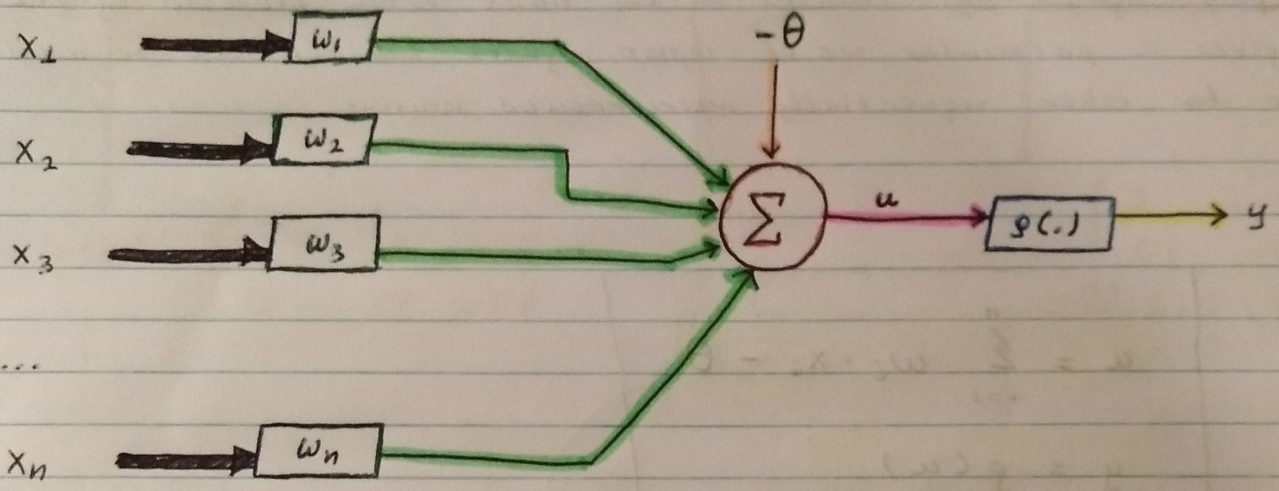
"Neural Networks"

"Artificial neuron - The perceptron network"



a) **Input signals** $(x_1, x_2, x_3, \ldots, x_n)$ are the signals or samples coming from the external environment and representing the values assumed by the variables of a particular application. The input signals are usually normalized in order to enhance the computational efficiency of learning algorithms.

b) **Synaptic weights** $(w_1, w_2, w_3, \ldots, w_n)$ are the values used to weight each one of the input variables, which enables the quantification of their relevance with respect to the functionality of the neuron.

c) **Linear aggregator** (Σ) gathers all input signals weighted by the synaptic weights to produce an activation voltage.

d) **Activation threshold** or **bias** (θ) is a variable used to specify the proper threshold that the result produced by the linear aggregator should have to generate a trigger value toward the neural output.

e) **Activation potential** (u) is the result produced by the difference between the linear aggregator and the activation threshold. if $u \geq \theta$ then the neuron produces an excitatory potential, otherwise it will be inhibitory.

f) **Activation function** (g) whose goal is limiting the neuron output within a reasonable range of values, assured by its own functional image.

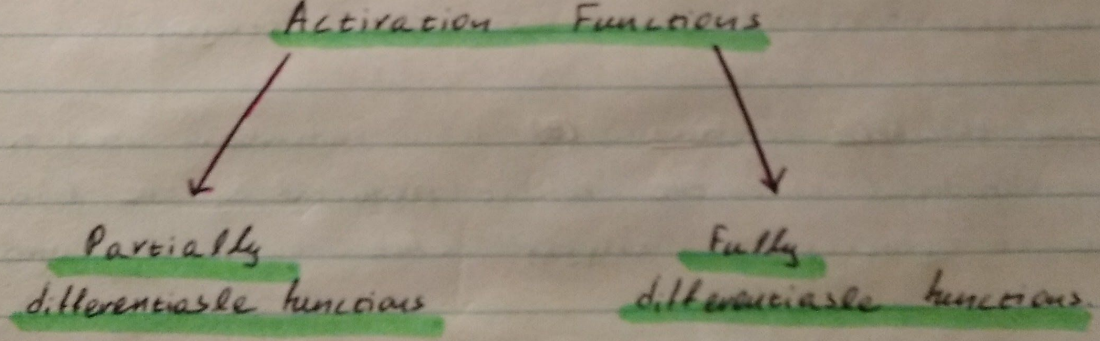
g) **Output signal** (y) consists on the final value produced by the neuron given a particular set of input signals, and can also be used as input for other sequentially interconnected neurons.

$$u = \sum_{i=1}^{n} w_i \cdot x_i - \theta$$

$$y = g(u)$$

Thus the artificial neuron operation can be summarized as follows:

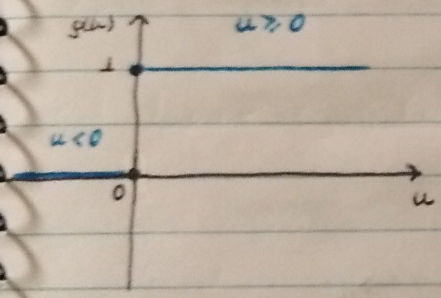
1. Present a set of values to the neuron, representing the input.
2. Multiply each input of the neuron to it's corresponding weight.
3. Obtain the activation potential produced by the weighted sum of the input signals and subtract the activation threshold.
4. Apply a proper activation function to limit the neuron output.
5. Compile the output by employing the neural activation function in the activation potential.



