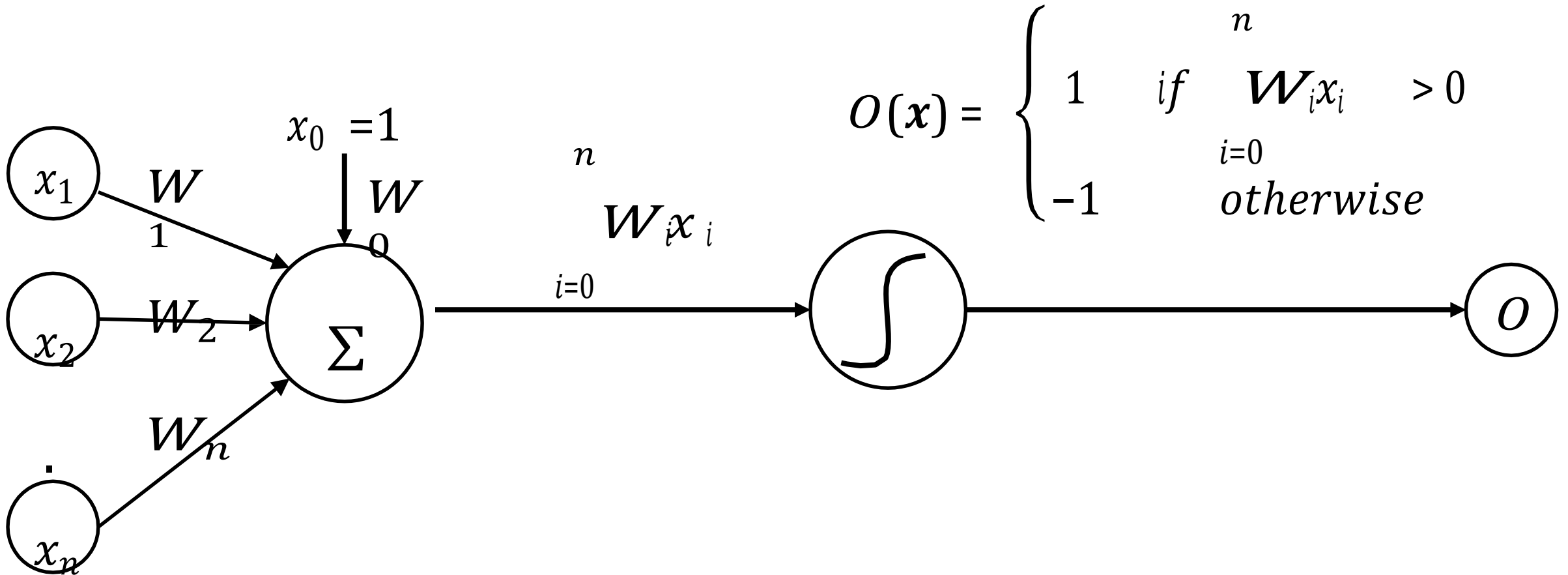


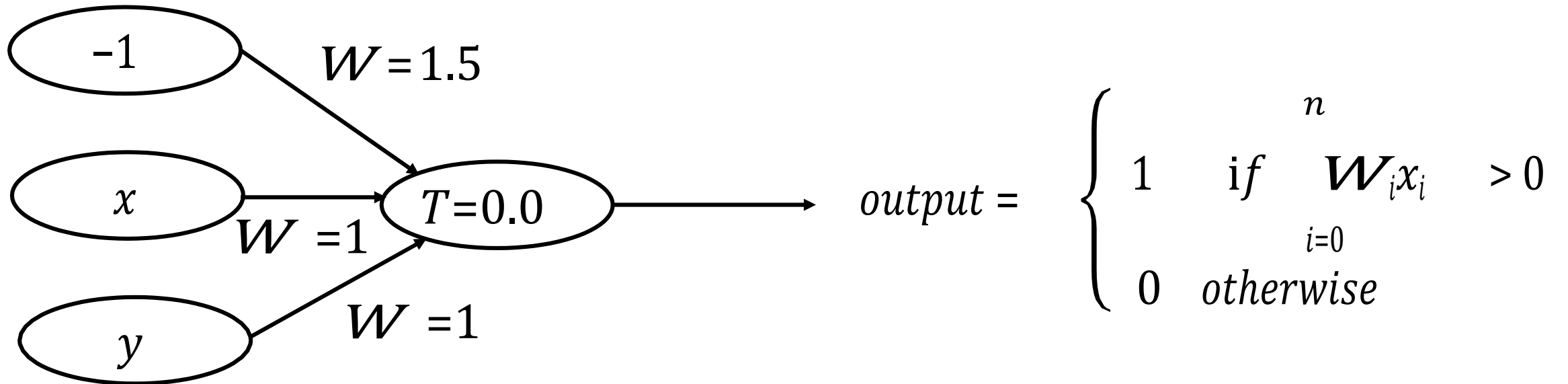
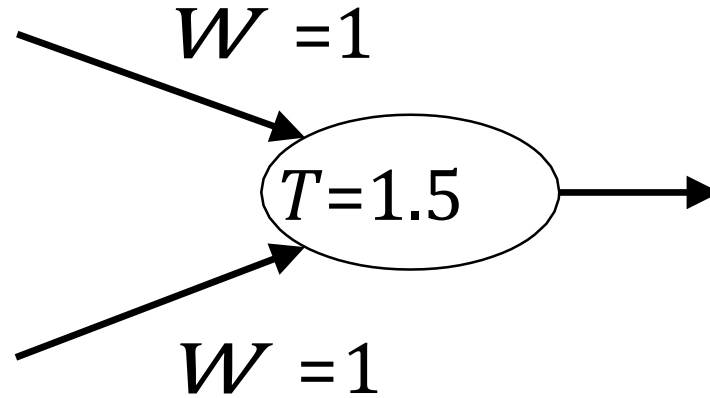
# Neural Networks

# Perceptron

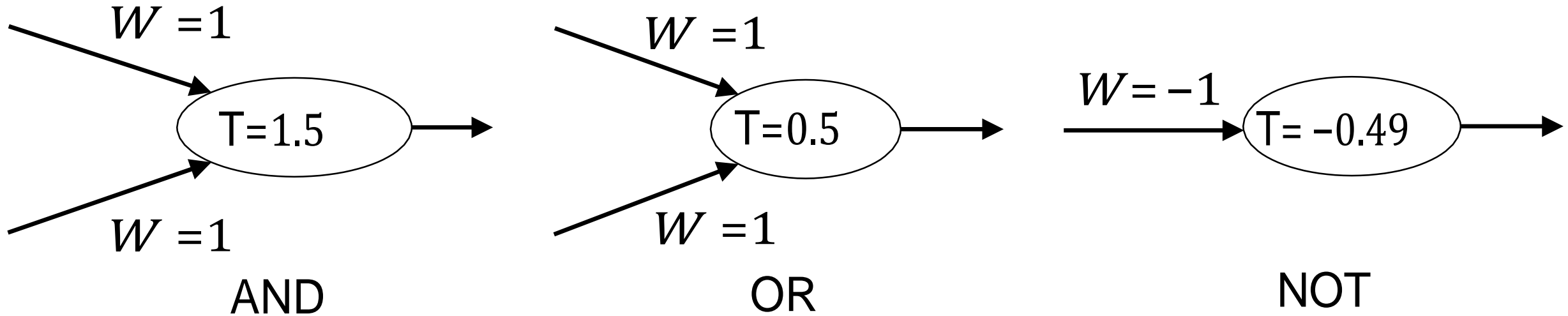
- Linear threshold unit (LTU)



