# Machine Learning (ML)

- Machine Learning is the study of how to build computer systems that adapt and improve with experience. It is a subfield of Artificial Intelligence and intersects with cognitive science, information theory, and probability theory, …etc
- Intelligent agents must be able to mange through the course of their interactions with the world, as well as through the experience of their own internal states and processes.

- Simon's definition (1983) describes learning as allowing the system to "perform better the second time." : "Learning is any process by which a system improves performance from experience."

# Why Machine Learning

- Machine learning is important in several real life problem because of the following reasons:

→ Some tasks cannot be defined well except by example

→Working environment may not be known at design time

→ Explicit knowledge encoding may be difficult and not available

→Environments change over time

- learning is widely used in a number of application areas such as:
  - Data mining and knowledge discovery
  - Speech/image/video (pattern) recognition
  - Adaptive control
  - Autonomous vehicles/robots
  - Decision support systems
  - Bioinformatics and WWW

# Tasks of ML

- Prediction: To predict the desired output for a given input based on previous input/output pairs. E.g., to predict the value of a stock given other inputs like interest rates etc.
- Categorization: To classify an object into one of several categories based on features of the object. E.g., a spam email based on subject
- Clustering: To organize a group of objects into homogeneous segments. E.g., a satellite image analysis system which groups land areas into forest, urban…etc, for better utilization of natural resources.
- Planning: To generate an optimal sequence of actions to solve a particular problem. E.g., an Automated Vehicle which plans its path to avoid obstacles

# Models of Learning

- Classical AI deals mainly with deductive reasoning, learning represents inductive reasoning.

→Deductive reasoning arrives at answers to queries relating to a particular situation starting from a set of general axioms,

→ inductive reasoning arrives at general axioms from a set of particular instances.

- Classical AI often suffers from the knowledge acquisition problem in real life applications where obtaining and updating the knowledge base is costly and contains errors.
- Machine learning serves to solve the knowledge acquisition bottleneck by obtaining the result from data by induction.

# Inductive Learning

Simplest way to describe it: learn a function (table-tree) from examples

- Ignores prior knowledge
- Assumes a deterministic, observable environment
- Assumes examples are given
- Assumes the "agent" wants to learn the function ( for a reason so and so)

# Inductive Learning

- Induction, which is learning a generalization from a set of examples, is one of the most fundamental learning tasks.

Different Approaches:

  - Symbolic approach
  - Neural Nets
  - genetic and evolutionary learning.

- Strongest models of learning we have, may be seen in the human and animal systems that have evolved towards equilibration with the world. This approach to learning through adaptation is reflected in genetic algorithms, genetic programming
- In the real world this information is often not immediately available  AI needs to be able to learn from experience

# Different kinds of learning

Supervised learning:

- Someone gives us examples and the right answer for those examples
- We have to predict the right answer for unseen examples

Unsupervised learning:

- We see examples but get no feedback
- We need to find patterns in the data

Reinforcement learning:

- We take actions and get rewards
- Have to learn how to get high rewards

# Symbolic vs. sub-symbolic AI

"Old"-Fashioned AI is inherently symbolic

Physical Symbol System Hypothesis: ***A necessary and sufficient condition for intelligence is the representation and manipulation of symbols.***

Ex. Turing Machines

alternatives to symbolic AI

A

- connectionist models – based on a brain metaphor

  model individual neurons and their connections

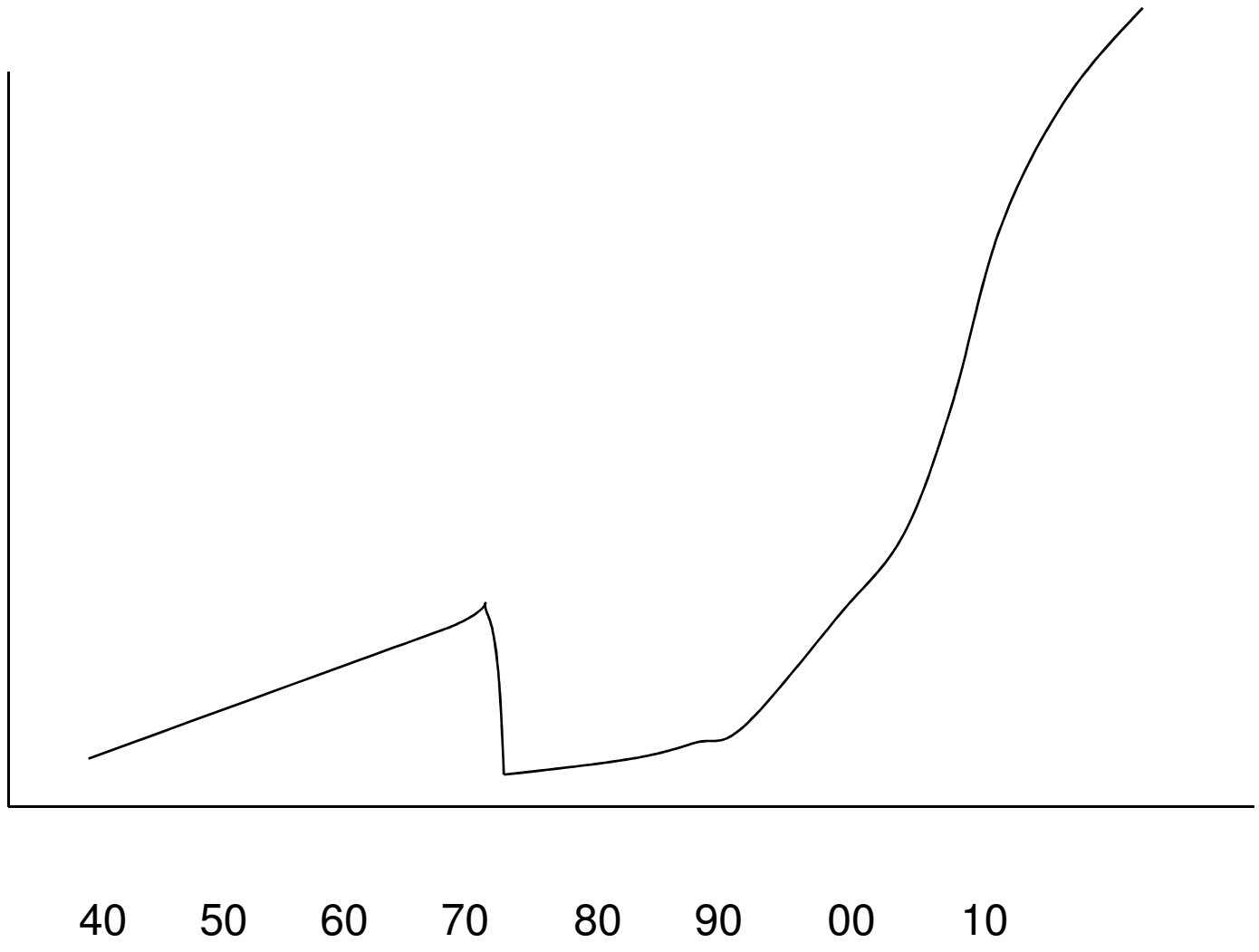
  ***properties:*** parallel, distributed, sub-symbolic

  ***examples:*** neural nets, perceptrons, backpropagation NN, associative memories (Hopfield networks)

# SubSymbolic Representations

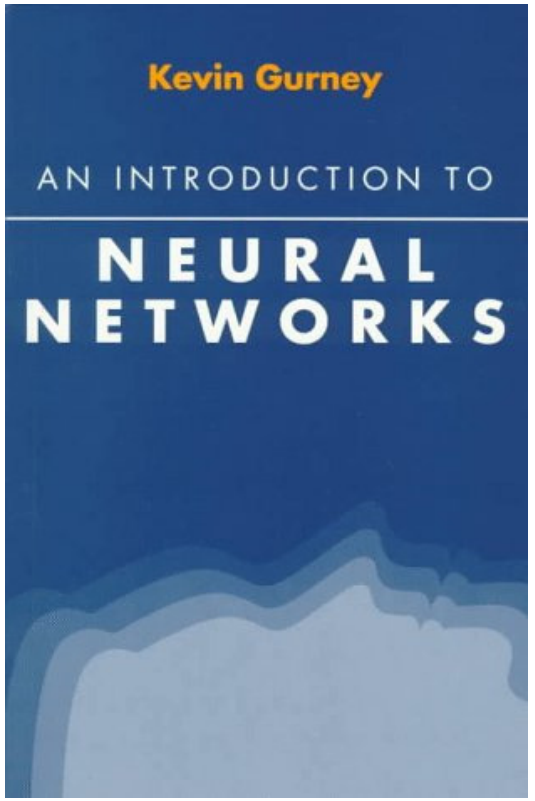
- Decision trees can be easily read
  - A disjunction of conjunctions (logic), We call this a symbolic representation
- Non-symbolic representations
  - More numerical in nature, more difficult to read
- Artificial Neural Networks (ANNs)
  - A Non-symbolic representation scheme
  - They embed a giant mathematical function
    - To take inputs and compute an output which is interpreted as a categorisation
  - Often shortened to “Neural Networks”
    - Don’t confuse them with real neural networks (in heads)

# Interest in Subsymbolic AI



# Helpful Resource (in addition to Luger)

Gurney, Kevin. *An Introduction to Neural Networks*, 1996.



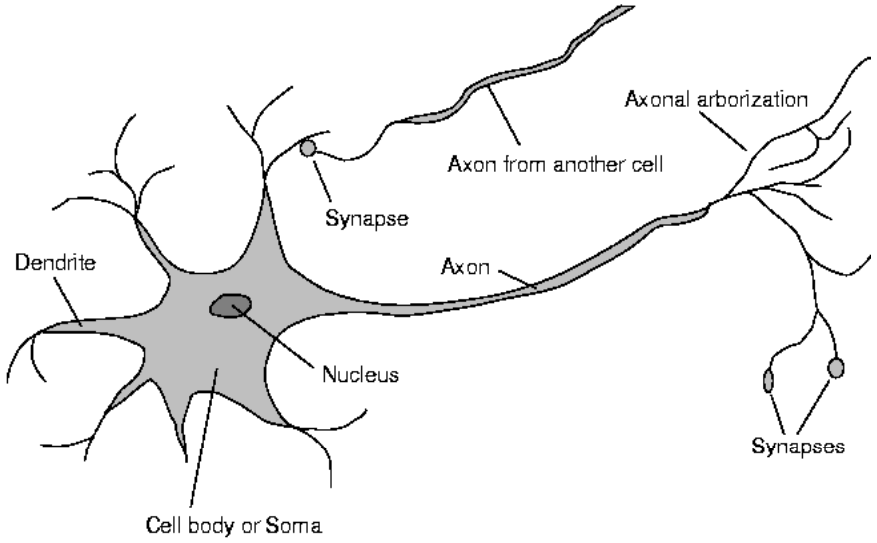
# Neural Networks

- Neural Networks can be :

  - **Biological** models
  - **Artificial** models

- Desire to produce artificial systems capable of sophisticated computations similar to the human brain.

# Biological analogy




general brain architecture:

- many (relatively) slow neurons, interconnected
- Dendrites(التغصنات) serve as input devices (receive electrical impulses from other neurons)
- cell body "sums" inputs from the dendrites (possibly inhibiting or exciting)
- if sum exceeds some threshold, the neuron fires an output impulse along axon, sending an electrical pulse to other neurons

# Biological analogy

The brain is composed of a mass of interconnected neurons, each neuron is connected to many other neurons

**The brain:**

$10^{11}$ neurons of $> 20$ types, $10^{14}$ synapses, 1ms–10ms cycle time

Signals are noisy "spike trains" of electrical potential

Neurons transmit signals to each other

Whether a signal is transmitted to an event or not (the electrical potential in the cell body of the neuron is thresholded)

Whether a signal is sent, depends on the strength of the bond (synapse-المشبك) between two neurons

# Connectionist models (neural nets)

**Definitions**

Neuron: The cell that performs information processing in the brain.

- Fundamental functional unit of all nervous system tissue.

A Neural Network: is a system composed of many simple processing elements operating in parallel which can acquire, store, and utilize experiential knowledge.

- Each element of NN is a node called unit.
- Units are connected by links.
- Each link has a numeric weight.

# History of Neural Networks

**1943:** McCulloch and Pitts proposed a model of a neuron --> Perceptron

**1960:** Widrow and Hoff explored Perceptron networks (which they called “Adelines”)

**1962:** Rosenblatt proved the convergence of the perceptron training rule.

**1969:** Minsky and Papert showed that the Perceptron cannot deal with nonlinearly-separable data sets---even those that represent simple function such as X-OR.

**1970-1985:** Very little research on Neural Nets

**1986:** Invention of Backpropagation which can learn from nonlinearly-separable data sets. [Rumelhart and McClelland, but also Parker and earlier on: Werbos]

Since 1985: A lot of research in Neural Nets!

# How NN learns a task.

- Initializing the weights.

- Use of a learning algorithm.

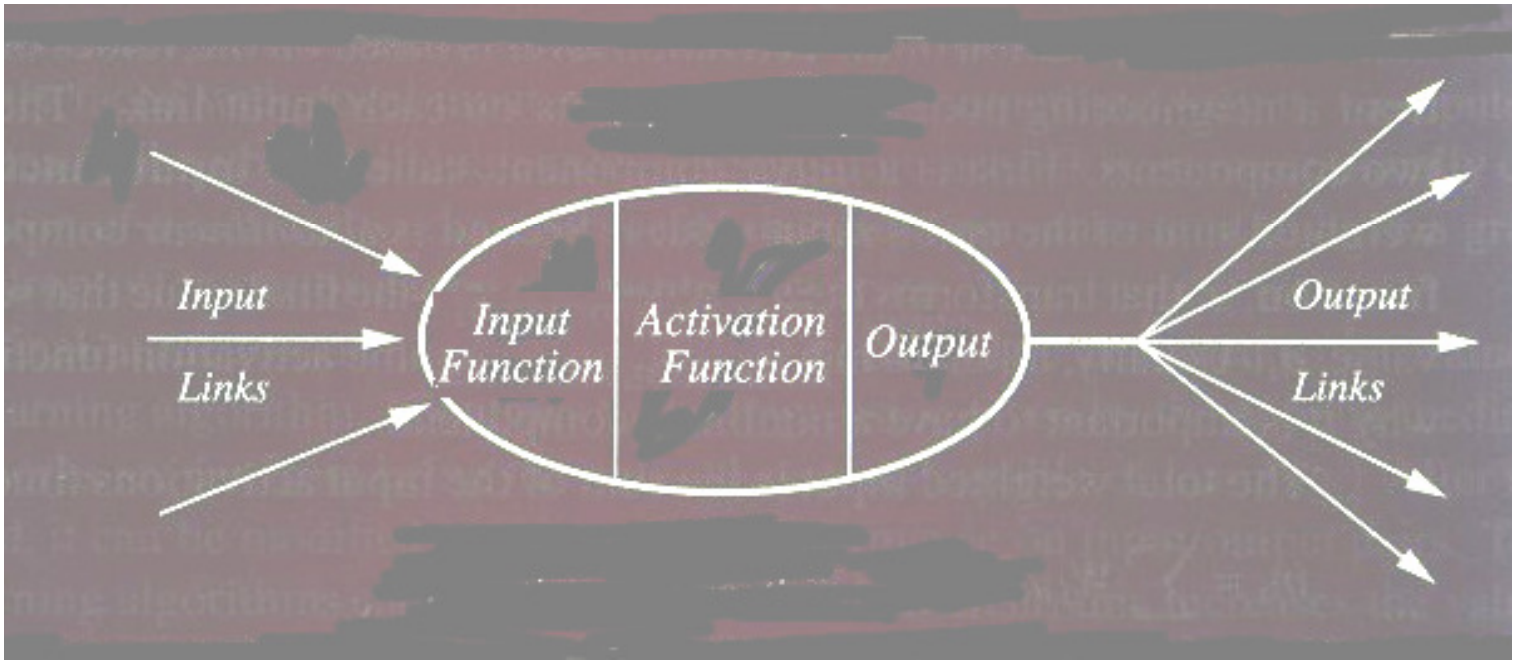
- Set of training examples.