Partially differentiable Activation functions

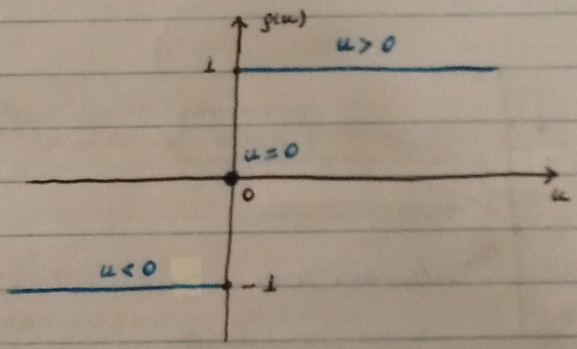
step function

$$g(u) = \begin{cases} 1, & \text{if } u \geq 0 \\ 0, & \text{if } u < 0 \end{cases}$$



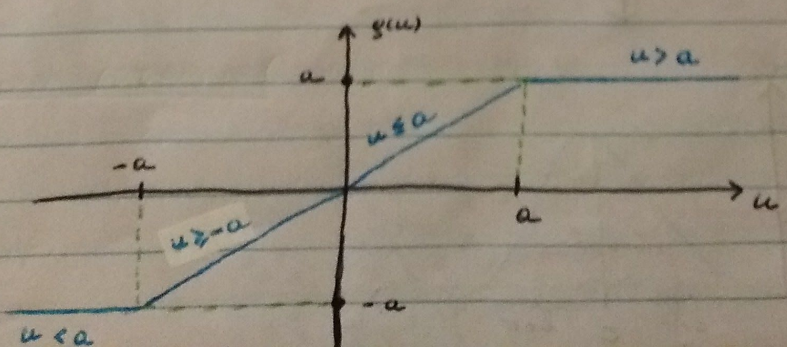
Bipolar step function

$$g(u) = \begin{cases} 1, & \text{if } u > 0 \\ 0, & \text{if } u = 0 \\ -1, & \text{if } u < 0 \end{cases}$$



Symmetric ramp function

$$g(u) = \begin{cases} a, & \text{if } u > a \\ u, & \text{if } -a \leq u \leq a \\ -a, & \text{if } u < a \end{cases}$$



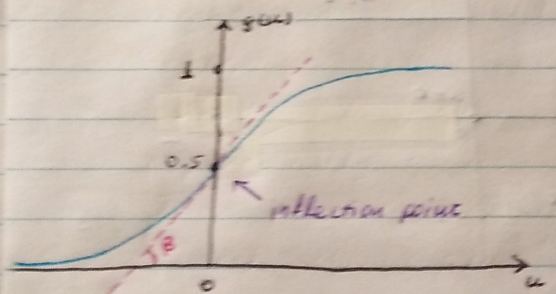
Fully differentiable activation functions

- Logistic function
- Hyperbolic tangent function
- Linear Function → $g(u) = u$

wormhole, entry

Logistic function

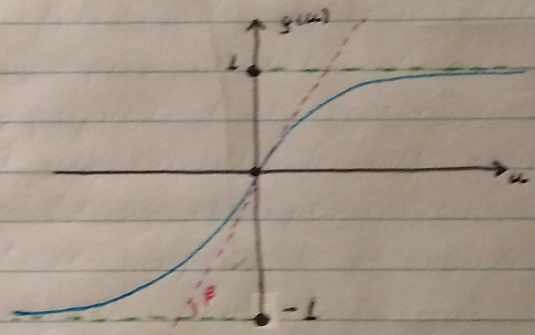
$$g(u) = \frac{1}{1 + e^{-\beta \cdot u}}$$



when $\beta \to +\infty$, logistic function approximates the step function.

Hyperbolic tangent function

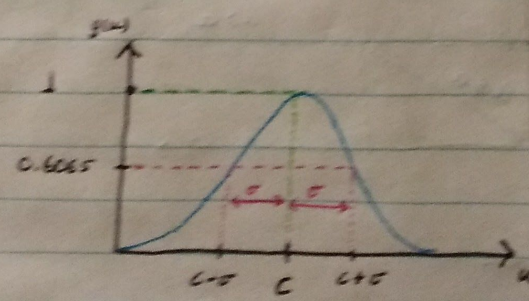
$$g(u) = \frac{1 - e^{-\beta \cdot u}}{1 + e^{-\beta u}}$$



when $\beta \to +\infty$, hyperbolic tangent function approximates the bipolar step function

wormhole, exit

Gaussian Function → $g(u) = e^{-\frac{(u-c)^2}{2\sigma^2}}$

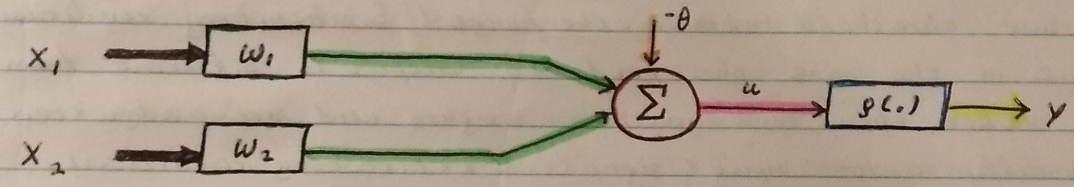


ExecutionHandler

perceptron classification: The perceptron network belongs to the class of single-layer feedforward architectures, because the information that flows in its structure is always from the input layer to the output layer, without any feedback from the output produced by its single output neuron.

