Neural Network II

Important Things We Have Learned So Far (I)

A big error means a bigger correction is needed, and a tiny error means we need the teeniest of nudges to c.

We have walked through the very core process of learning in a neural network - we've trained the machine to get better and better at giving the right answer, i.e., **iteratively** improving an answer bit by bit.

Important Things We Have Learned So Far (II)

We can use simple math to understand the relationship between the output error of a linear classifier and the adjustable slope parameter. That is the same as knowing how much to adjust the slope to remove that output error.

A problem with doing these adjustments naively, is that the model is updated to best match the last training example only, effectively ignoring all previous training examples. A good way to fix this is to moderate the updates with a learning rate so no single training example totally dominates the learning.

Training examples from the real world can be noisy or contain errors. Moderating updates in this way helpfully limits the impact of these false examples.

Important Things We Have Learned So Far (III)

Neural networks learn by refining their link weights. This is guided by the error (the difference between the right answer given by the training data and their actual output.

The error at the output nodes is simply the difference between the desired and actual output.

However the error associated with internal nodes is not obvious. One approach is to split the output layer errors in proportion to the size of the connected link weights, and then recombine these bits at each internal node.

Important Things We Have Learned So Far (IV)

Backpropagating the error can be expressed as a matrix multiplication.

This allows us to express it concisely, irrespective of network size, and also allows computer languages that understand matrix calculations to do the work more efficiently and quickly.

This means both feeding signals forward and error backpropagation can be made efficient using matrix calculations.

How Do We Actually Update Weights?

We can't do fancy algebra to work out the weights directly because the maths is too hard. There are just too many combinations of weights, and too many functions of functions of functions ... being combined when we feed forward the signal through the network.

For a simple 3 layer neural network with 3 nodes in each layer

$$o_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}\left(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}}\right)}}$$

How Do We Actually Update Weights?

Complexities:

The mathematical expressions showing how all the weights result in a neural network's output are too complex to easily untangle.

The weight combinations are too many to test one by one to find the best.

The training data might not be sufficient to properly teach a network.

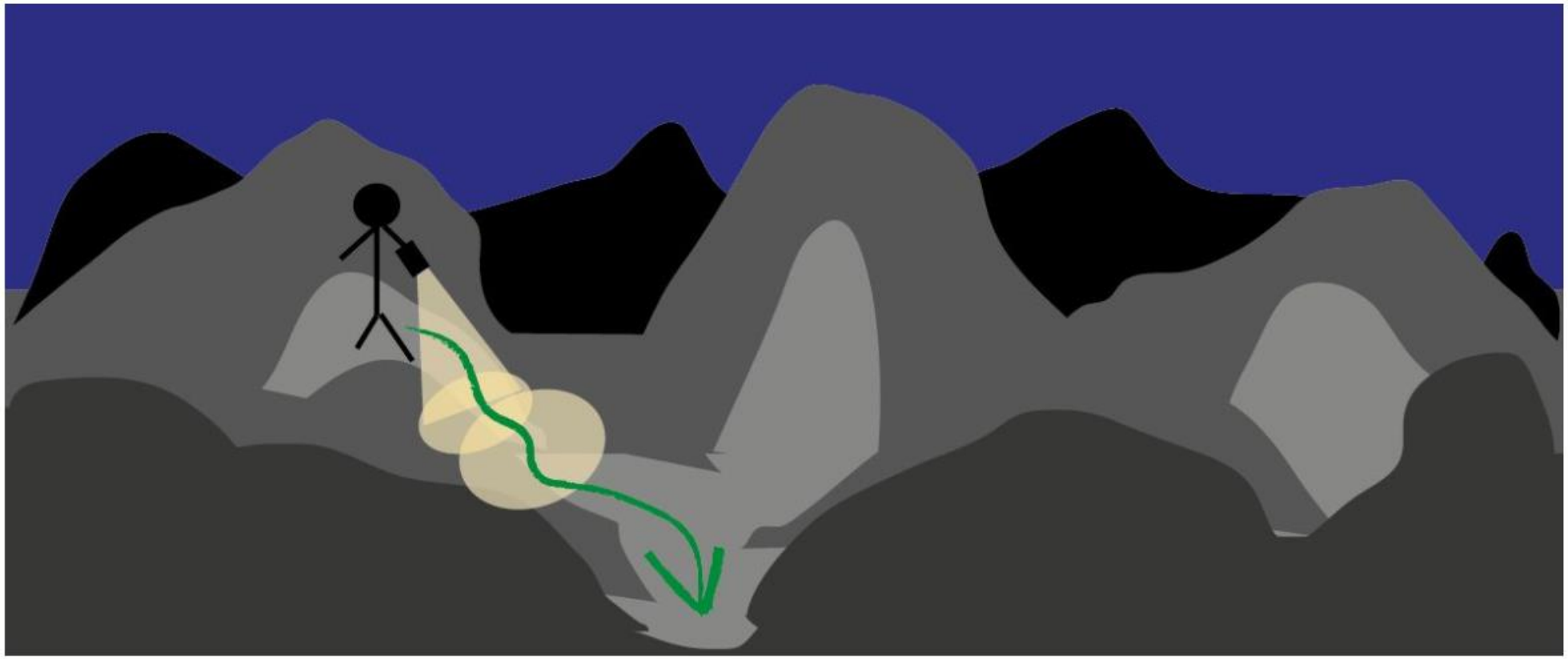
The training data might have errors so our assumption that it is the perfect truth, something to learn from, is then flawed.

The network itself might not have enough layers or nodes to model the right solution to the problem.

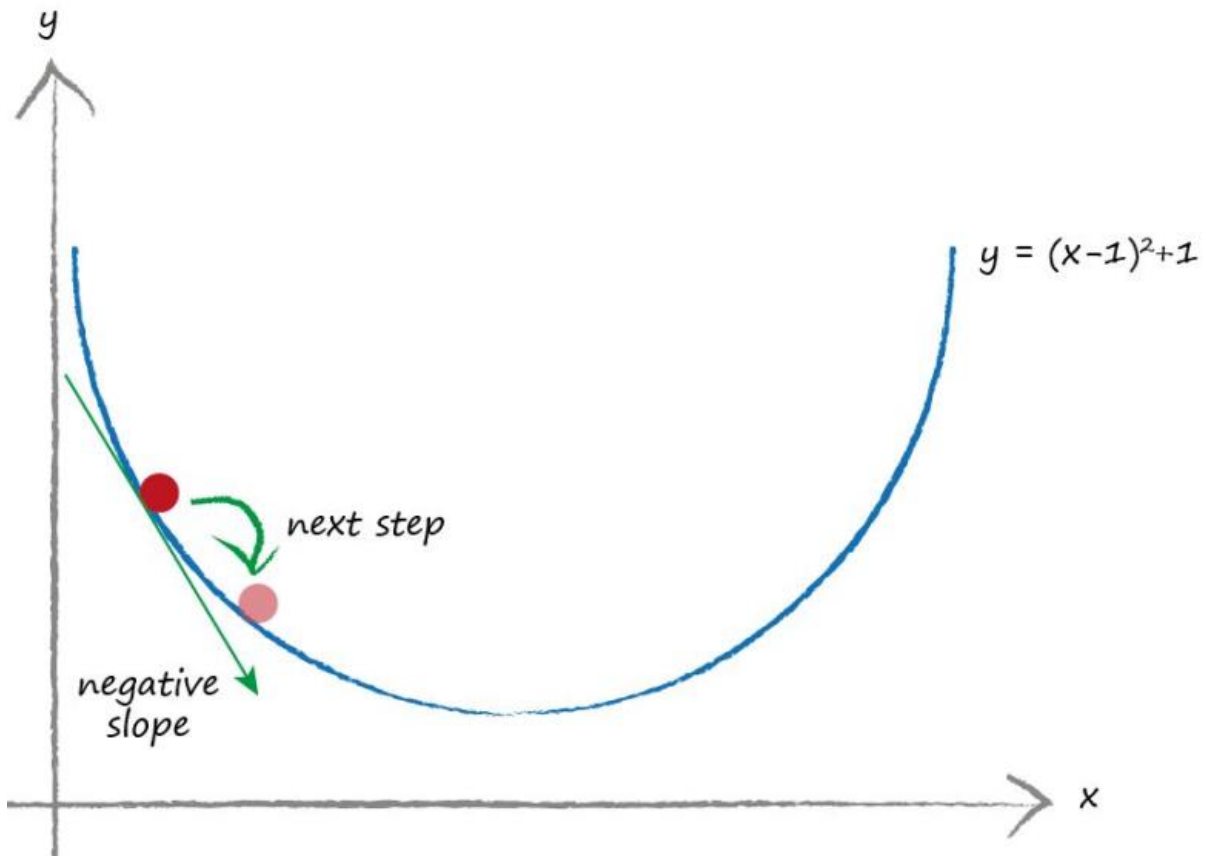
How Do We Actually Update Weights?