- Encode the examples as inputs.

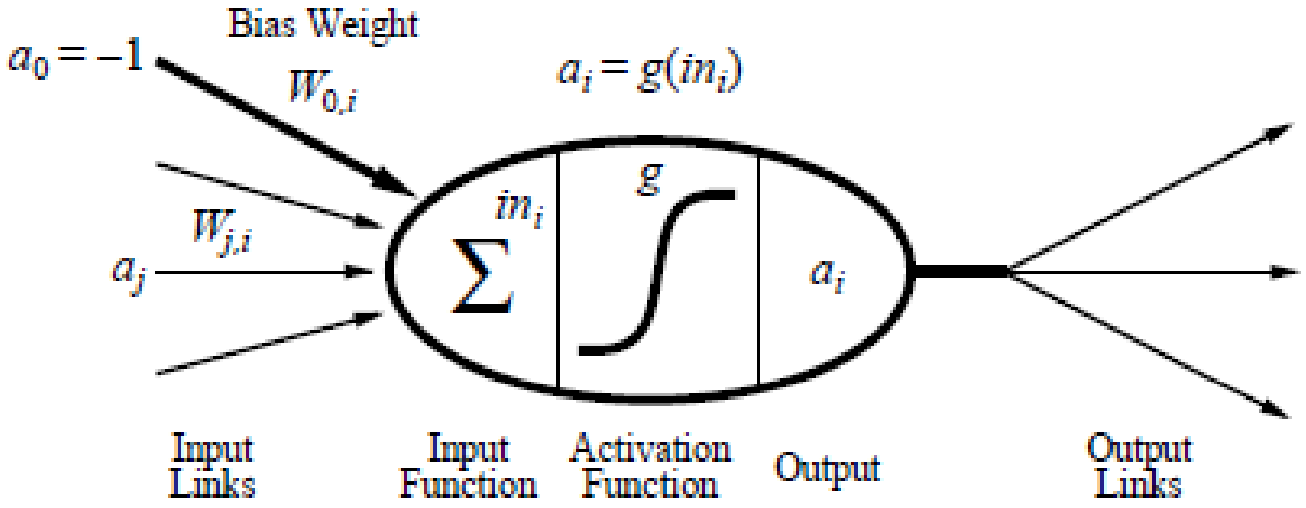
- Convert output into meaningful results.

# Computing Elements

A typical unit:



# Computing Elements




The output : $a_i = g(\Sigma_j W_{j,i} a_j)$

# Simple Computations in this network

-There are **2 types of components:** Linear and Non-linear.

- **Linear:** Input function
  - calculate weighted sum of all inputs.

- **Non-linear:** Activation function
  - transform sum into activation level.

# Activation Functions

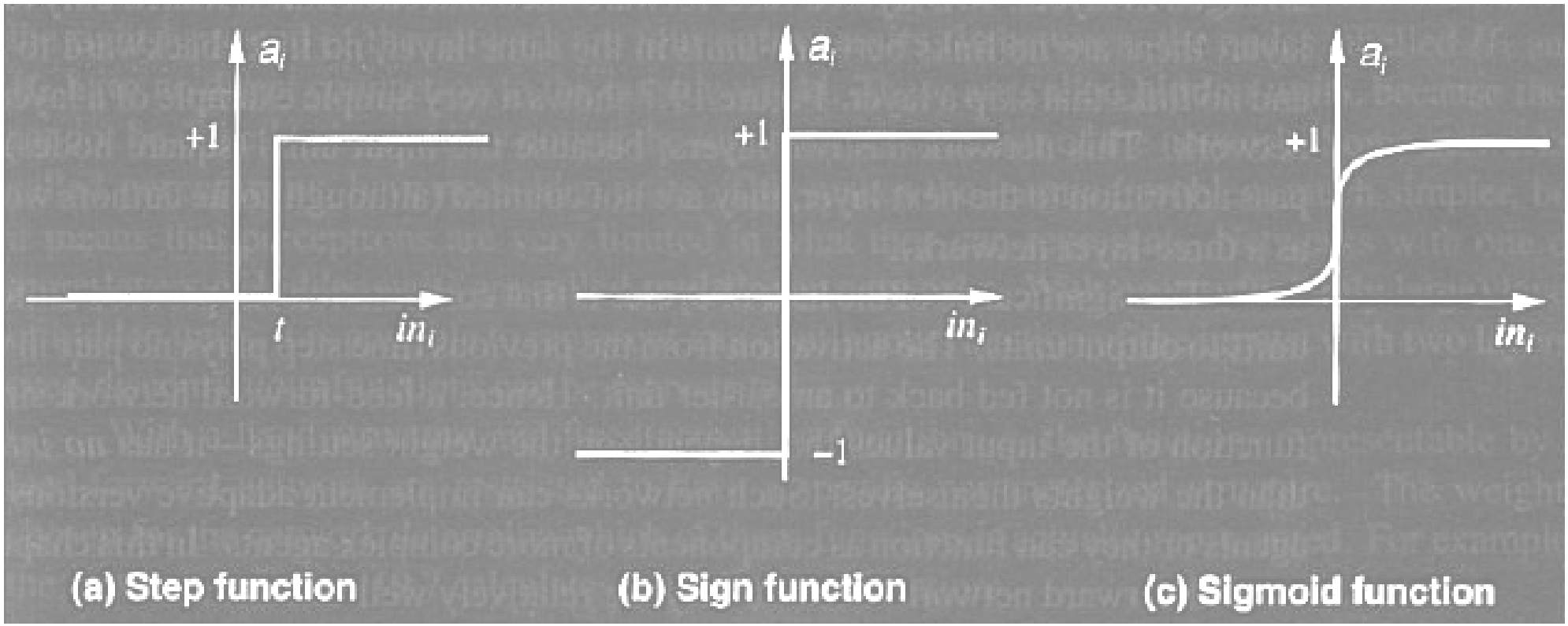
- Use different functions to obtain different models.

- 3 most common choices :

    1) Step function
    2) Sign function
    3) Sigmoid function

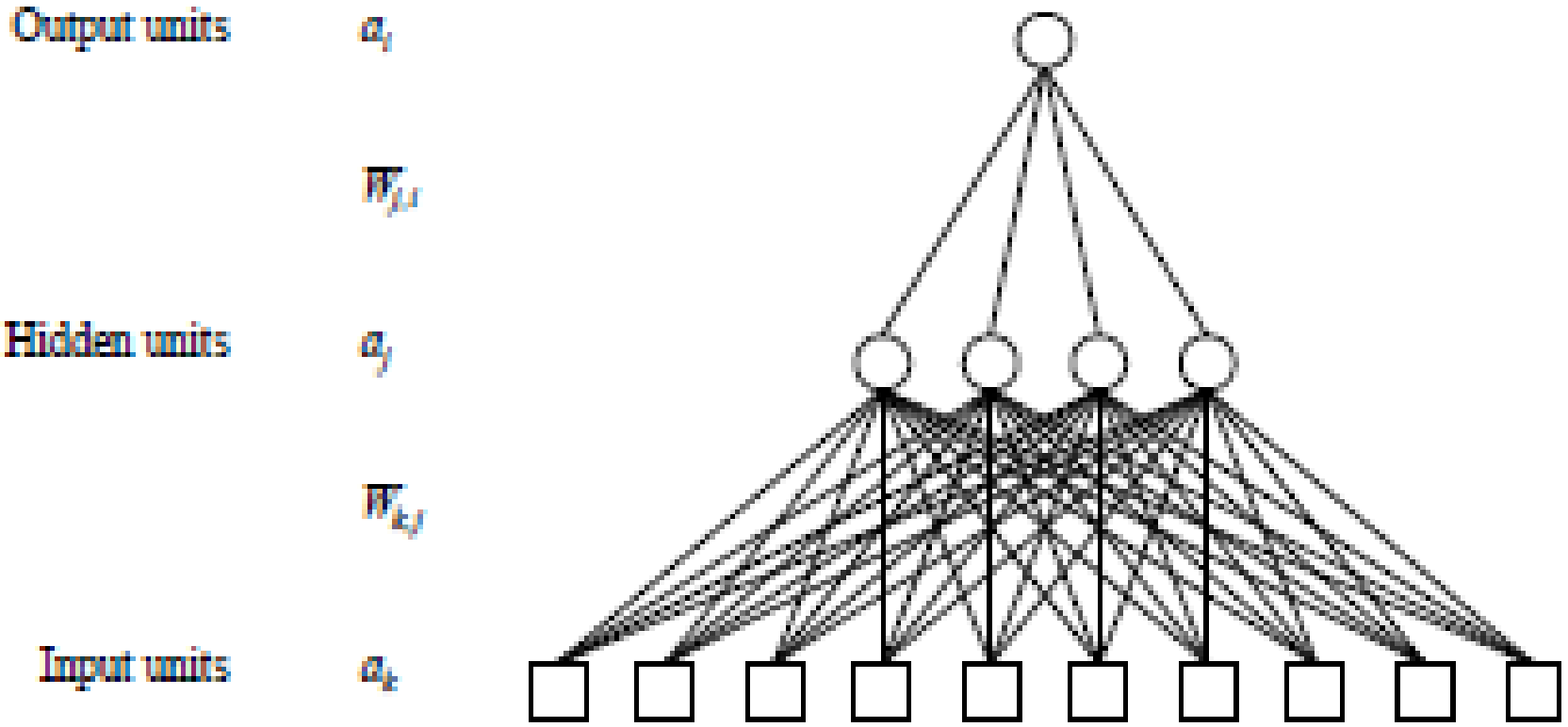
- An output of **1 represents firing** of a neuron down the axon.

# Activation Functions




$$\text{step}_t(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases} \qquad \text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \qquad \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

# Two-Layer Feed-Forward Neural Network



# Standard structure of an artificial neural network

## Input units

represents the input as a fixed-length vector of numbers (user defined)

## Hidden units

calculate threshold weighted sums of the inputs

represent intermediate calculations that the network learns

## Output units

represent the output as a fixed length vector of numbers

# Network Structure

**Feed-forward neural nets:**

Links can only go in one direction.

**Recurrent neural nets:**

Links can go anywhere and form arbitrary topologies.

# Feed Forward Network

- Arranged in ***layers***.
- Each unit is linked only in the unit in next layer.
- No units are linked between the same layer, back to the previous layer or skipping a layer.

- Computations can proceed uniformly from input to output units.

# Multi-layer Networks

- Have one or more layers of hidden units.

- With two possibly very large hidden layers, it is possible to implement any function.

# Perceptrons

- Networks without hidden layer are called perceptrons.

- Perceptrons are very limited in what they can represent, but this makes their learning problem much simpler.

# Perceptron

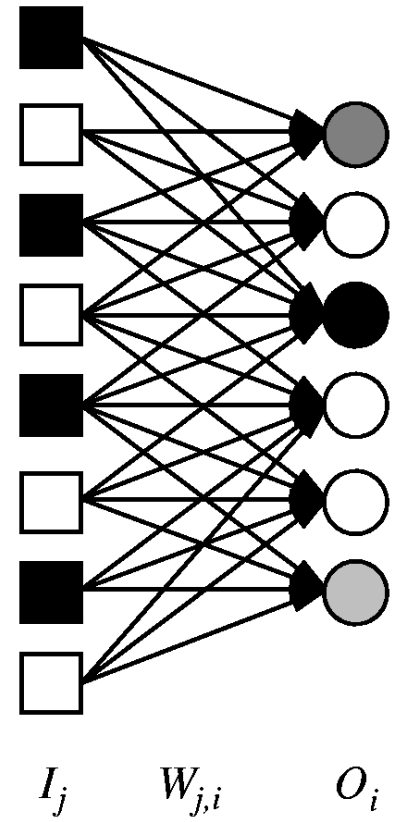
- First studied in the late 1950s.

- Also known as Layered Feed-Forward Networks.

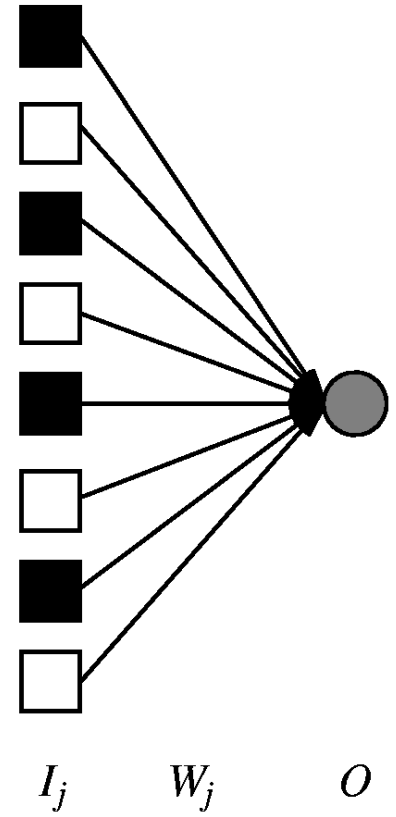
- The only efficient learning element at that time was for single-layered networks.

- Today, used as a synonym for a single-layer, feed-forward network.