"Mathematical Analysis of the Perceptron Network"

From the mathematical analysis of the Perceptron and by considering the bipolar step function, it becomes possible to verify that such network can be considered a typical case of a linear discriminator. To demonstrate this scenario, consider a Perceptron with only two inputs, as illustrated below:



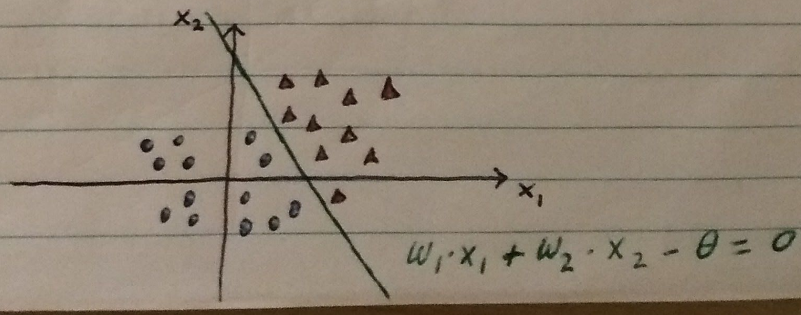
In mathematical notation, the perceptron output, which uses the bipolar step activation function, is given by:

$$y = \begin{cases} 1, & \text{if } \sum w_i \cdot x_i - \theta \geq 0 \iff w_1 \cdot x_1 + w_2 \cdot x_2 - \theta \geq 0 \\ -1, & \text{if } \sum w_i \cdot x_i - \theta < 0 \iff w_1 \cdot x_1 + w_2 \cdot x_2 - \theta < 0 \end{cases}$$

As the above inequalities are represented by linear equations, the classification boundary for this case (perceptron with two inputs) will be a straight line given by:

$$w_1 \cdot x_1 + w_2 \cdot x_2 - \theta = 0$$

Thus it is possible to conclude that the perceptron behaves as a pattern classifier whose purpose is to divide linearly seperable classes.



A perceptron with three inputs (three dimensions) will have the decision boundary represented by a plane.
A perceptron with n inputs (n dimensions) will have the decision boundary represented by a hyperplane.

Perceptron convergence theorem: For a perceptron to be applied as a pattern classifier the classes of the problem being mapped must be linearly seperable.

"Training process of the perceptron"

Hebb's learning rule: if the output produced by the perceptron coincides with the desired output, it's synaptic weights and threshold remain unchanged (inhibitory condition). Otherwise in the case the produced output is different from the desired value, then its synaptic weights and are adjusted proportionally to its input signals (excitatory condition).
This process is repeated sequentially for all training samples until the output produced by the perceptron is similar to the desired output for all samples. In mathematical notation, the rules for adjusting the synaptic weight w_i and threshold θ can se expressed respectively, by the following equations:

$$w_i^{current} = w_i^{previous} + (d^{(k)} - y) \cdot x_i^{(k)}$$

$$\theta^{current} = \theta^{previous} + (d^{(k)} - y) \cdot (-1)$$

As the same adjustment rule is applied to both the synaptic weights and threshold, it is possible to insert the threshold value θ within the synaptic weight vector. Thus

$$\vec{w}^{current} = \vec{w}^{previous} + \eta (d^{(k)} - y) \cdot x^{(k)}$$

In algorithmic notation, the inner processing performed by the perceptron can be described by the following expression:

$$\vec{w} \leftarrow \vec{w} + \eta \cdot (d^{(k)} - y) \cdot x^{(k)}$$

where:

$\vec{w} = (\theta \quad w_1 \quad w_2 \quad \dots \quad w_n)^T$ is the vector containing the threshold and the weights.

$X^{(k)} = (-1 \quad x_1^{(k)} \quad x_2^{(k)} \quad \dots \quad x_n^{(k)})$ is the kth training sample.

$d^{(k)}$ is the desired value for the kth training sample.

y is the output produced by the perceptron.

η is a constant that defines the learning rate. $(0 < \eta < 1)$

Perceptron training algorithm

```
procedure perceptron_train(X, D):
        X = {x^(1), x^(2), ..., x^(n)}  training sample.
        D = {d^(1), d^(2), ..., d^(n)}  output of each sample

W := {w_1, w_2, ..., w_n}
for ∀w_i ∈ W:  w_i := rand()
η := a,  a ∈ (0, 1)
epoch := 0
do:
    error := "none"
    for ∀x_i ∈ X and ∀d_i ∈ D:
        u := W^T · X_i
        y := signal(u)
        if y ≠ d_i:
            W := W + η · (d_i - y) · x_i
            error := "existent"
    epoch := epoch + 1
while error == "existent"
end
```

Perceptron operation phase algorithm

procedure perceptron − predict(W, x_i):

x_i = some unseen input sample.

W = the adjusted weights of the perceptron.

$u := W^T \cdot x_i$

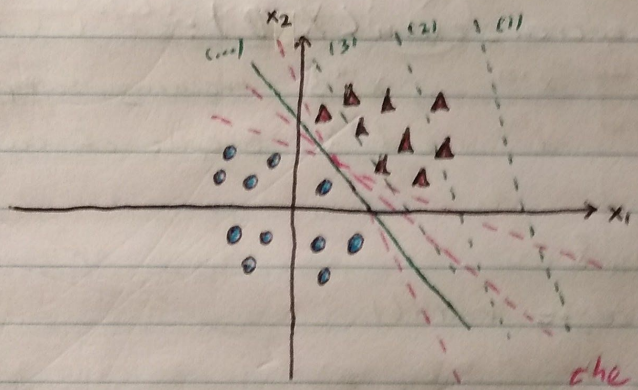
$y := signal(u)$

if $y == -1$: $x \in$ { the class associated with -1 }

else: $x \in$ { the class associated with 1 }

end

The perceptron training process tends to move the classification hyperplane continuously until it meets a decision boundary that allows the seperation of both classes.