We need an approach that is realistic, and recognizes the limitations. We may find an approach which isn't mathematically perfect but does actually give us better results because it doesn't make false idealistic assumptions.

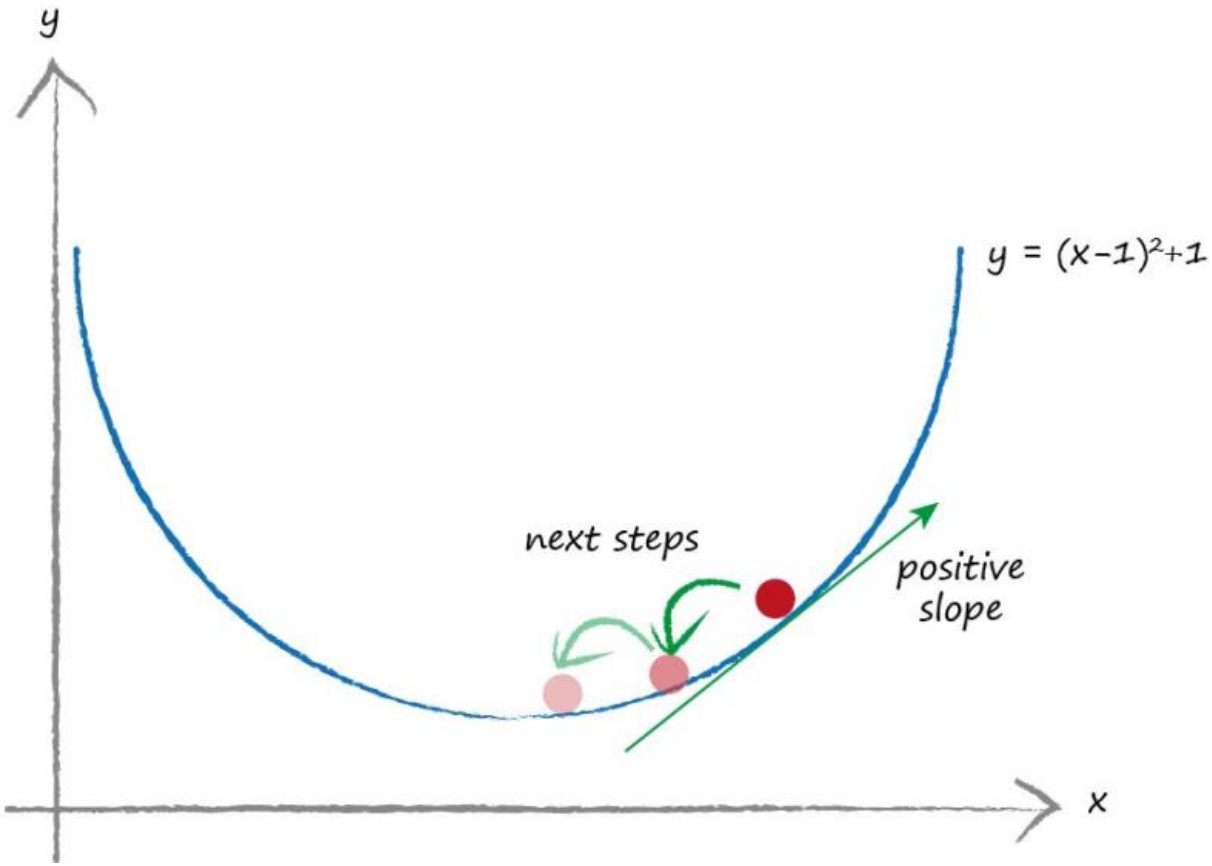
Gradient descent approach



How Do We Actually Update Weights?

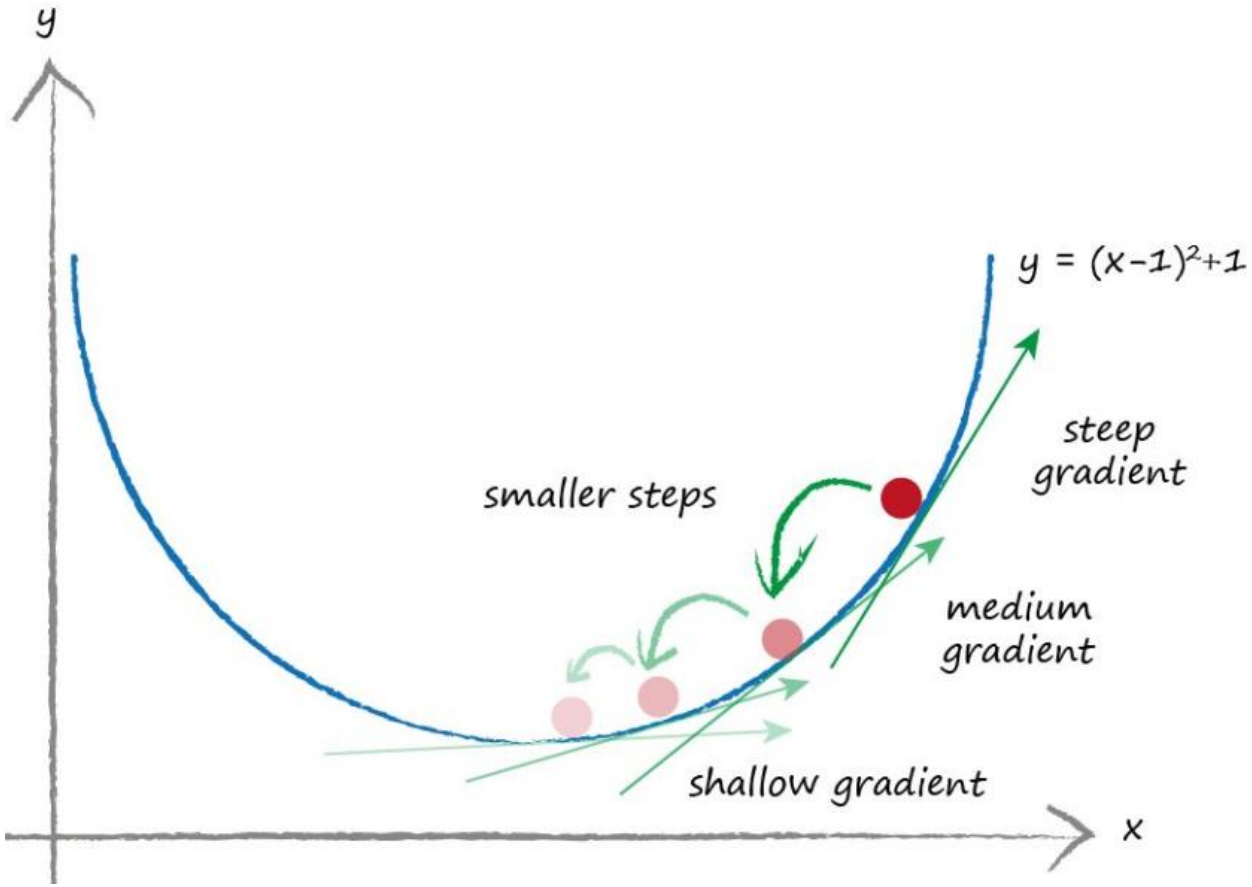


The slope beneath our feet is negative, so we move to the right. That is, we **increase x** a little.



