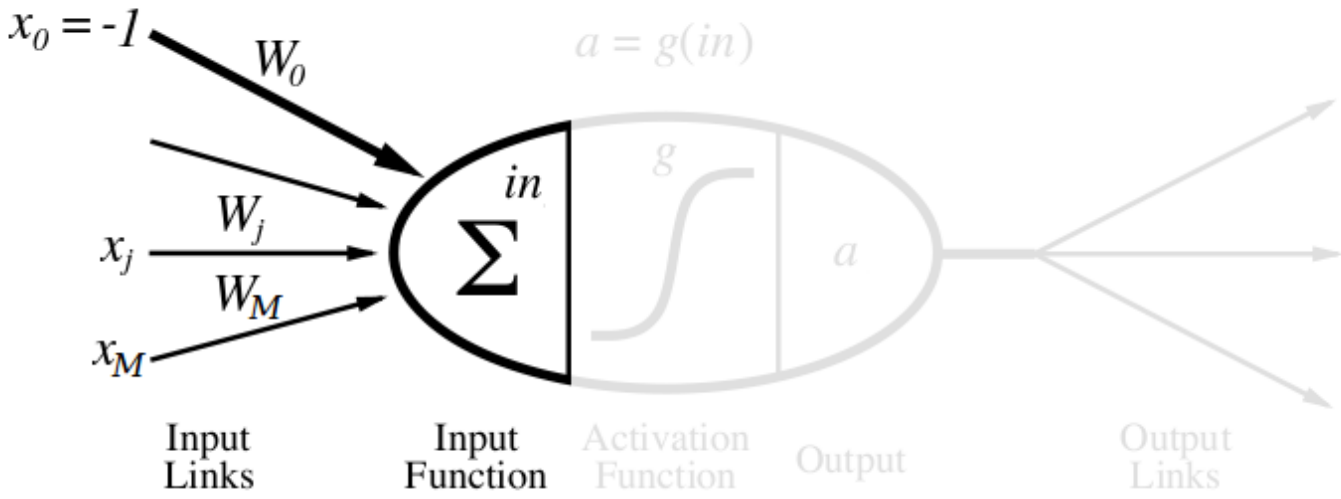
What it does in a nutshell: It fires when a **linear combination of its inputs exceeds some threshold**.

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 (cont.)

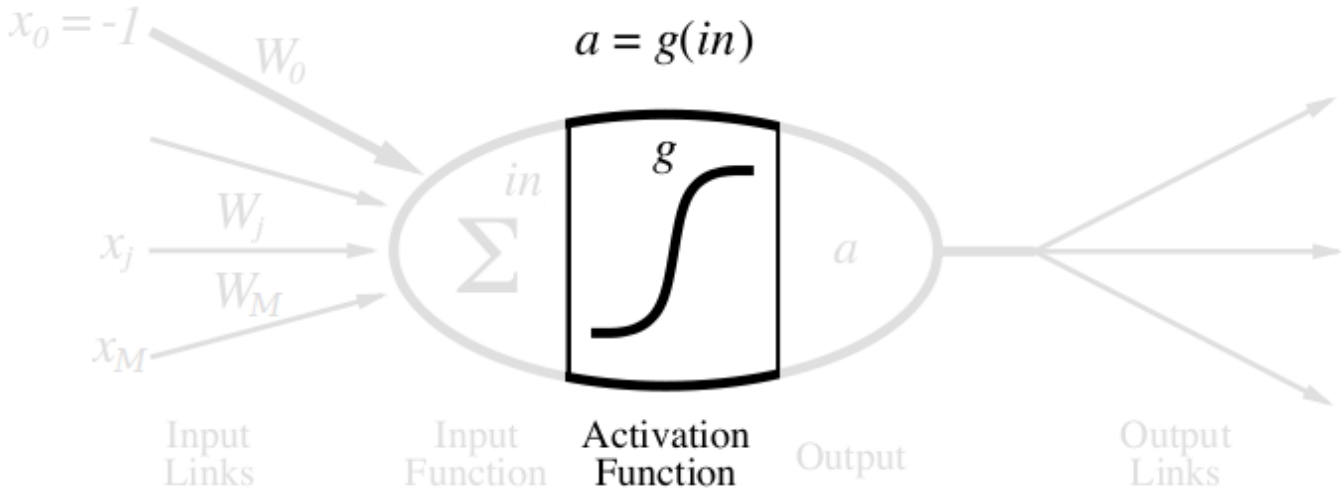


The inputs are multiplied with their corresponding connection weights and summed at the neuron:

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

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

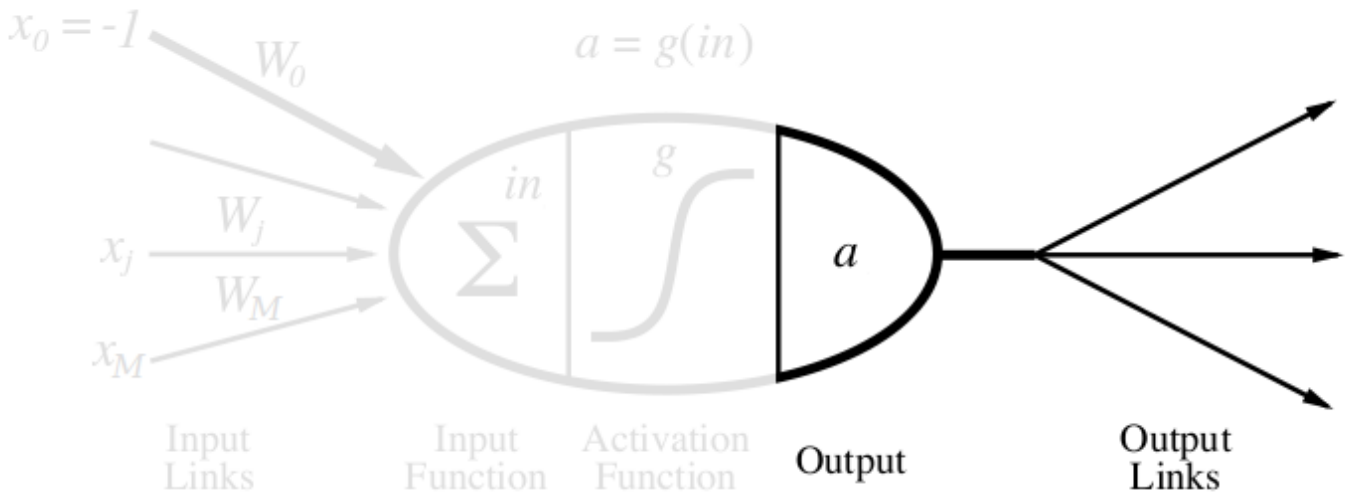


An **activation function** g is then applied to the weighted-sum inputs to yield the output:

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

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

Output

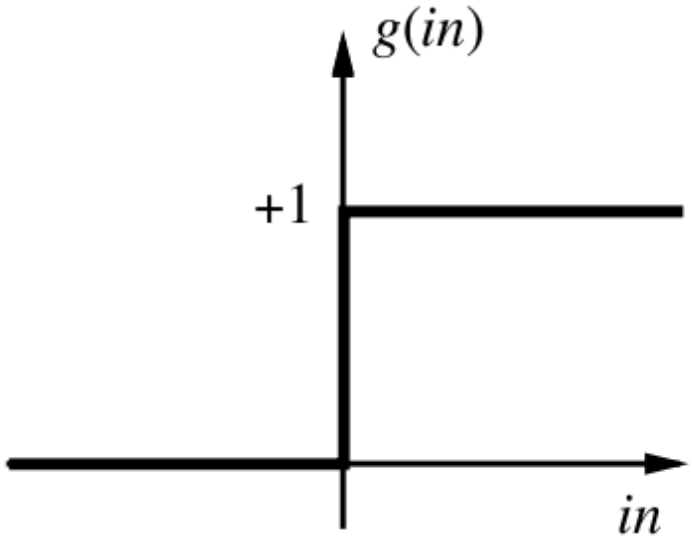


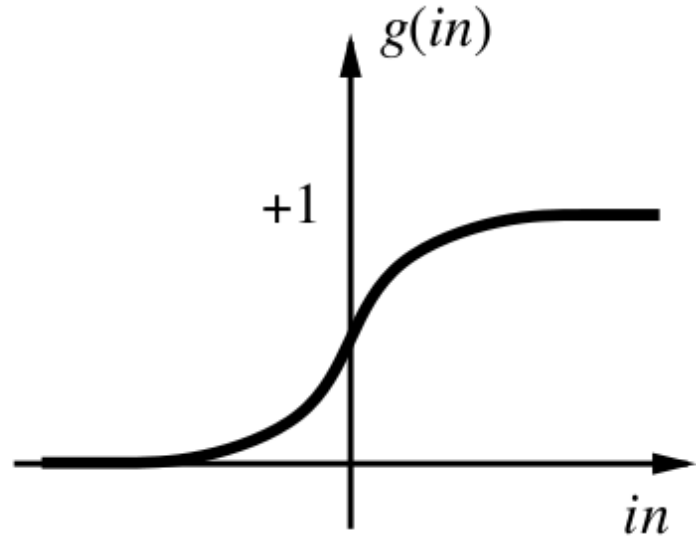
The output of the neuron can be taken as the overall output of the system or be transmitted to other neurons for further processing.

Activation functions

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:



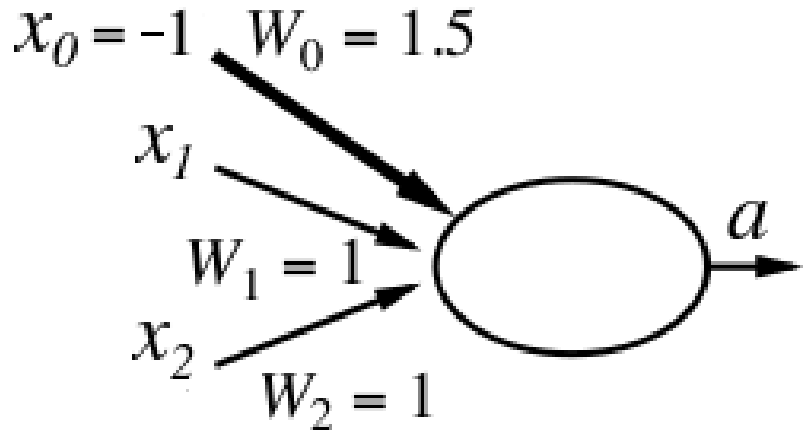


$$g_{th}(in) = \begin{cases} 1 & \text{if } \ in \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

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

Activation functions (cont.)