# Perceptrons
## Early neural nets

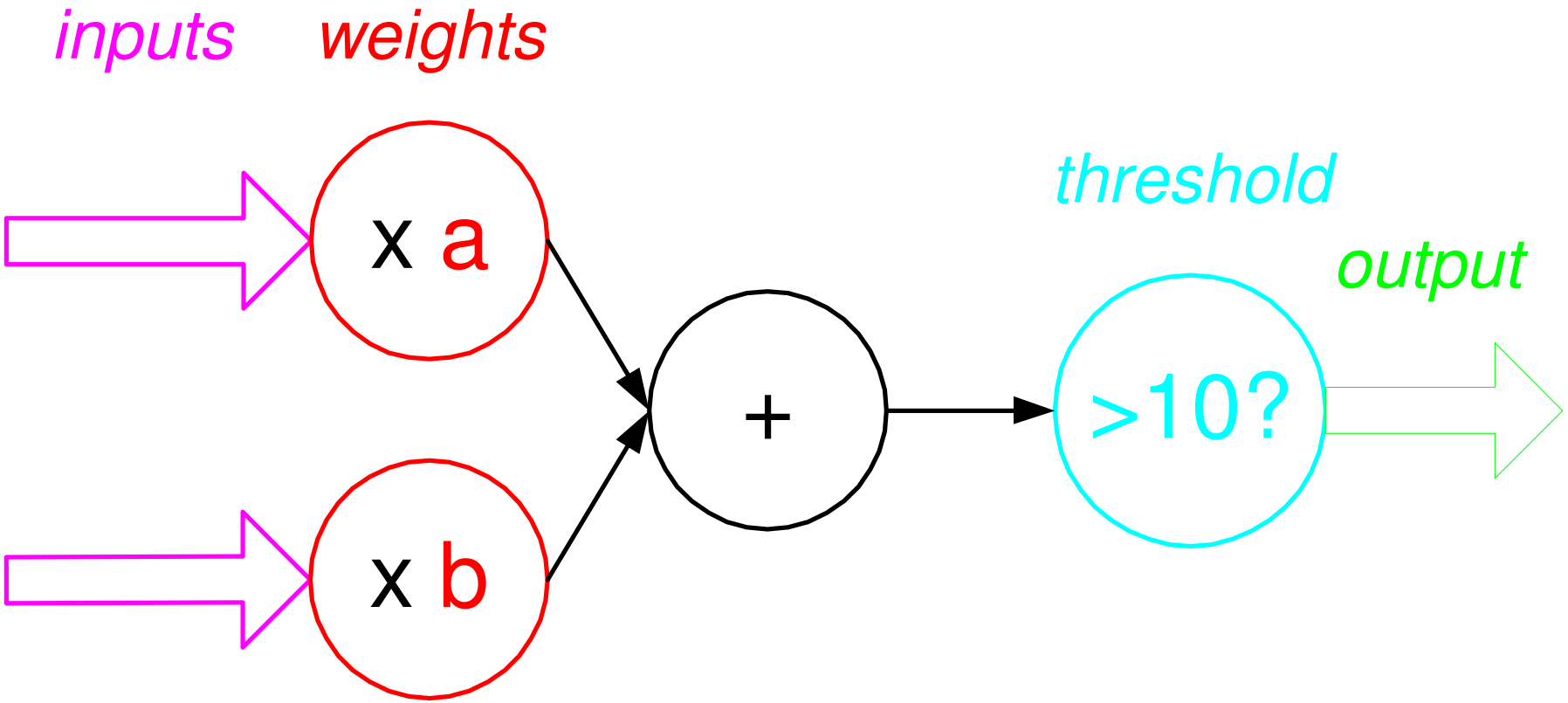



**Perceptron Network**




**Single Perceptron**

# Perceptron (artificial neuron)



# Example of Training

Inputs and outputs are 0 (no) or 1 (yes)

Initially, weights are random

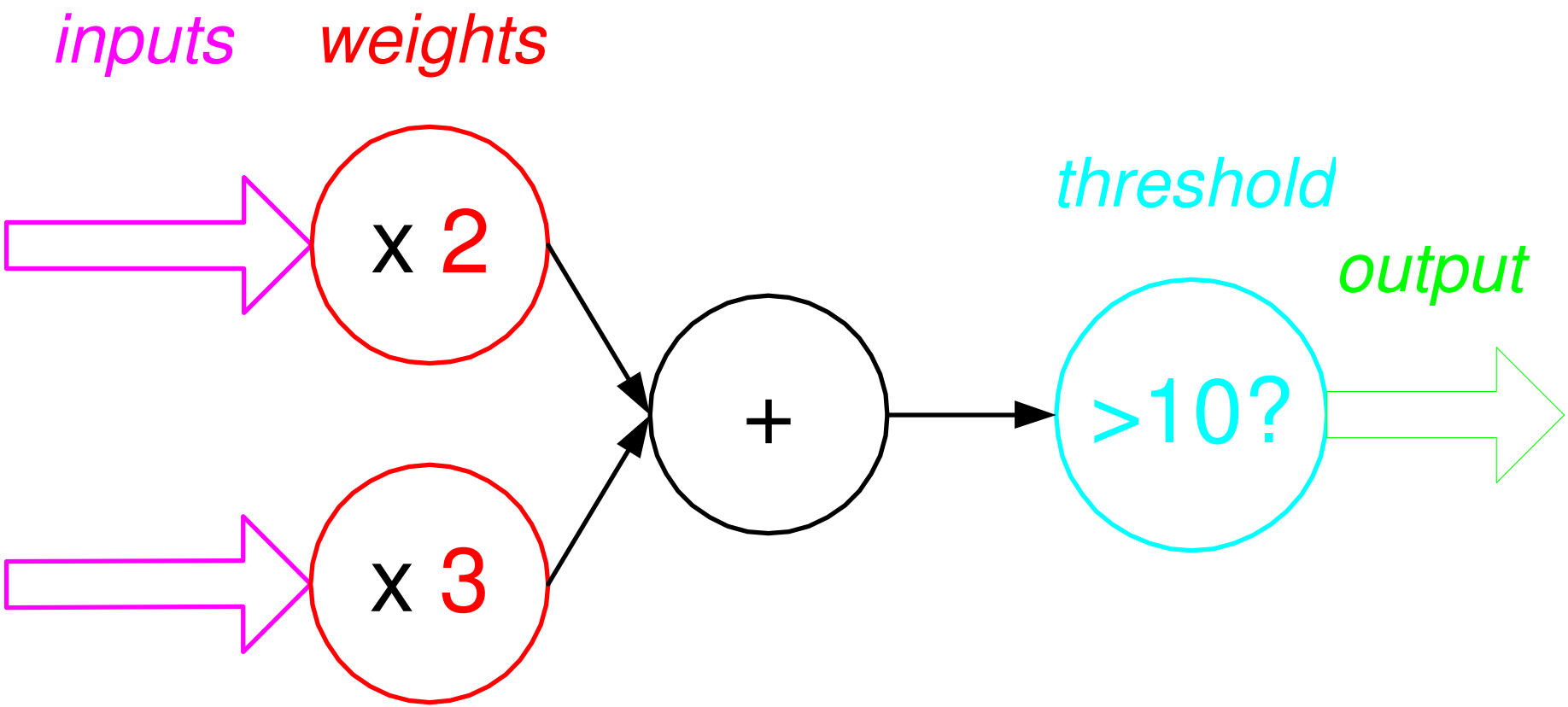
Provide training input

Compare output of neural network to desired output

- If same, reinforce patterns
- If different, adjust weights

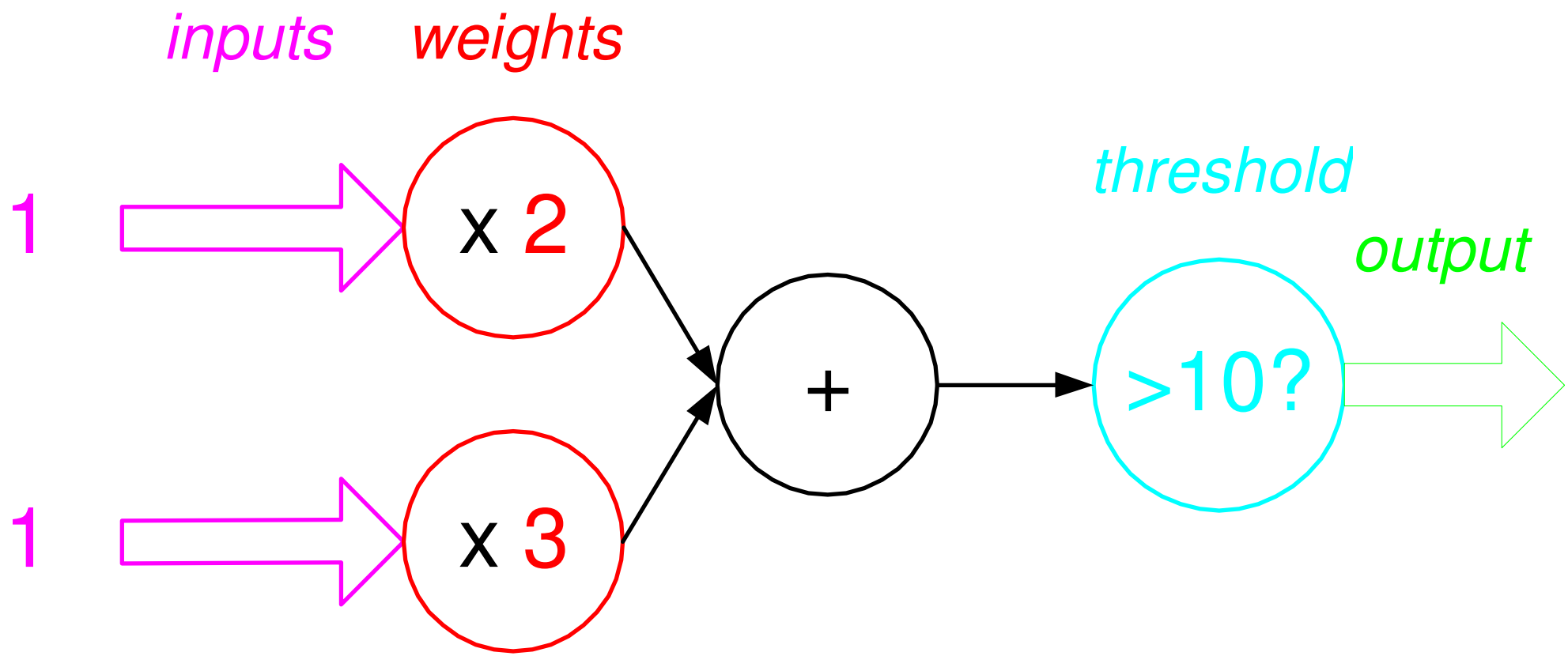
# Example

If both inputs are 1, output should be 1.



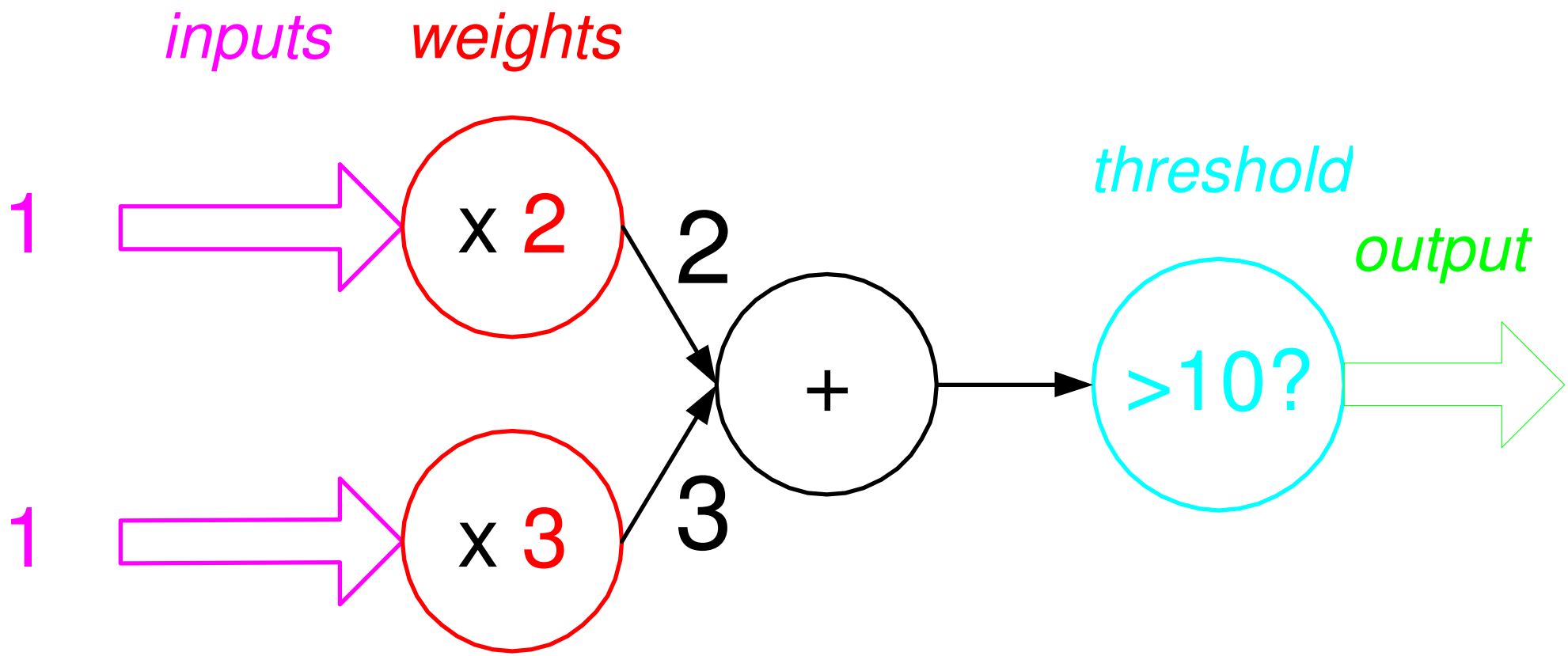
# Example

If both inputs are 1, output should be 1.



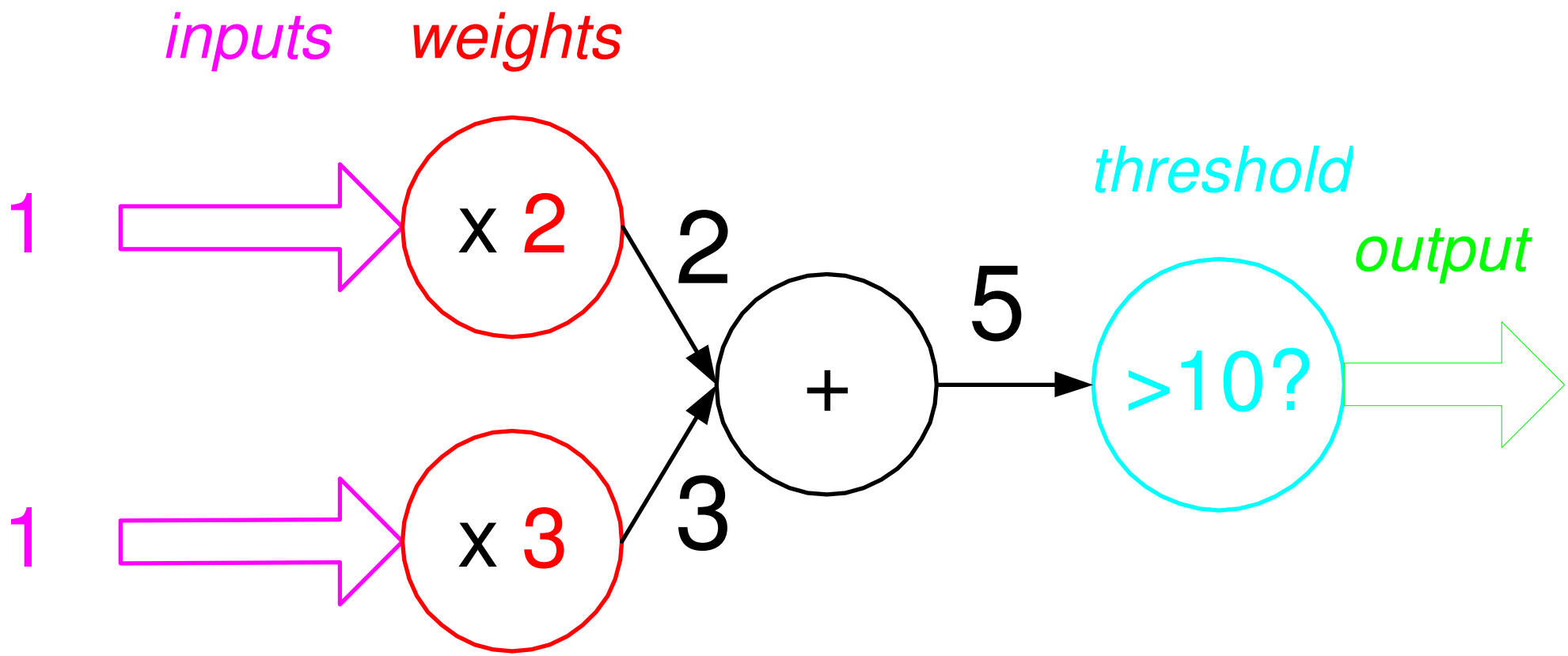
# Example

If both inputs are 1, output should be 1.



