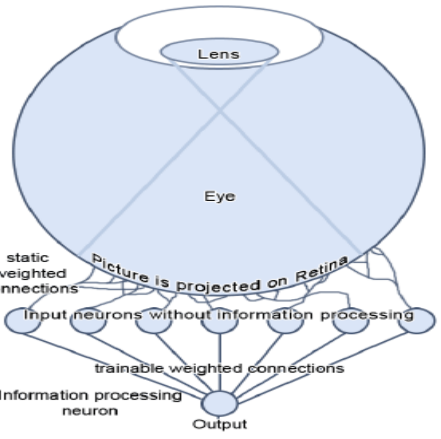
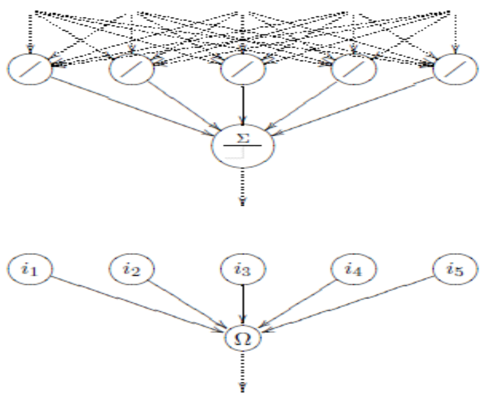
The perceptron, Delta rule

Perceptron is a feed forward network has a layer of scanner neurons (retina) with statically weighted connections to the following layer and is called input layer; but the weights of all other layers are allowed to be changed. All neurons subordinate to the retina are pattern detectors. Here we initially use a binary perceptron with every output neuron having exactly two possible output values (e.g. $\{0, 1\}$ or $\{-1, 1\}$). Thus, a binary threshold function is used as activation function, depending on the threshold value of the output neuron. In a way, the binary activation function represents an IF query which can also be negated by means of negative weights.

.

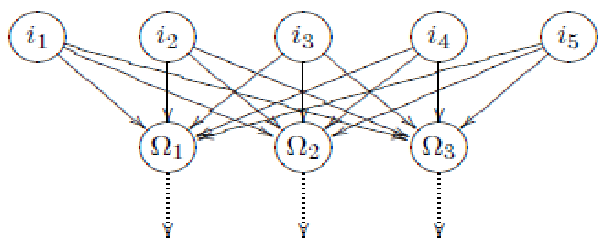
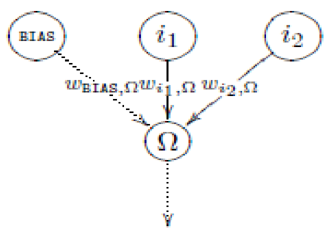

Lens
Eye
static weighted connections
Picture is projected on Retina
Input neurons without information processing
trainable weighted connections
Information processing neuron
Output



Definition (Information processing neuron)**.** ***Information processing neurons*** somehow process the input information; A ***binary neuron*** sums up all inputs by using the weighted sum as propagation function. Then the activation function of the neuron is the binary threshold function.

Definition 5 (Single layer perceptron)**.** A ***single layer perceptron*** (***SLP***) is a perceptron having only one layer of variable weights and one layer of output neurons**.**