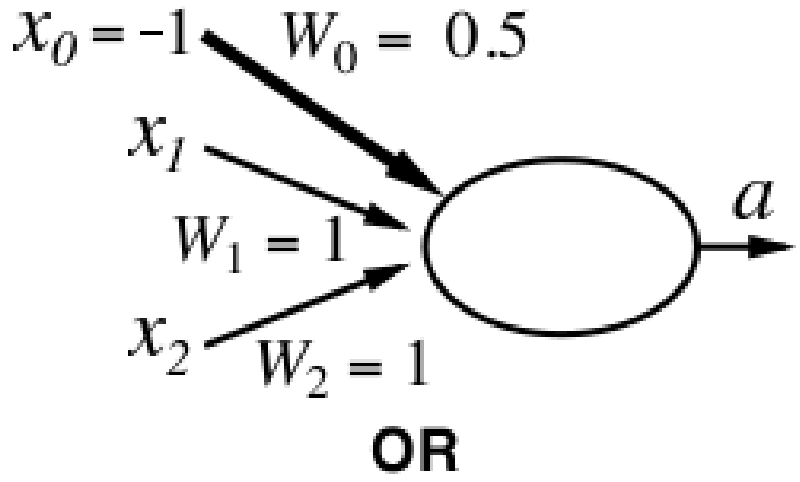
Example: Implementing Boolean functions with a neuron (of two different activation functions).



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 functions (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 $\implies$ weakly activated, output close to 1 $\implies$ strongly activated.

The bias weight

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

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

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**.

The bias weight (cont.)

Example: $x_1 = 0$, $x_2 = 1$ from the AND network. The weighted sum of x_1 and x_2 is

$$W_1 x_1 + W_2 x_2 = (0 \times 1) + (1 \times 1) = 1$$

which is smaller than the bias weight 1.5. Hence the neuron does not fire at all ($a = 0$) for the threshold function, or fires weakly ($a = 0.3775$) for the sigmoid function.

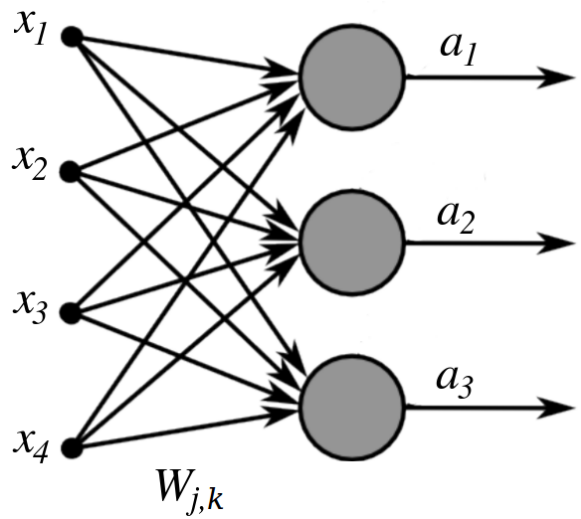
Example: $x_1 = 1$, $x_2 = 1$ from the OR network. The weighted sum of x_1 and x_2 is

$$W_1 x_1 + W_2 x_2 = (1 \times 1) + (1 \times 1) = 2$$

which is larger than the bias weight 0.5. Hence the neuron is fully activated ($a = 1$) for the threshold function, or fires strongly ($a = 0.9241$) for the sigmoid function.

Perceptrons

Perceptrons (Rosenblatt, 1957): A **neural network** with only a single layer of neurons. Each input has a connection to every neuron in the network.



In this example there are four input values x_1, x_2, x_3 and x_4, and three neurons a_1, a_2 and a_3.

The weight $W_{j,k}$ connects the j-th input to the k-th neuron (for clarity, we omit drawing of the bias input $x_0 = -1$ and the associated weights $W_{0,k}$).

Training neural networks

The actual function represented by a neural network depends on the connection weights. Recall from the AND and OR networks which differ only in their weights.

Once the weights for a network are determined, using the neural network is trivial— Simply obtain the weighted-sum of the inputs and apply the activation functions.

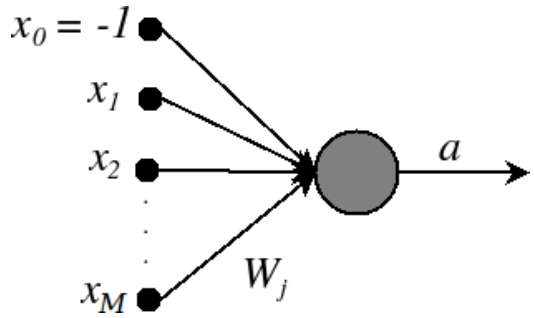
Training a neural network amounts to learning or estimating the connection weights.

This can be accomplished by showing the network a set of input and output pairs and modifying the weights accordingly — **supervised learning**.

The Perceptron Algorithm

Perceptron Algorithm: Supervised learning for Perceptrons.

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).

The Perceptron Algorithm (cont.)

