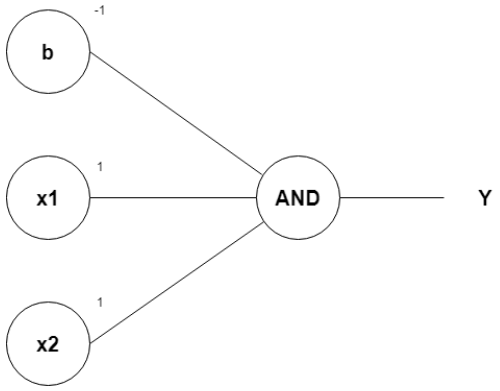
Logic Gate Design using Perceptron ANN with Bias

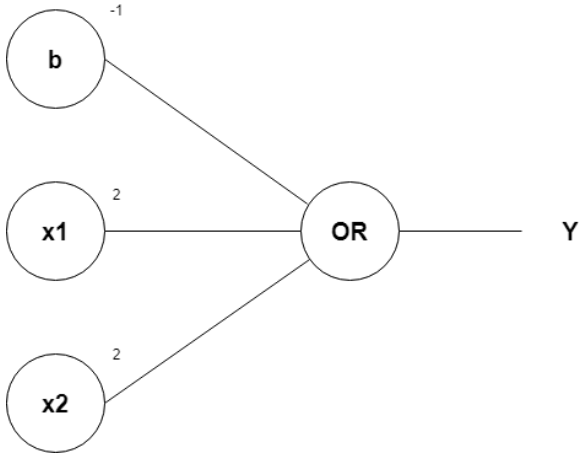
1. AND Gate

   The model to achieve an AND gate, using the Perceptron algorithm is:
   $x1+x2-1$



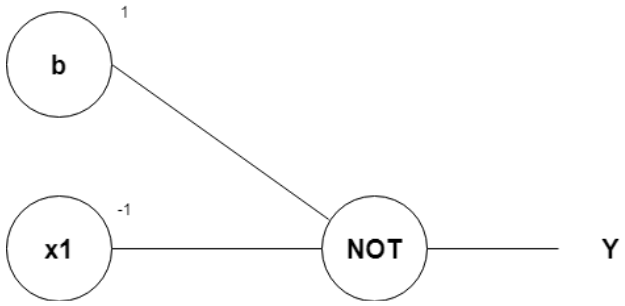
2. OR Gate

   The model to achieve an OR gate, using the Perceptron algorithm is:
   $2x1+2x2-1$