The slope beneath our feet is positive, so we move to the left. That is, we **decrease x** a little.

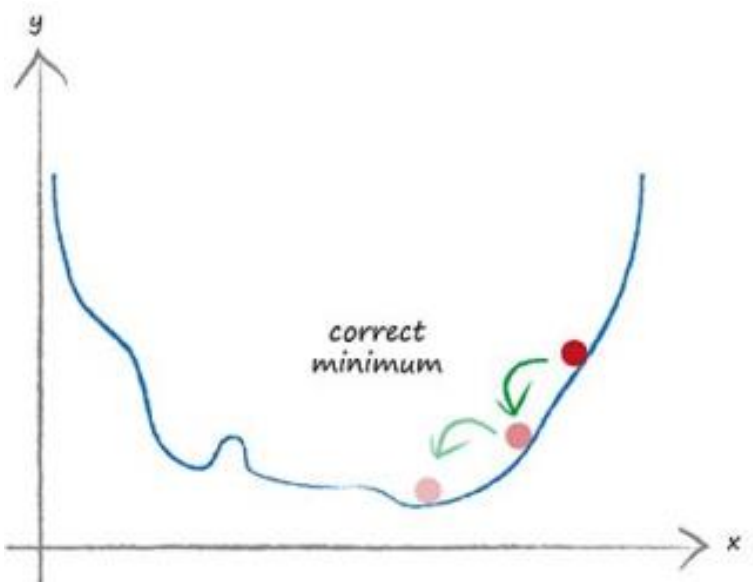
How Do We Actually Update Weights?

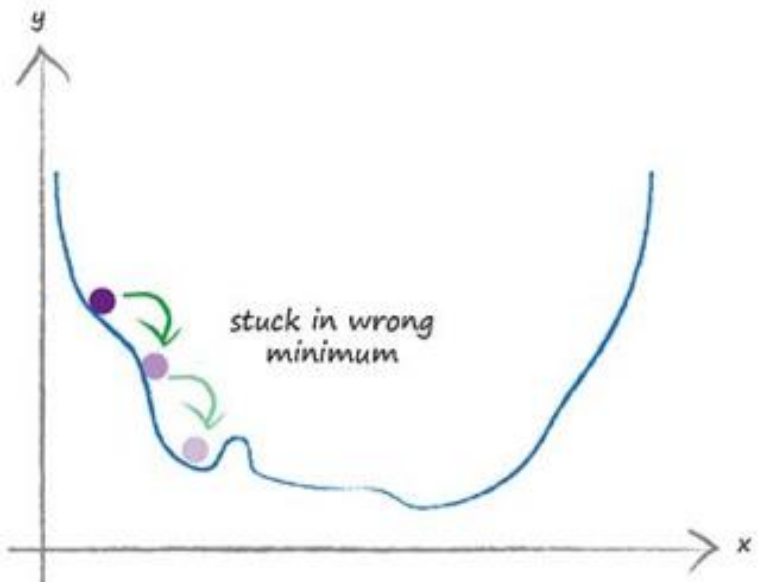


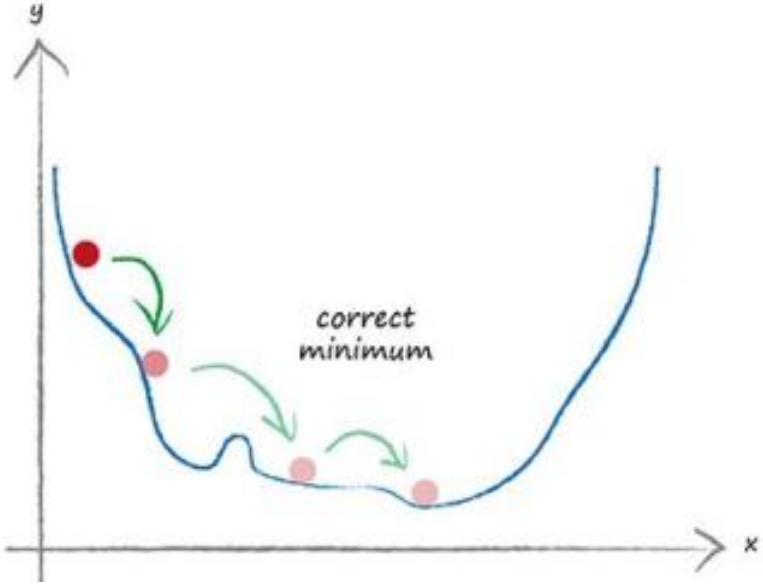
As we get closer to a minimum the slope does indeed get shallower. This is the idea of moderating step size as the function gradient gets smaller

How Do We Actually Update Weights?

Three different cases:







What we learnt

Gradient descent is a really good way of working out the minimum of a function, and it really works well when that function is so complex and difficult that we couldn't easily work it out mathematically using algebra.

What's more, the method still works well when there are many parameters, something that causes other methods to fail or become impractical.

This method is also resilient to imperfections in the data, we don't go wildly wrong if the function isn't quite perfectly described or we accidentally take a wrong step occasionally.

How Do We Actually Update Weights?

Network Output	Target Output	Error (target - actual)	Error |target - actual|	Error (target - actual)^2
0.4	0.5	0.1	0.1	0.01
0.8	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

How Do We Actually Update Weights?

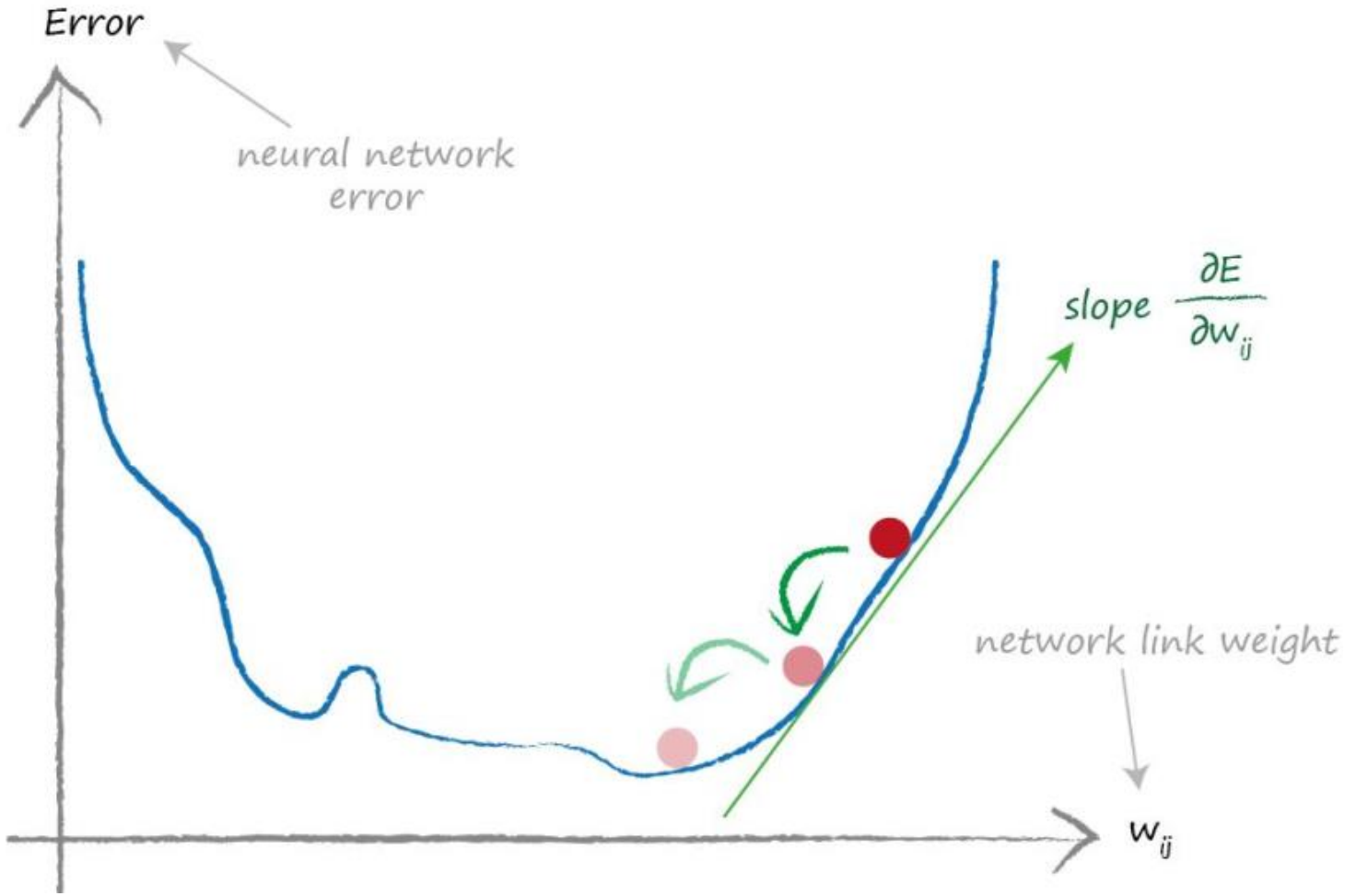
Reasons to prefer **(target - actual)^2**:

- The algebra needed to work out the slope for gradient descent is easy enough with this squared error.

- The error function is smooth and continuous making gradient descent work well – there are no gaps or abrupt jumps.

- The gradient gets smaller nearer the minimum, meaning the risk of overshooting the objective gets smaller if we use it to moderate the step sizes.

How Do We Actually Update Weights?



We want to refine network link weight to reduce the error.

How Do We Actually Update Weights?

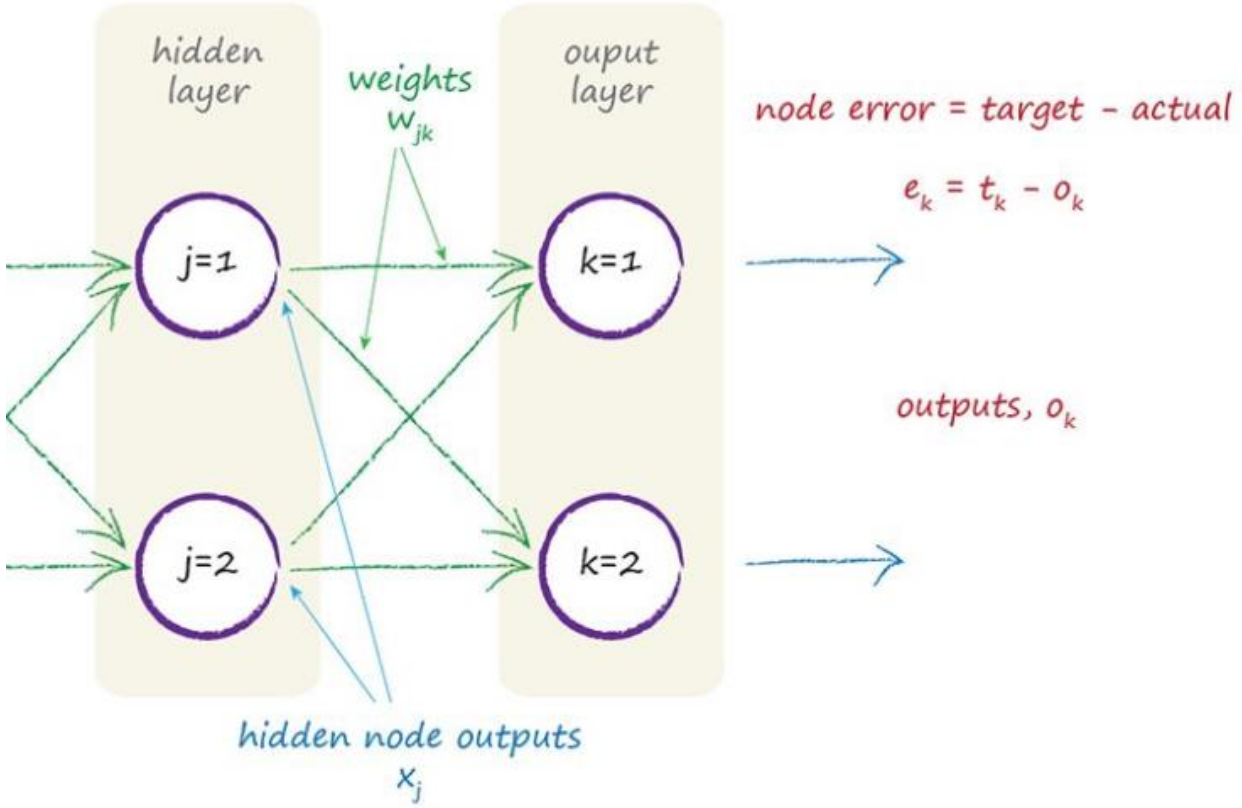
How does the error E change as the weight w_{jk} changes

$$\frac{\partial E}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \Sigma_n (t_n - o_n)^2$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$



How Do We Actually Update Weights?

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}} \qquad \frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{ik}} \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)$$

$$\frac{\partial}{\partial x} \text{sigmoid}(x) = \text{sigmoid}(x)\left(1 - \text{sigmoid}(x)\right)$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\right) \cdot \frac{\partial}{\partial w_{jk}}\left(\Sigma_j w_{jk} \cdot o_j\right)$$

$$= -2(t_k - o_k) \cdot \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\right) \cdot o_j$$

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\right) \cdot o_j$$

How Do We Actually Update Weights?

The slope of the error function for the weights between the hidden and output layers

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

The slope of the error function for the weights between the input and hidden layers

$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot \text{sigmoid}(\Sigma_i w_{ij} \cdot o_i)(1 - \text{sigmoid}(\Sigma_i w_{ij} \cdot o_i)) \cdot o_i$$

How Do We Actually Update Weights?

$$\text{new } w_{jk} = \text{old } w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