Require: A training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$

1: Randomly initialise the weights $\{W_j\}_{j=0}^{M}$
2: **repeat**
3: **for** $i = 1, \ldots, N$ **do**
4: $in \longleftarrow \sum_{j=0}^{M} W_j x_{i,j}$
5: $Err \longleftarrow y_i - g(in)$
6: **for** $j = 0, \ldots, M$ **do**
7: $W_j \longleftarrow W_j + \alpha \times Err \times g'(in) \times x_{i,j}$
8: **end for**
9: **end for**
10: **until** maximum number of epochs
11: **return** Perceptron with learned weights

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**.

The Perceptron Algorithm (cont.)

Here, g' is the first derivative of the activation function g, where

$$g'_{sig}(in) = \frac{e^{-in}}{(1+e^{-in})^2} \qquad \text{and} \qquad g'_{th}(in) = 1$$

The principle behind the Perceptron Algorithm is simple:

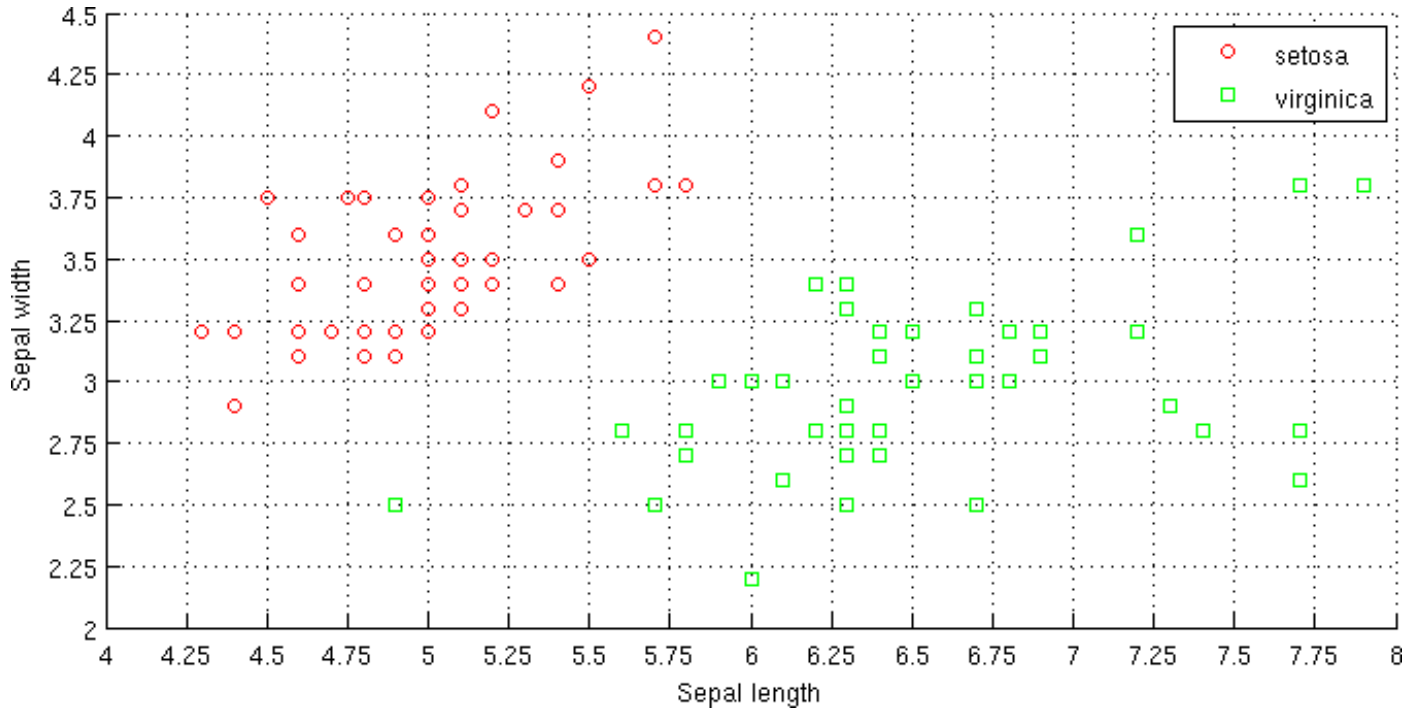
$$W_j \longleftarrow W_j + \alpha \times Err \times g'(in) \times x_{i,j}$$

If the error $Err = y_i - g(in)$ is positive, the output for the i-th example is too small, thus the weight is increased. The opposite happens if the error is negative, i.e. the weight is decreased.

Parameter α is the **learning rate** of the algorithm which controls how fast the algorithm converges. Setting the correct α is crucial: Too low $\implies$ training takes too long. Too high $\implies$ algorithm behaves erratically with significant transients.

The Iris data

We can train a Perceptron to classify (a portion of) the Iris data.



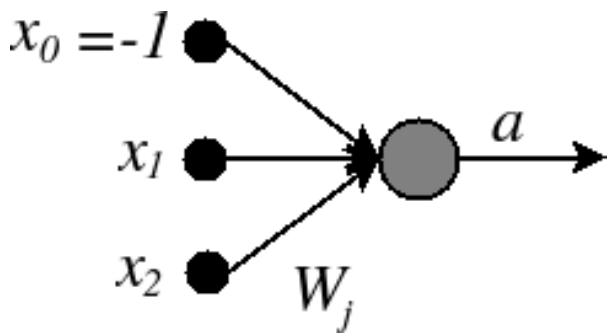
The Iris data (cont.)