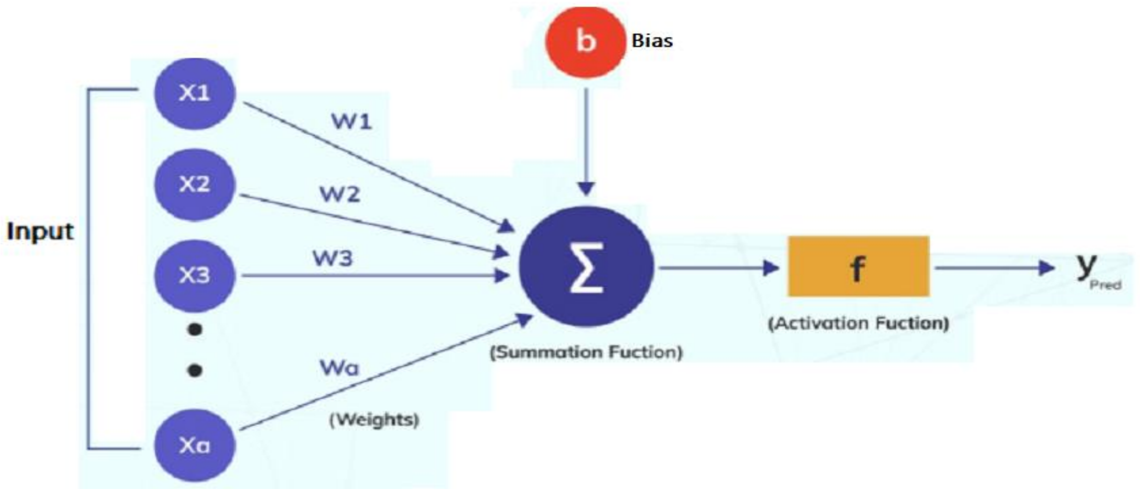
Certainly, the existence of several output neurons 1,2, . . ., *n* does not considerably change the concept of the perceptron: A perceptron with several output neurons can also be regarded as several different perceptron's with the same input.

 i_1 i_2 i_3 i_4 i_5 Ω_1 Ω_2 Ω_3	 BIAS i_1 i_2 $w_{\mathrm{BIAS},\Omega}$ $w_{i_1,\Omega}$ $w_{i_2,\Omega}$ Ω
Single layer perceptron with several output neuron	Single layer perceptron with single output neuron

Networks with threshold activation functions

A single layer supervised learning feed forward network consists of one or more output neurons ***o*** each of which is connected with a weighting factor w_{io} to all of the inputs ***i*** In the simplest case the network has only two inputs and a single output as sketched leave the output index o out. Each perceptron comprises four different parts:

1. **Input Values**: A set of values or a dataset for predicting the output value. They are also described as a dataset's features and dataset.
2. **Weights:** The real value of each feature is known as weight. It tells the importance of that feature in predicting the final value.
3. **Bias:** The activation function is shifted towards the left or right using bias. You may understand it simply as the y-intercept in the line equation.
4. **Summation Function:** The summation function binds the weights and inputs together. It is a function to find their sum.
5. **Activation Function:** It introduces non-linearity in the perceptron model.



The output of the network is formed by the activation of the output neuron, which is some function of the input

$$y = F\left(\sum_{i=1}^{2} w_i x_i + \theta\right)$$

The activation function ***F*** can be linear so that we have a linear network, or nonlinear. In this section we consider the threshold F(s) or Heaviside or *sgn* function:

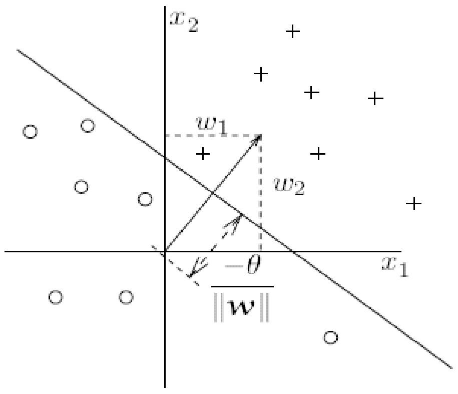
$$F(s) = \begin{cases} 1 & if\ s > 0 \\ -1 & otherwise. \end{cases}$$

The output of the network thus is either +1` or -1 depending on the input. The network can now be used for a classification task: it can decide whether an input pattern belongs to one of two classes. If the total input is positive, the pattern will be assigned to class +1. if the total input is negative, the sample will be assigned to class -1, The separation between the two classes in this case is a straight line, given by the equation:

$$w_1x_1 + w_2x_2 + \theta = 0$$

The single layer network represents a *linear discriminant function*:

The weights determine the slope of the line and the bias determines the "offset" i.e. how far the line is from the origin. Note that also the weights can be plotted in the input space: the weight vector is always perpendicular to the discriminant function



we come to the second issue: how do we learn the weights and biases in the network. We will describe two learning methods for these types of networks, the 'perceptron' learning rule and the ‘delta’ rule. Both methods are iterative procedures that adjust the weights. A learning sample is presented to the network. For each weight the new value is computed by adding a correction to the old value. The threshold is updated in a same way:

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

$$\theta(t+1) = \theta(t) + \Delta\theta(t)$$

Or update the weight the following:

$$w_{new} = w_{old} + (t-y) * x$$

The learning problem can now be formulated as: how do we compute $\Delta w_i(t)$ and $\Delta\theta(t)$ in order to classify the learning patterns correctly.