# Simple network (AND)



# Perceptron

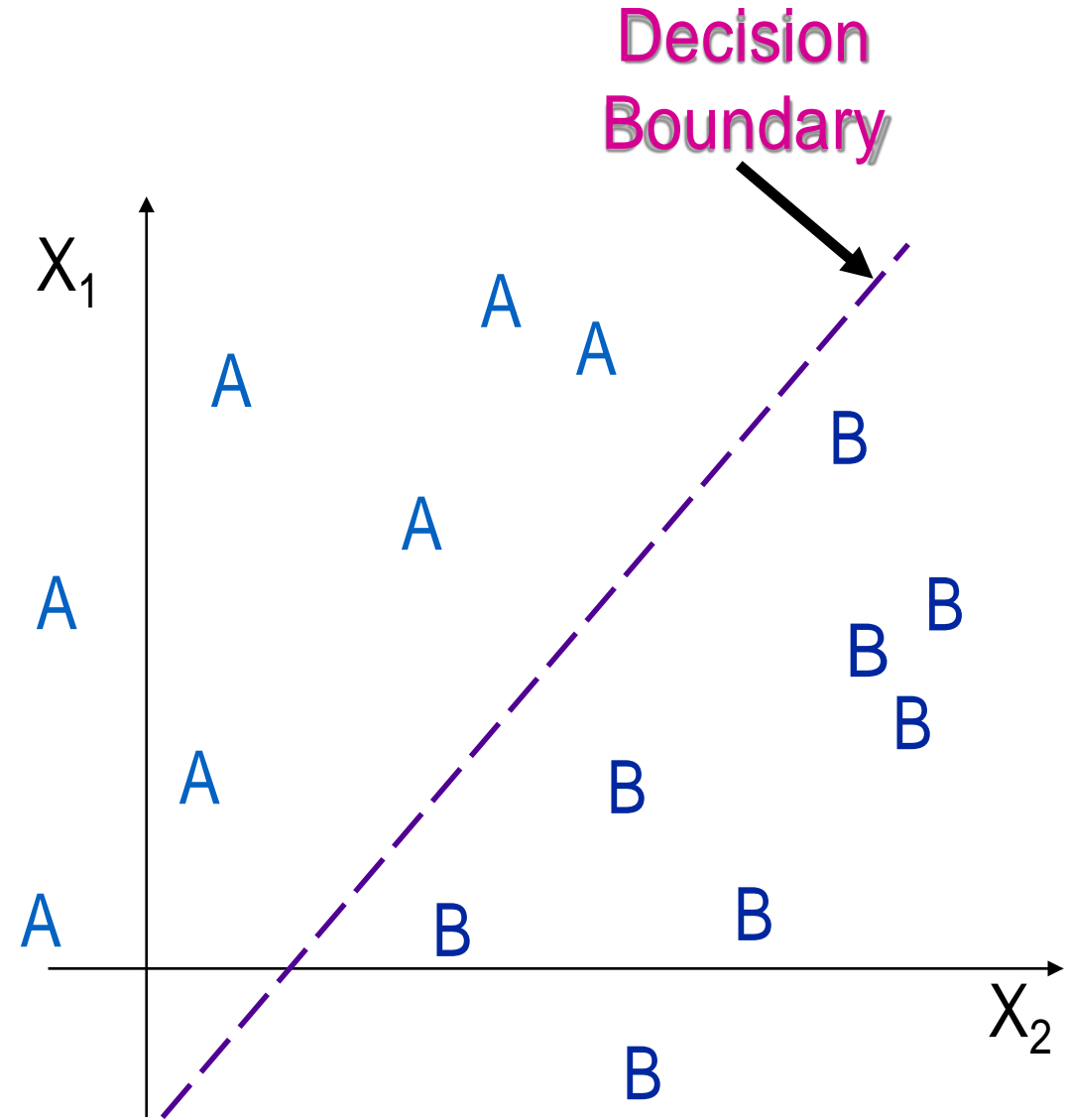


- Linear threshold is used.
- $W$  - weight value
- $t$  - threshold value

$$\text{output} = \begin{cases} 1 & \text{if } \sum_{i=0}^n \mathbf{W}_i x_i > T \\ 0 & \text{otherwise} \end{cases}$$

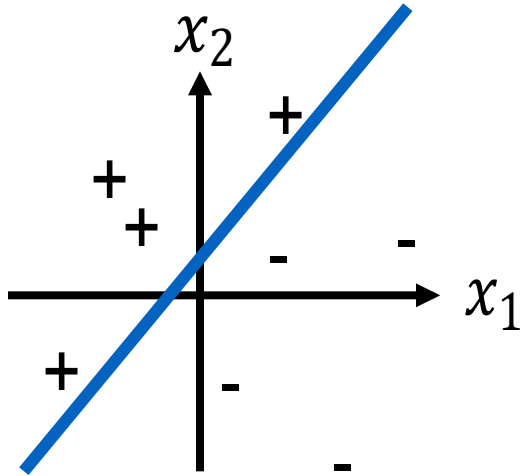
# Decision boundaries

- In simple cases, divide feature space by drawing a hyperplane across it.
- Known as a **decision boundary**.
- **Discriminant function**: returns different values on opposite sides. (straight line)
- Problems which can be thus classified are **linearly separable**.

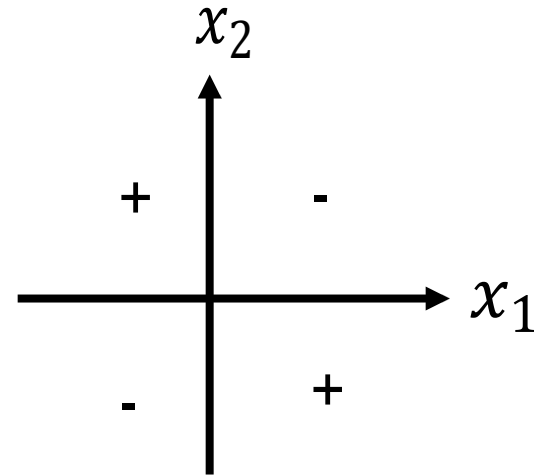


# Decision Surface of a Perceptron

- Perceptron is able to represent some useful functions
- $AND(x_1, x_2)$  choose weights  $w_0 = -1.5, w_1 = 1, w_2 = 1$
- But functions that are not linearly separable (e.g.  $XOR$ ) are not representable



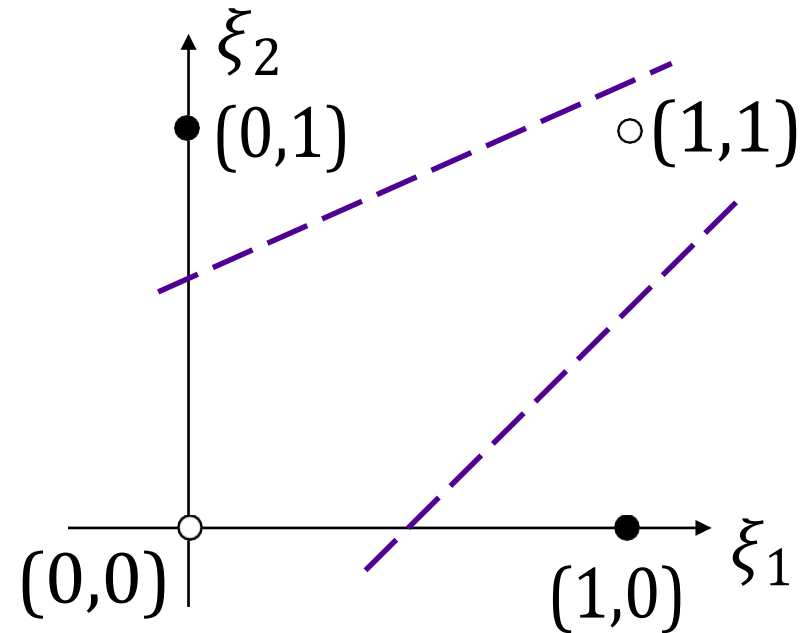
Linearly separable



Non-Linearly separable

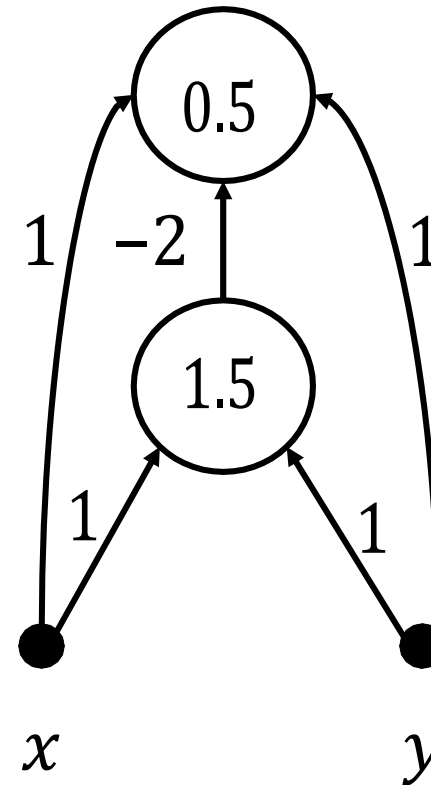
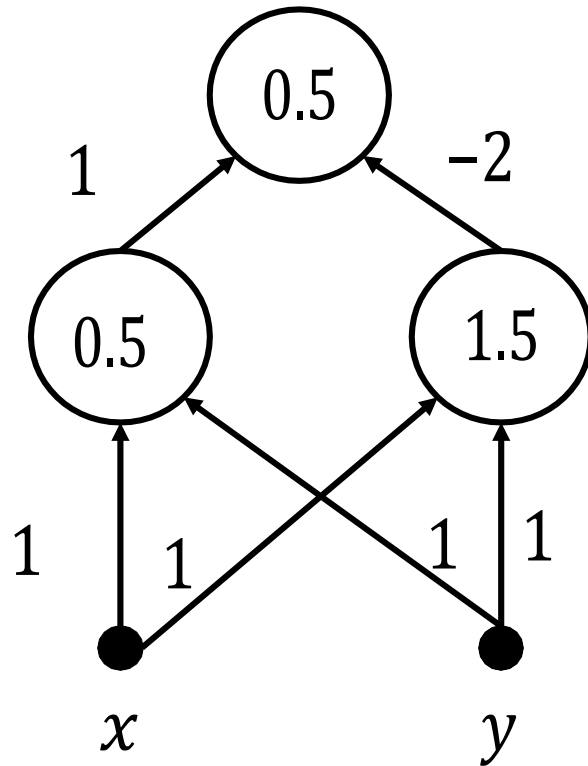
# Solution in 1980s: Multilayer Perceptrons

- Removes many limitations of single-layer networks
  - Can solve *XOR*
- **Exercise:** Draw a two-layer perceptron that computes the XOR function
  - 2 binary inputs  $\xi_1$  and  $\xi_2$
  - 1 binary output
  - One “hidden” layer
  - Find the appropriate weights and threshold