← separation line adjustment after a certain number of epochs.

the seperabiling line produced is not unique and depending on the initial values of W the number of epochs may vary.

a) the network will diverge if problem is nonlinearly seperable. The strategy for this scenario is to limit the training process by a maximum number of epochs.

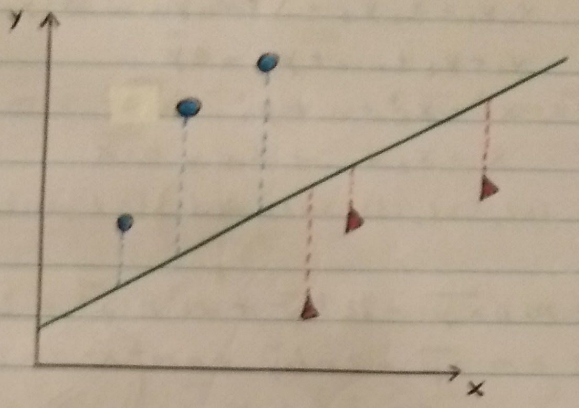
b) small value learning rate mitigate instability.

c) The number of epochs depends on the initial values of W.

d) The closer is the decision surface to the separating boundary, the fewer epochs are usually required for convergence.

e) normalizing input signals enhances performance.

"Lease mean square"
a.k.a "Delta rule"



$y = (mx + b)$

we want to find m, b such that it minimizes the squared error of all points from the line.

We define as error the vertical distance of a point from a line, namely $\text{error} = y_i - y = y_i - (mx_i + b)$. The reason we take the square is because of some nice mathematical properties such as being differentiable everywhere and having uniquely nice geometric properties, which the absolute does not have.

The total square error against the line $y = mx + b$ is the sum of the squared errors of each point (x_i, y_i) against the line.

$$E(m,b) = (y_1 - (mx_1 + b))^2 + (y_2 - (mx_2 + b))^2 + \dots + (y_n - (mx_n + b))^2$$

We want to find $(m^*, b^*) \in \mathbb{R}^2$ s.t. $E(m^*, b^*) \leq E(m, b)$

$$\begin{aligned}
E(w,b) &= y_1^2 - 2(mx_1 + b)y_1 + (mx_1 + b)^2 \\
&+ y_2^2 - 2(mx_2 + b)y_2 + (mx_2 + b)^2 \\
&+ \dots \\
&+ y_n^2 - 2(mx_n + b)y_n + (mx_n + b)^2 \\
&= y_1^2 - 2y_1 mx_1 - 2by_1 + m^2x_1^2 + 2mx_1 b + b^2 \\
&+ y_2^2 - 2y_2 mx_2 - 2by_2 + m^2x_2^2 + 2mx_2 b + b^2 \\
&+ \dots \\
&+ y_n^2 - 2y_n mx_n - 2by_n + m^2x_n + 2mx_n b + b^2 \\
&= (y_1^2 + y_2^2 + \dots + y_n^2) - 2m(x_1y_1 + x_2y_2 + \dots + x_ny_n) - 2b(y_1 + y_2 + \dots + y_n) \\
&+ m^2(x_1^2 + x_2^2 + \dots + x_n^2) + 2mb(x_1 + x_2 + \dots + x_n) + nb^2
\end{aligned}$$

we know that:

$$y_1^2 + y_2^2 + \dots + y_n^2 = n \cdot \overline{y^2} \quad (1)$$

$$x_1 y_1 + x_2 y_2 + \dots + x_n y_n = \overline{x \cdot y} \quad (2)$$

$$y_1 + y_2 + \dots + y_n = n\bar{y} \quad (3)$$

$$x_1^2 + x_2^2 + \dots + x_n^2 = n\overline{x^2} \quad (4)$$

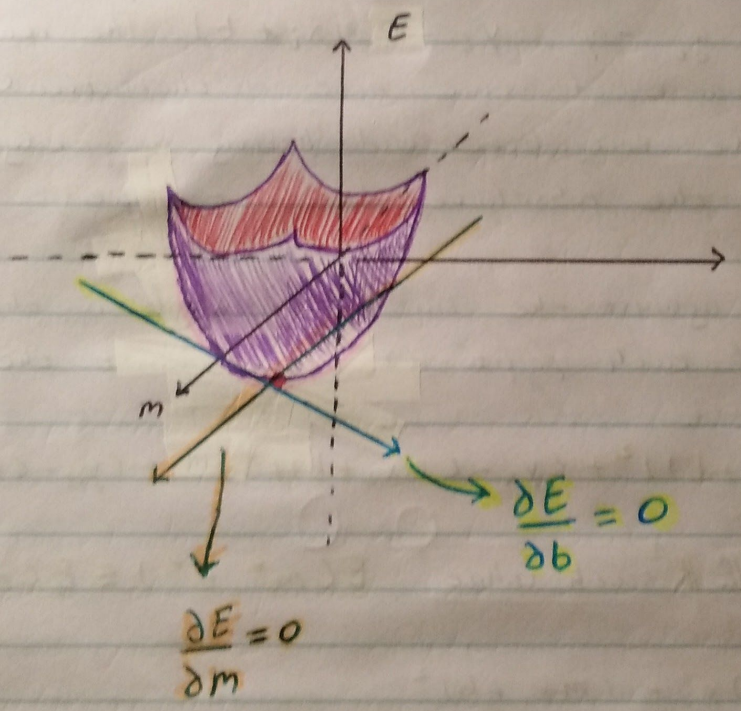
$$x_1 + x_2 + \dots + x_n = n \cdot \bar{x} \quad (5)$$