Perceptron learning algorithm and convergence theorem

Suppose the set of learning samples consisting of an input vector x and a desired output d(x). For a classification task the d(x) is usually (-1or +1) ,The perceptron learning rule is very simple and can be stated as follows:

Step 1: Initialization

Set initial weights $w_1, w_2, \ldots, w_n$ and threshold ϴ to random numbers in the range (0.5, -0.5).

Step 2: Activation

Activate the perceptron by applying inputs $x_1(p); x_2(p); \ldots ; x_n(p)$ and desired output $Y_d(p)$. Calculate the actual output at iteration p =1.

$$Y(p) = step\left[\sum_{i=1}^{n} x_i\,(p) w_i(p) - \theta\right]$$

where n is the number of the perceptron inputs, and step is a step activation function.

Step 3: Weight training

If y (p) ≠ d(x) (the perceptron gives an incorrect response), Update the weights of the perceptron

$$w_i(p+1) = w_i(p) + \Delta w_i(p)$$

Where: $\Delta w_i(p)$ is the weight correction at iteration p.

The weight correction is computed by:

$$\Delta w_i(p) = x_i(p) \times E_i(p)$$

$$E_i(p) = Y_d(p) - Y(p)$$

Where : the actual output is Y(p) and the desired output is $Y_d(p)$

Step 4: Iteration

Increase iteration p by one, go back to Step 2 and repeat the process until convergence.

```
while $\exists p \in P$ and error too large do
  Input $p$ into the network, calculate output $y$ {$P$ set of training patterns}
  for all output neurons $\Omega$ do
    if $y_\Omega = t_\Omega$ then
      Output is okay, no correction of weights
    else
      if $y_\Omega = 0$ then
        for all input neurons $i$ do
          $w_{i,\Omega} := w_{i,\Omega} + o_i$ {...increase weight towards $\Omega$ by $o_i$}
        end for
      end if
      if $y_\Omega = 1$ then
        for all input neurons $i$ do
          $w_{i,\Omega} := w_{i,\Omega} - o_i$ {...decrease weight towards $\Omega$ by $o_i$}
        end for
      end if
    end if
  end for
end while
```

Algorithm 1: Perceptron learning algorithm. The perceptron learning algorithm reduces the weights to output neurons that return 1 instead of 0, and in the inverse

Suppose that we have a single layer perceptron with randomly set weights which we want to teach a function by means of training samples. The set of these training samples is called *P*. It contains, as already defined, the pairs (*p*, *t*) of the training samples *p* and the associated teaching input *t*. I also want to remind you that:

. *x* is the input vector,

. *y* is the output vector of a neural network,

. Output neurons are referred to as 1,2, . . . ,|O|,

. *i* is the input and

. *o* is the output of a neuron.

Additionally, we defined that

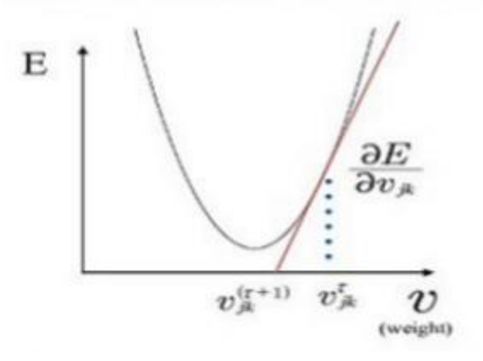
. The error vector E_p represents the difference (t–y) under a certain training sample *p*.

. Furthermore, let *O* be the set of output neurons and

. *I* be the set of input neurons.