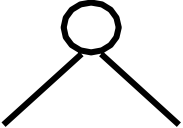
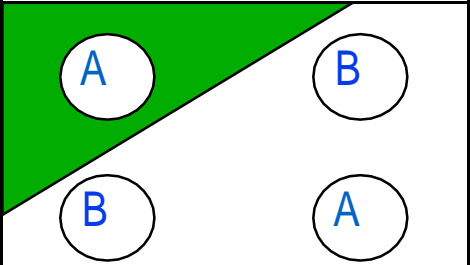
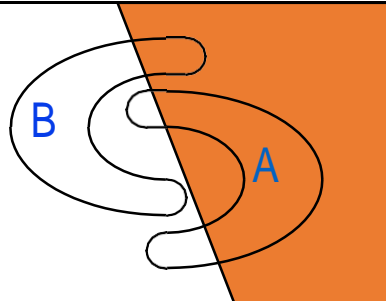
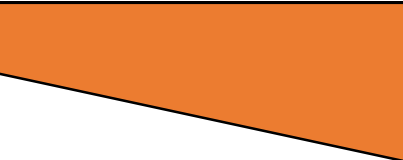
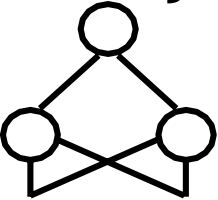
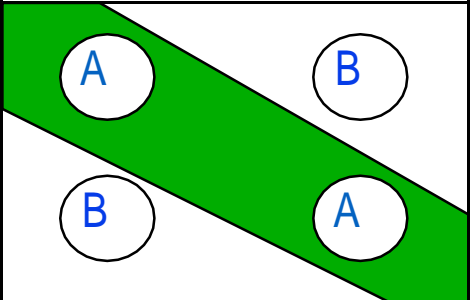
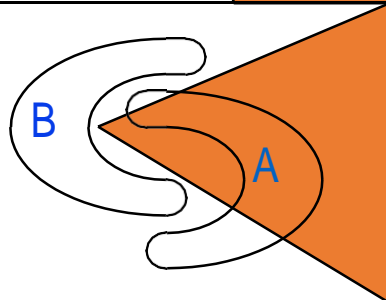
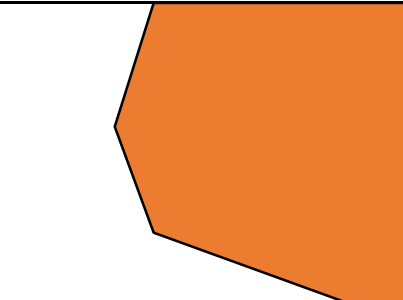
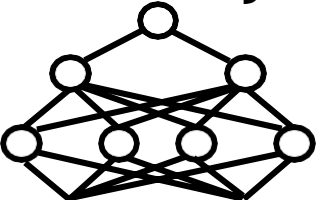
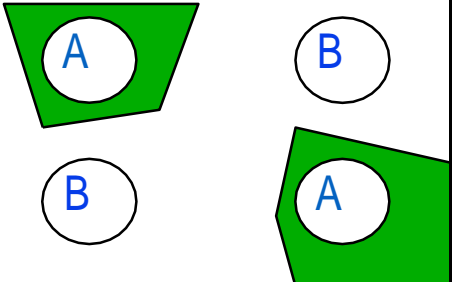
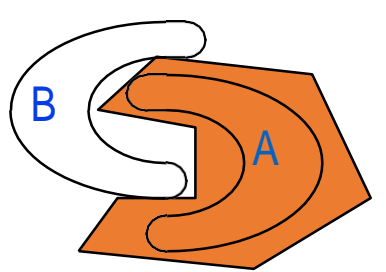
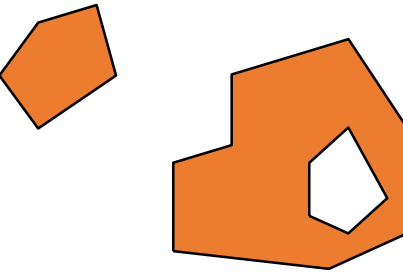
# Solution in 1980s: Multilayer Perceptrons

- Examples of two-layer perceptrons that compute *XOR*
  - E.g. **Right side network** Output is 1 if and only if  $x + y - 2(x + y - 1.5 > 0) - 0.5 > 0$
  - Left side network  $(x + y - 0.5 > 0) - 2(x + y - 1.5 > 0) - 0.5 > 0$

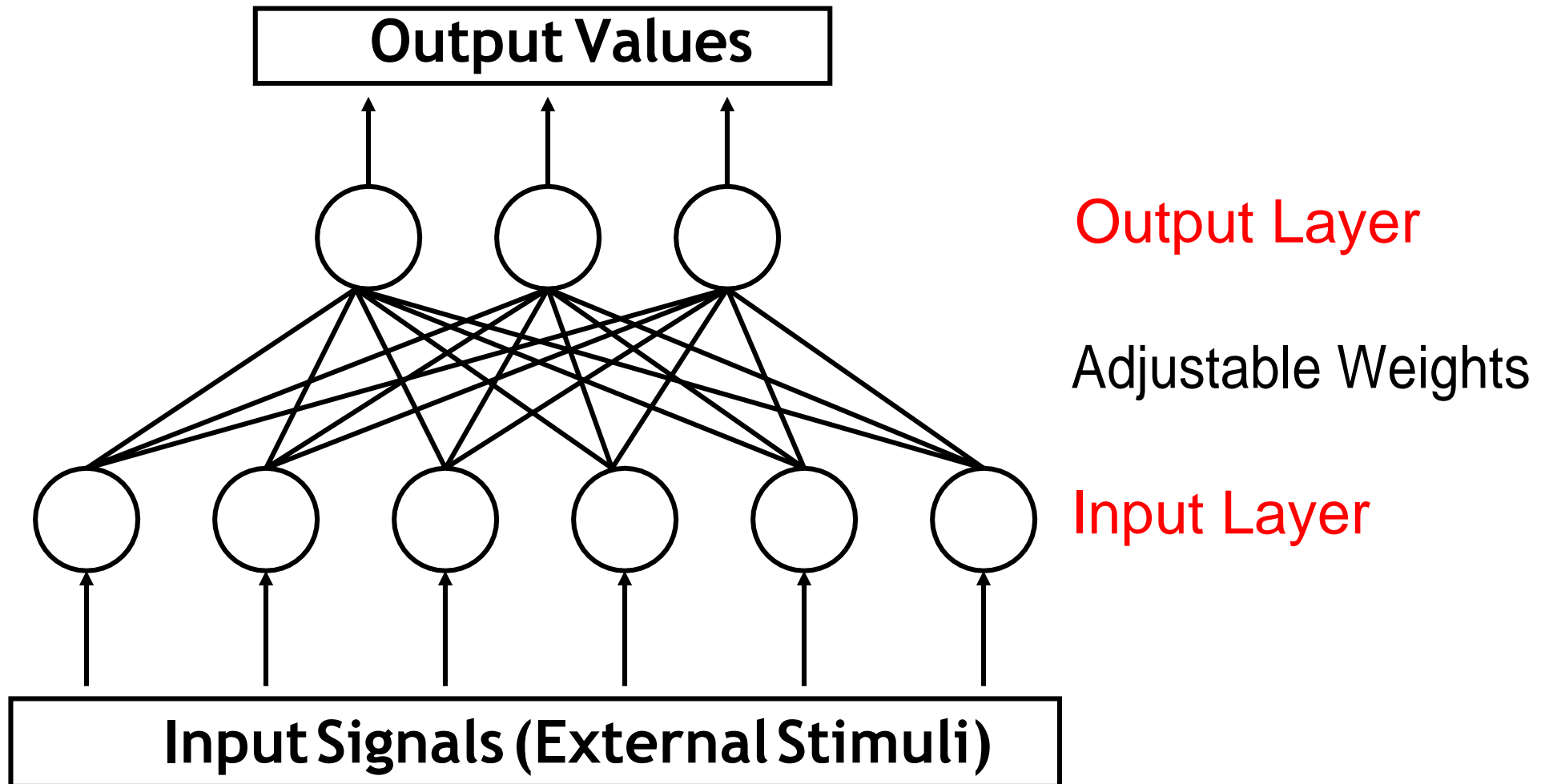




# Different Non-Linearly Separable Problems

<i>Structure</i>	<i>Types of Decision Regions</i>	<i>Exclusive-OR Problem</i>	<i>Classes with Meshed regions</i>	<i>Most General Region Shapes</i>
<i>Single-Layer</i> 	<i>Half Plane Bounded By Hyperplane</i>			
<i>Two-Layer</i> 	<i>Convex Open Or Closed Regions</i>			
<i>Three-Layer</i> 	<i>Arbitrary (Complexity Limited by No. of Nodes)</i>			

# Multilayer Perceptron (MLP)

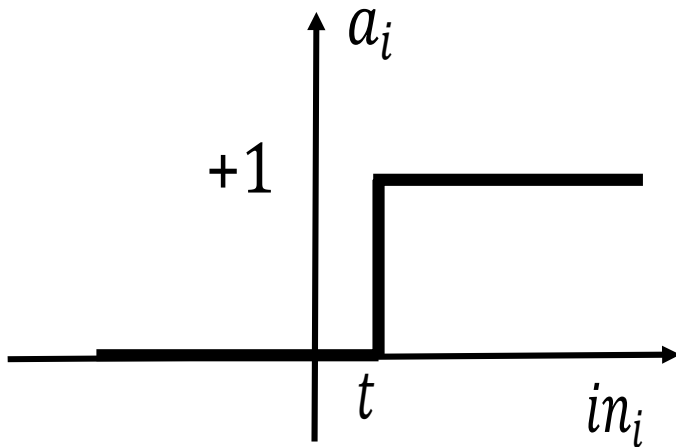


# Types of Layers

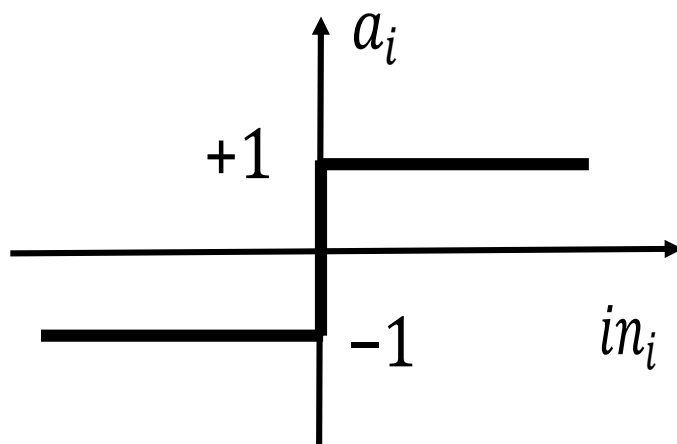
- The input layer.
  - Introduces input values into the network.
  - No activation function or other processing.
- The hidden layer(s).
  - Perform classification of features
  - Two hidden layers are sufficient to solve any problem
  - Features imply more layers may be better
- The output layer.
  -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

# Activation functions

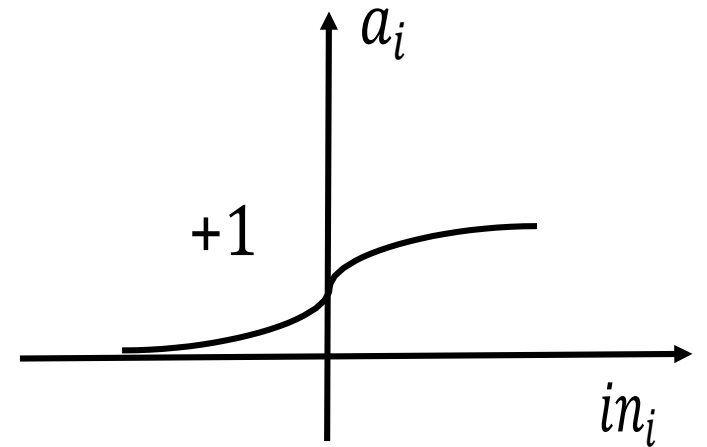
- Transforms neuron's input into output.
- Features of activation functions:
  - A squashing effect is required
    - Prevents accelerating growth of activation levels through the network.
  - Simple and easy to calculate