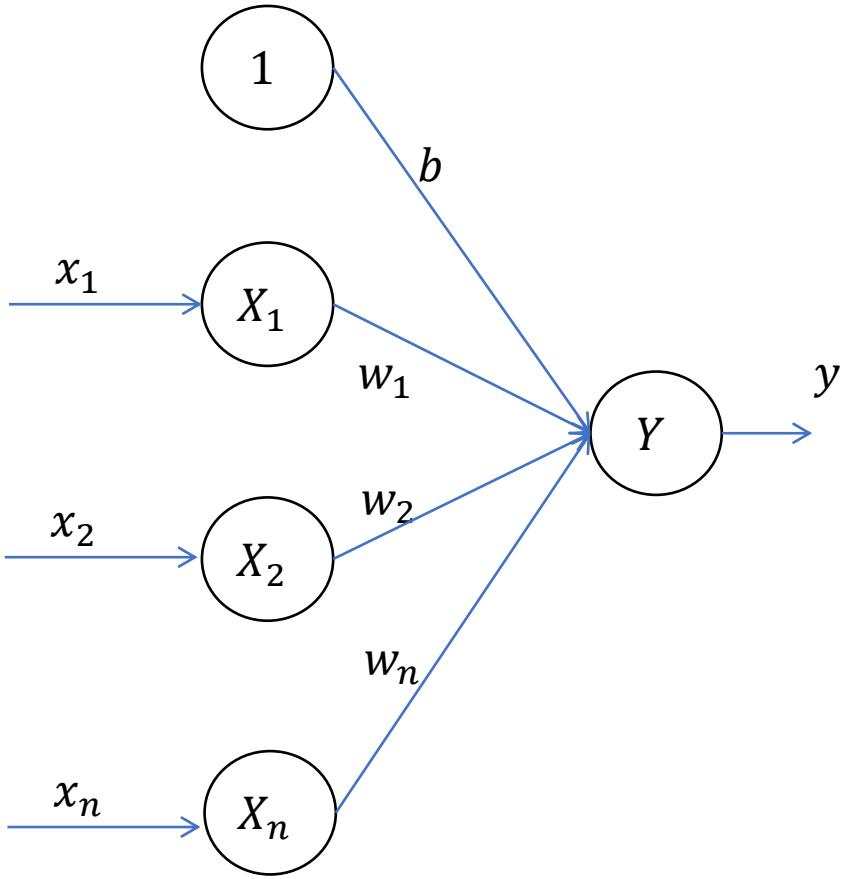
Reference

Make Your Own Neural Network

by

Tariq Rashid

Simple Net with bias



McCulloch-Pitts Neuron

M-P neurons are connected by directed weighted paths.

Activation of M-P neurons is binary that is at any time step the neuron may fire or may not fire.

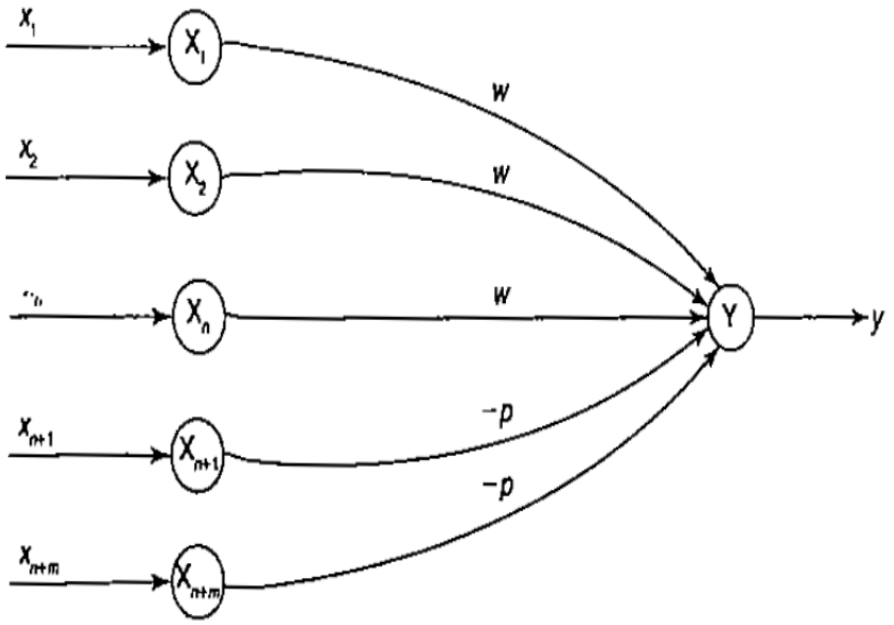
Weights associated with communication links may be excitatory (weights are positive)/inhibitory (weights are negative).

Threshold plays major role.

It is excitatory with weight w (w>0) / inhibitory with weight –p (p<0).

Activation function is defined as

$$f(x) = \begin{cases} 1 \ if \ y_{in} \geq \theta \\ 0 \ if \ y_{in} < \theta \end{cases}$$

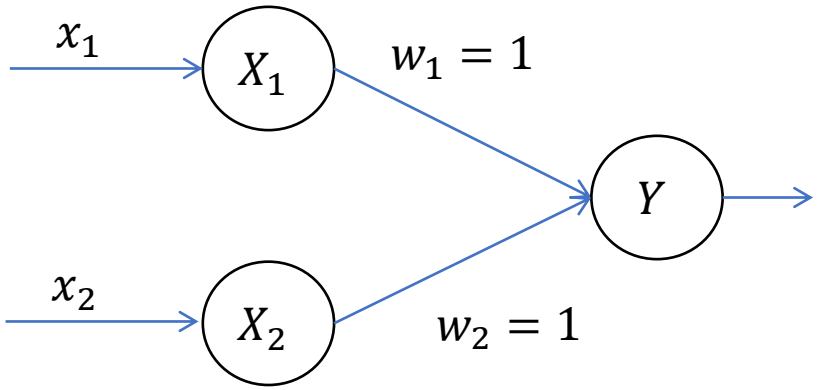


Designing a M-P neuron to implement AND function

We assume $w_1 = 1$ and $w_2 = 1$

x_1	x_2	y	y_{in}
1	1	1	2
1	0	0	1
0	1	0	1
0	0	0	0

$$y = f(y_{in}) = \begin{cases} 1 & if\, y_{in} \geq 2 \\ 0 & if\, y_{in} < 2 \end{cases}$$



Designing a M-P neuron to implement ANDNOT function

We assume $w_1 = 1$ and $w_2 = 1$

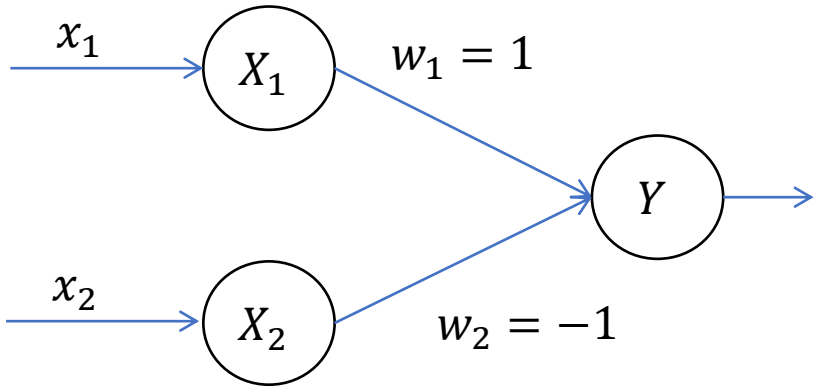
x_1	x_2	y	y_{in}
0	0	0	2
0	1	0	1
1	0	1	1
1	1	0	0

It is not possible to fire neuron for input (1,0) only.

We assume $w_1 = 1$ and $w_2 = -1$

x_1	x_2	y	y_{in}
0	0	0	0
0	1	0	-1
1	0	1	1
1	1	0	0

$$y = f(y_{in}) = \begin{cases} 1 & if\, y_{in} \geq 1 \\ 0 & if\, y_{in} < 1 \end{cases}$$



Hebb Network

Hebb or Hebbian learning rule was proposed by Donald O Hebb.

It is used for pattern classification.

It is a single layer neural network, i.e. it has one input layer and one output layer.

The input layer can have many units, say n.

The output layer only has one unit.

This network is suitable for bipolar data.

The Hebbian learning rule is generally applied to logic gates.

The weights are updated as:

$$W\ (new) = w\ (old) + x*y$$

Hebb Network

STEP 1:Initialize the weights and bias to **'0'** i.e w1=0,w2=0, …., wn=0.