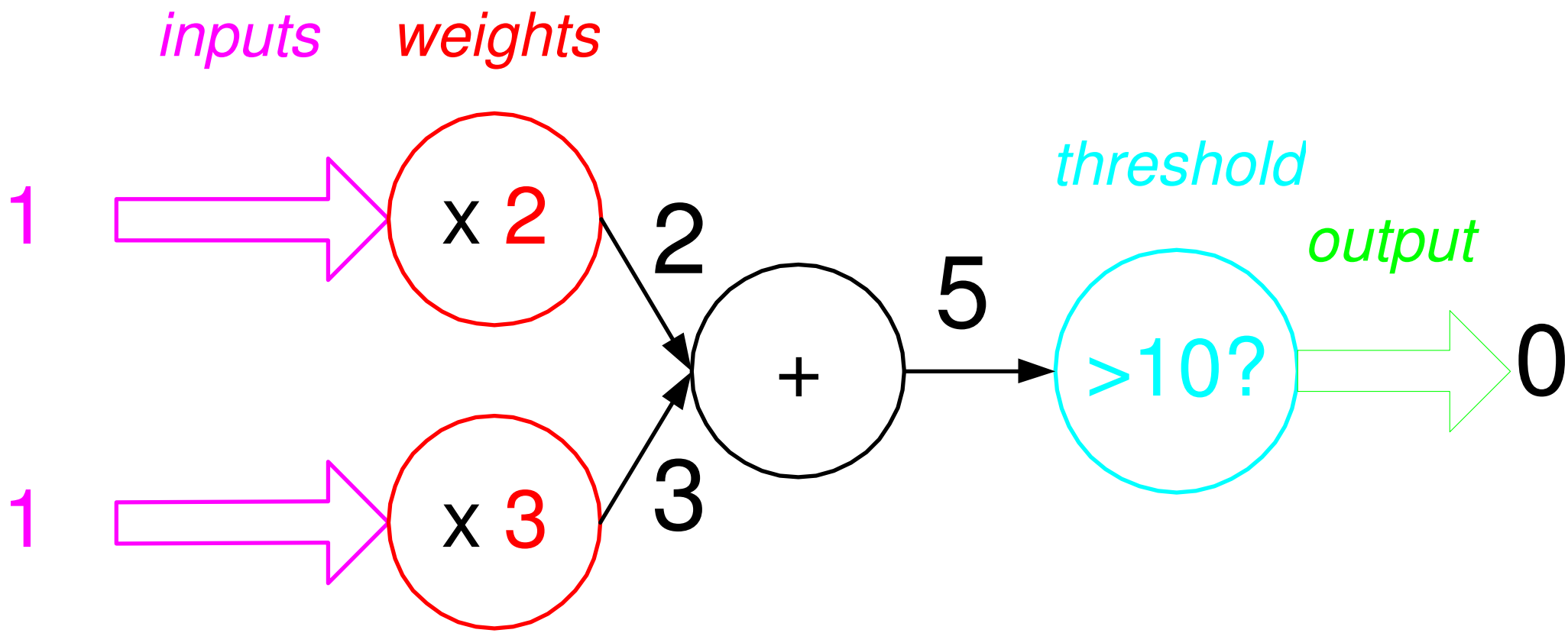
# Example

If both inputs are 1, output should be 1.



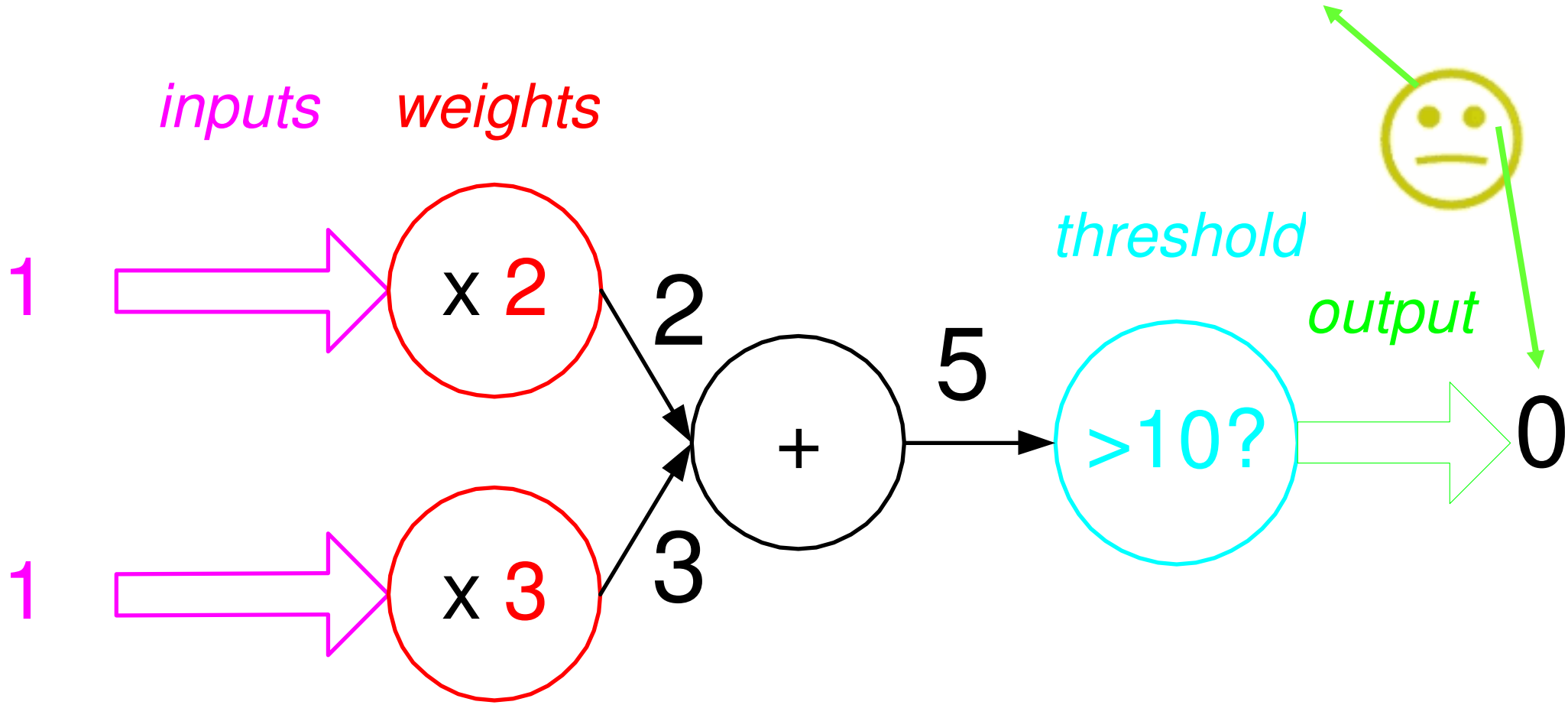
# Example

If both inputs are 1, output should be 1.



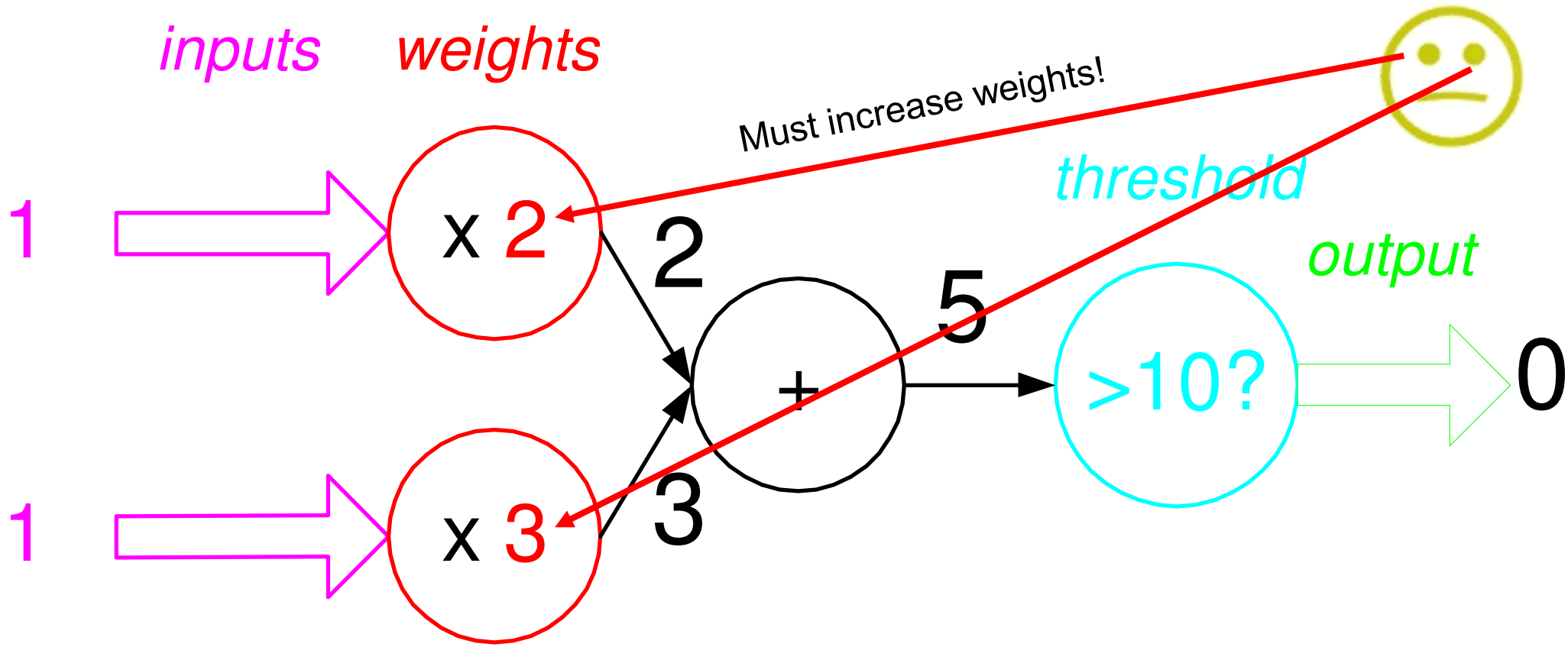
# Example

If both inputs are 1, output should be 1.



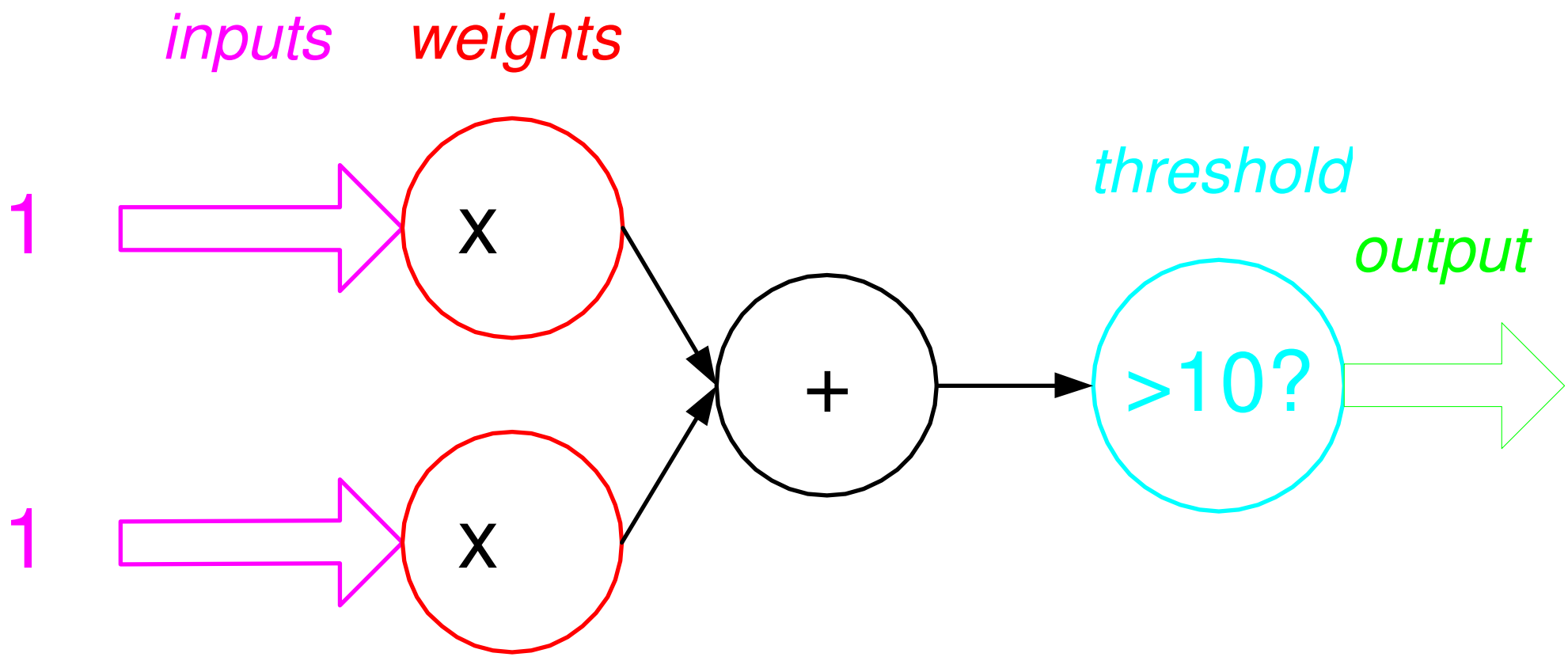
# Example

If both inputs are 1, output should be 1.



# Example

If both inputs are 1, output should be 1.




Repeat for all inputs until weights stop changing.