Step function



Sign function

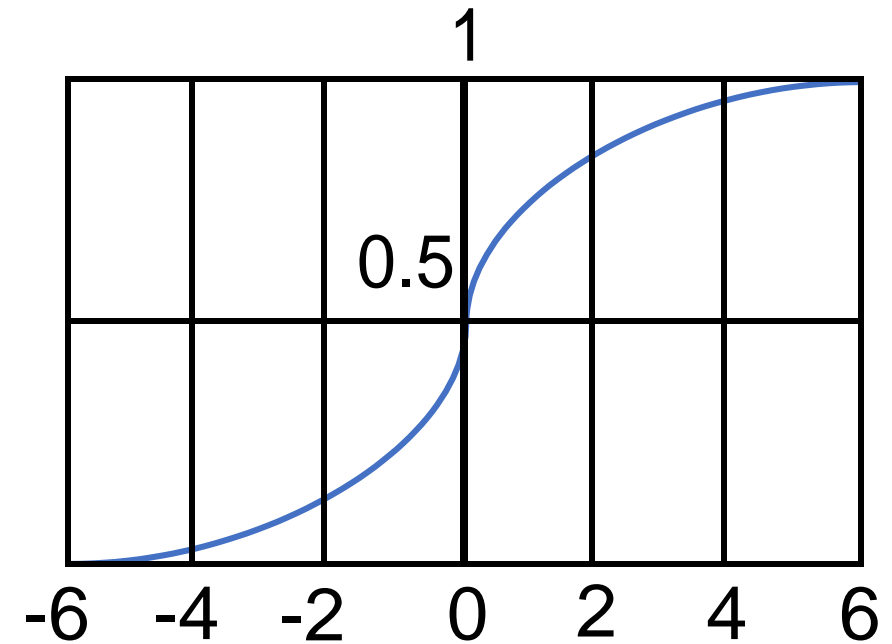


Sigmoid function

# Standard activation functions

- 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)
  - Both functions have a simple differential
  - Only the shape is important

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



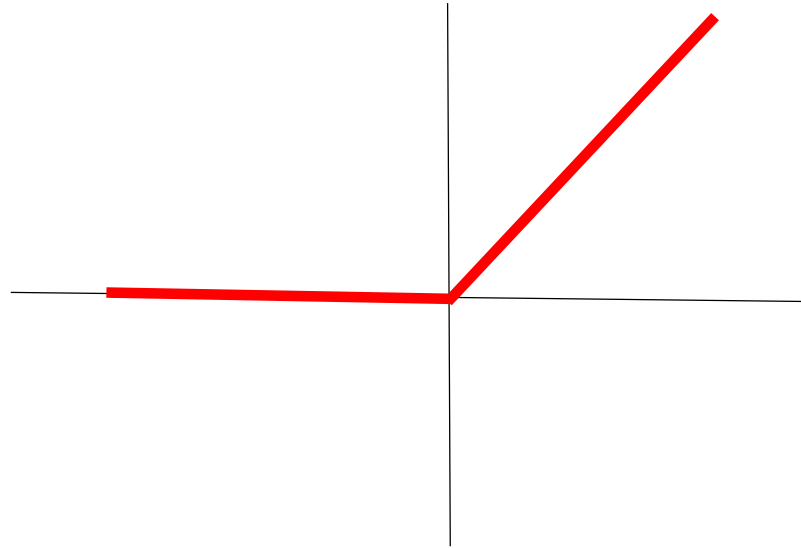
# Standard activation functions

- tanh

$$\frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

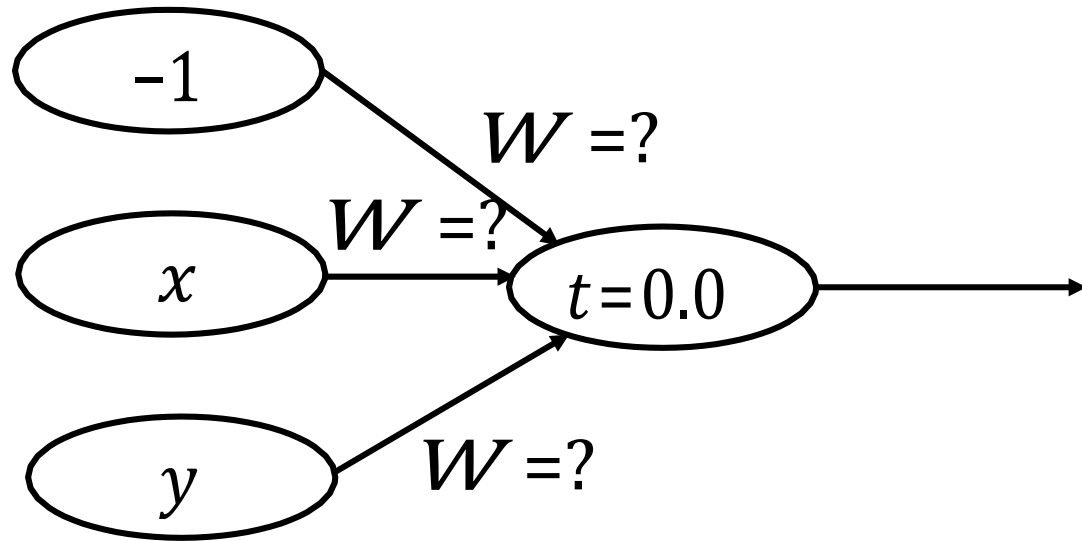
- relu

$$\text{relu}(z) = \max(0, z)$$



# Training Perceptrons

- What are the weight values?



For AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

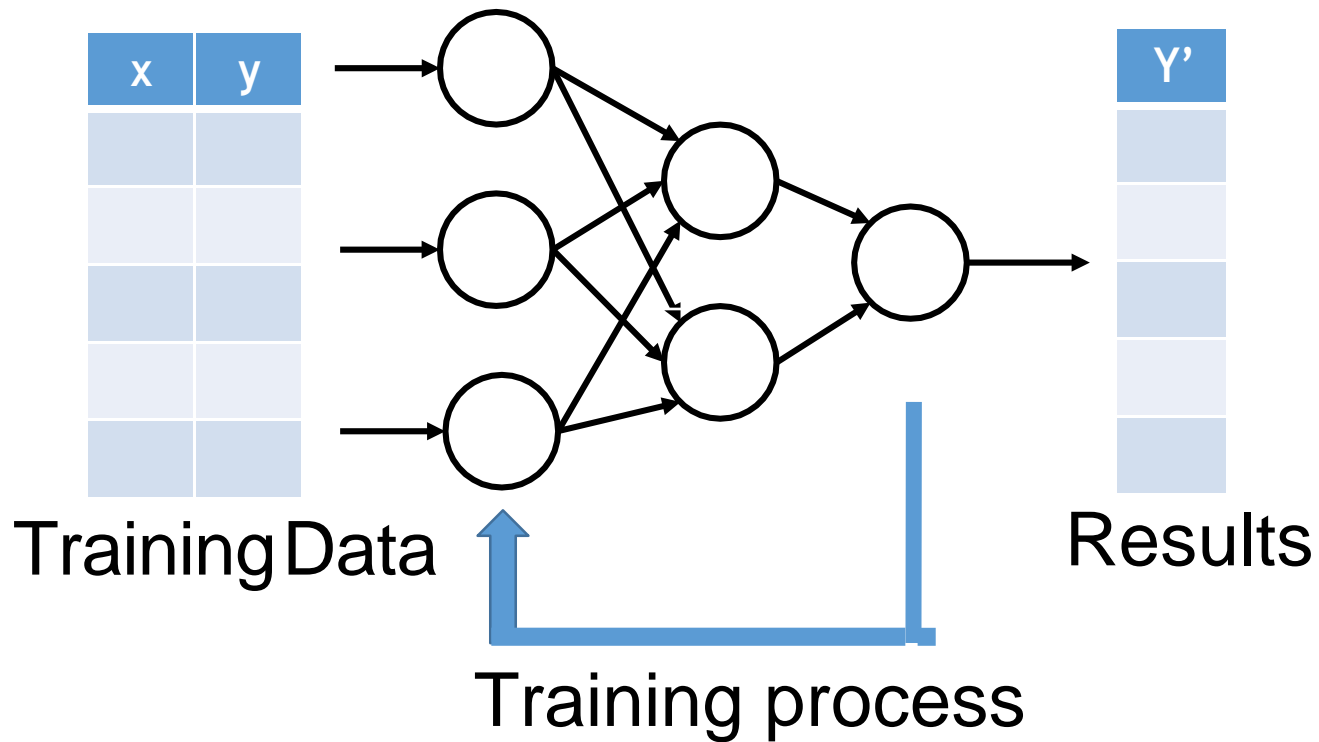
# Training Algorithms

- 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.
- The purpose is to learn to generalize
  - Recognize features which are common to good and bad exemplars



# Supervised Learning

- Training and test data sets
- Training set; input & target



X1	X2	X3	X4	Class
5.1	3.5	1.4	0.2	0
4.9	3.0	1.4	0.2	2
4.7	3.2	1.3	0.2	0
4.6	3.1	1.5	0.2	1

# Learning algorithm

- **Epoch** : Presentation of the entire training set to the neural network. In the case of the AND function an epoch consists of four sets of inputs being presented to the network (i.e.  $[0,0]$ ,  $[0,1]$ ,  $[1,0]$ ,  $[1,1]$ )
- **Error**: The error value is the amount by which the value output by the network differs from the target value. For example, if we required the network to output 0 and it output a 1, then  $\text{Error} = -1$