By substituting (1), (2), (3), (4) and (5) into $E(m,b)$:

$$E(m,b) = n \cdot \overline{y^2} - 2mn\overline{x \cdot y} - 2nb\bar{y} + m^2 n \cdot \overline{x^2} + 2mbn\bar{x} + nb^2 \iff$$

$$E(m,b) = n \cdot \overline{y^2} - 2 \cdot n \cdot m\ \overline{xy} - 2nb\bar{y} + nm^2\bar{x}^2 + 2nmb\bar{x} + nb^2$$

Because we want to find the minimums of $E(m,b)$, that occurs when the tangent line with respect to m is flat and parallel to the m-axis and the tangent line with respect to b is flat and parallel to the b-axis.



or equivalently $\nabla E(m,b) = \vec{0} \iff \left(\frac{\partial E}{\partial m}, \frac{\partial E}{\partial b}\right) = \vec{0}$

$$\iff \left(\frac{\partial E}{\partial m}, \frac{\partial E}{\partial b}\right) = (0,0) \iff \frac{\partial E}{\partial m} = 0 \text{ and } \frac{\partial E}{\partial b} = 0$$

And we know from theory that maximums or minimums exist at the points where the derivatives are equal to zero.

$$\frac{\partial E}{\partial m} = -2n\,\bar{xy} + 2n\,\bar{x}^2 m + 2bn\,\bar{x} = 0 \quad (6)$$

$$\frac{\partial E}{\partial b} = -2n\,\bar{y} + 2mn\,\bar{x} + 2bn = 0 \quad (7)$$

We have two equations with two unknowns. Without breaking the symmetry, we multiply (6) with $\frac{1}{2n}$:

$$-\frac{2n}{2n}\overline{xy} + \frac{2n}{2n}\overline{x^2}m + \frac{2n}{2n}b\bar{x} = 0 \Leftrightarrow -\overline{xy} + m\overline{x^2} + b\bar{x} = 0 \quad (8)$$

equivalently we multiply (7) with $\frac{1}{2n}$:

$$-\frac{2n}{2n}\bar{y} + \frac{2n}{2n}m\bar{x} + \frac{2n}{2n}b = 0 \Leftrightarrow -\bar{y} + m\bar{x} + b = 0 \quad (9)$$

From (8), (9) we get: $\left\{ \begin{array}{l} \overline{x^2}m + b\bar{x} = \overline{xy} \\ m\bar{x} + b = \bar{y} \end{array} \right\} \Rightarrow$

$b = \bar{y} - m\bar{x}$ and $m\overline{x^2} + (\bar{y} - m\cdot\bar{x})\bar{x} = \overline{xy} \Leftrightarrow$

$$m\overline{x^2} + \bar{y}\bar{x} - m(\bar{x})^2 = \overline{xy} \Leftrightarrow$$

$$-m(\bar{x})^2 + m\overline{x^2} = \overline{xy} - \bar{x}\bar{y}$$

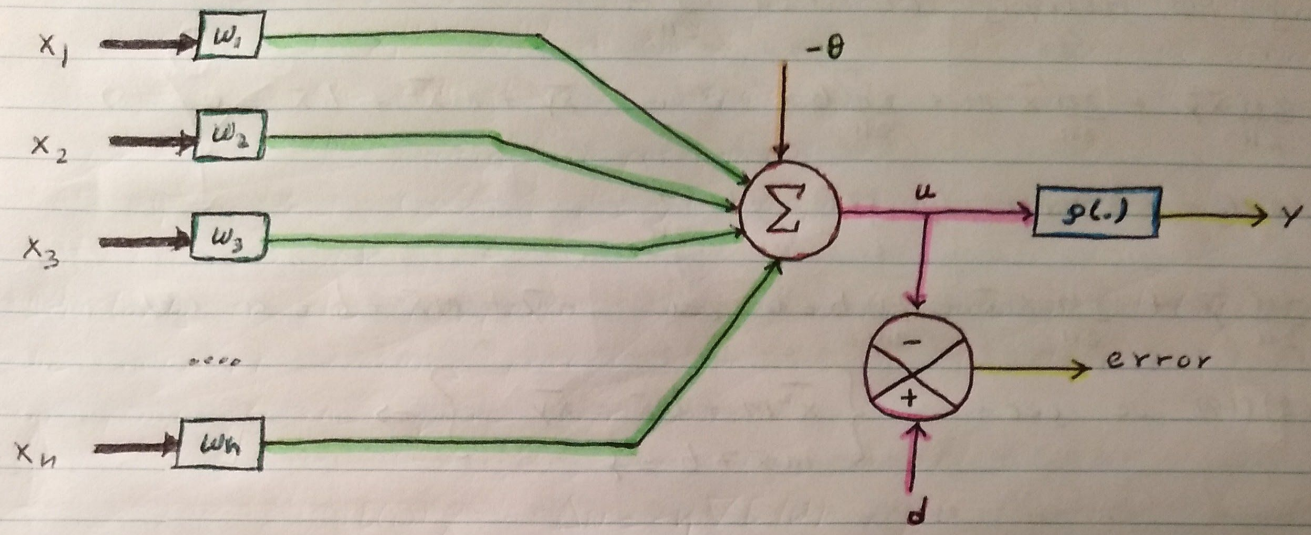
$$m(\overline{x^2} - (\bar{x})^2) = \overline{xy} - \bar{x}\bar{y} \Leftrightarrow$$

$$m = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - (\bar{x})^2}$$

Therefore:

$$\boxed{b = \bar{y} - m\cdot\bar{x}} \quad \text{and} \quad \boxed{m = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - (\bar{x})^2}}$$

"The Adaline Network"



The Adaline network is very similar to the perceptron network, that is, it belongs to the single-layer feedforward architecture and the classes from the problem being mapped must be linearly seperable in order to be completely identified by the network.