STEP 2: 2–4 have to be performed for each input training vector and target output pair **i.e. s:t** (s=training input vector, t=training output vector)

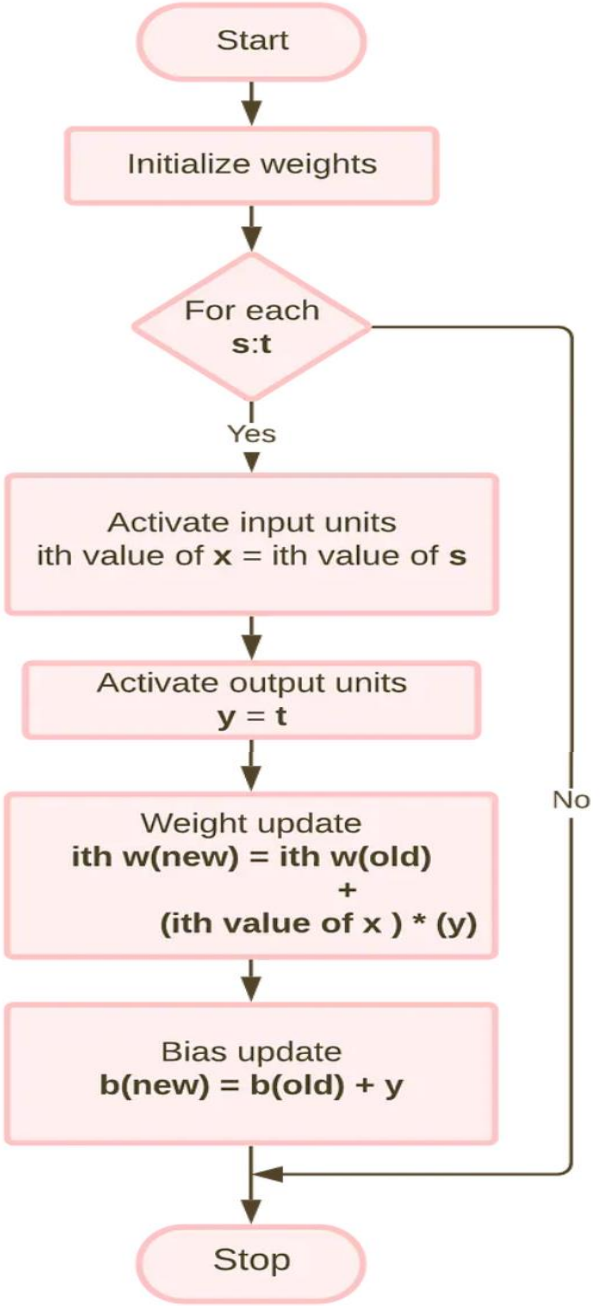
STEP 3: Input units activation are set and in most of the cases is an identity function(one of the types of an activation function) for the input layer;

ith value of x = ith value of s for i=1 to n

STEP 4: Output units activations are set y:t

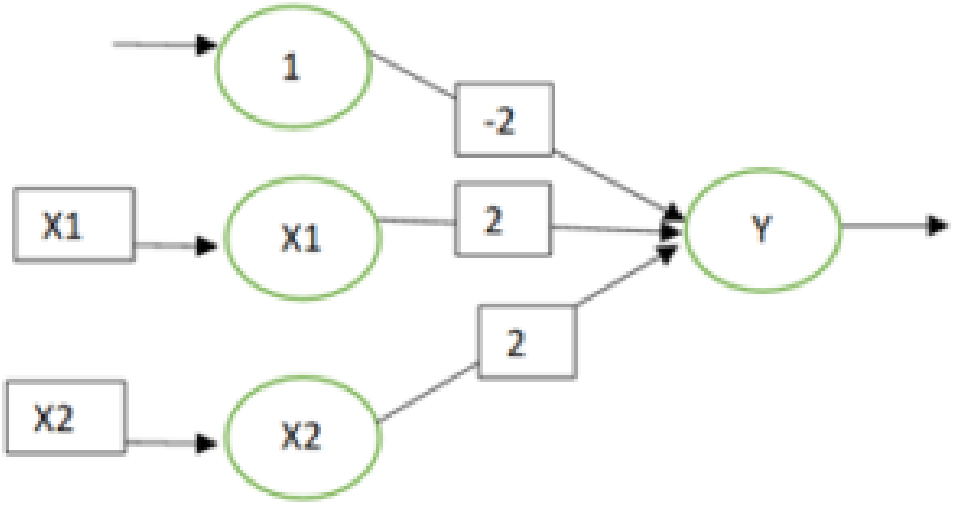
STEP 5: Weight adjustments and bias adjustments are performed;

ith value of w(new) = ith value of w(old) + (ith value of x * y)
new bias(value) = old bias(value) + y



Designing a Hebb network to implement AND function

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		New Bias
X1	X2	b	y	Δw1 X1*y	Δw2 X1*y	Δb y	w1	w2	b
							0	0	0
1	1	1	1	1	1	1	1	1	1
1	-1	1	-1	-1	1	-1	0	2	0
-1	1	1	-1	1	-1	-1	1	1	-1
-1	-1	1	-1	1	1	-1	2	2	-2



Designing a Hebb network to implement OR function

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		New Bias
X1	X2	b	y	Δw1 X1*y	Δw2 X1*y	Δb y	w1	w2	b
1	1	1	1						
1	-1	1	1						
-1	1	1	1						
-1	-1	1	-1						

Designing a Hebb network to implement OR function

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		New Bias
X1	X2	b	y	Δw1 X1*y	Δw2 X1*y	Δb y	w1	w2	b
1	1	1	1	1	1	1	1	1	1
1	-1	1	1	1	-1	1	2	0	2
-1	1	1	1	-1	1	1	1	1	3
-1	-1	1	-1	1	1	-1	2	2	2

Designing a Hebb network to implement XOR function

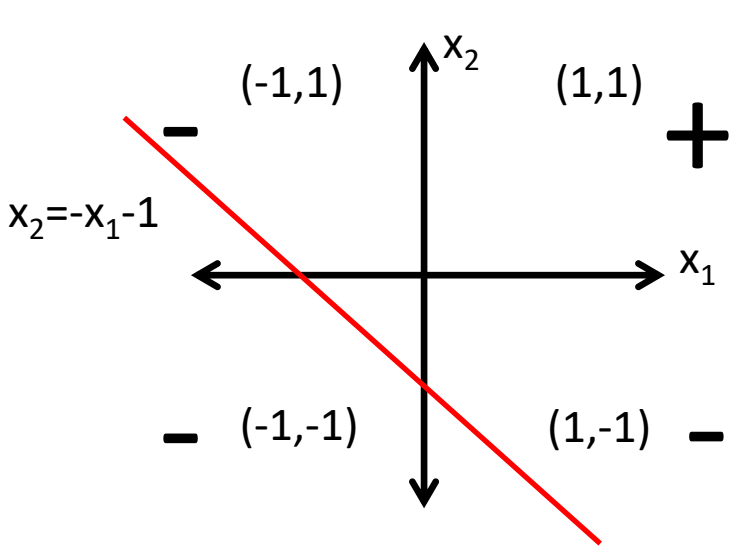
Inputs		**Bias**	**Target Output**	**Weight Changes**		**Bias Changes**	**New Weights**		**New Bias**
X1	X2	b	y	Δw1 X1*y	Δw2 X1*y	Δb y	w1	w2	b
1	1	1	-1	-1	-1	-1	-1	-1	-1
1	-1	1	1	1	-1	1	0	-2	0
-1	1	1	1	-1	1	1	-1	-1	1
-1	-1	1	-1	1	1	-1	0	0	0

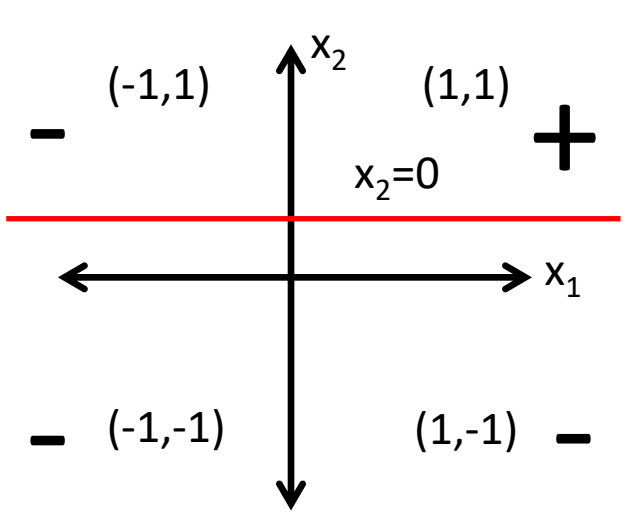
Designing a Hebb network to implement XOR function