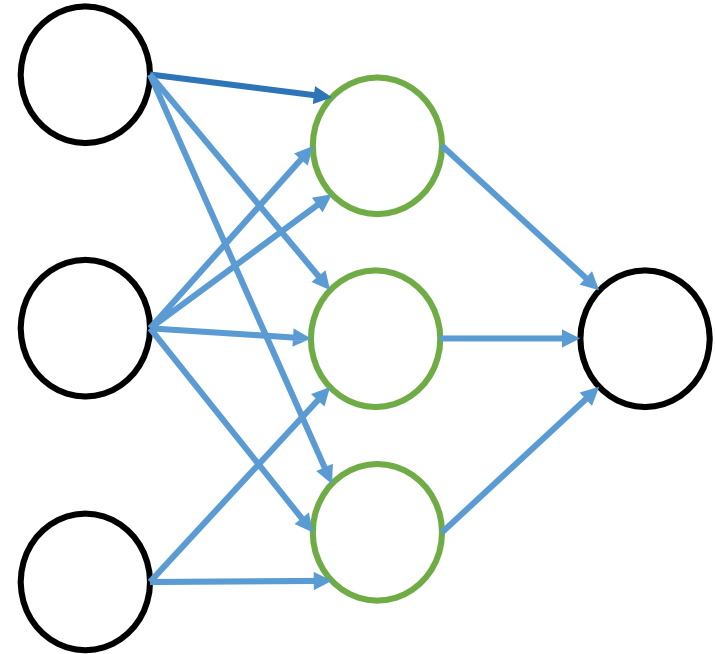
# Learning algorithm

- **Target Value, T** : When we are training a network we not only present it with the input but also with a value that we require the network to produce. For example, if we present the network with  $[1,1]$  for the AND function the training value will be 1
- **Output , O** : The output value from the neuron
- **$I_j$** : Inputs being presented to the neuron
- **$W_j$** : Weight from input neuron ( $I_j$ ) to the output neuron
- **LR( $\alpha$ )**: The learning rate. This dictates how quickly the network converges. It is set by a matter of experimentation. It is typically 0.1

# Training Perceptrons

Training the neural network

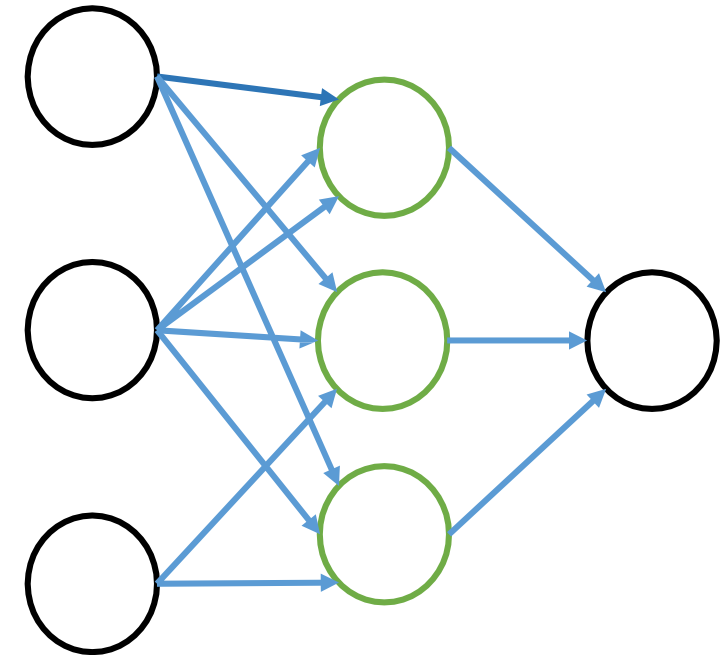
$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0



