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

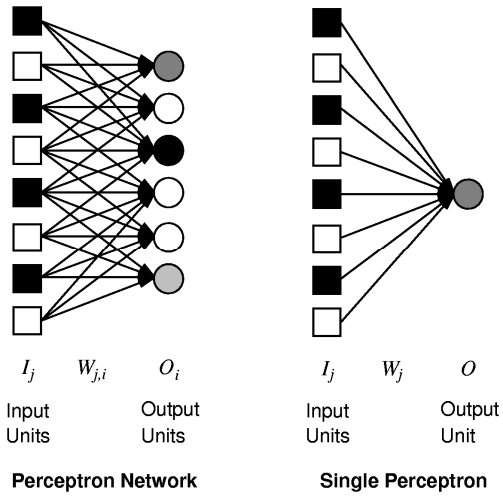
- First studied in the late 1950s.

- Also known as Layered Feed-Forward Networks.

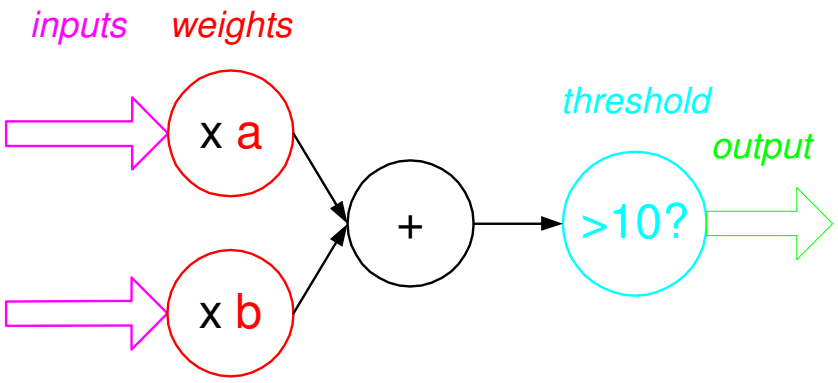
- The only efficient learning element at that time was for single-layered networks.

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

Perceptrons
Early neural nets



Perceptron (artificial neuron)



Example of Training

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

Initially, weights are random

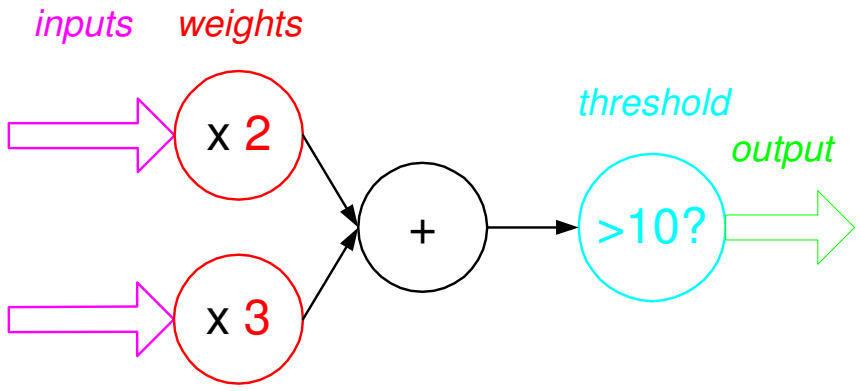
Provide training input

Compare output of neural network to desired output

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

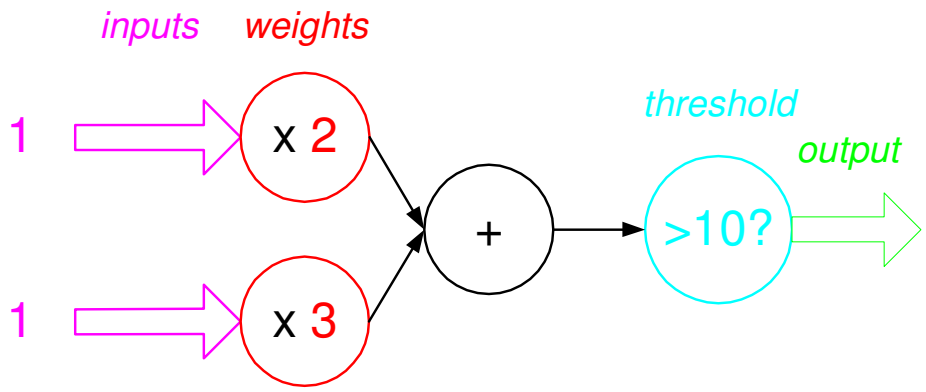
Example

If both inputs are 1, output should be 1.