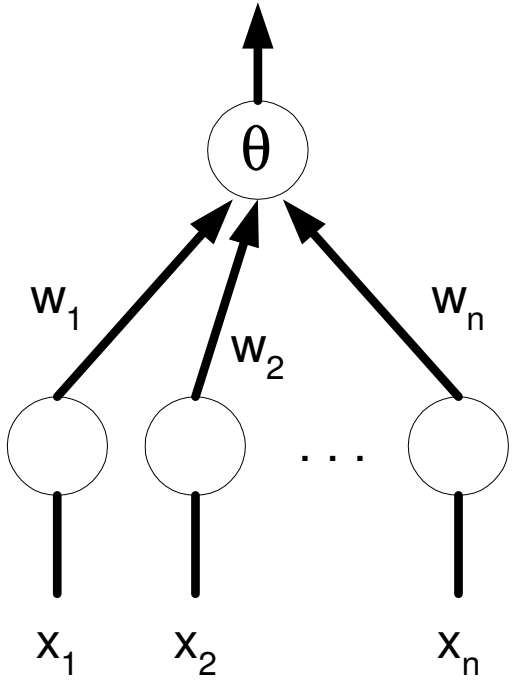
# Computations

## Consider the perceptron:

Multiple input nodes

Single output node

- Takes a weighted sum of the inputs, call this S
- Unit function calculates the output for the network

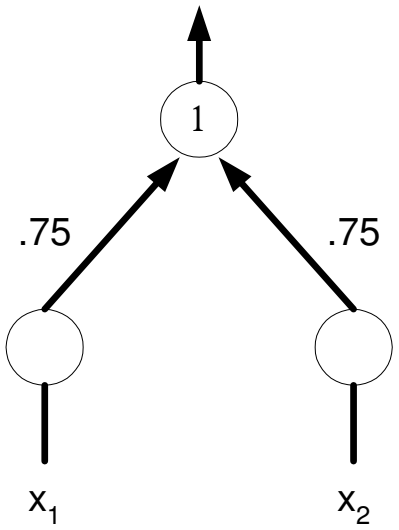



if $\Sigma w_i x_i >= \theta$, output = 1

if $\Sigma w_i x_i < \theta$, output = 0

# Computation via activation function

can view an artificial neuron as a computational element *accepts* or *classifies* an input if the output fires




INPUT: $x_1 = 1, x_2 = 1$

.75*1 + .75*1 = 1.5 >= 1 ➔ OUTPUT: 1

INPUT: $x_1 = 1, x_2 = 0$

.75*1 + .75*0 = .75 < 1 ➔ OUTPUT: 0

INPUT: $x_1 = 0, x_2 = 1$

.75*0 + .75*1 = .75 < 1 ➔ OUTPUT: 0

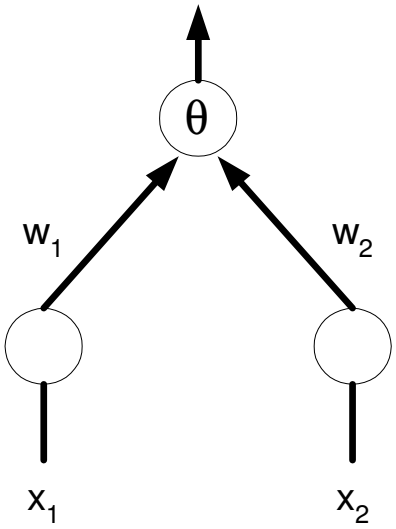
INPUT: $x_1 = 0, x_2 = 0$

.75*0 + .75*0 = 0 < 1 ➔ OUTPUT: 0

this neuron *computes* the AND function

# Exercise

## specify weights and thresholds to compute OR




INPUT: $x_1 = 1, x_2 = 1$

$w_1*1 + w_2*1 >= \theta$ ➔ OUTPUT: 1

INPUT: $x_1 = 1, x_2 = 0$

$w_1*1 + w_2*0 >= \theta$ ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 1$

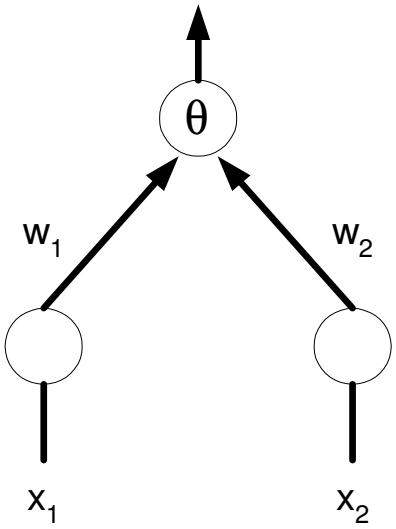
$w_1*0 + w_2*1 >= \theta$ ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 0$

$w_1*0 + w_2*0 < \theta$ ➔ OUTPUT: 0

# Another exercise?

specify weights and thresholds to compute XOR




INPUT: $x_1 = 1, x_2 = 1$

$w_1*1 + w_2*1 >= \theta$ ➔ OUTPUT: 0

INPUT: $x_1 = 1, x_2 = 0$

$w_1*1 + w_2*0 >= \theta$ ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 1$

$w_1*0 + w_2*1 >= \theta$ ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 0$

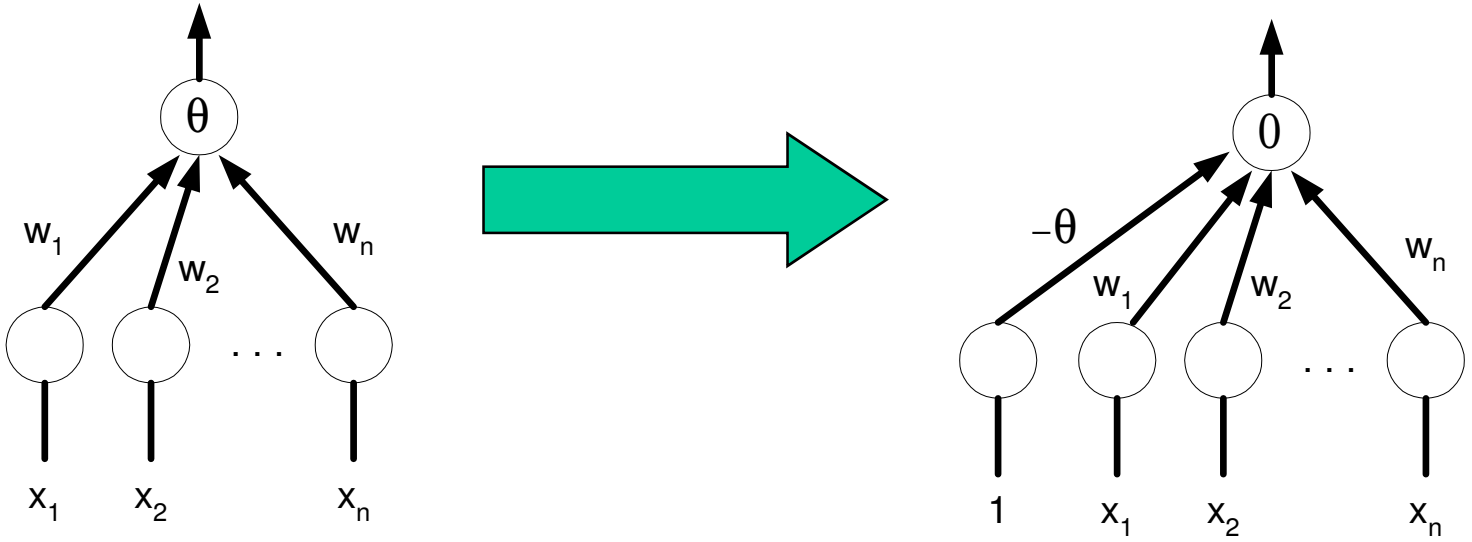
$w_1*0 + w_2*0 < \theta$ ➔ OUTPUT: 0

we'll come back to this later…

# Normalizing thresholds

to make life more uniform, can normalize the threshold to 0

simply add an additional input $x_0 = 1$, $w_0 = -\theta$

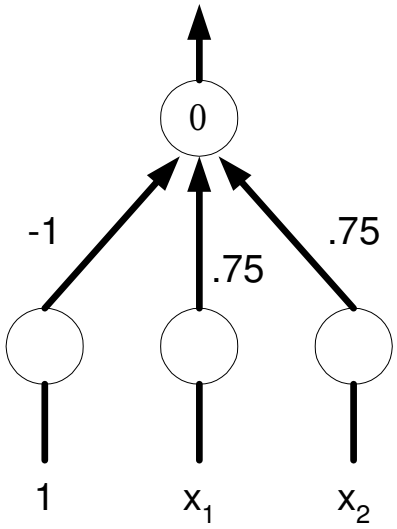



advantage: threshold = 0 for all neurons

$\Sigma w_i x_i >= \theta$ $\equiv$ $-\theta * 1 + \Sigma w_i x_i >= \mathbf{0}$

# Normalized examples

AND



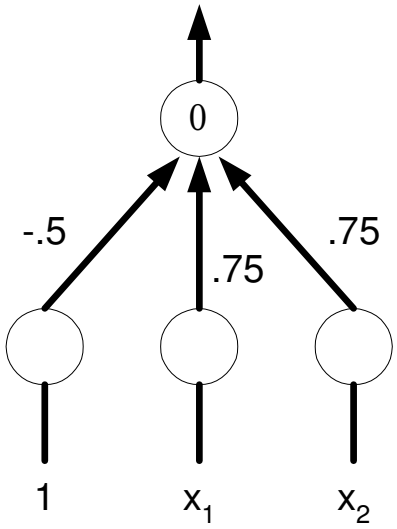

INPUT: $x_1 = 1, x_2 = 1$
1*-1 + .75*1 + .75*1 = .5 >= 0 ➔ OUTPUT: 1

INPUT: $x_1 = 1, x_2 = 0$
1*-1 +.75*1 + .75*0 = -.25 < 0 ➔ OUTPUT: 0

INPUT: $x_1 = 0, x_2 = 1$
1*-1 +.75*0 + .75*1 = -.25 < 0 ➔ OUTPUT: 0

INPUT: $x_1 = 0, x_2 = 0$
1*-1 +.75*0 + .75*0 = -1 < 0 ➔ OUTPUT: 0

OR




INPUT: $x_1 = 1, x_2 = 1$
1*-.5 + .75*1 + .75*1 = 1 >= 0 ➔ OUTPUT: 1

INPUT: $x_1 = 1, x_2 = 0$
1*-.5 +.75*1 + .75*0 = .25 >0 ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 1$
1*-.5 +.75*0 + .75*1 = .25 < 0 ➔ OUTPUT: 1

INPUT: $x_1 = 0, x_2 = 0$
1*-.5 +.75*0 + .75*0 = -.5 < 0 ➔ OUTPUT: 0

# Perceptrons

Rosenblatt (1958) devised a learning algorithm for artificial neurons

start with a training set (example inputs & corresponding desired outputs)

train the network to recognize the examples in the training set (by adjusting the weights on the connections)

once trained, the network can be applied to new examples

*Perceptron* learning algorithm:

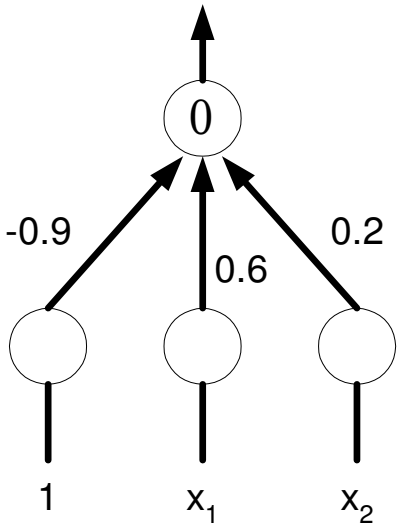
1. Set the weights on the connections with random values.
2. Iterate through the training set, comparing the output of the network with the desired output for each example.
3. If all the examples were handled correctly, then DONE.
4. Otherwise, update the weights for each incorrect example:
   - if should have fired on $x_1, \ldots, x_n$ but didn't, $w_i \mathrel{+}= x_i$ (0 <= i <= n)
   - if shouldn't have fired on $x_1, \ldots, x_n$ but did, $w_i \mathrel{-}= x_i$ (0 <= i <= n)
5. GO TO 2

# Example: perceptron learning

Suppose we want to train a perceptron to compute AND

training set:

$x_1 = 1, x_2 = 1 \rightarrow 1$
$x_1 = 1, x_2 = 0 \rightarrow 0$
$x_1 = 0, x_2 = 1 \rightarrow 0$
$x_1 = 0, x_2 = 0 \rightarrow 0$



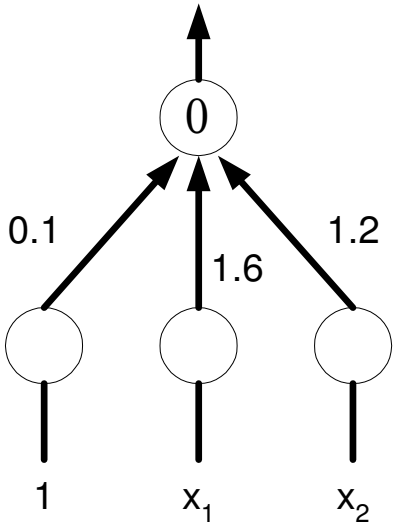

randomly, let: $w_0 = -0.9$, $w_1 = 0.6$, $w_2 = 0.2$

using these weights:

| | | | |
|---|---|---|---|
| $x_1 = 1, x_2 = 1$: | -0.9*1 + 0.6*1 + 0.2*1 | = -0.1 → 0 | WRONG |
| $x_1 = 1, x_2 = 0$: | -0.9*1 + 0.6*1 + 0.2*0 | = -0.3 → 0 | OK |
| $x_1 = 0, x_2 = 1$: | -0.9*1 + 0.6*0 + 0.2*1 | = -0.7 → 0 | OK |
| $x_1 = 0, x_2 = 0$: | -0.9*1 + 0.6*0 + 0.2*0 | = -0.9 → 0 | OK |

new weights:
$w_0 = -0.9 + 1 = 0.1$
$w_1 = 0.6 + 1 = 1.6$
$w_2 = 0.2 + 1 = 1.2$

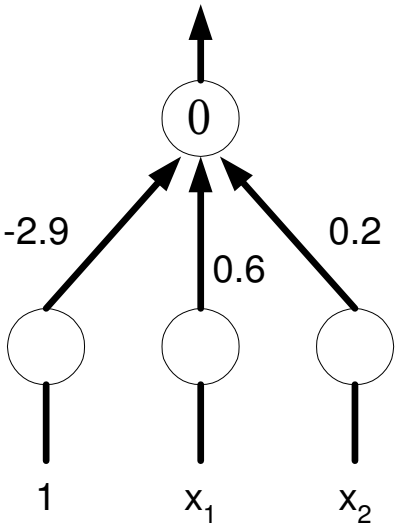
# Example: perceptron learning




using these updated weights:

| | | | |
|---|---|---|---|
| $x_1 = 1, x_2 = 1$: | 0.1*1 + 1.6*1 + 1.2*1 | = 2.9 → 1 | OK |
| $x_1 = 1, x_2 = 0$: | 0.1*1 + 1.6*1 + 1.2*0 | = 1.7 → 1 | WRONG |
| $x_1 = 0, x_2 = 1$: | 0.1*1 + 1.6*0 + 1.2*1 | = 1.3 → 1 | WRONG |
| $x_1 = 0, x_2 = 0$: | 0.1*1 + 1.6*0 + 1.2*0 | = 0.1 → 1 | WRONG |

new weights:
$w_0 = 0.1 - 1 - 1 - 1 = -2.9$
$w_1 = 1.6 - 1 - 0 - 0 = 0.6$
$w_2 = 1.2 - 0 - 1 - 0 = 0.2$