Training data

$x_1$	$x_2$	$x_3$	Target Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

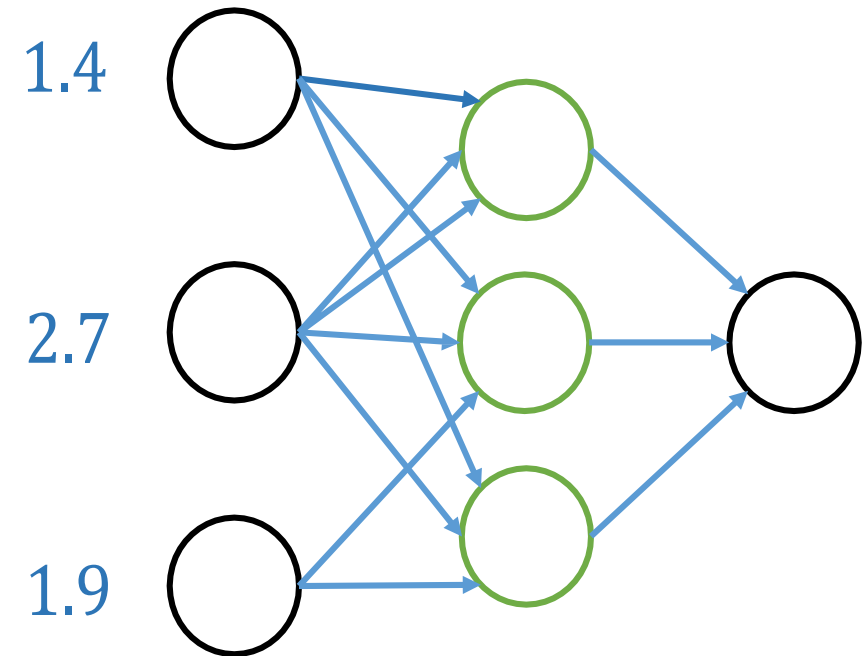
Initialise with random weights



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

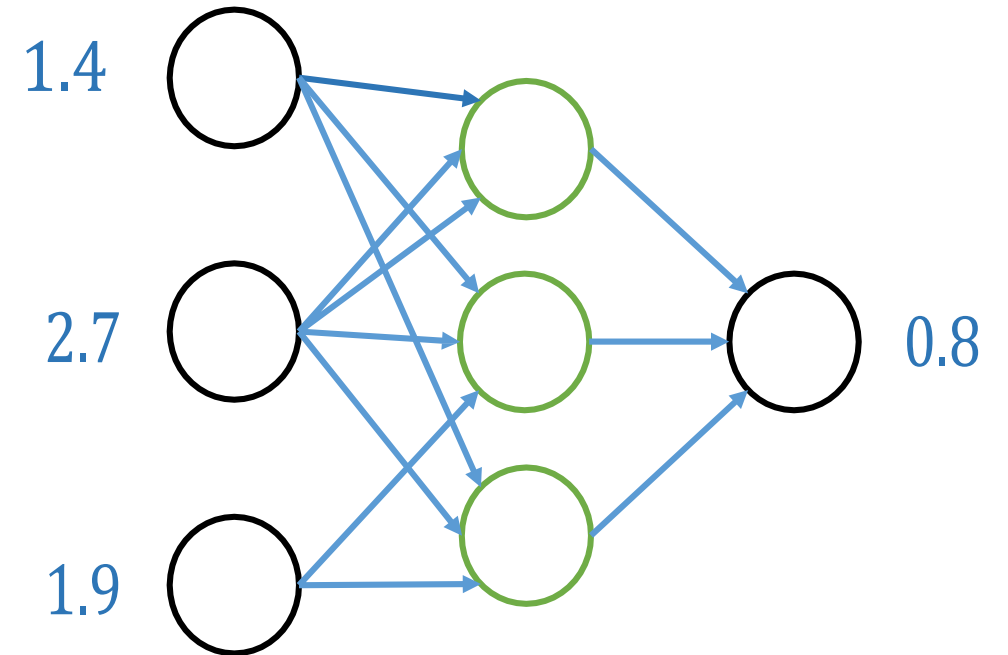
Present a training pattern



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

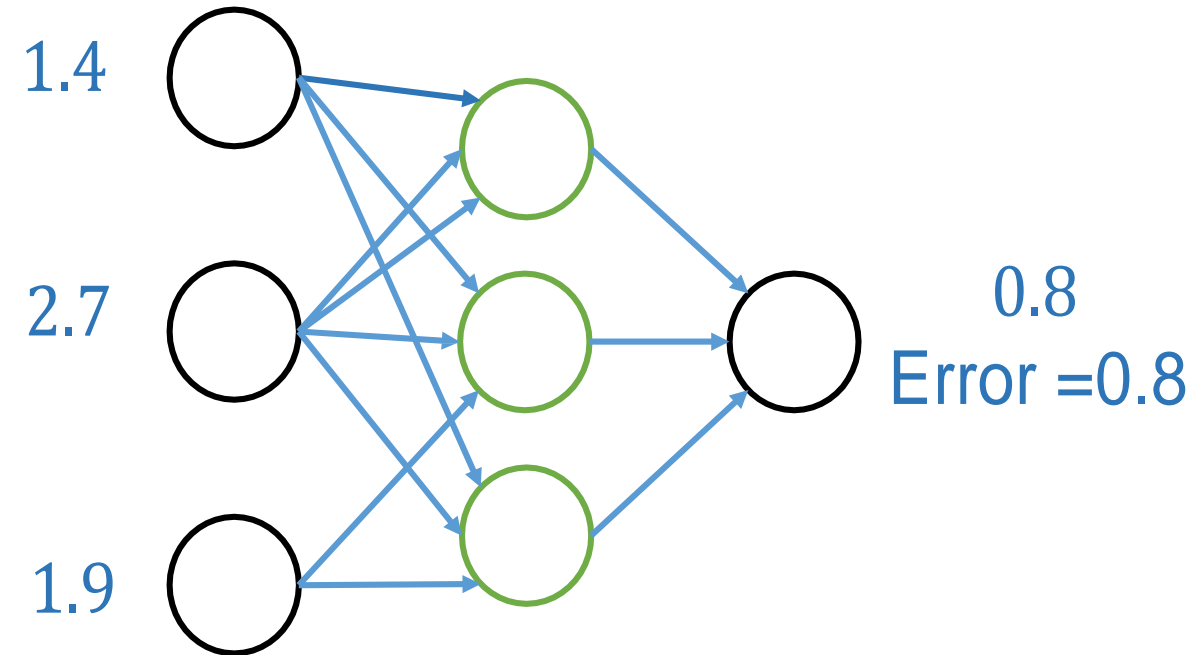
Feed it through to get output



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

Compare with target output

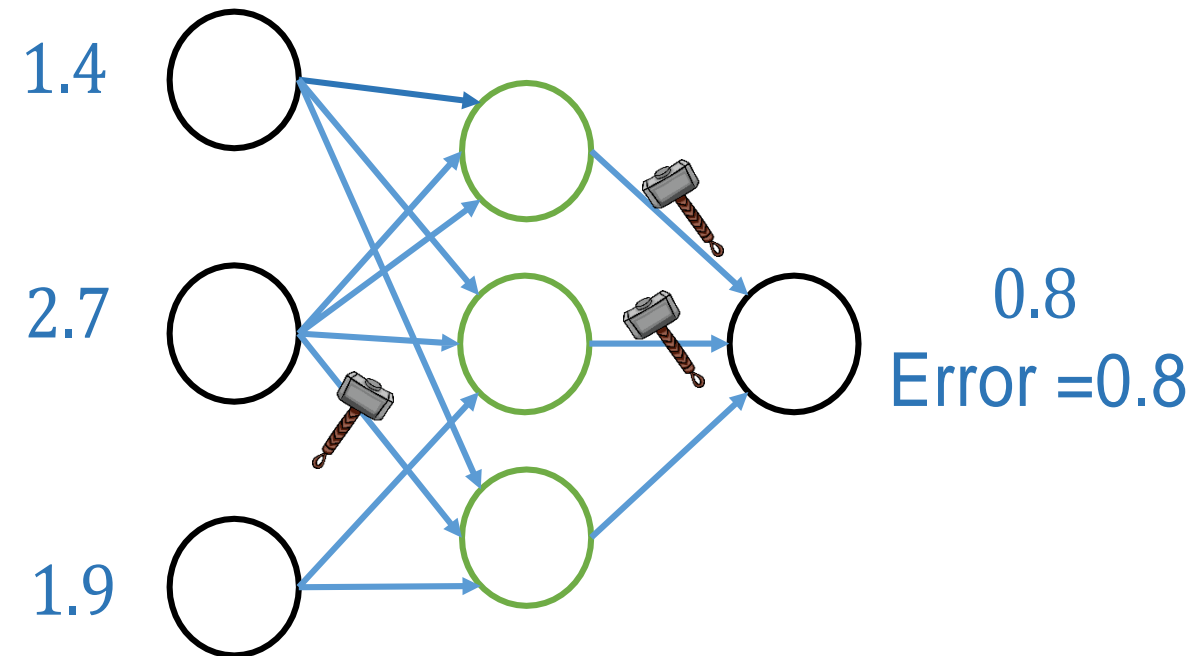




Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

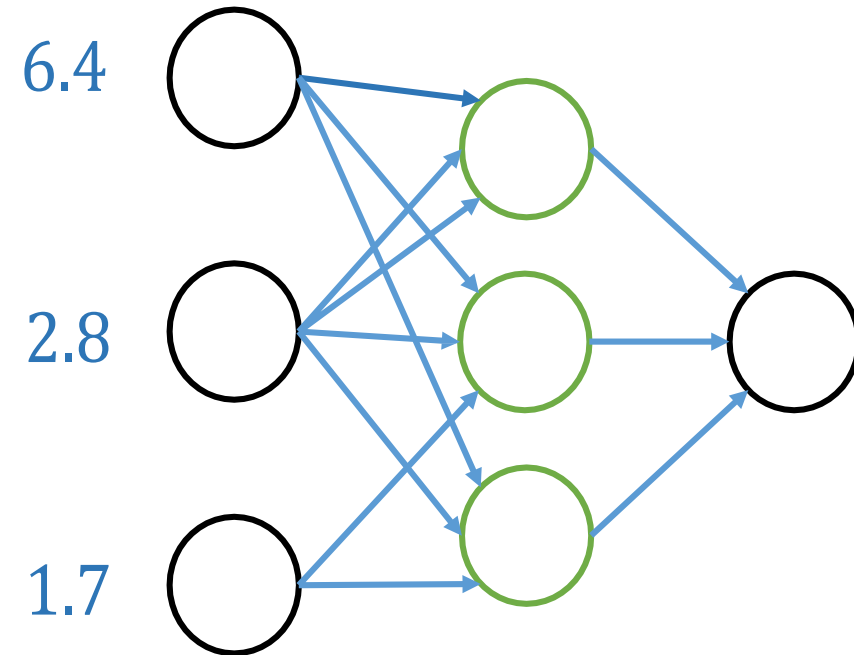
Adjust weight parameters



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

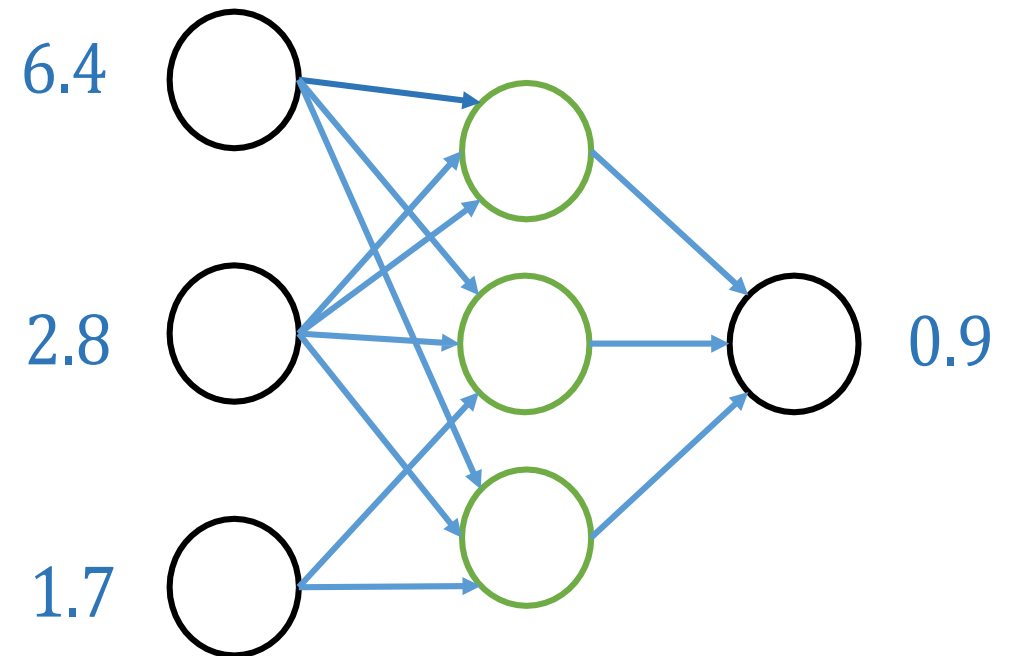
Present a training pattern