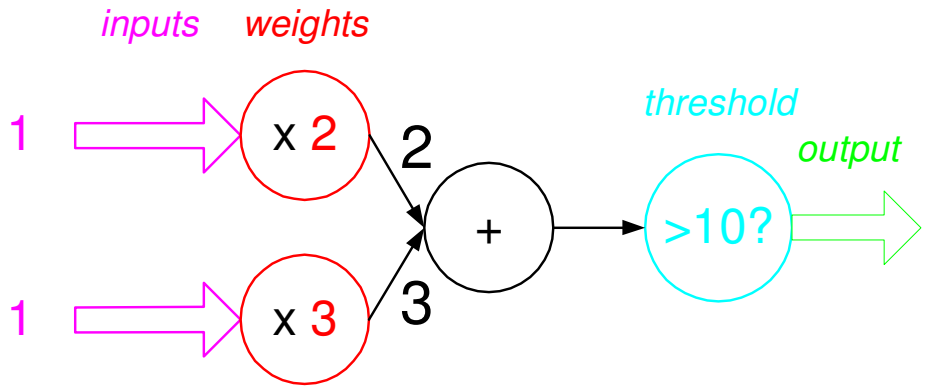
Example

If both inputs are 1, output should be 1.



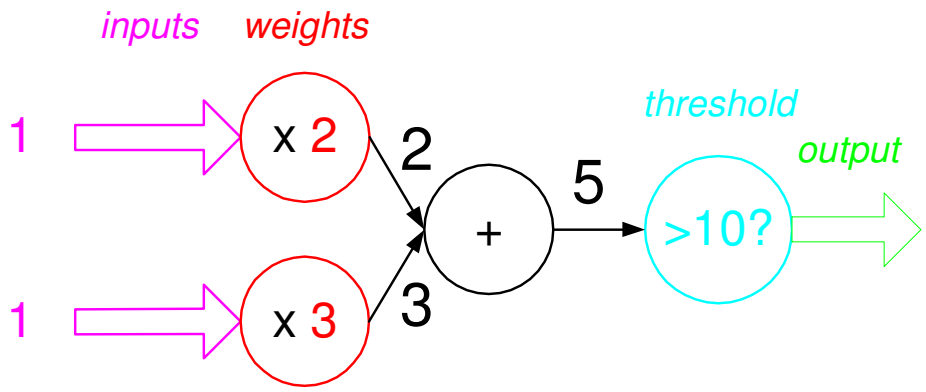
Example

If both inputs are 1, output should be 1.



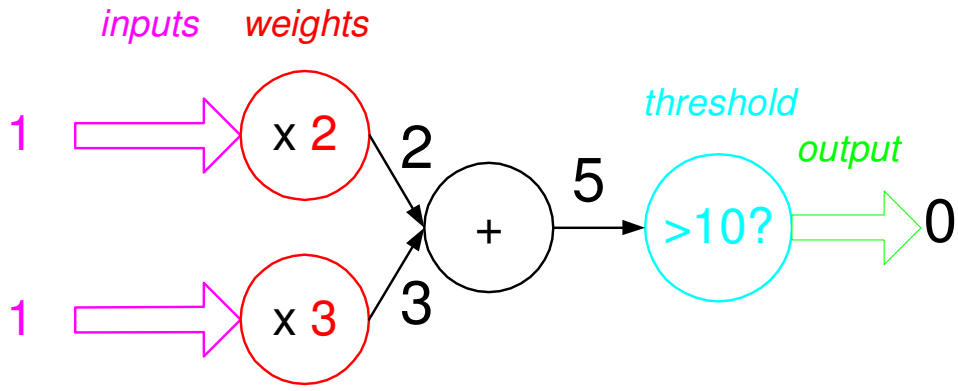
Example

If both inputs are 1, output should be 1.