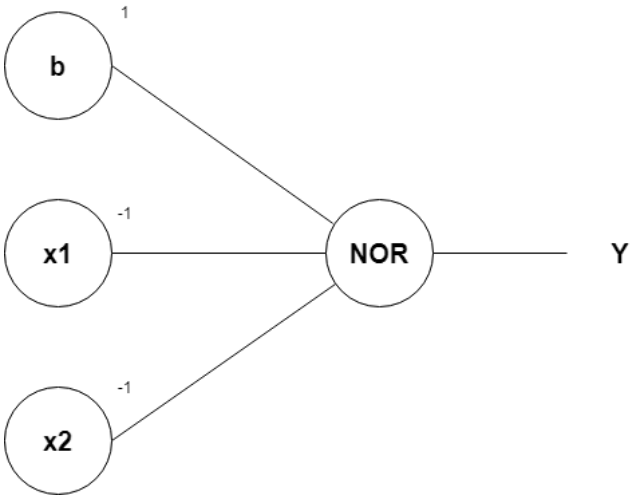
3. NOT Gate

   The model to achieve a NOT gate, using the Perceptron algorithm is:
   $-x1+1$



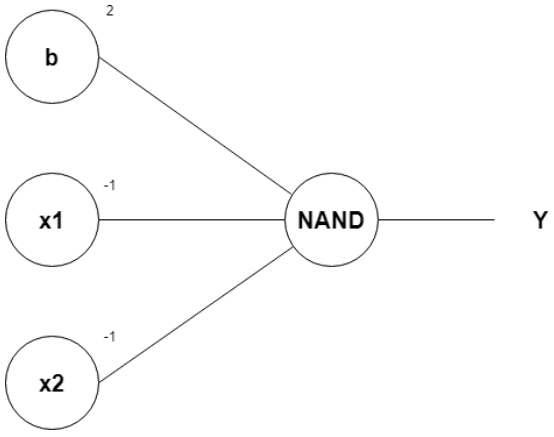
4. NOR Gate

The model to achieve a NOR gate, using the Perceptron algorithm is:
$-x1-x2+1$



5. NAND Gate

The model to achieve a NAND gate, using the Perceptron algorithm is:
$-x1-x2+2$

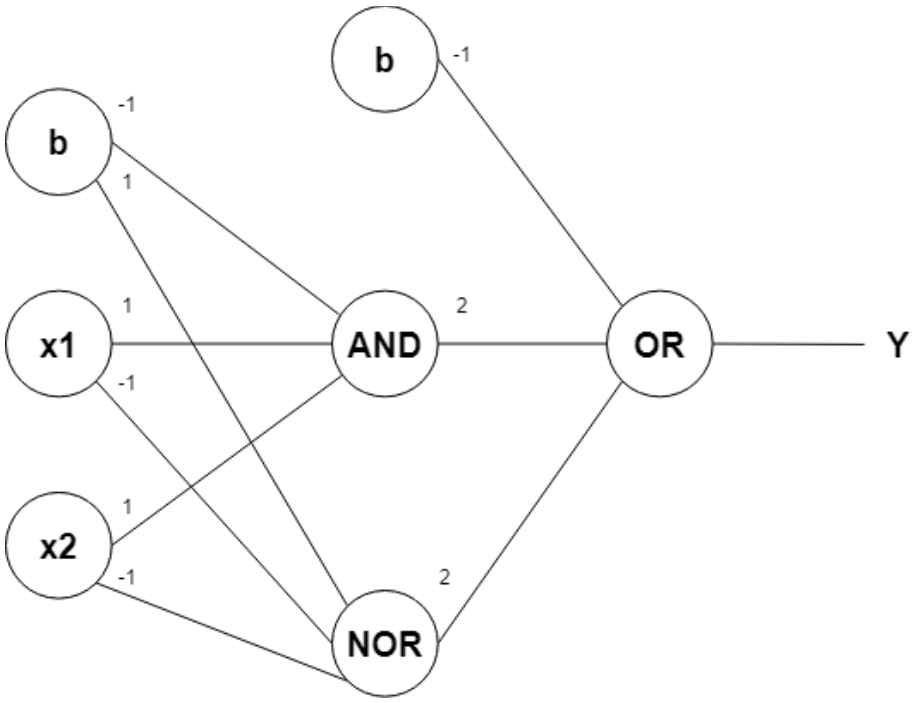


6. XNOR Gate

Single layer Perceptron can't be directly used to implement XOR and XNOR gates. Because both of XOR and XNOR are not linearly separable. But we can combine the basic gates AND, OR, NAND, NOR, NOT to produce an XOR and XNOR gate.
The boolean representation of an XNOR gate is: $x1x2 + x1\ `x2\ `$

From the expression, we can say that the XNOR gate consists of an AND gate (x1x2) a NOR gate (x1`x2`) and an OR gate. This means we will have to combine 3 Perceptrons:

- AND (x1+x2−1)
- NOR (-x1-x2+1)
- OR (2x1+2x2−1)



7. XOR Gate

The boolean representation of an XOR gate is;

$x1x`2 + x`1x2$

We first simplify the boolean expression

$x`1x2 + x1x`2 + x`1x1 + x`2x2$

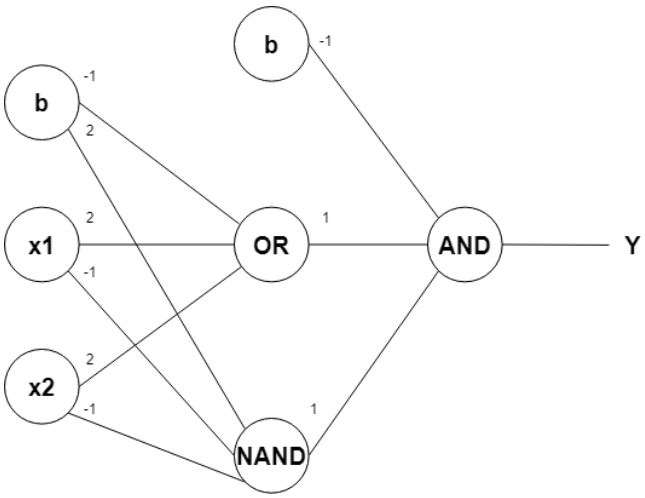
$x1(x`1 + x`2) + x2(x`1 + x`2)$

$(x1 + x2)(x`1 + x`2)$

$(x1 + x2)(x1x2)`$

This means we will have to combine 3 perceptrons:

- OR (2x1+2x2−1)
- NAND (-x1-x2+2)
- AND (x1+x2−1)