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

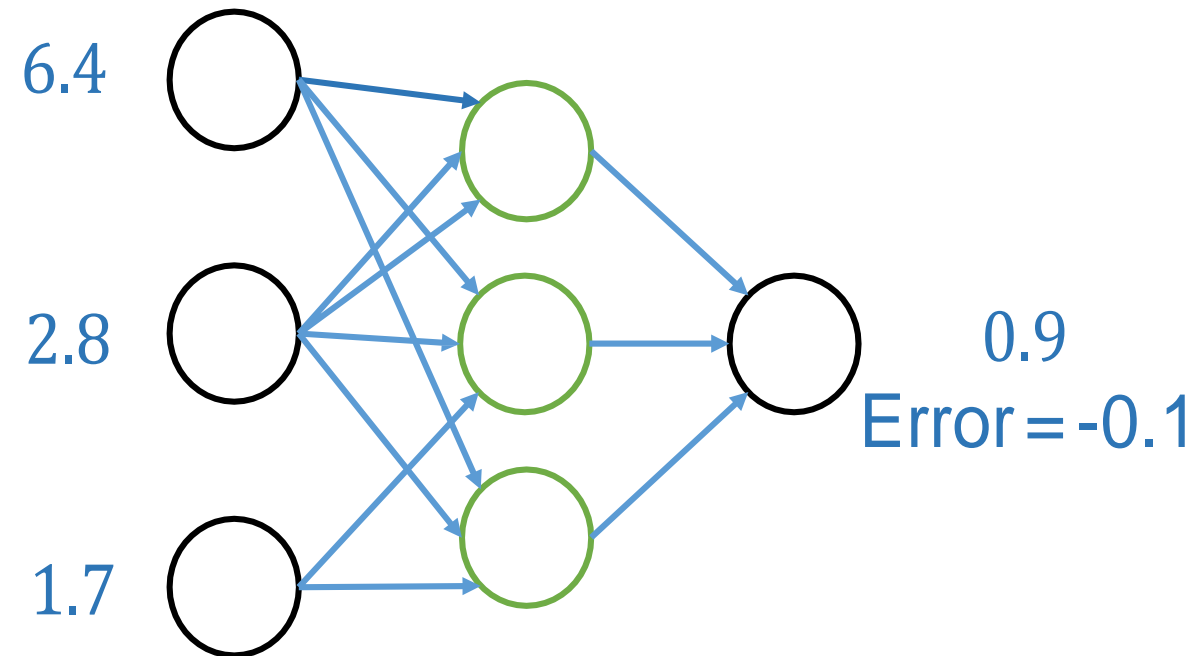
Feedthrough to get output



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

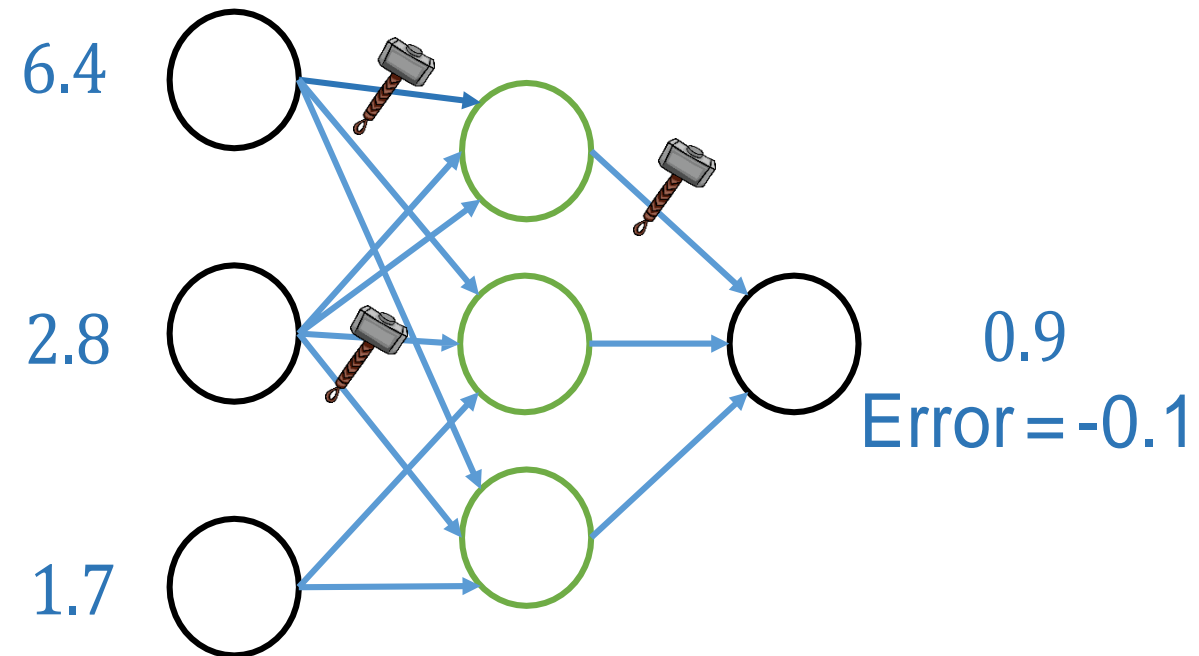
Compare with target output



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

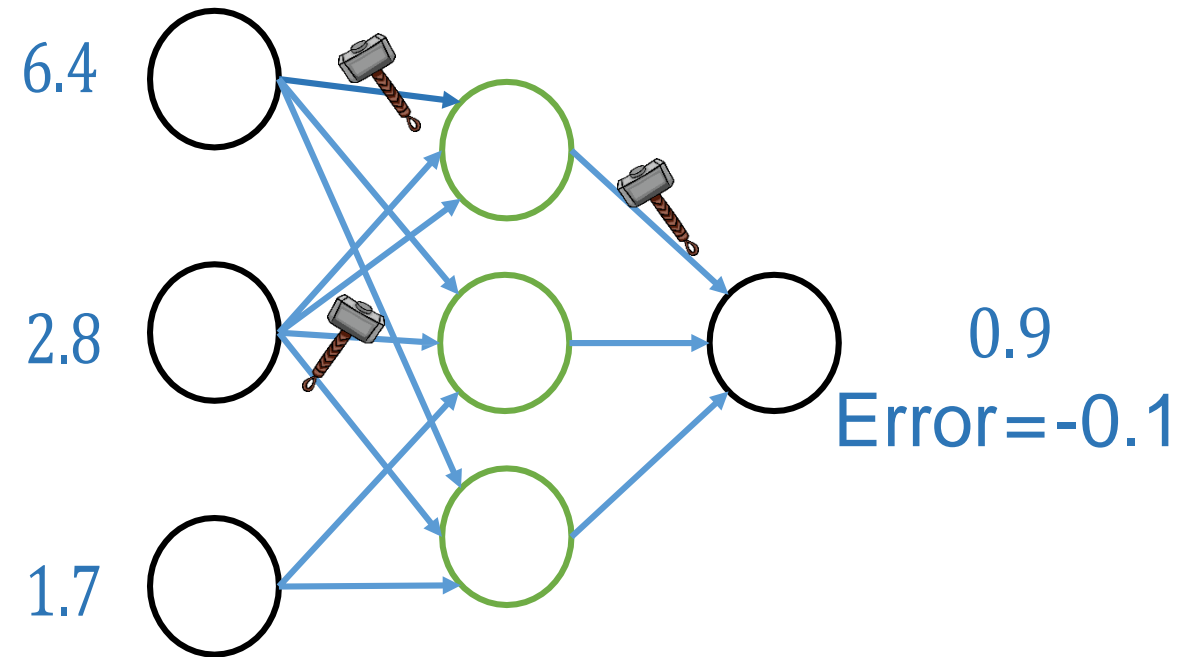
Adjust weights based on error



Training data

$x_1$	$x_2$	$x_3$	Class
1.4	2.7	1.9	0
3.8	3.4	3.2	0
6.4	2.8	1.7	1
4.1	0.1	0.2	0

And so on ...

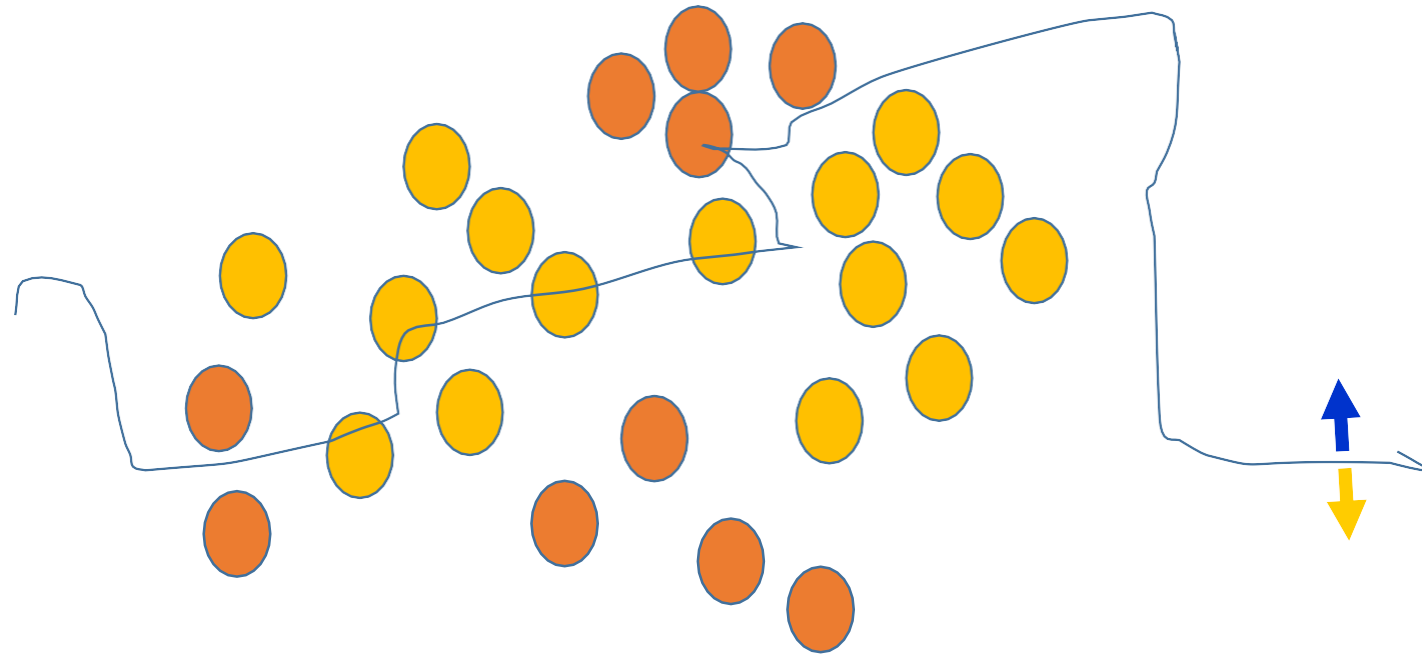


Repeat this thousands, maybe millions of times - each time taking a random training instance, and making slight weight adjustments

*Algorithms for weight adjustment are designed to make changes that will reduce the error*

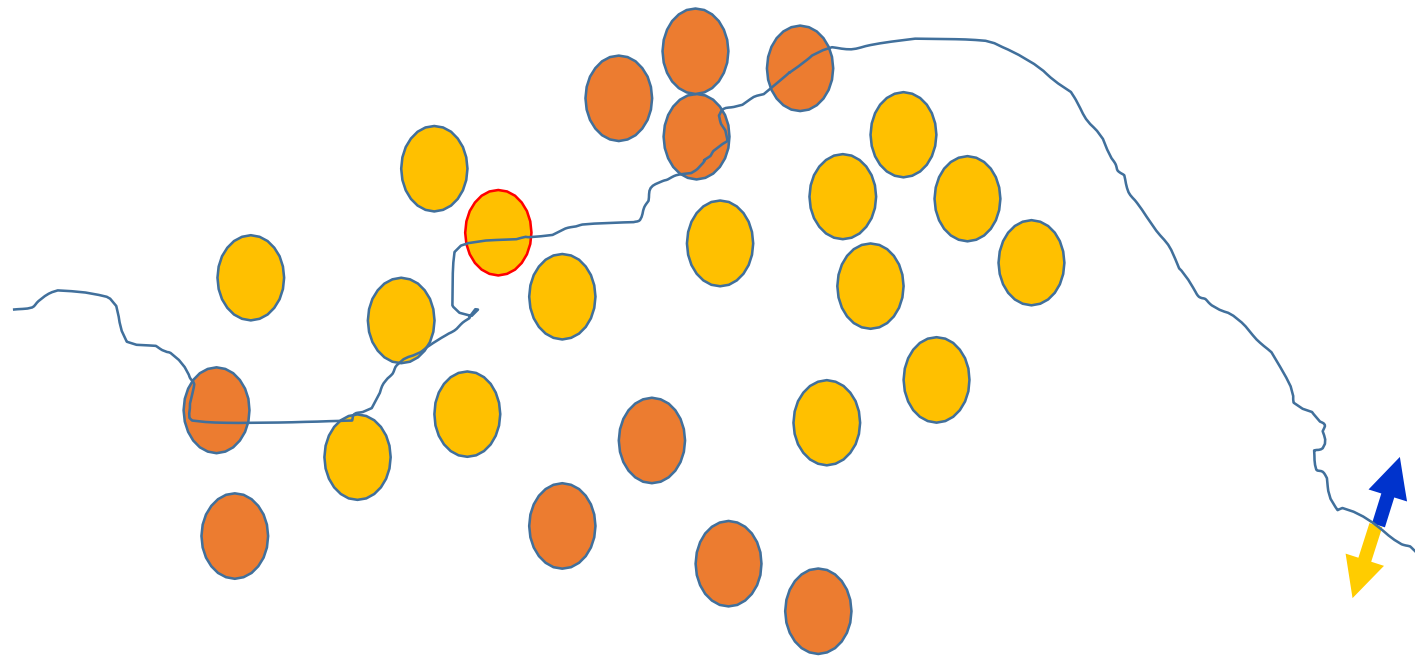
# The decision boundary perspective...