Delta rule

The development of the perceptron was a big step towards the goal of creating useful connectionist networks capable of learning complex relations between inputs and outputs. This rule is similar to the perceptron learning rule but is also characterized by a mathematical utility and elegance missing in the perceptron and other early learning rules.



$$v_{jk}^{(\tau+1)} = v_{jk}^{\tau} + \Delta v_{jk}$$

$$\Delta v_{jk} = -\eta \frac{\partial E}{\partial v_{jk}}$$

$v_{jk}^{(\tau+1)}$	new weight
v_{jk}^{τ}	current weight
η	learning rate
E	Error Function

The Delta Rule uses the difference between target activation (i.e., target output values) and obtained activation to drive learning. Thus, the activation function is called a Linear Activation function .graphical depiction of a simple two-layer network capable of deploying the Delta Rule is given in the figure below:

$$\boldsymbol{E_p = \frac{1}{2}\sum_n (t_{jn} - a_n)^2}$$

where 'Ep' is total error over the training pattern, ½ is a value applied to simplify the function's derivative, 'n' represents all output nodes for a given training pattern, 'tj' sub n represents the Target value for node n in output layer j, and 'aj' sub n represents the actual activation for the same node.

.If we determine, analogously to the aforementioned derivation, only provides the output oi of the predecessor neuron *i* and if the function *g* is the difference between the desired activation *t* and the actual activation *a*, we will receive the ***delta rule***, also known as ***Widrow-Hoff rule***:

$$\Delta w = w - w_{old} = -\eta \frac{\partial E}{\partial w} = +\eta \delta x \quad \text{or} \quad w = w_{old} + \eta \delta x$$

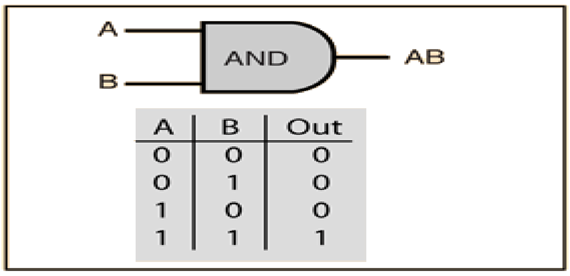
Where δ = $y_{target} - y$ and η is a constant that controls the learning rate (amount of increment/update Δw at each training step).

***Note*:** Delta rule is similar to the Perceptron Learning rule with some differences:

1. Error (δ) in delta rule is not restricted to having values of 0, 1, or -1 (as in perceptron) but may have any value.
2. Delta rule can be derived for any *differentiable* output/activation function *f*, whereas in perceptron only works for threshold output function.
3. Note that the rule will be different for not linear *f*.

Example : AND Gate

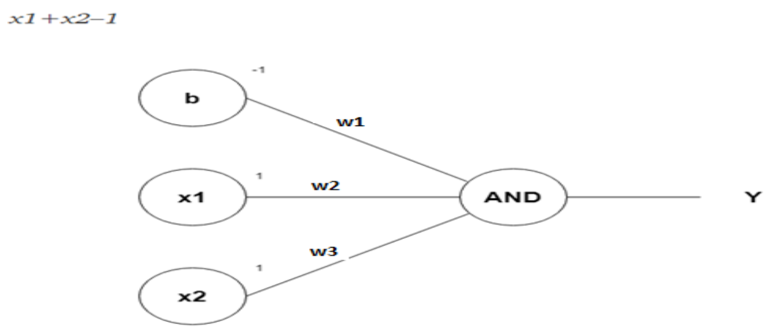
From our knowledge of logic gates, we know that an AND logic table is given by the diagram below



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

The question is, what are the weights and bias for the AND perceptron?

First, we need to understand that the output of an AND gate is 1 only if both inputs (in this case, x1 and x2) are 1. So, following the steps listed above;



Row 1

- From $w_1*x_1+w_2*x_2+b$, initializing w1, w2, as 1 and b as –1, we get;

  $x_1(1)+x_2(1)–1$

- Passing the first row of the AND logic table ($x_1=0$, $x_2=0$), we get;

  $0+0–1 = –1$

- From the Perceptron rule, if $Wx+ b\leq 0$, then y`=0. Therefore, this row is correct, and no need for Back propagation.

Row 2

- Passing ($x_1=0$ and $x_2=1$), we get;

  $0+1–1 = 0$

- From the Perceptron rule, if $W_x+b\leq0$, then y`=0. This row is correct, as the output is 0 for AND gate.