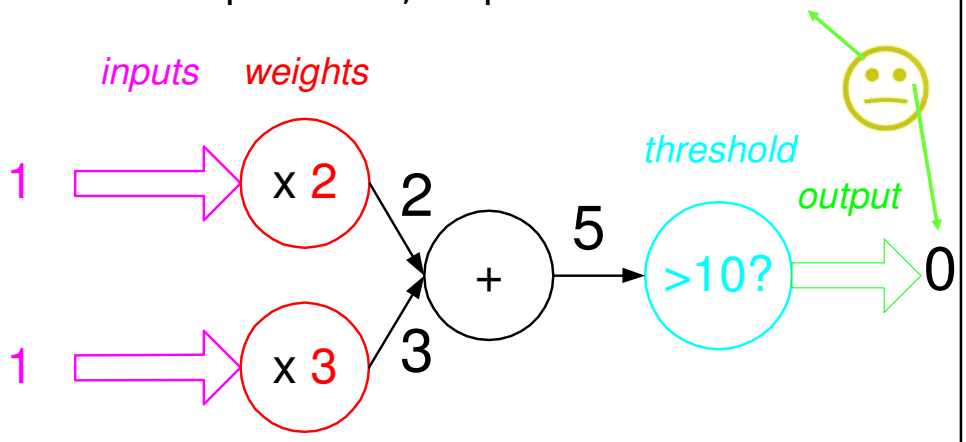
Example

If both inputs are 1, output should be 1.



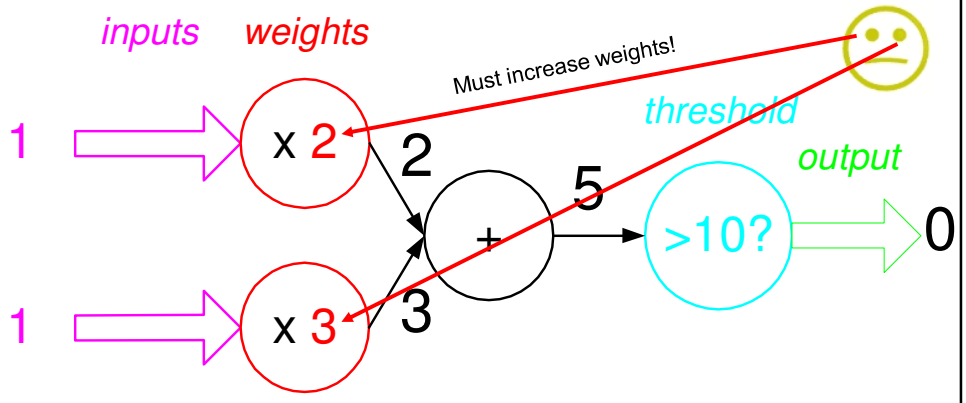
Example

If both inputs are 1, output should be 1.



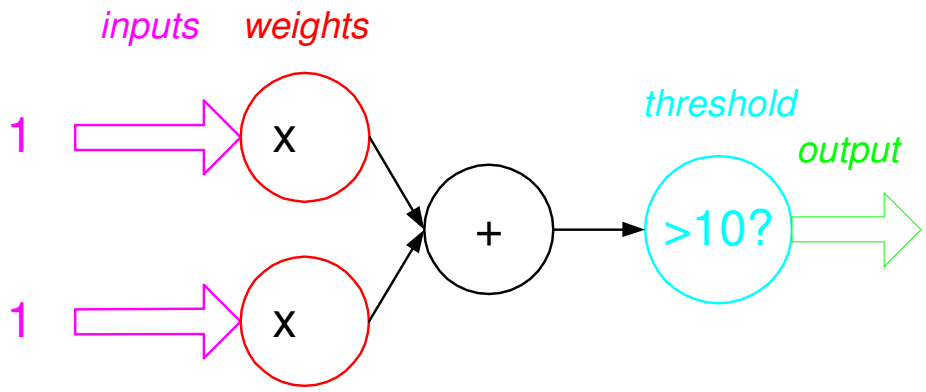
Example

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.