Summary of Learning Rules

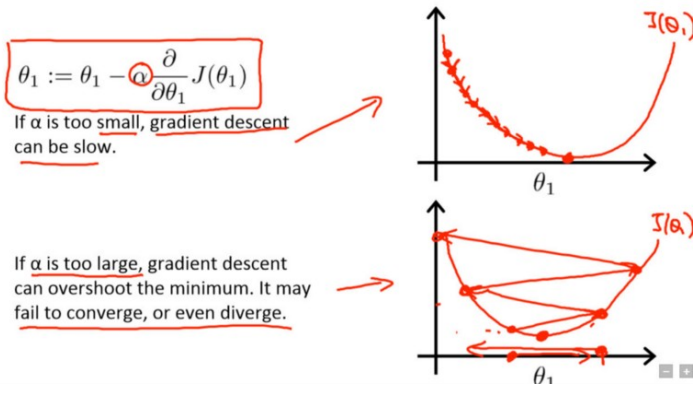
Summary of learning rules and their properties.

Learning rule	Single weight adjustment Δw_{ij}	Initial weights	Learning	Neuron characteristics	Neuron / Layer
Hebbian	$co_i x_j$ $j = 1, 2, \ldots, n$	0	U	Any	Neuron
Perceptron	$c\left[d_i - \text{sgn}\,(\mathbf{w}_i^t \mathbf{x})\right] x_j$ $j = 1, 2, \ldots, n$	Any	S	Binary bipolar, or Binary unipolar*	Neuron
Delta	$c(d_i - o_i) f'(net_i) x_j$ $j = 1, 2, \ldots, n$	Any	S	Continuous	Neuron
Widrow-Hoff	$c(d_i - \mathbf{w}_i^t \mathbf{x}) x_j$ $j = 1, 2, \ldots, n$	Any	S	Any	Neuron
Correlation	$cd_i x_j$ $j = 1, 2, \ldots, n$	0	S	Any	Neuron
Winner-take-all	$\Delta w_{mj} = \alpha(x_j - w_{mj})$ m-winning neuron number $j = 1, 2, \ldots, n$	Random Normalized	U	Continuous	Layer of p neurons
Outstar	$\beta(d_i - w_{ij})$ $i = 1, 2, \ldots, p$	0	S	Continuous	Layer of p neurons

c, α, β are positive learning constants
S — supervised learning, U — unsupervised learning

Learning Rate:

Learning rate (C) is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minima, it could also mean that we'll be taking a long time to converge.



Hebbian Learning Rule

For the Hebbian learning rule the learning signal is equal simply to the neuron's output (Hebb **1949).**

$$\Delta w_{ij} = co_i x_j, \quad \text{for } j = 1, 2, \ldots, n$$

This learning rule requires the weight initialization at small random values around wi = **0** prior to learning. The Hebbian learning rule represents a purely feedforward, unsupervised learning.

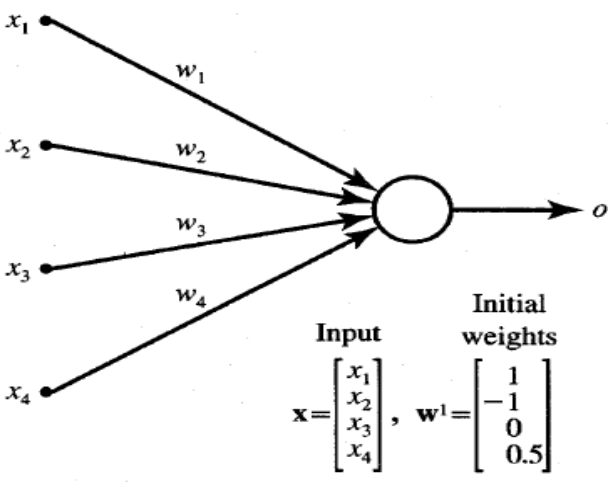
This example illustrates Hebbian learning with binary and continuous activation functions of a very simple network. Assume the network shown in Figure 2.22 with the initial weight vector

$$\mathbf{w}^1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

needs to be trained using the set of three input vectors as below

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix}, \quad \mathbf{x}_3 = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix}$$

for an arbitrary choice of learning constant $c = 1$. Since the initial weights are of nonzero value, the network has apparently been trained beforehand. Assume first that bipolar binary neurons are used, and thus $f(net) = \text{sgn}(net)$.