Initial random weights



# The decision boundary perspective...

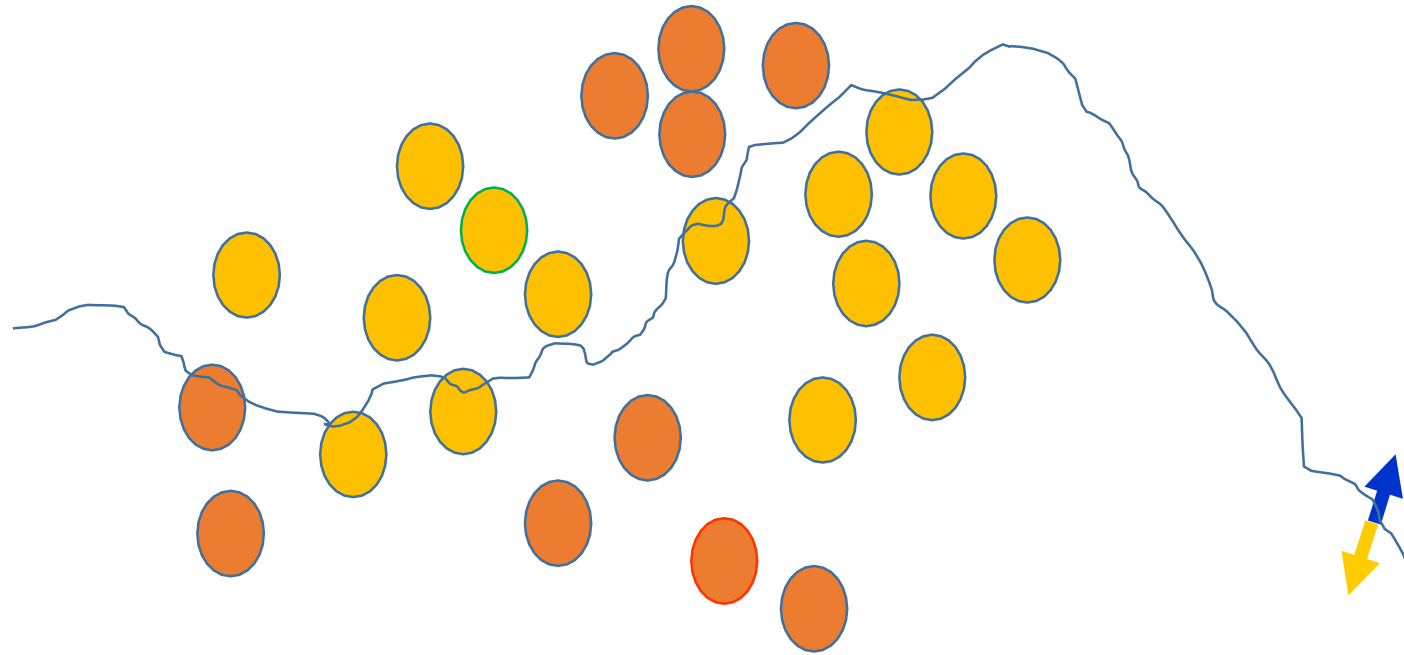
## Present a training instance / adjust the weights





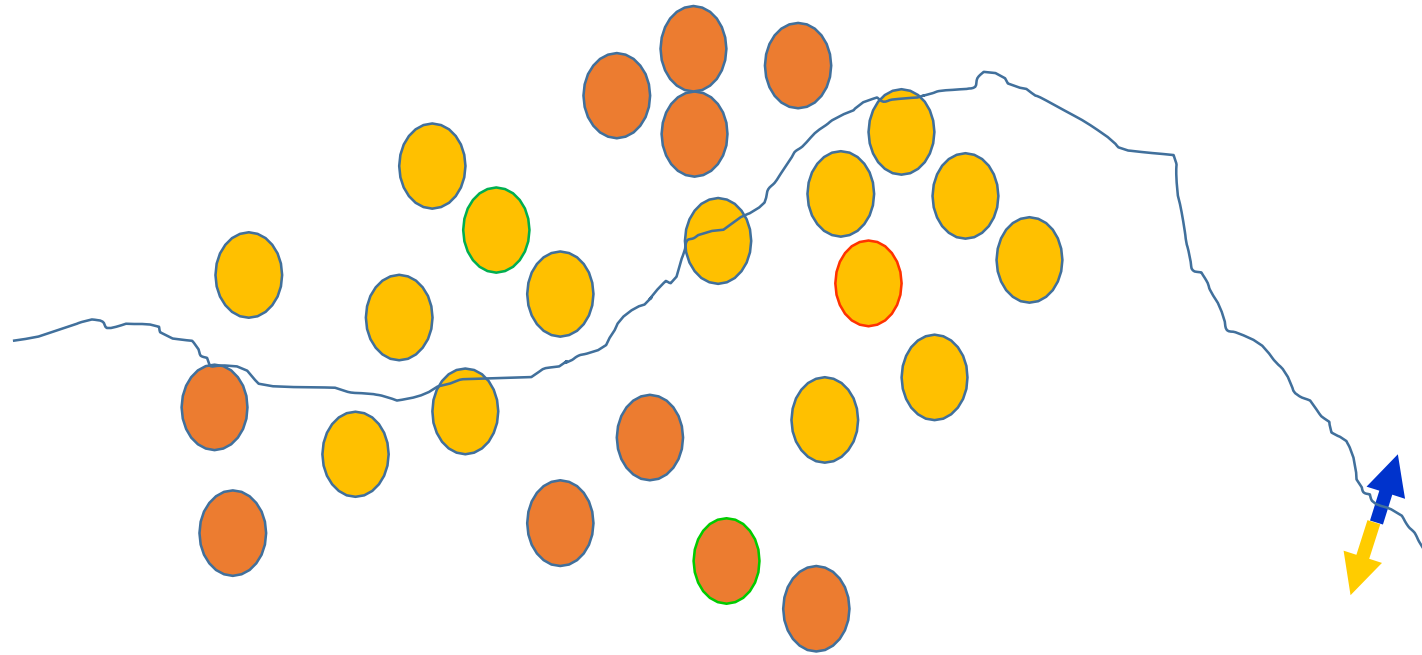
# The decision boundary perspective...

Present a training instance / adjust the weights



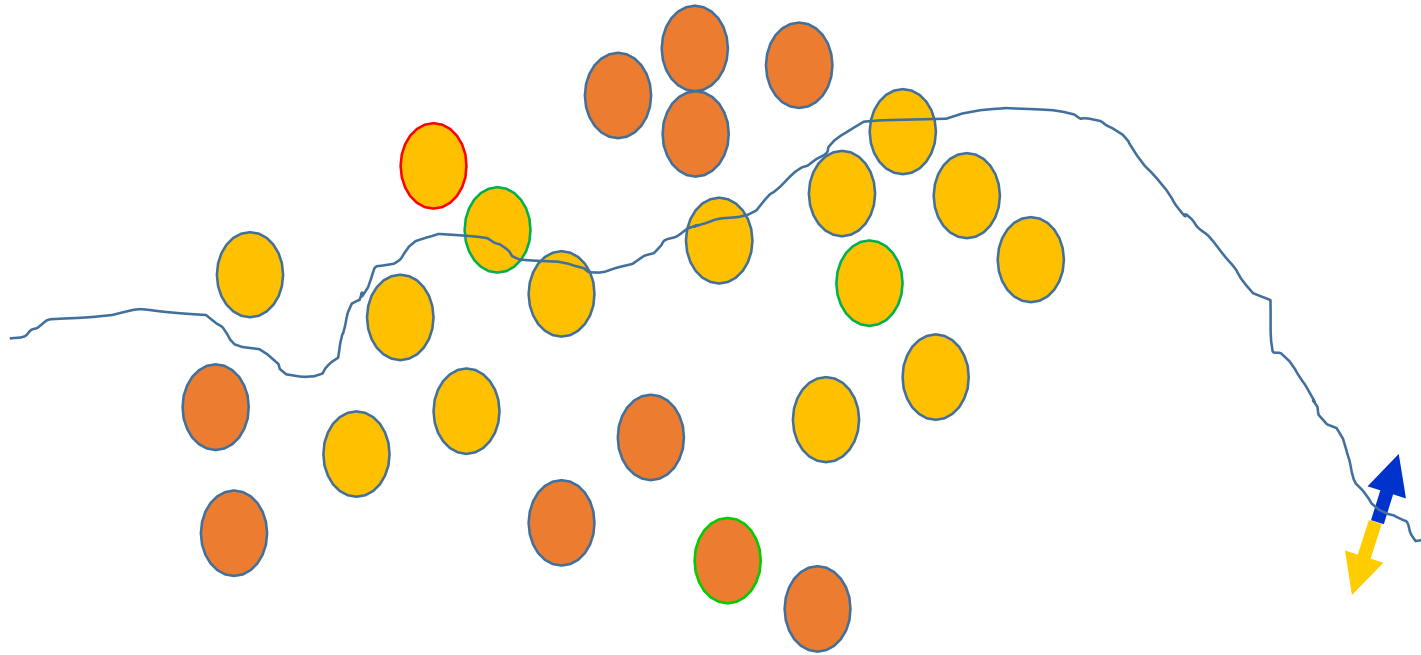
# The decision boundary perspective...

Present a training instance / adjust the weights



# The decision boundary perspective...

Present a training instance / adjust the weights



# Other Perspective

Eventually ....