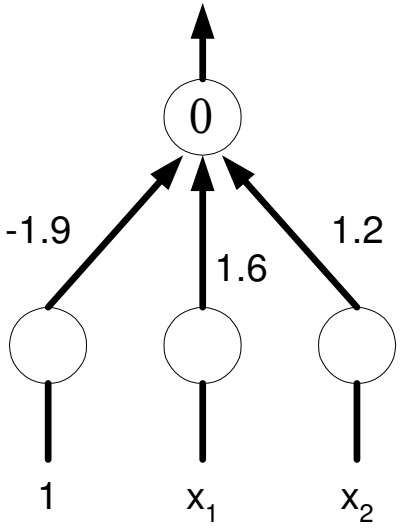



using these updated weights:

| | | | |
|---|---|---|---|
| $x_1 = 1, x_2 = 1$: | -2.9*1 + 0.6*1 + 0.2*1 | = -2.1 → 0 | WRONG |
| $x_1 = 1, x_2 = 0$: | -2.9*1 + 0.6*1 + 0.2*0 | = -2.3 → 0 | OK |
| $x_1 = 0, x_2 = 1$: | -2.9*1 + 0.6*0 + 0.2*1 | = -2.7 → 0 | OK |
| $x_1 = 0, x_2 = 0$: | -2.9*1 + 0.6*0 + 0.2*0 | = -2.9 → 0 | OK |

new weights:
$w_0 = -2.9 + 1 = -1.9$
$w_1 = 0.6 + 1 = 1.6$
$w_2 = 0.2 + 1 = 1.2$

# Example: perceptron learning




using these updated weights:

| | | | |
|---|---|---|---|
| $x_1 = 1, x_2 = 1$: | -1.9*1 + 1.6*1 + 1.2*1 | = 0.9 → 1 | OK |
| $x_1 = 1, x_2 = 0$: | -1.9*1 + 1.6*1 + 1.2*0 | = -0.3 → 0 | OK |
| $x_1 = 0, x_2 = 1$: | -1.9*1 + 1.6*0 + 1.2*1 | = -0.7 → 0 | OK |
| $x_1 = 0, x_2 = 0$: | -1.9*1 + 1.6*0 + 1.2*0 | = -1.9 → 0 | OK |

DONE!

EXERCISE: train a perceptron to compute OR

# Learning Algorithm

- Weights, initially, are set randomly
- For each training example E

  Calculate the observed output from the ANN, o(E)

  If the target output t(E) is different from o(E)

  - Then tweak all the weights so that o(E) gets closer to t(E)
  - Tweaking is done by perceptron training rule
  - This routine is done for every example E
- Don't necessarily stop when all examples used

  Repeat the cycle again (an 'epoch') Until the ANN produces the correct output for "all " the examples in the training set (or good enough)

# Perceptron training alg.

$$\Delta w_i = c(d\text{-}sign((\Sigma x_i w_i ))x_i$$

Where c is the learning rate, d is the desired output and $sign(\Sigma x_i w_i )$ is the actual output

If the desired output and actual output are equal, do nothing

If the actual value is -1 and should be 1, increment the weights on the ith line by "$2c\, x_i$"

If the actual value is 1 and should be -1, decrement the weights on the ith line by "$2c\, x_i$"

*i.e.* We can think of the addition of $\Delta w_i$ as the movement of the weight in a direction Which will improve the networks performance with respect to the example. Multiplication by xi, Moves it more if the input is bigger

# The Learning Rate

$$\Delta w_i = c(d\text{-}sign((\Sigma x_i w_i ))x_i$$

- c (in some books η) is called the learning rate, Usually set to something small (e.g., 0.1 or less), d is the desired output
- To control the movement of the weights Not to move too far for one example Which may over-compensate for another example
- If a large movement is actually necessary for the weights to correctly categorise the example

  This will occur over time with multiple epochs

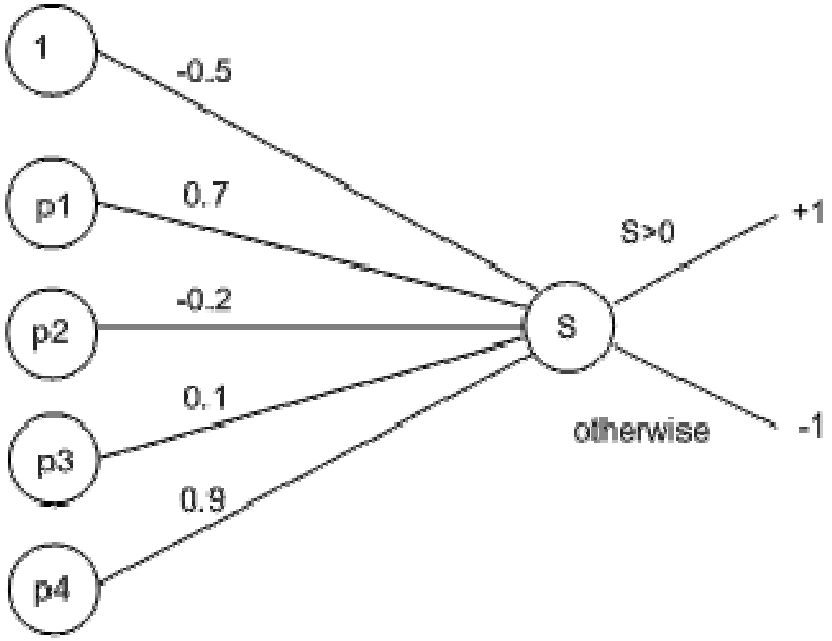
# Example

Suppose we want to train this network with

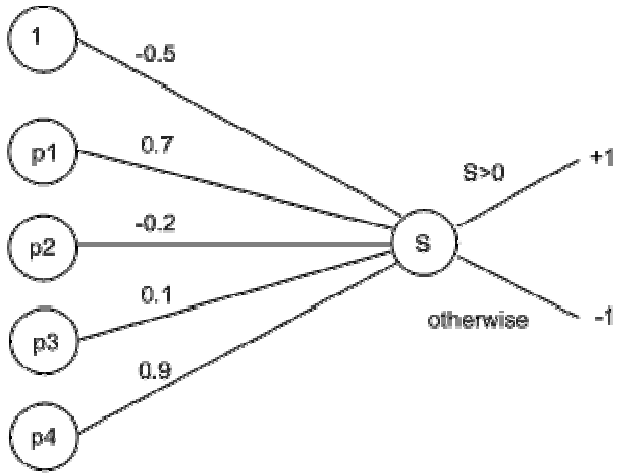
Inputs: x1 = -1, x2 = 1, x3 = 1, x4 = -1, and output 1

Use a learning rate of η = 0.1

Suppose we have set random weights:



# The Error Values




$\Delta w_i = \eta(d\text{-}sign((\Sigma x_i w_i))x_i$ , $\eta = 0.1$

x1 = -1, x2 = 1, x3 = 1, x4 = -1

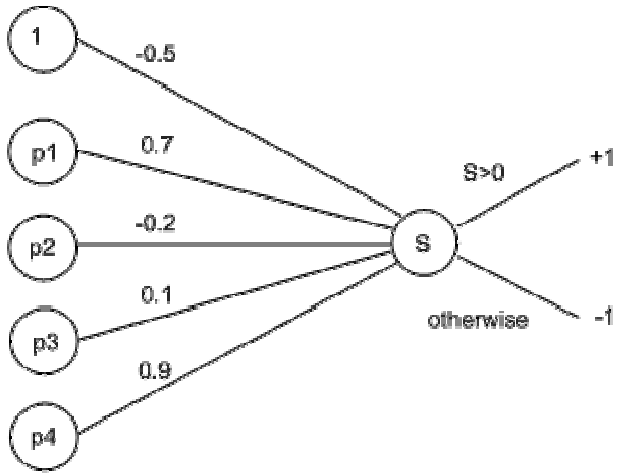
Propagate this information through the network:

S = (-0.5 * 1) + (0.7 * -1) + (-0.2 * 1) + (0.1 * 1) + (0.9 * (-1)) = -2.2

Hence the network outputs -1

But this should have been +1

# The Error Values




$\Delta w_i = \eta(d\text{-}sign((\Sigma x_i w_i))x_i$ , $\eta = 0.1$

x1 = -1, x2 = 1, x3 = 1, x4 = -1

Now: real output d=1 while calculated $sign\Sigma x_i w_i = -1$

$\Delta w_0 = 0.1 * (1 - (-1)) * (1) = 0.1 * (2) = 0.2$

$\Delta w_1 = 0.1 * (1 - (-1)) * (-1) = 0.1 * (-2) = -0.2$

$\Delta w_2 = 0.1 * (1 - (-1)) * (1) = 0.1 * (2) = 0.2$

$\Delta w_3 = 0.1 * (1 - (-1)) * (1) = 0.1 * (2) = 0.2$

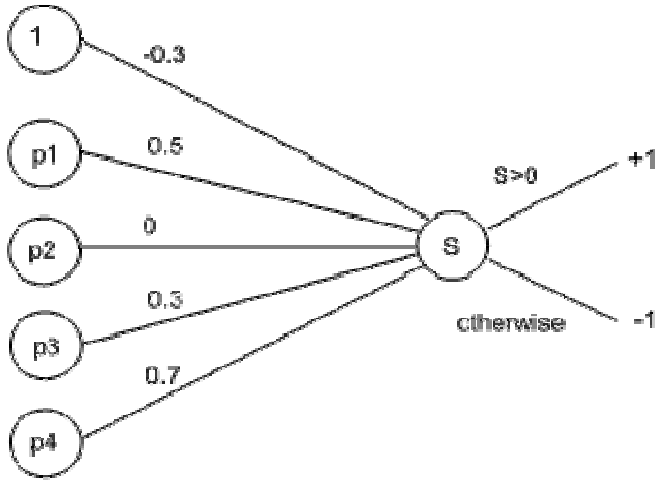
$\Delta w_4 = 0.1 * (1 - (-1)) * (-1) = 0.1 * (-2) = -0.2$

New Weights: $w'_0 = -0.5 + \Delta w_0 = -0.5 + 0.2 = -0.3$

$w'_1 = 0.7 + -0.2 = 0.5$ $\quad$ $w'_2 = -0.2 + 0.2 = 0$

$w'_3 = 0.1 + 0.2 = 0.3$ $\quad$ $w'_4 = 0.9 - 0.2 = 0.7$

# New Perceptron




Using the new weights:

$w'_0$=-0.3, $w'_1$=0.5, $w'_2$=0,$w'_3$=0.3,$w'_4$ = 0.7

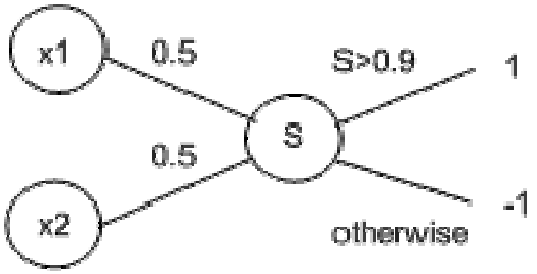
Calculating again: (x1 = -1, x2 = 1, x3 = 1, x4 = -1)

S = (-0.3 * 1) + (0.5 * -1) + (0 * +1) + (0.3 * +1) + (0.7 * -1) = -1.2

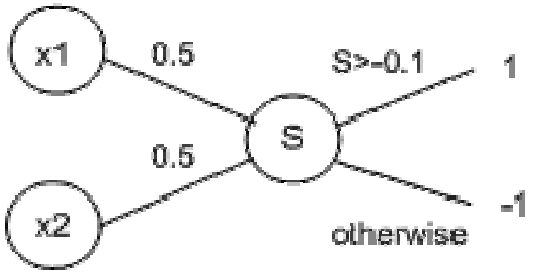
Still gets the wrong categorisation

- But the value is closer to zero (from -2.2 to -1.2)
- In a few epochs time, this example will be correctly categorised

# Boolean Functions as Perceptrons




An ANN for AND

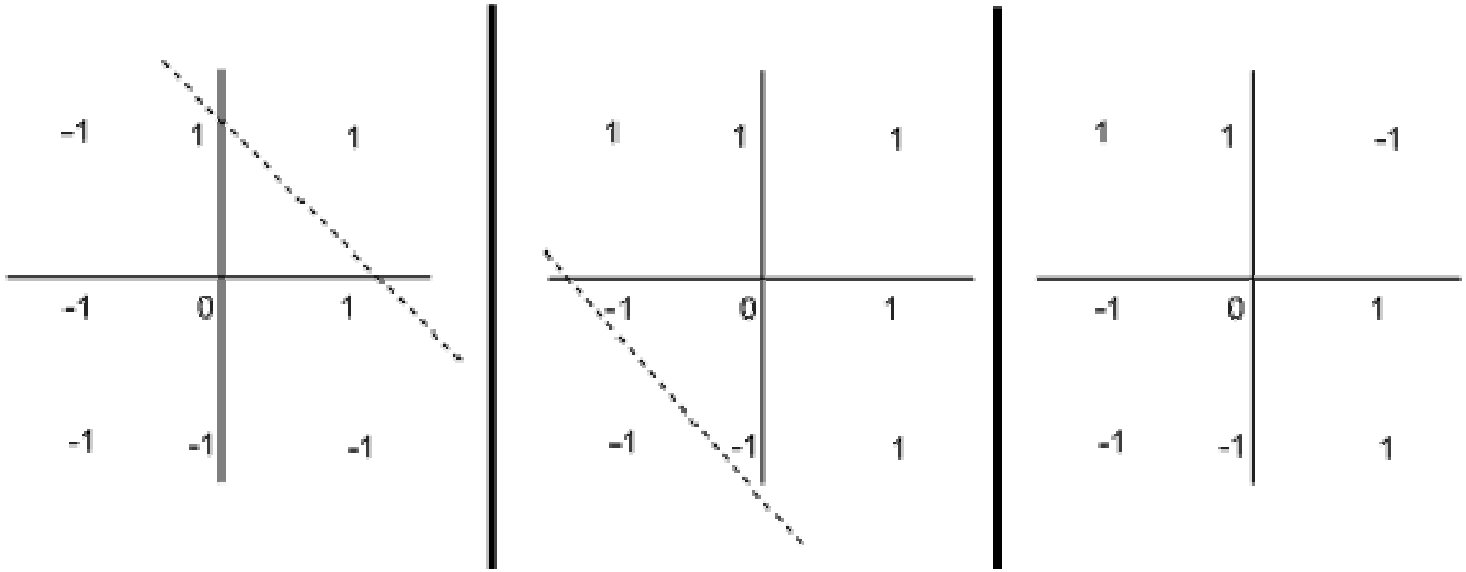



An ANN for OR

- Perceptrons are very simple networks
- Perceptrons cannot learn some simple Boolean functions.
- Killed the ANNs in AI for many years
  - People thought it represented a fundamental limitation
  - But perceptrons are the simplest network ANNS were revived by neuroscientists later, etc.
- XOR boolean function cannot be represented as a perceptron because it is NOT

linearly separable

# Linearly Separable Boolean Functions

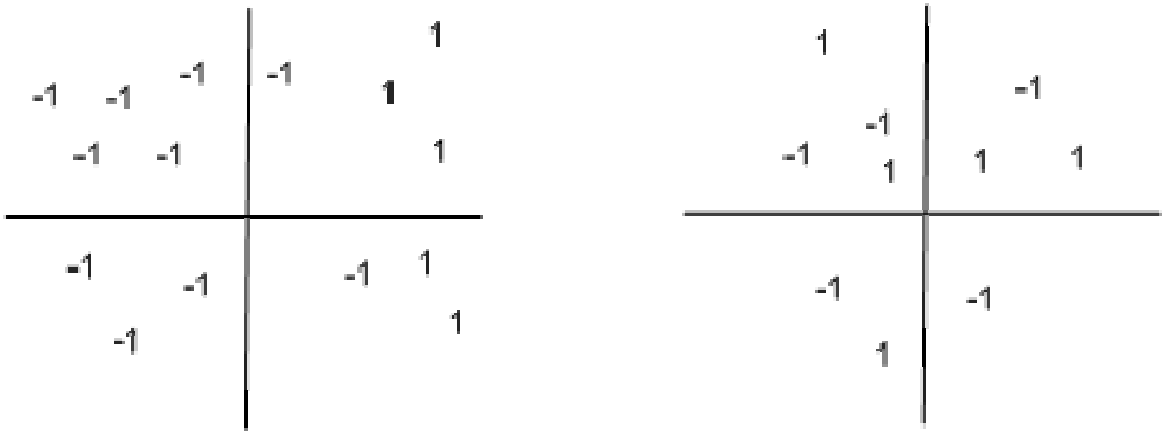



Linearly separable:

Can use a line (dotted) to separate +1 and –1

Theorem: There is a perceptron that will learn any linearly separable function, given enough training examples.

# Linearly Separable Functions




Result extends to functions taking many inputs

And outputting +1 and –1

Also extends to higher dimensions for outputs

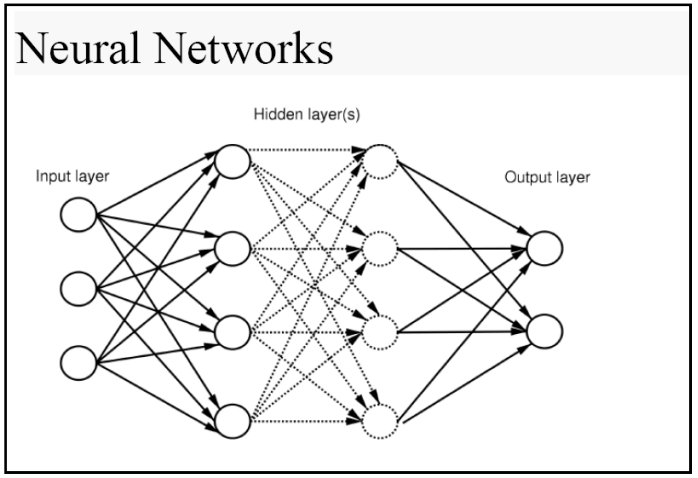
# Perceptron Convergence

**Perceptron convergence theorem**: If the data is linearly separable and therefore a set of weights exist that are consistent with the data, then the Perceptron algorithm will eventually converge to a consistent set of weights.

**Perceptron cycling theorem**: If the data is not linearly separable, the Perceptron algorithm will eventually repeat a set of weights and threshold at the end of some epoch and therefore enter an infinite loop.

By checking for repeated weights+threshold, one can guarantee termination with either a positive or negative result.

# N-layer Feed Forward Network




Layer 0 is input nodes

Layers 1 to N-1 are hidden nodes

Layer N is output nodes

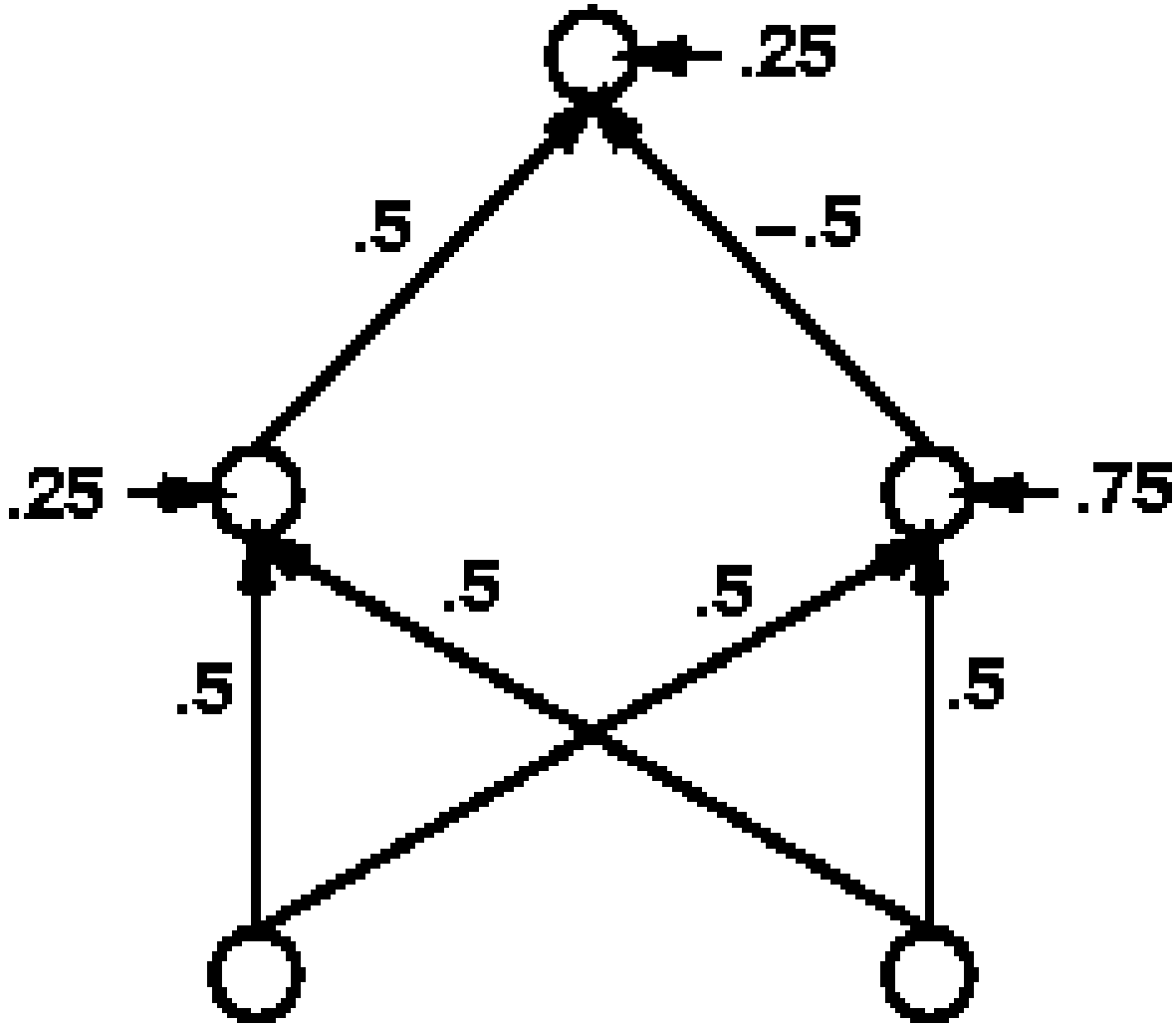
All nodes at any layer k are connected to all nodes at layer k+1

There are no cycles

Theorem:

Given an arbitrary number of hidden units, any Boolean function can be computed with a single hidden layer.

# XOR Solution



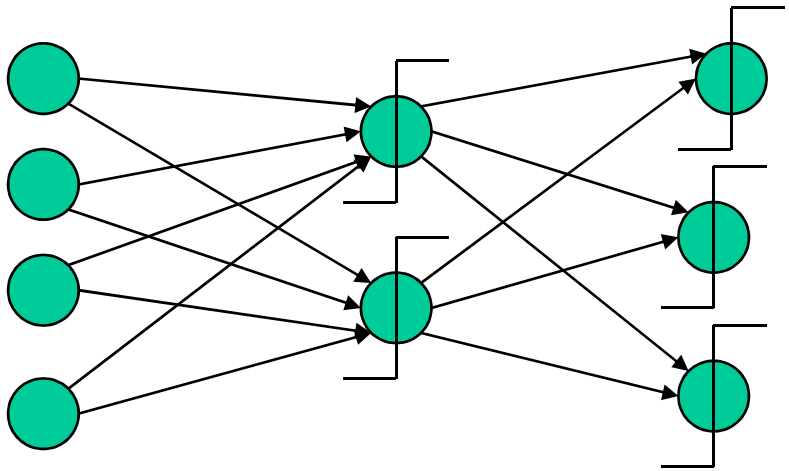
# Multi-Layer Networks Built from Perceptron Units

Perceptrons are not able to learn certain concepts

Can only learn linearly separable functions

But they can be the basis for larger structures

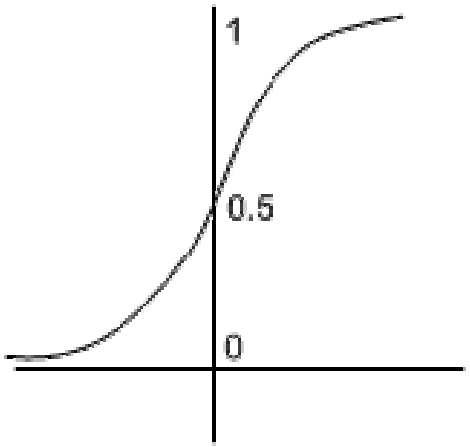
- Which can learn more sophisticated concepts
- Say that the networks have "perceptron units"



# Problem With Perceptron Units

- Needs the output of a unit to be a differentiable function
- That is: The learning rule relies on minimizing the error. Finding minima by differentiating.
- Step functions aren't differentiable. They are not continuous
- Alternative threshold function are to be used
  - Must be differentiable
  - Must be similar to step function
- Sigmoid units used for backpropagation
  (There are other alternatives that may be used)

# Sigmoid Units




Take in weighted sum of inputs, S

Then the out isoutput:

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

Advantages:

- Looks very similar to the step function
- Is differentiable
- Derivative easily expressible in terms of σ itself:

$$\frac{d\sigma(S)}{dS} = \sigma(S)(1 - \sigma(S))$$

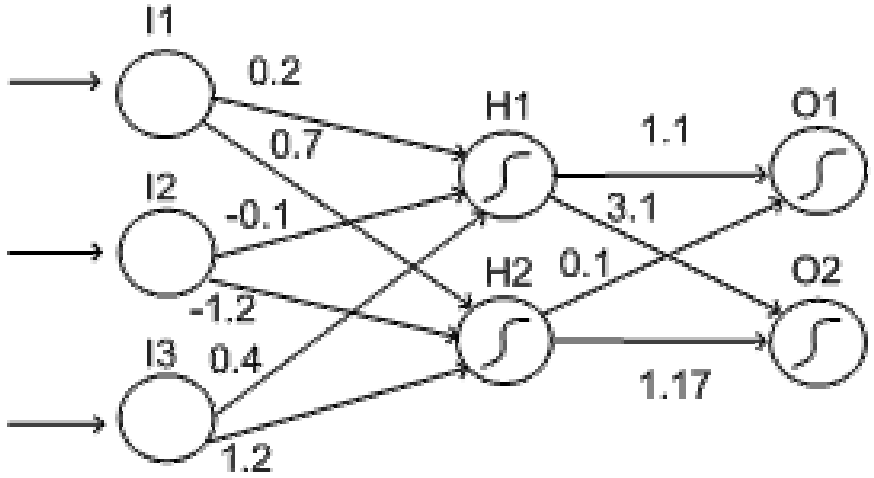
# Example ANN with Sigmoid Units

Feed forward network

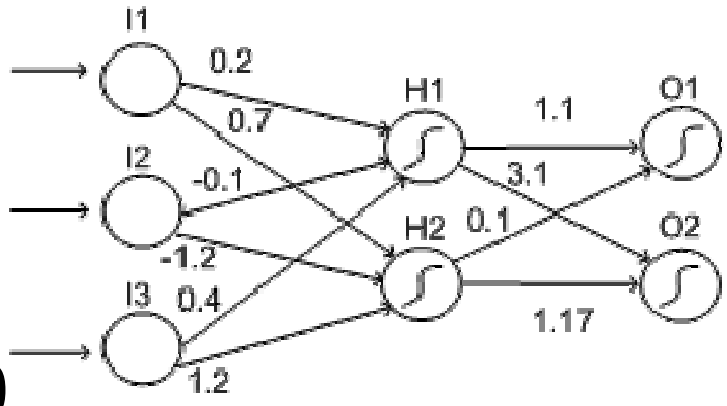
- Feed inputs on the left, propagate numbers forward

Suppose we have this ANN used for categoriztion

- With weights set arbitrary



# Propagation of Example




With an example E:

Suppose the input to this ANN is 10, 30, 20

First calculate weighted sums to the hidden layer:

$S_{H1} = (0.2*10) + (-0.1*30) + (0.4*20) = 2-3+8 = 7$

$S_{H2} = (0.7*10) + (-1.2*30) + (1.2*20) = 7-36+24= -5$

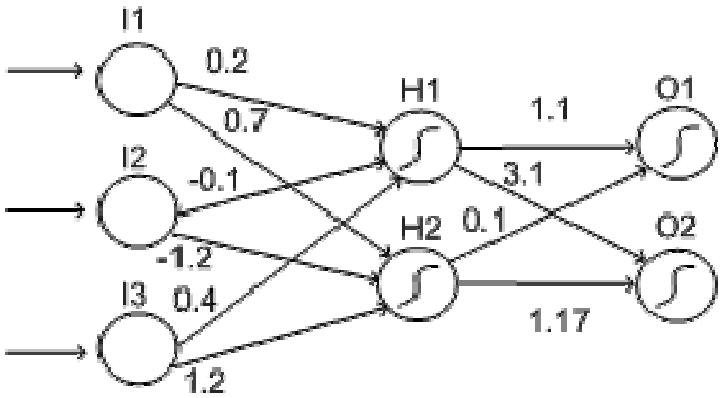
Next calculate the output from the hidden layer Using:

$\sigma(S) = 1/(1 + e^{-S})$

$\sigma(S_{H1}) = 1/(1 + e^{-7}) = 1/(1+0.000912) = 0.999$

$\sigma(S_{H2}) = 1/(1 + e^{5}) = 1/(1+148.4) = 0.0067$

# Propagation of Example




Next calculate the weighted sums into the output layer:

$S_{O1} = (1.1 * 0.999) + (0.1 * 0.0067) = 1.0996$

$S_{O2} = (3.1 * 0.999) + (1.17 * 0.0067) = 3.1047$

Finally, calculate the output from the ANN

$\sigma(S_{O1}) = 1/(1+e^{-1.0996}) = 1/(1+0.333) = 0.750$

$\sigma(S_{O2}) = 1/(1+e^{-3.1047}) = 1/(1+0.045) = 0.957$

Output from O2 > output from O1

So, the ANN predicts category associated with O2

For the example input (10,30,20)

# Propagate E through the Network

Feed E through the network (as in example above)

Record the target and observed values for example E

- i.e., determine weighted sum from hidden units, do sigmoid calc
- Let $t_i(E)$ be the target values for output unit i
- Let $o_i(E)$ be the observed value for output unit i

For categorisation learning tasks,

- Each $t_i(E)$ will be 0, except for a single $t_j(E)$, which will be 1
- But $o_i(E)$ will be a real valued number between 0 and 1

Also record the outputs from the hidden units

- Let $h_i(E)$ be the output from hidden unit i

# Backpropagation Learning Algorithm

- Same task as in perceptrons
  - Learn a multi-layer ANN to correctly categorise unseen examples
  - We'll concentrate on ANNs with one hidden layer
- Overview of the routine
  - Fix architecture and sigmoid units within architecture
    - i.e., number of units in hidden layer; the way the input units represent example; the way the output units categorises examples
  - Randomly assign weights to the the whole network
    - Use small values (between –0.5 and 0.5)
  - Use each example in the set to retrain the weights
  - Have multiple epochs (iterations through training set)
    - Until some termination condition is met (not necessarily 100% acc)

# Weight Training

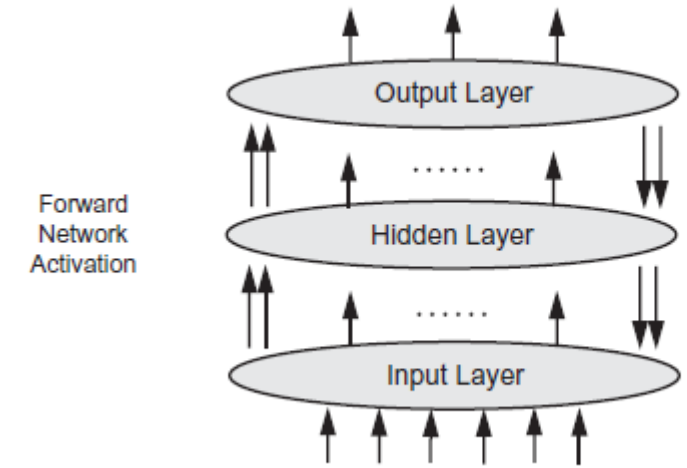



Figure 11.9 Backpropagation in a connectionist network having a hidden layer.

The Back propagation algorithm is

1. start at the output layer and
2. propagate error backwards through the hidden layer

- Use notation $w_{ij}$ to specify: Weight between unit i and unit j
- Look at the calculation with respect to example E
- Calculate a value $\Delta_{ij}$ for each $w_{ij}$ And add $\Delta_{ij}$ on to $w_{ij}$
- Do this by calculating **error terms** for each unit
- The error term for output units is found And then this information is used to calculate the error terms for the hidden units

So, the error is propagated back through the ANN

# Error terms for each unit

The Error Term for output unit k is calculated as:

$$\delta_{O_k} = o_k(E)(1 - o_k(E))(t_k(E) - o_k(E))$$

The Error Term for hidden unit k is:

$$\delta_{H_k} = h_k(E)(1 - h_k(E)) \sum_{i \in outputs} w_{ki}\delta_{O_i}$$

i.e. For hidden unit h, add together all the errors for the output units, multiplied by the appropriate weight. Then multiply their sum by $h_k(E)(1 - h_k(E))$

# Final Calculations

Choose a learning rate, η (= 0.1 say)

For each weight $w_{ij}$

Between input unit i and hidden unit j

Calculate: $\Delta w_{ij} = \eta \delta_{Hi} x_i$

Where $x_i$ is the input to the system to input unit i for E

For each weight $w_{ij}$ between hidden unit i and output unit j

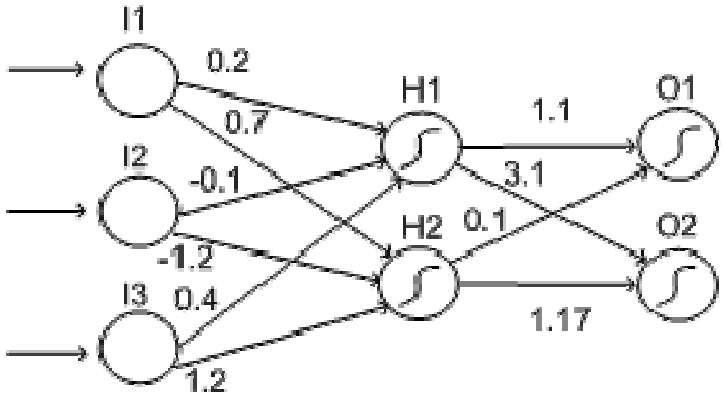
Calculate: $\Delta w_{ij} = \eta \delta_{Oi} h_i(E)$

Where $h_i(E)$ is the output from hidden unit i for E

Finally, add on each $\Delta w_{ij}$ on to $w_{ij}$

# Worked Backpropagation Example

Start with the previous ANN




We will retrain the weights

- In the of example E = (10,30,20)
- Assume that E should have been categorised as O1 (not O2 as the calculated result)
- Will use a learning rate of η = 0.1

# Previous Calculations

Need the calculations from when we propagated E through the ANN:

| Input units | | Hidden units | | | Output units | | |
|---|---|---|---|---|---|---|---|
| Unit | Output | Unit | Weighted Sum Input | Output | Unit | Weighted Sum Input | Output |
| I1 | 10 | H1 | 7 | 0.999 | O1 | 1.0996 | 0.750 |
| I2 | 30 | H2 | -5 | 0.0067 | O2 | 3.1047 | 0.957 |
| I3 | 20 | | | | | | |

$o_1(E) = 0.750$ and $o_2(E) = 0.957$

$t_1(E) = 1$ and $t_2(E) = 0$ [Assumption says it should be O1]

# Error Values for Output Units

$t_1(E) = 1$ and $t_2(E) = 0$ $o_1(E) = 0.750$ and $o_2(E) = 0.957$

So:

$$\delta_{O_k} = o_k(E)(1 - o_k(E))(t_k(E) - o_k(E))$$

$$\delta_{O1} = o_1(E)(1 - o_1(E))(t_1(E) - o_1(E)) = 0.750(1-0.750)(1-0.750) = 0.0469$$

$$\delta_{O2} = o_2(E)(1 - o_2(E))(t_2(E) - o_2(E)) = 0.957(1-0.957)(0-0.957) = -0.0394$$

# Error Values for Hidden Units

Now: $\delta_{O1} = 0.0469$ and $\delta_{O2} = -0.0394$

$h_1(E) = 0.999$ and $h_2(E) = 0.0067$ (output of hidden from the table)

$$\delta_{H_k} = h_k(E)(1 - h_k(E)) \sum_{i \in outputs} w_{ki}\delta_{O_i}$$

---

So, for H1, we add together:

$(w_{11}*\delta_{01}) + (w_{12}*\delta_{O2}) = (1.1*0.0469)+(3.1*-0.0394) = -0.0706$

And multiply by: $h_1(E)(1-h_1(E))$ to give us:

$\delta_{H1} = -0.0706 * (0.999 * (1-0.999)) = 0.0000705$

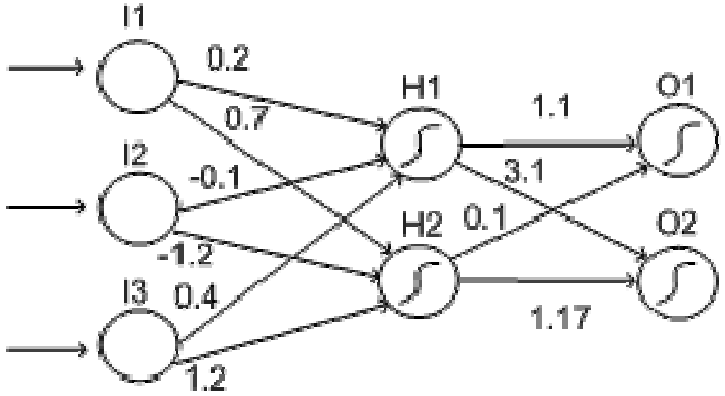
For H2, we add together:

$(w_{21}*\delta_{01}) + (w_{22}*\delta_{O2}) = (0.1*0.0469)+(1.17*-0.0394) = -0.0414$

And multiply by: $h_2(E)(1-h_2(E))$ to give us:

$\delta_{H2} = -0.0414 * (0.067 * (1-0.067)) = -0.00259$

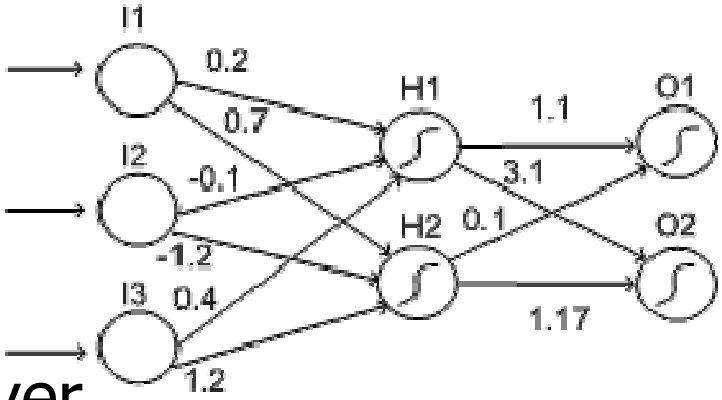
# Calculation of Weight Changes




For weights between the input and hidden layer each $w_{ij}$ And add $\Delta_{ij}$ on to $w_{ij}$

| Input unit | Hidden unit | $\eta$ | $\delta_H$ | $x_i$ | $\Delta = \eta * \delta_H * x_i$ | Old weight | New weight |
|---|---|---|---|---|---|---|---|
| I1 | H1 | 0.1 | -0.0000705 | 10 | -0.0000705 | 0.2 | 0.1999295 |
| I1 | H2 | 0.1 | -0.00259 | 10 | -0.00259 | 0.7 | 0.69741 |
| I2 | H1 | 0.1 | -0.0000705 | 30 | -0.0002115 | -0.1 | -0.1002115 |
| I2 | H2 | 0.1 | -0.00259 | 30 | -0.00777 | -1.2 | -1.20777 |
| I3 | H1 | 0.1 | -0.0000705 | 20 | -0.000141 | 0.4 | 0.39999 |
| I3 | H2 | 0.1 | -0.00259 | 20 | -0.00518 | 1.2 | 1.1948 |

# Calculation of Weight Changes




For weights between hidden and output layer

| Hidden unit | Output unit | $\eta$ | $\delta_O$ | $h_i(E)$ | $\Delta = \eta * \delta_O * h_i(E)$ | Old weight | New weight |
|---|---|---|---|---|---|---|---|
| H1 | O1 | 0.1 | 0.0469 | 0.999 | 0.000469 | 1.1 | 1.100469 |
| H1 | O2 | 0.1 | -0.0394 | 0.999 | -0.00394 | 3.1 | 3.0961 |
| H2 | O1 | 0.1 | 0.0469 | 0.0067 | 0.00314 | 0.1 | 0.10314 |
| H2 | O2 | 0.1 | -0.0394 | 0.0067 | -0.0000264 | 1.17 | 1.16998 |

Weight changes are not very large

- Small differences in weights can make big differences in calculations
- But it might be a good idea to increase $\eta$

# Calculation of Network Error

- Could calculate Network error as
  - Proportion of mis-categorised examples
- But there are multiple output units, with numerical output
  - So we use a more sophisticated measure:

$$\frac{1}{2} \sum_{E \in examples} \left( \sum_{k \in outputs} (t_k(E) - o_k(E))^2 \right)$$

  - Square the difference between target and observed
    - Squaring ensures we get a positive number
- Add up all the squared differences
  - For every output unit and every example in training set

# Backpropagation Training Algorithm

The algorithm is composed of two parts that get repeated over and over a number of ***epochs***.

I. The ***feedforward*** : the activation values of the hidden and then output units are computed.

II. The ***backpropagation*** : the weights of the network are updated--starting with the hidden to output weights and followed by the input to hidden weights--with respect to the sum of squares error, the ***Delta Rule***.

# Backpropagation Training Algorithm

Until all training examples produce the correct value (within ε), or mean squared error stops to decrease, or other termination criteria:

Begin epoch

For each training example, E, do:

Calculate network output for E's input values

Compute error between current output and correct output or E

Update weights by backpropagating error and using learning rule

End epoch

# Backpropagation: The Momentum

Backpropagation has the disadvantage of being too slow if $\eta$, the learning rate, is small and it can oscillate too widely if $\eta$ is large.

To solve this problem, we can add a ***momentum*** to give each connection some inertia, forcing it to change in the direction of the downhill "force".

Old Delta Rule: $\Delta w_{ij} = \eta \delta_{Hi} x_i$ , $\Delta w_{ij} = \eta \delta_{Oi} h_i(E)$

**New Delta Rule:** $\Delta w_{ij}(t+1) = \eta \delta_{Hi} x_i + \alpha \Delta w_{ij}(t)$

And $\Delta w_{ij}(t+1) = \eta \delta_{Oi} h_i(E) + \alpha \Delta w_{ij}(t)$

where i,j are any input and hidden, or, hidden and output units;
t is a time step or epoch;
and $\alpha$ is the momentum parameter which regulates the amount of inertia of the weights.

# Backpropagation Training Algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- In practice, it does converge to low error for many large networks on real data.
- Many epochs (thousands) may be required, hours or days of training for large networks.
- To avoid local-minima problems, run several trials starting with different random weights (***random restarts***).

# Hidden Unit Representations

Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.

On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc..

However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.

# ANN Representing function

**Boolean functions**: Any Boolean function can be represented by a two-layer network with sufficient hidden units.

**Continuous functions**: Any bounded continuous function can be approximated with arbitrarily small error by a two-layer network.

**Arbitrary function**: Any function can be approximated to arbitrary accuracy by a three-layer network.

# Applications of ANNs

Fraud detection

9 of top 10 US credit card companies use Falcon uses neural nets to model customer behavior, identify fraud claims

Prediction & Financial Analysis

In Banks: financial forecasting, investing, marketing analysis

control & optimization

- Intel – computer chip manufacturing quality control
- AT&T (cell phones) – echo & noise control in phone lines (filters and compensates)
- Ford engines utilize neural net chip to diagnose misfirings, reduce emissions

Text to Speech (NetTalk)

Handwriting recognition

Face recognition

Optical character recognition (OCR)

Game Playing