- From the Perceptron rule, this works (for both row 1, row 2 and 3).

Row 4

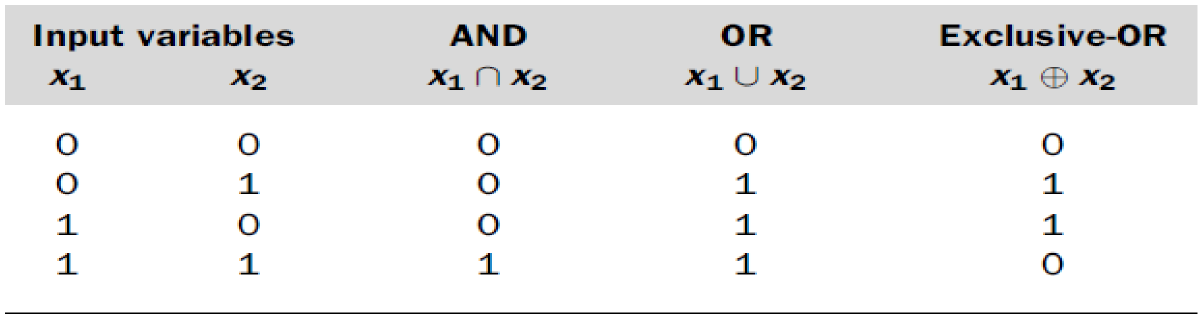
- Passing (x_1=1 and x_2=1), we get;

  $1+1–1 = 1$

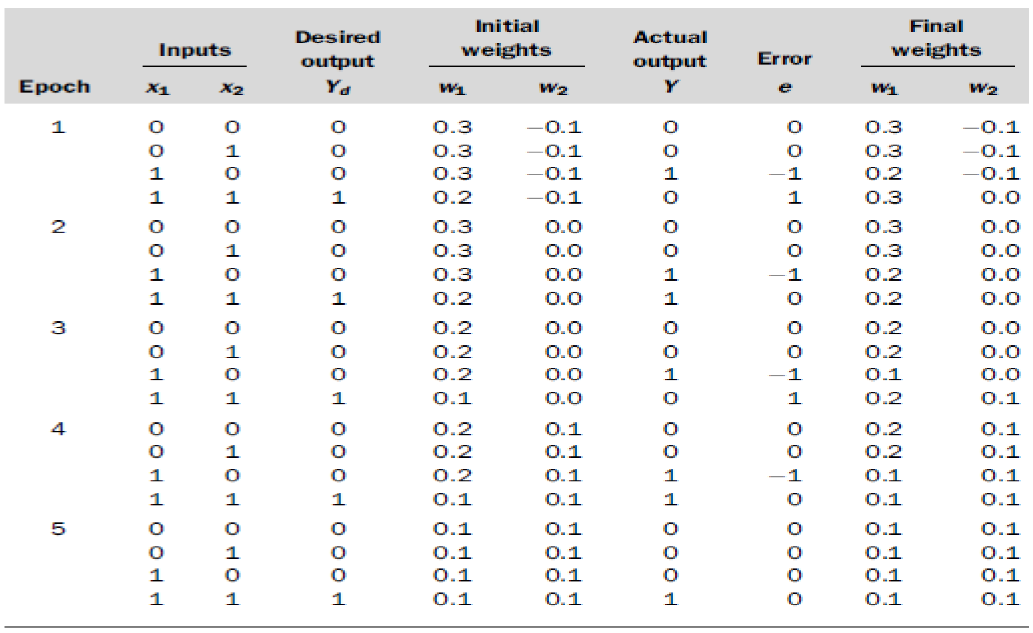
- Again, from the perceptron rule, this is still valid.

Therefore, we can conclude that the model to achieve an AND gate, using the Perceptron algorithm is;

$x_1+x_2–1$



Input variables		AND	OR	Exclusive-OR
x_1	x_2	$x_1 \cap x_2$	$x_1 \cup x_2$	$x_1 \oplus x_2$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
Epoch	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	−1	0.2	−0.1
	1	1	1	0.2	−0.1	0	1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	−1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	−1	0.1	0.0
	1	1	1	0.1	0.0	0	1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	−1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1