We use $Setosa$ as the positive class ($y_i = 1$) and $Virginica$ as the negative class ($y_i = 0$). This can be reversed with no implications to the training (apart from the reversed outputs).

Here's how the training set looks like:

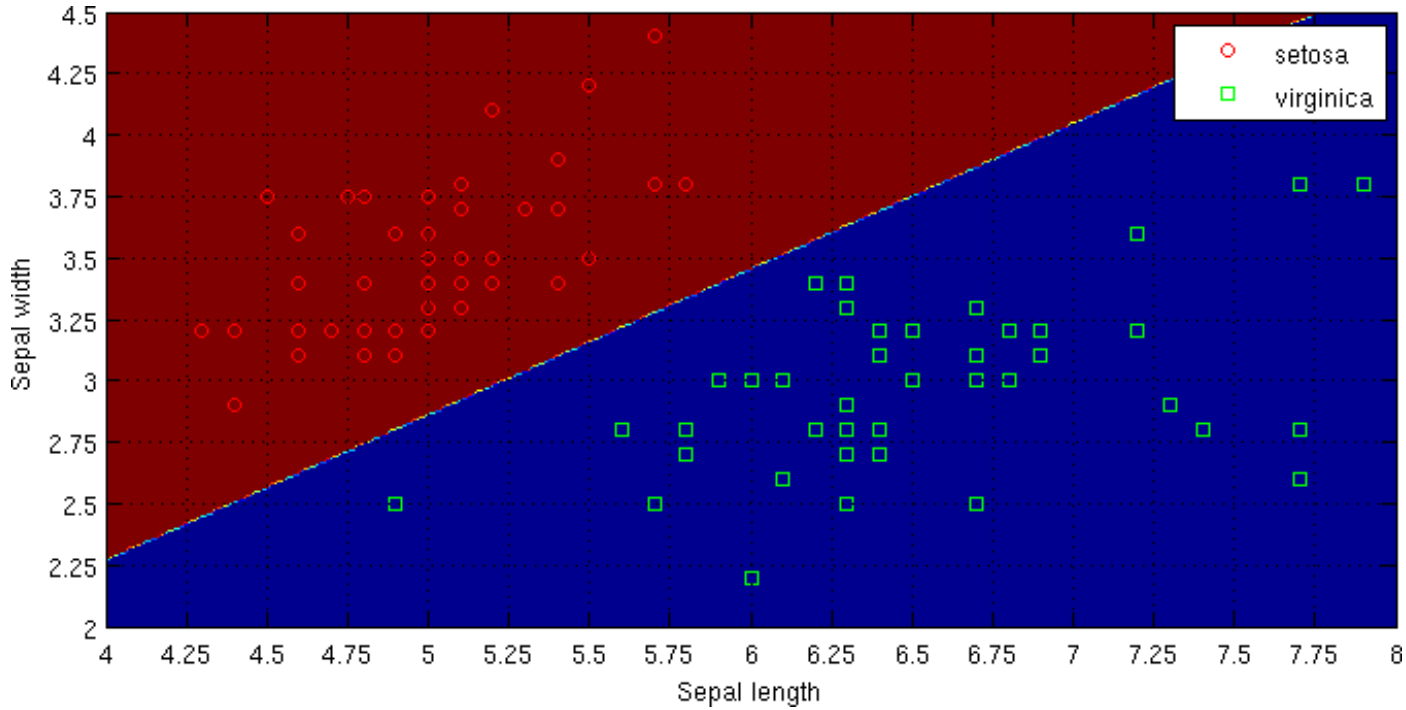
	Attributes		*Label*
	$Sepal\ length, x_{i,1}$	$Sepal\ width, x_{i,2}$	y_i
$\mathbf{x}_1$	*4.3*	*3.2*	*1*
$\mathbf{x}_2$	*4.4*	*2.9*	*1*
$\mathbf{x}_3$	*6.2*	*3.4*	*0*
$\mathbf{x}_4$	*4.5*	*2.3*	*1*
$\mathbf{x}_5$	*6.1*	*2.6*	*0*
$\mathbf{x}_6$	*6.1*	*3.0*	*0*
$\mathbf{x}_7$	*4.4*	*3.2*	*1*
$\mathbf{x}_8$	*6.2*	*2.8*	*0*
⋮	⋮	⋮	⋮

The corresponding Perceptron is relatively simple:



The Iris data (cont.)

Here are the decision regions after ≈ 25 epochs on the Perceptron Algorithm (using the threshold activation function).



Observe that the two classes are cleanly separated. The network weights after the training terminates are $W_0 = -1.9293$, $W_1 = -11.2857$, $W_2 = 19.0435$.