The main difference between the Adaline and the perceptron is on the learning rule used to adjust weights and threshold.

The rule performs the minimization of the squared error between u and d to adjust the weight vector $\vec{w} = [\theta \; w_1 \; w_2 \; \dots \; w_n]^T$ of the network. In summary the objective is to obtain an optimal $\vec{w}^*$ so that the squared error $E(\vec{w}^*)$ of the whole sample set is as low as possible. In mathematical notation, considering an optimal weight configuration, it can be stated that:

$$E(\vec{w}^*) \leq E(\vec{w}) \qquad \forall \vec{w} \in \mathbb{R}^{n+1}$$

The function of the squared error related to the p training samples is defined by:

$$E(\vec{w}) = \frac{1}{2} \sum_{k=1}^{p} (d^{(k)} - u)^2 \qquad (1)$$

$$u = \sum_{i=1}^{n} w_i \cdot x_i - \theta \qquad (2)$$

substituting (2) into (1) we get:

$$E(\vec{\omega}) = \frac{1}{2} \sum_{k=1}^{p} \left(d^{(k)} - \left(\sum_{i=1}^{n} w_i \cdot x_i^{(k)} - \theta \right) \right)^2 \quad \Leftrightarrow$$

$$E(\vec{\omega}) = \frac{1}{2} \sum_{k=1}^{p} \left(d^{(k)} - \left(\omega^T \cdot x^{(k)} - \theta \right) \right)^2 \qquad (3)$$

Thus equation (3) computes the mean squared error by processing the p training samples provided for the training process of the Adaline network.

The next step consists of the application of the gradient operator on the mean squared error with respect to vector ω, in order to search for an optimal value for the squared error function given by (3), that is:

$$\nabla E(\vec{\omega}) = \frac{\delta E(\vec{\omega})}{\delta \vec{\omega}} \quad \Leftrightarrow$$

$$\nabla E(\vec{\omega}) = 2 \cdot \frac{1}{2} \sum_{k=1}^{p} \left(d^{(k)} - (\omega^T \cdot x^{(k)} - \theta) \cdot (-x^{(k)}) \right) \quad \Leftrightarrow$$

$$\nabla E(\vec{\omega}) = -\sum_{k=1}^{p} \left(d^{(k)} - (\omega^T \cdot x^{(k)} - \theta) \cdot x^{(k)} \right)$$

Finally the steps for adapting the weight vector must be executed in the opposite direction of the gradient, because the optimization goal is to minimize the squared error which in mathematical terms means to find the global minimum of $E(\vec{\omega})$. We know from calculus that the global extrema are located at $\vec{\omega}$ s.t. $\nabla E(\vec{\omega}) = \vec{0}$. But because computers cannot solve $\nabla E(\vec{\omega}) = \vec{0}$ the way we humans do, we have to approximate it via iterative steps η until convergence. In this condition, the variation $\Delta\vec{\omega}$ to update the Adaline weights vector is given by: $\Delta\omega = -\eta \nabla E(\vec{\omega}) \Leftrightarrow$

$$\vec{\omega}^{\,current} - \vec{\omega}^{\,previous} = \eta \sum_{k=1}^{p} \left(d^{(k)} - u \right) \cdot x^{(k)} \quad \Leftrightarrow$$

$$\vec{\omega}^{\,current} = \vec{\omega}^{\,previous} + \eta \cdot \left(d^{(k)} - u \right) \cdot x^{(k)}, \quad \forall k \in \{1, 2, \ldots, p\}$$

In algorithmic notation:

$$\vec{\omega}^{\,current} \leftarrow \vec{\omega}^{\,previous} + \eta \cdot \left(d^{(k)} - u \right) \cdot x^{(k)}, \quad \text{with } k = 1, 2, \ldots, p$$

where:

$\vec{w} = [\theta \quad w_1 \quad w_2 \quad \dots \quad w_n]^T$, vector containing threshold and weights

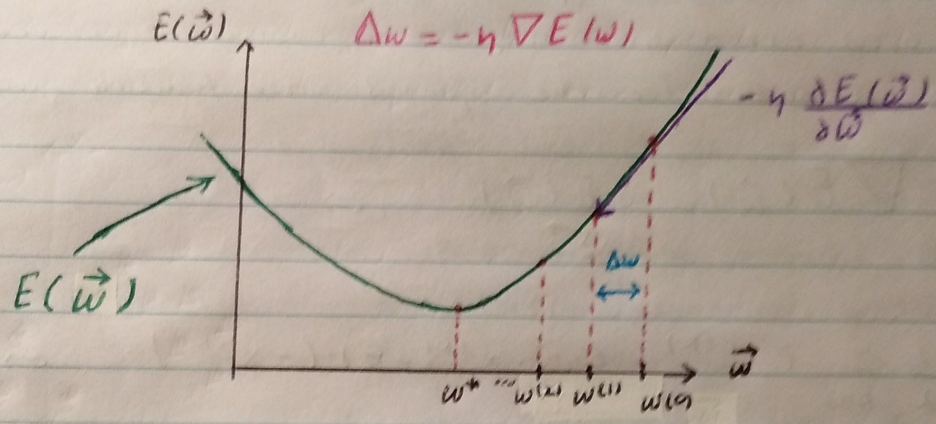
$x^{(k)} = [-1 \quad x_1^{(k)} \quad x_2^{(k)} \quad \dots \quad x_n^{(k)}]^T$, is the kth training sample

$d^{(k)}$ is the kth desired output signal.

u, is the activation potential.