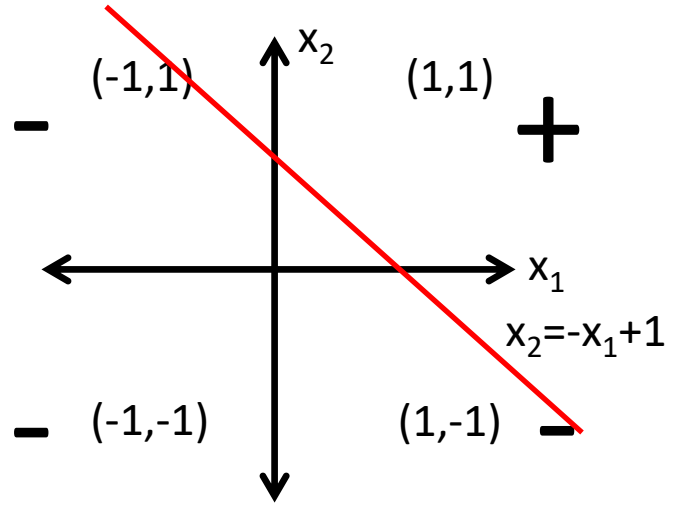
Inputs		**Bias**	**Target Output**	**Weight Changes**		**Bias Changes**	**New Weights**		**New Bias**
X1	X2	b	y	Δw1 X1*y	Δw2 X1*y	Δb y	w1	w2	b
1	1	1	-1						
1	-1	1	1						
-1	1	1	1						
-1	-1	1	-1						

Designing a Hebb network to implement AND function

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		New Bias
X1	X2	b	y	Δw1	Δw2	Δb	w1	w2	b
1	1	1	1	1	1	1	1	1	1
1	-1	1	-1	-1	1	-1	0	2	0
-1	1	1	-1	1	-1	-1	1	1	-1
-1	-1	1	-1	1	1	-1	2	2	-2

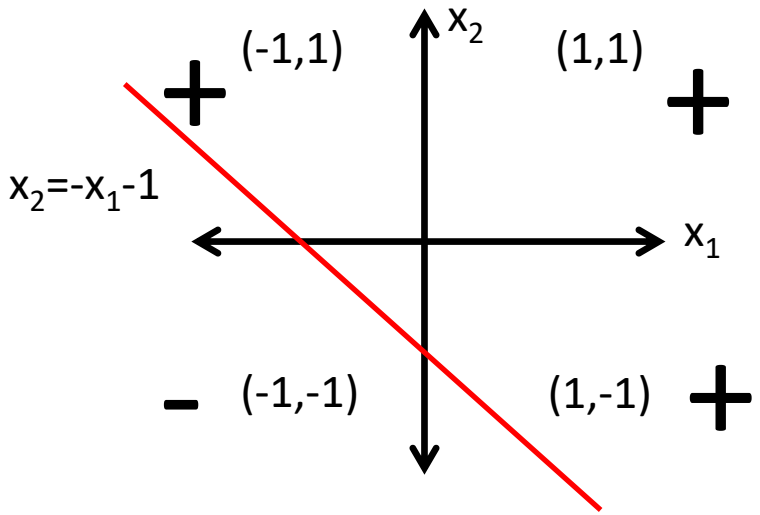






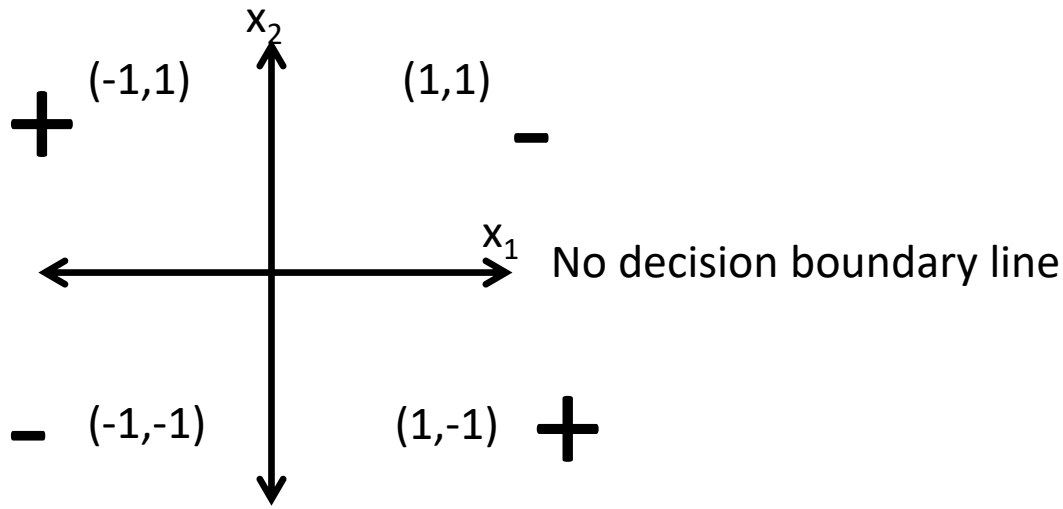
Designing a Hebb network to implement OR function

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		New Bias
X1	X2	b	y	Δw1	Δw2	Δb	w1	w2	b
1	1	1	1	1	1	1	1	1	1
1	-1	1	1	1	-1	1	2	0	2
-1	1	1	1	-1	1	1	1	1	3
-1	-1	1	-1	1	1	-1	2	2	2



Designing a Hebb network to implement XOR function

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		New Bias
X1	X2	b	y	Δw1	Δw2	Δb	W1	W2	b
1	1	1	-1	-1	-1	-1	-1	-1	-1
1	-1	1	1	1	-1	1	0	-2	0
-1	1	1	1	-1	1	1	-1	-1	1
-1	-1	1	-1	1	1	-1	0	0	0



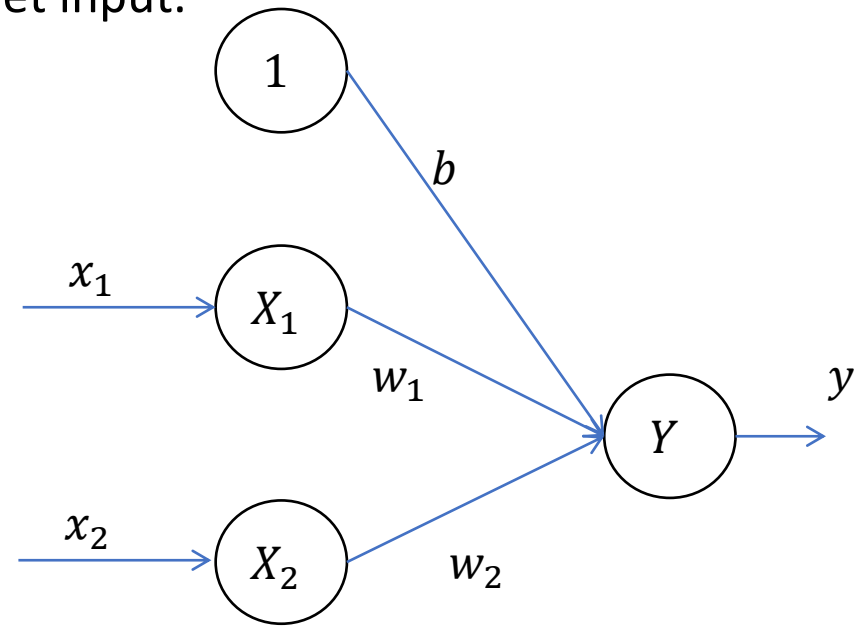
Perceptron

Perceptron networks come under single-layer feed-forward networks and are also called simple perceptrons.
Various types of perceptrons were designed by Rosenblatt (1962) and Minsky-Papert (1969, 1988).