Multiclass classification

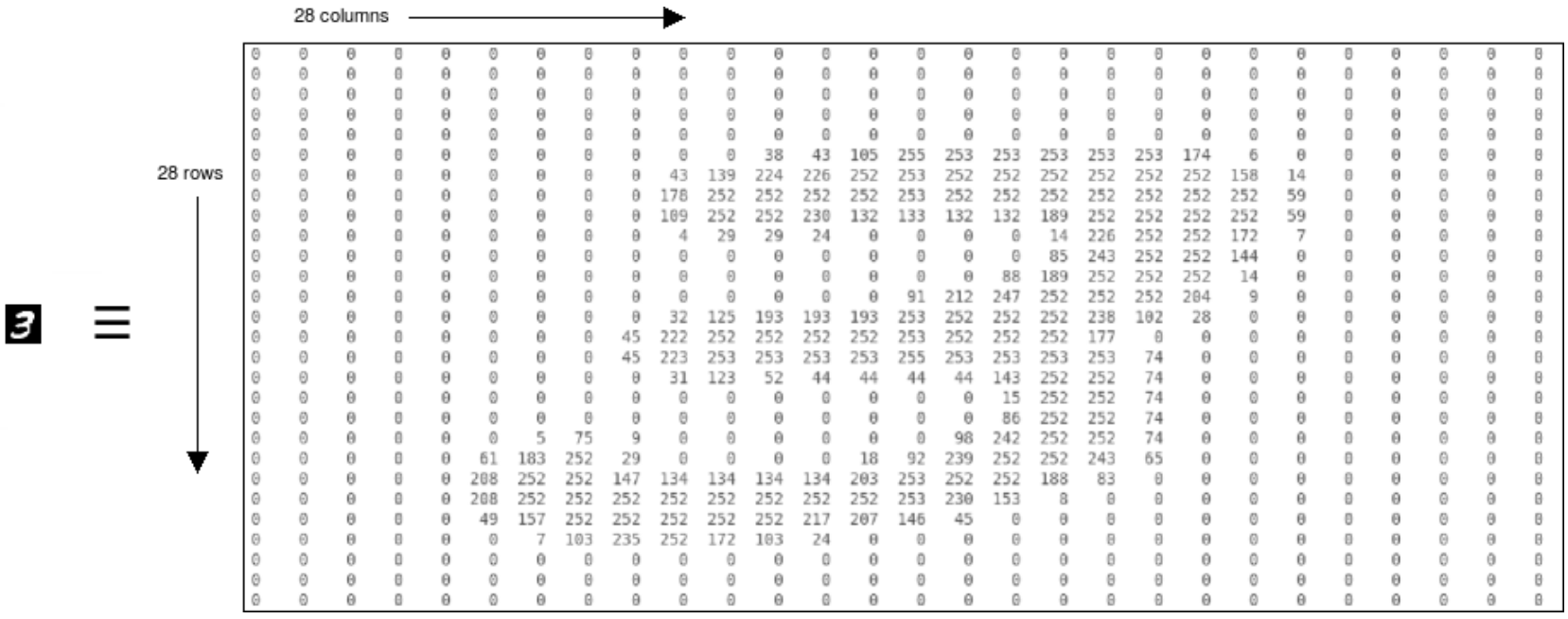
The Perceptron Algorithm deals with binary classification problems. What if we have a **multiclass** problem?

For example, handwritten digit recognition:

How do we train a Perceptron to do it?

Multiclass classification (cont.)

First, we have to convert the data into a form that is useful for a Perceptron. To this end we note that a digital image is a matrix of pixels, where each pixel has a value that is proportional to the pixel brightness. For example, a 28×28 pixel image of a digit-3.



We **raster scan** each digital image, i.e. take values one-by-one from the image column-first and insert them into a vector of length 784 $(= 28 \times 28)$.

Multiclass classification (cont.)

Now we have a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ where class label $y_i \in \{0, 1, \ldots, 9\}$.

Each $\mathbf{x}_i$ is a rasterised image vector appended with -1 for the bias.

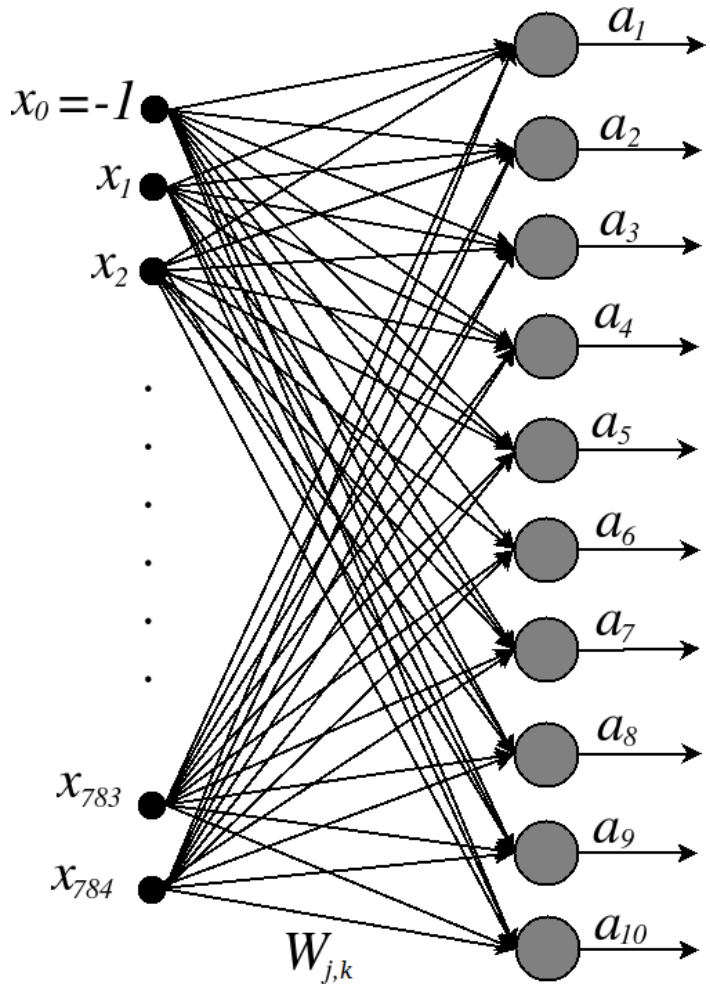
We convert each y_i into a 10-dimensional vector $\mathbf{y}_i$ defined as