Step 1 Input $\mathbf{x}_1$ applied to the network results in activation net^1 as below:

$$net^1 = \mathbf{w}^{1t}\mathbf{x}_1 = \begin{bmatrix} 1 & -1 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = 3$$

The updated weights are

$$\mathbf{w}^2 = \mathbf{w}^1 + \text{sgn}\,(net^1)\mathbf{x}_1 = \mathbf{w}^1 + \mathbf{x}_1$$

and plugging numerical values we obtain

$$\mathbf{w}^2 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 1.5 \\ 0.5 \end{bmatrix}$$

where the superscript on the right side of the expression denotes the number of the current adjustment step.

Step 2 This learning step is with $\mathbf{x}_2$ as input:

$$net^2 = \mathbf{w}^{2t}\mathbf{x}_2 = \begin{bmatrix} 2 & -3 & 1.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -0.5 \\ -2 \\ -1.5 \end{bmatrix} = -0.25$$

The updated weights are

$$\mathbf{w}^3 = \mathbf{w}^2 + \text{sgn}\,(net^2)\mathbf{x}_2 = \mathbf{w}^2 - \mathbf{x}_2 = \begin{bmatrix} 1 \\ -2.5 \\ 3.5 \\ 2 \end{bmatrix}$$

Step 3 For input $\mathbf{x}_3$, we obtain in this step

$$net^3 = \mathbf{w}^{3t}\mathbf{x}_3 = \begin{bmatrix} 1 & -2.5 & 3.5 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ -1 \\ 1.5 \end{bmatrix} = -3$$

The updated weights are

$$\mathbf{w}^4 = \mathbf{w}^3 + \text{sgn}(net^3)\mathbf{x}_3 = \mathbf{w}^3 - \mathbf{x}_3 = \begin{bmatrix} 1 \\ -3.5 \\ 4.5 \\ 0.5 \end{bmatrix}$$

It can be seen that learning with discrete $f(net)$ and $c = 1$ results in adding or subtracting the entire input pattern vectors to and from the weight vector, respectively. In the case of a continuous $f(net)$, the weight incrementing/decrementing vector is scaled down to a fractional value of the input pattern.

Revisiting the Hebbian learning example, with continuous bipolar activation function $f(net)$, using input $\mathbf{x}_1$ and initial weights $\mathbf{w}^1$, we obtain neuron output values and the updated weights for $\lambda = 1$ as summarized in Step 1. The only difference compared with the previous case is that instead of $f(net) = \text{sgn}(net)$, now the neuron's response is computed from (2.3a).

Given the following continuous activation function

$$f(net) = \frac{2}{1 + e^{-\lambda net}} - 1$$

Step 1

$$f(net^1) = 0.905$$

$$\mathbf{w}^2 = \begin{bmatrix} 1.905 \\ -2.81 \\ 1.357 \\ 0.5 \end{bmatrix}$$

Subsequent training steps result in weight vector adjustment as below:

Step 2

$$f(net^2) = -0.077$$

$$\mathbf{w}^3 = \begin{bmatrix} 1.828 \\ -2.772 \\ 1.512 \\ 0.616 \end{bmatrix}$$

Step 3

$$f(net^3) = -0.932$$

$$\mathbf{w}^4 = \begin{bmatrix} 1.828 \\ -3.70 \\ 2.44 \\ -0.783 \end{bmatrix}$$

Ex/ Implement logical AND function with bipolar inputs using Hebbian Learning. X1 and X2 are inputs, b is the bias taken as 1, the target value is the output of logical AND operation over inputs.

#1) Initially, the weights are set to zero and bias is also set as zero.

W1=w2=b=0

#2) First input vector is taken as [x1 x2 b] = [1 1 1] and target value is 1.

The new weights will be:

W (new) = w (old) +x*y
W1 (n) = w1 (o) + x1*y = 0 + 1*1 = 1
W2 (n) = w2 (o) + x2*y = 0 + 1*1 = 1
B (n) =b (o) + y = 0+1 =1
The change in weights is: Δw1= x1 *y =1 Δw2= x2 *y =1Δb= y =1

#3) The above weights are the final new weights. When the second input is passed, these become the initial weights.

#4) Take the second input = [1 -1 1]. The target is -1.

The weights vector is = [w1 w2 b] = [1 1 1]
The change is weights is: Δw1= x1 *y =-1 Δw2= x2 *y =1 Δb= y =-1
The new weights will be w1 (n) = w1 + Δw1 => 1 + (-1) = 0
W2 (n) = w2 + Δw2 => 1 + (1) = 2
B (n) = b + Δb => 1 + (-1) = 0

#5) Similarly, the other inputs and weights are calculated.

Inputs		Bias	Target Output	Weight Changes		Bias Changes	New Weights		
X1	X2	b	y	?w1	?w2	?b	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

Hebb Net for AND Function