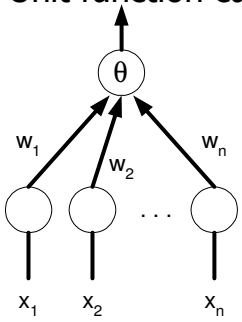
Computations

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

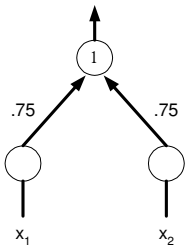


if $\Sigma w_i x_i >= \theta$, output = 1

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

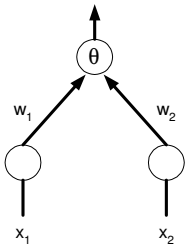
INPUT: $x_1 = 0, x_2 = 1$
$.75*0 + .75*1 = .75 < 1$ ➔ OUTPUT: 0

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

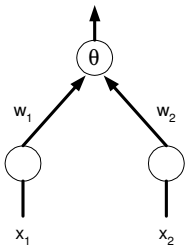
INPUT: $x_1 = 1, x_2 = 0$
$w_1*1 + w_2*0 >= \theta$ ➔ OUTPUT: 1

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: $x_1 = 0, x_2 = 0$

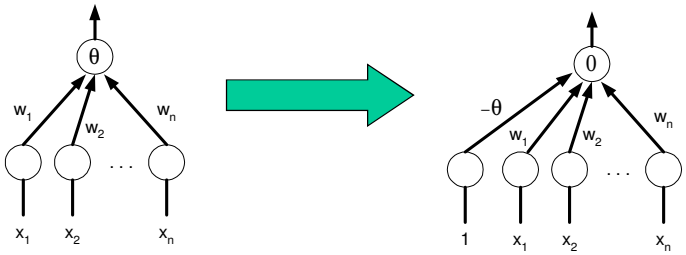
$w_1*0 + w_2*0 < \theta$ ➔ OUTPUT: 0

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 $x_0 = 1$, $w_0 = -\theta$

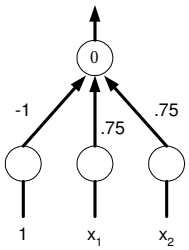


advantage: threshold = 0 for all neurons

$\Sigma w_i x_i >= \theta$ ≡ $-\theta*1 + \Sigma w_i x_i >= 0$

Normalized examples

AND



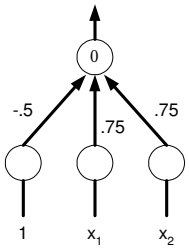
INPUT: $x_1 = 1, x_2 = 1$
1*-1 + .75*1 + .75*1 = .5 >= 0 ➔ OUTPUT: 1

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

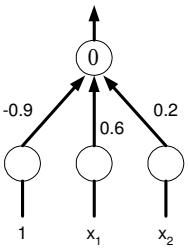
Perceptron learning algorithm:

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



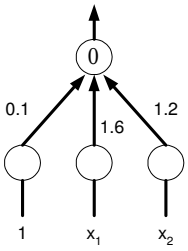
randomly, let: $w_0 = -0.9$, $w_1 = 0.6$, $w_2 = 0.2$

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

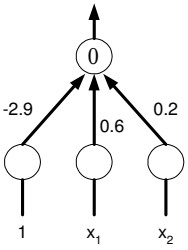
Example: perceptron learning



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

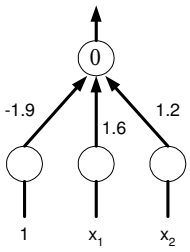


using these updated weights:

$x_1 = 1, x_2 = 1$:	-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

Perceptrons

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

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.

$$\Delta w_i = c\,(d - sign((\Sigma x_i\, w_i))\, x_i$$

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 Δw_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

$$\Delta w_i = c\,(d - sign((\Sigma x_i\, w_i))\, x_i$$

- c (in some books η) is called the learning rate, Usually set to something small (e.g., 0.1), 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