$$\begin{aligned}
\mathbf{y}_i &= [\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ]^T, \quad \text{if } y_i = 0 \\
\mathbf{y}_i &= [\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ]^T, \quad \text{if } y_i = 1 \\
&\vdots \\
\mathbf{y}_i &= [\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ ]^T, \quad \text{if } y_i = 9.
\end{aligned}$$

This is called **one-of-K coding scheme**.

Multiclass classification (cont.)

We then construct and train a Perceptron with 10 neurons $a_1, a_2, \ldots, a_{10}$, **one for each of the digits** to be classified:



Given a sample $\mathbf{x}_i$, the target label for each neuron is given by $\mathbf{y}_i$, i.e. 1 at the corresponding digit and 0 elsewhere.

Multiclass classification (cont.)

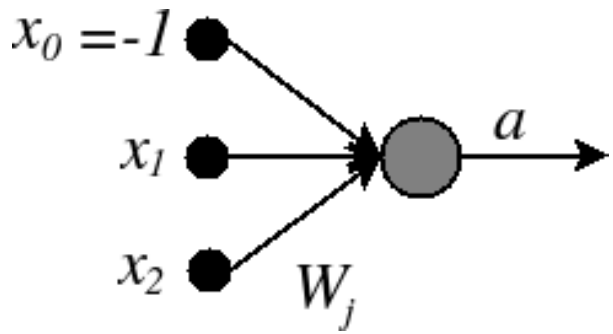
Observe that weights $W_{j,k}$ are **not shared by the neurons**—Thus we can simply run the Perceptron Algorithm **separately on each neuron** to only update the weights linked to it!

Require: A training set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ with 1-of-K coding scheme for the target labels.

1: Randomly initialise the weights of the Perceptron with K neurons.
2: **for** $k = 1, \ldots, K$ **do**
3: $\quad y_i \leftarrow \mathbf{y}_i[k]$ for $i = 1, \ldots, N$.
4: $\quad$ Run Perceptron Algorithm using $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ to update weights $\{W_{j,k}\}_{j=0}^M$.
5: **end for**
6: **return** Perceptron with learned weights

Convergence of Perceptron Algorithm

From the results on the Iris data, the weights are $W_0 = -1.9293$, $W_1 = -11.2857$ and $W_2 = 19.0435$ for the Perceptron

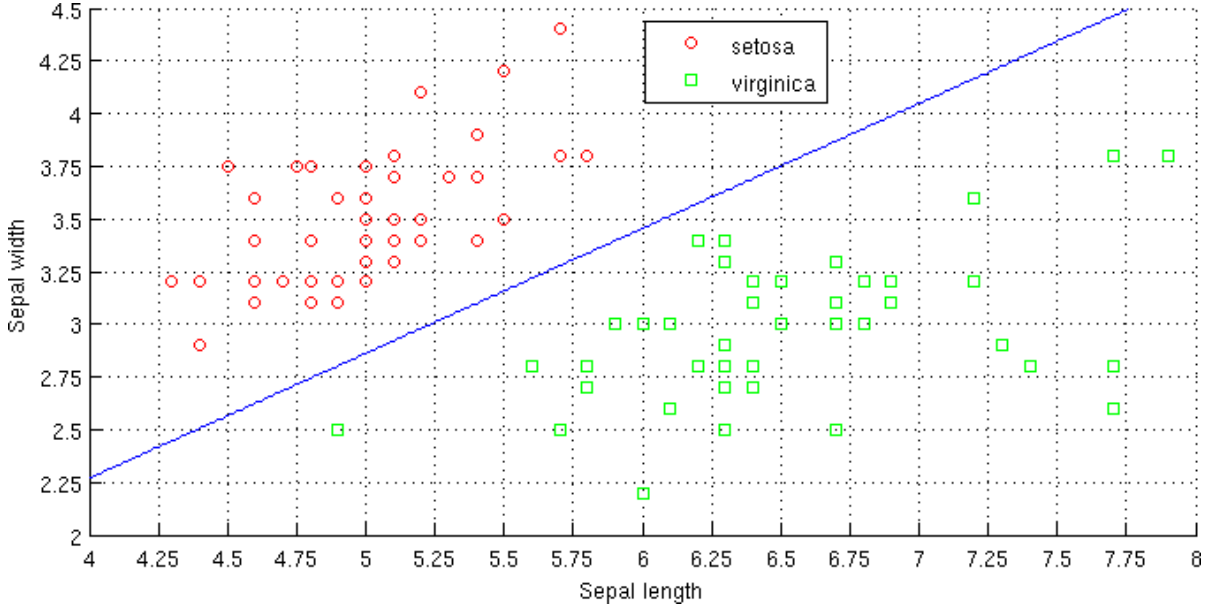


Thus the output of the Perceptron is evaluated as the following equation:

$$a = g(in) = g(-11.2857x_1 + 19.0435x_2 + 1.9293)$$

Convergence of Perceptron Algorithm (cont.)