η, the learning rate ($0 < \eta < 1$)

Similarly to the perceptron, the learning rate η defines how fast the network training process advances in the direction of the minimum point of the square error function.



The stopping criterion is stipulated by employing the mean square error $\bar{E}(\vec{w})$ with respect to all training samples, and is defined:

$$\bar{E}(\vec{w}) = \frac{1}{P} \sum_{k=1}^{P} (d^{(k)} - w)^2 .$$

The algorithm converges when the difference of the mean square between two successive epochs is small enough, that is:

$$| \bar{E}(\vec{w}^{\,current}) - \bar{E}(\vec{w}^{\,previous}) | \leq \varepsilon$$

where ε is the precision required for the convergence process. It is specified by taking into account the performance requirements of the application mapped by the Adaline network.

```
procedure   adaline_train(X, D, η, ε):
                X = { x^(1), x^(2), ..., x^(n) }, training sample.
                D = { d^(1), d^(2), ..., d^(n) }, output of each sample.
                η = learning rate.
                ε = precision of convergence.

W := { w_1, w_2, ..., w_n }
for ∀w_i ∈ W:  w_i := rand()
epochs := 0
do:
    mse_previous := mse(W, X, D)
    for ∀x_i ∈ X and ∀d_i ∈ D:
        u := W^T · x_i
        w := w + η·(d_i - u)·x_i
    epochs := epochs + 1
    mse_current := mse(W, X, D)
while abs(mse_current - mse_previous) ≤ ε
end


procedure   mse(W, X, D):
                W = [θ w_1 w_2 ... w_n]^T, the weights vector.
                X = { x^(1), x^(2), ..., x^(n) }, the training sample.
                D = { d^(1), d^(2), ..., d^(n) }, output of each sample.

value := 0
for ∀x_i ∈ X and ∀d_i ∈ D:
    u := W^T · x_i
    value := value + (d_i - u)^2
value := value / 101
end
```

procedure adaline_predict(X, W):

X = Some unseen input sample.

$W = [\theta \; w_1 \; w_2 \; \dots \; w_n]^T$, adjusted weights.

$u := W^T \cdot X$

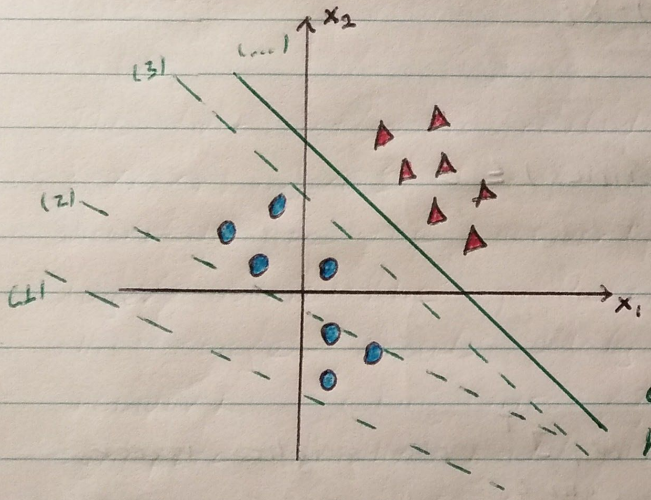
$y := signal(u)$

if $y == -1$: $x \in$ { the class associated with -1 }

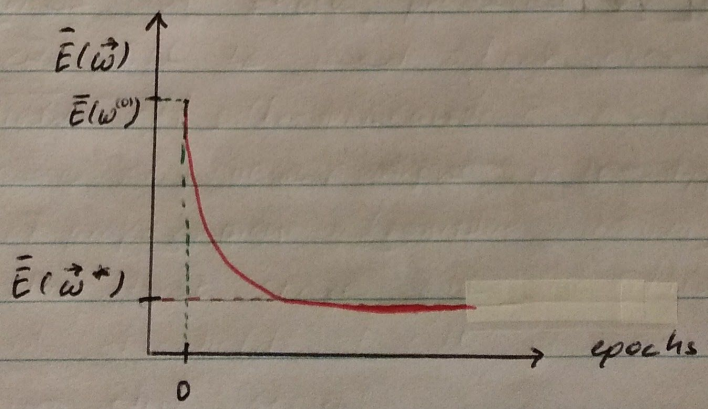
if $y == 1$: $x \in$ { the class associated with 1 }