The perceptron learning rule is used in the weight updation.

Consider a finite "n" number of input training vectors, with their associated target (desired) values x(n) and t(n), where “n” ranges from 1 to N. The target is either +1 or -1. The output "y" is obtained on the basis of the net input calculated and activation function being applied over the net input.

$$y = f(y_{in}) = \begin{cases} 1 & if\ y_{in} > \theta \\ 0 & if\ -\theta \le y_{in} \le \theta \\ -1 & if\ y_{in} < -\theta \end{cases}$$



If $y \neq t$ then

$$w(new) = w(old) + \Delta w \qquad \Delta w = \alpha t x$$

$$b(new) = b(old) + \Delta b \qquad \Delta b = \alpha t$$

Else

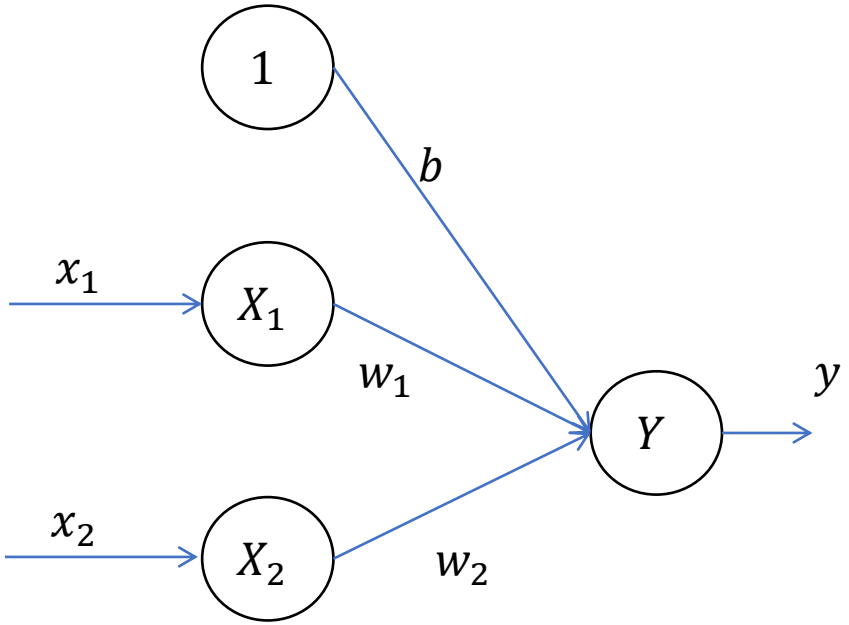
$$w(new) = w(old)$$

α is learning rate.

Designing a perceptron to implement AND function with bipolar inputs and targets

x_1	x_1	t
1	1	1
1	-1	-1
-1	1	-1
1	-1	-1

$$y = f(y_{in}) = \begin{cases} 1 & if\, y_{in} > 0 \\ 0 & if\, y_{in} = 0 \\ -1 & if\, y_{in} < 0 \end{cases}$$



$$w_i(new) = w_i(old) + \Delta w_i \qquad \Delta w_i = \alpha t x_i$$

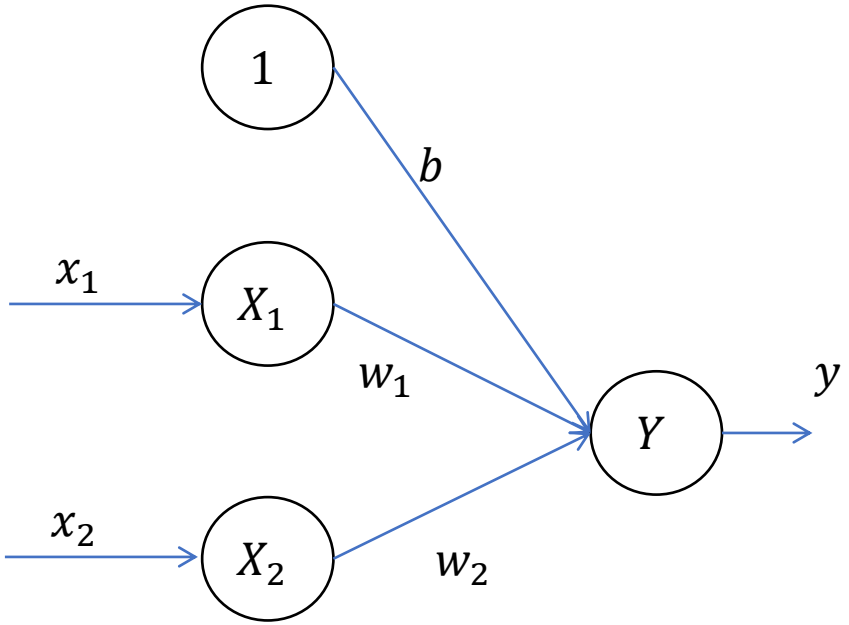
$$b(new) = b\,(old) + \Delta b \qquad \Delta b = \alpha t$$

Inputs		**Bias**	**Target Output**	**Net Input**	**Calculated ouput**	**Weight Changes**			**New Weights**		
x_1	x_2	b	t	y_{in}	y	Δw_1 tx_1	Δw_2 tx_2	Δb t	w_1	w_2	b
									0	0	0
1	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**
1	**-1**	**1**	**-1**	**1**	**1**	**-1**	**1**	**-1**	**0**	**2**	**0**
-1	**1**	**1**	**-1**	**2**	**1**	**-1**	**-1**	**-1**	**1**	**1**	**-1**
1	**-1**	**1**	**-1**	**-3**	**-1**	**0**	**0**	**0**	**1**	**1**	**-1**
1	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**-1**
1	**-1**	**1**	**-1**	**-1**	**-1**	**0**	**0**	**0**	**1**	**1**	**-1**
-1	**1**	**1**	**-1**	**-1**	**-1**	**0**	**0**	**0**	**1**	**1**	**-1**
1	**-1**	**1**	**-1**	**-3**	**-1**	**0**	**0**	**0**	**1**	**1**	**-1**

Designing a perceptron to implement OR function with binary inputs and bipolar targets

x_1	x_1	t
1	1	1
1	0	1
0	1	1
0	0	-1

$$y = f(y_{in}) = \begin{cases} 1 & if\, y_{in} > 0.2 \\ 0 & if\, -0.2 \le y_{in} \le 0.2 \end{cases}$$



$$w_i(new) = w_i(old) + \Delta w_i \qquad \Delta w_i = \alpha t x_i$$

$$b(new) = b\,(old) + \Delta b \qquad \Delta b = \alpha t$$

Inputs		**Bias**	**Target Output**	**Net Input**	**Calculated ouput**	**Weight Changes**			**New Weights**		
x_1	x_2	b	t	y_{in}	y	Δw_1 tx_1	Δw_2 tx_2	Δb t	w_1	w_2	b
									0	0	0
1	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**
1	**0**	**1**	**1**	**2**	**1**	**0**	**0**	**0**	**1**	**1**	**1**
0	**1**	**1**	**1**	**2**	**1**	**0**	**0**	**0**	**1**	**1**	**0**
0	**0**	**1**	**-1**	**1**	**1**	**0**	**0**	**-1**	**1**	**1**	**0**
1	**1**	**1**	**1**	**2**	**1**	**0**	**0**	**0**	**1**	**1**	**0**
1	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**
0	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**
0	**0**	**1**	**-1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**-1**
1	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**-1**
1	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**2**	**1**	**0**
0	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**2**	**1**	**0**
0	**0**	**1**	**-1**	**0**	**0**	**0**	**0**	**-1**	**2**	**1**	**-1**