The equation $-11.2857x_1 + 19.0435x_2 + 1.9293 = 0$ is simply the line separating the two classes!



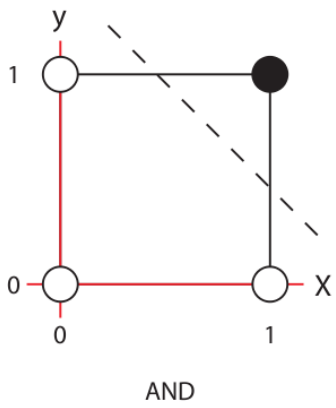
In general, this is true of Perceptrons, i.e.,

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

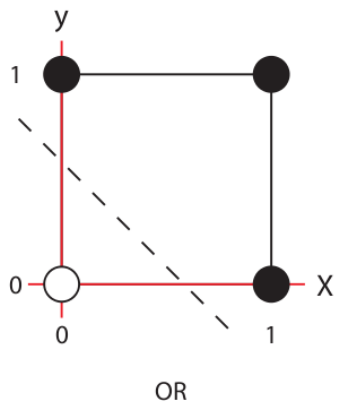
defines a linear hyperplane in the space of M dimensions.

Convergence of Perceptron Algorithm (cont.)

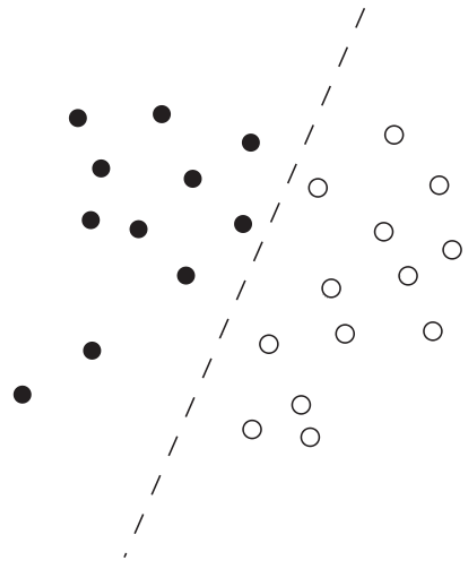
Thus data that can be separated by a linear hyperplane can be solved using Perceptrons. Such classification problems are called **linearly separable** problems.



AND



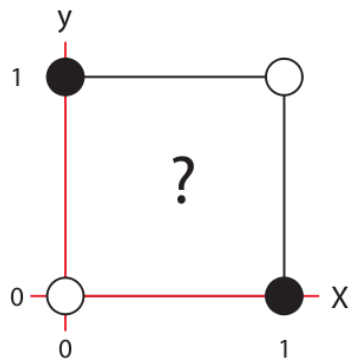
OR



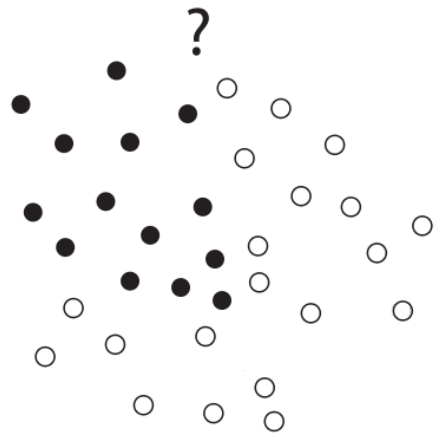
If the problem is linearly separable, the Perceptron Algorithm is guaranteed to converge in a finite number of iterations (Novikoff, 1962).

Limitation of Perceptrons

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



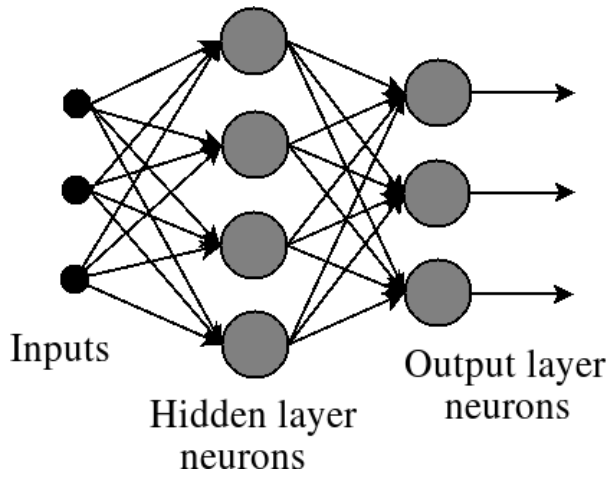
XOR



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

Multilayer Perceptrons

One approach to solve linearly nonseparable problems with Neural Networks is to use **Multilayer Perceptrons (MLP)**, i.e.



MLPs are more expressive than Perceptrons since they can learn highly non-linear class boundaries. However, this comes at the price of more complex training algorithms and difficulty in parameter tuning (e.g., how many layers? How many neurons per layer?).

What alternatives do we have?