end

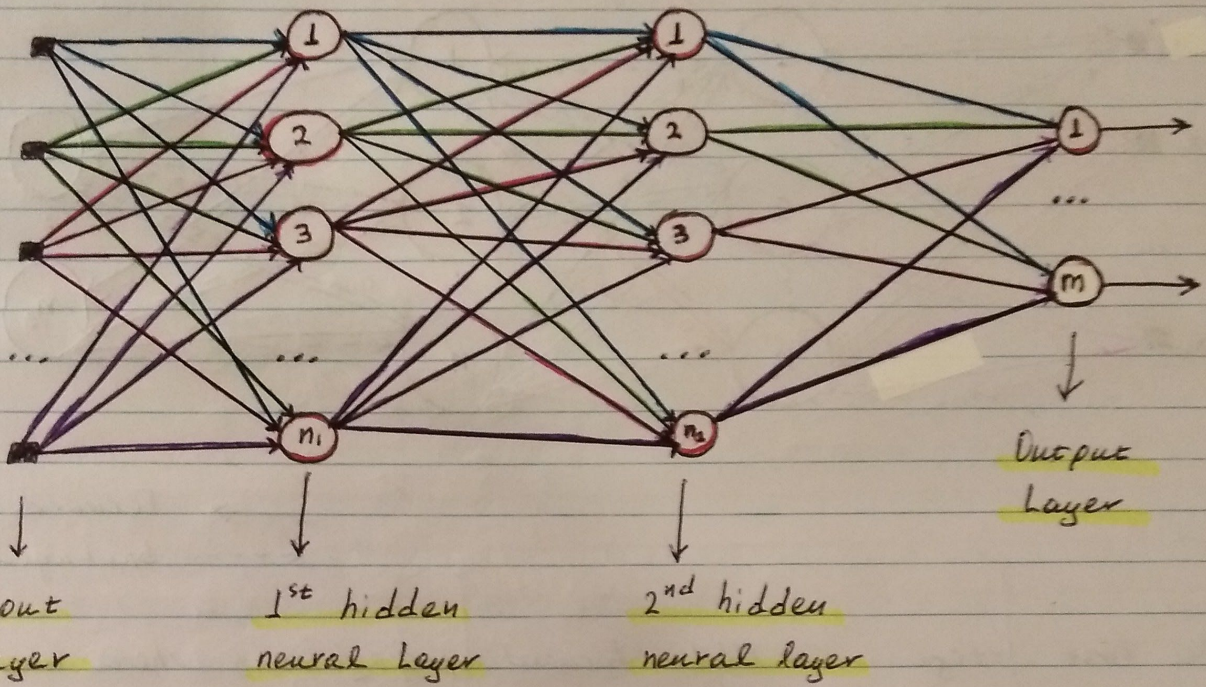
The convergence process moves the hyperplane toward the optimal seperasility boundary, which corresponds to the w^* value that minimizes the squared error function.



The mean squared error curve of the Adaline is always descendant meaning it decreases as the number of training epochs increases. It estaslishes at a constant value when the minimum point of the squared error function is reached.

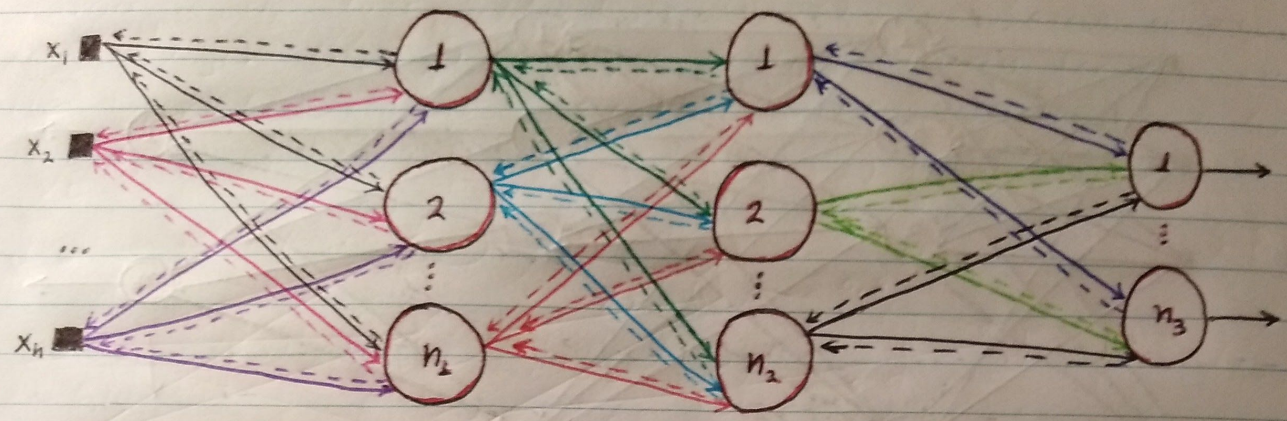


"Multilayer Perceptron Networks"



In summary, in contrast to the Perceptron or Adaline networks, in which a single neuron is responsible for the full mapping of the whole process, the knowledge related to the behavior of the input and outputs of the system will be distributed among all neurons composing the MLP. The stimuli or signals are presented to the network on its input layer. The intermediate layers, on the other hand, extract the majority of the information related to the system behavior and codify them using synaptic weights and thresholds of their neurons, thus forming a representation of the environment where the particular system exists. Finally the neurons of the output layer receive the stimuli from the neurons of the last intermediate layer, producing a response pattern which will be the output generated by the network.

"Training process of the Multilayer perceptron"



→ Forward stage

←--- backward stage

The first stage is called forward propagation, where the signals $\{x_1, x_2, \ldots, x_n\}$ of a given sample from the training set are inserted into the network inputs and are propagated layer-by-layer until the production of the corresponding outputs. Thus this stage intends solely in obtaining the responses from the network, taking into account only the current values of the synaptic weights and thresholds of its neurons, which will remain unmodified during the execution stage.

Next, the response produced by the network outputs are compared to the respective available desired responses, since it is a supervised learning process, as mentioned earlier. It is important to note that, considering an MLP network with n_3 neurons in its output layer, the respective n_3 deviations (errors) between the desired responses and those produced by the output neurons are calculated and will be used after that to adjust the weights and thresholds of all neurons.

Therefore, because of these errors, it is applied the second stage of the backpropagation algorithm, known as backward propagation. Unlike the first stage the modifications of the synaptic weights and thresholds of all neurons of the network are executed during this stage.

This successive application of forward and backward stages results in the gradual reduction of the sum of the errors.