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.

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

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 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

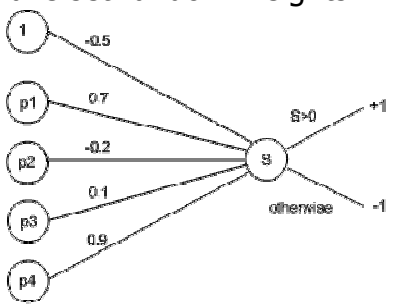
$$\Delta w_i = c(d\text{-}sign((\Sigma x_i w_i))x_i$$

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

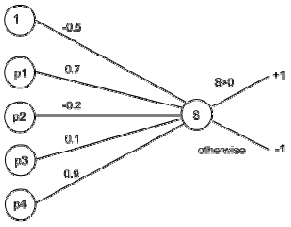
  This will occur over time with multiple epochs

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



$\Delta w_i = \eta(d\text{-}sign((\Sigma x_i w_i))x_i$, η = 0.1
x1 = -1, x2 = 1, x3 = 1, x4 = -1

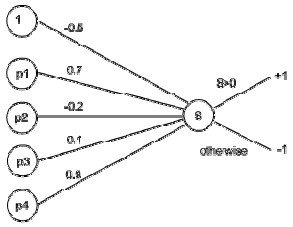
Propagate this information through the network:

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 $sign\Sigma x_i w_i = -1$

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

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

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

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

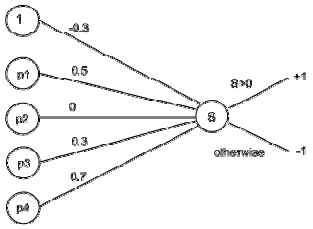
$\Delta w_4 = 0.1 * (1 - (-1)) * (-1) = 0.1 * (-2) = -0.2$

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

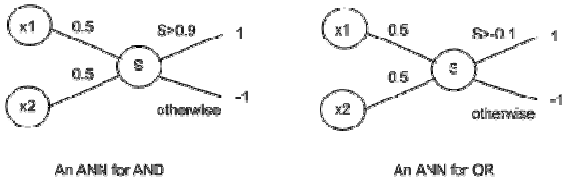
Calculating again: (x1 = -1, x2 = 1, x3 = 1, x4 = -1)

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

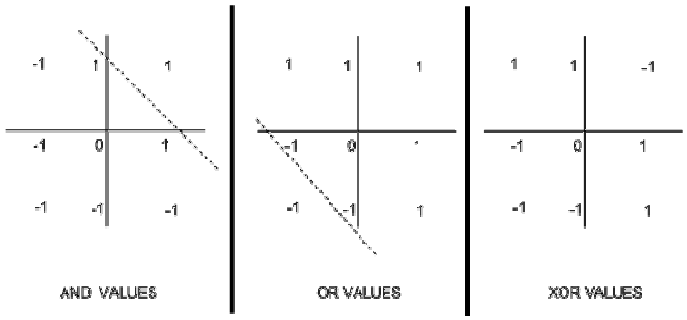
Boolean Functions as Perceptrons



- 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

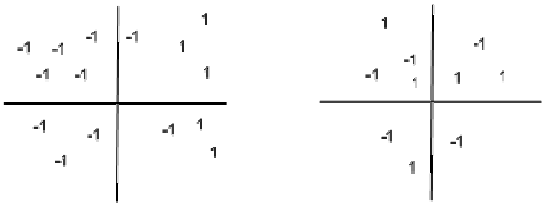


Linearly separable:

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.

Perceptron as Hill Climbing

The space being searched: is a set of weights and a threshold.

Goal: is to minimize classification error on the training set.

Perceptron does hill-climbing (gradient descent) in this space, changing the weights a small amount at each point to decrease training set error.

For a single model neuron, the space is well behaved with a single minima.

In practice, converges fairly quickly for linearly separable data.

Perceptron and DT