Neural Networks

Chp.3 : Learning

• Given a layered structure and the input x, the output of the neural network will depend on the weights :

• $o = F(W, x)$

• We arrange the weights so that the neural network performs a given task (possibly with an acceptable performance). Such tasks are e.g. classification, pattern recognition, function approximation, memorizing patterns, etc...

• **Learning** is a systematic way of changing the weights so that the given task is achieved (either after a finitely many steps, or asymptotically)

• Usually we have a training set set $\mathcal{S}_{Tr} = \{x^1, x^2, \ldots x^N\}$

• We start with an initial structure, and hence an initial weight $W(1)$

• Choose a training set example, say x^1 $\rightarrow$ Find $o(1) = F(W(1)x^1)$

• Then, based on some criterion we **change = update** the weights :

• $W(2) \leftarrow W(1) + \Delta W(1)$

• The way we select ΔW is called a **learning rule**

• Then choose a training set example (which is not chosen before), say x^2 $\rightarrow$ Find $o(2) = F(W(2)x^2)$

• Then, based on the same criterion we **change = update** the weights :

• $W(3) \leftarrow W(2) + \Delta W(2)$

• We keep this procedure till all the training set examples are exhausted. This is called an **epoch** in learning. After the end of the epoch, we obtain a set of weights $W(n)$. If this weight is not sufficient for our performance requirements, we start **another** epoch, and continue this way, till our performance measures are satisfied.

• Sometimes, this process may not converge (i.e. performance may degrade). Then we may stop the training, and start the process with different initial weight, or change some parameters of the learning process...

• There are basically 3 types of learning :

• **1 : Supervised Learning**. Here, at each pattern in the training set, we know what the correct output should be. And we decide on ΔW accordingly. This is called **learning with a teacher**.

• **2 : Unsupervised Learning**. Here, we do not know the correct output for the training patters. The network should organize itself $\rightarrow$ **Self-Organization**

• **3 : Reinforcement Learning**. Here, we have an **idea** about the **good** outputs, and **bad** outputs, and we decide on ΔW accordingly (reward and punishment) $\rightarrow$ game learning.

• In terms of updating, there are 2 different types :

• **1 : Stepwise Learning** One input is given, weights are updated. Then in the next input, last updated weights are used. In this case, there is no need to store the weight increments. This is sometimes called **on-line learning**.

• **2 : Batch Learning** One input is given, necessary weight increments are found, **but the weights are not updated**. When the next input is given, previous weights are used, and again the necessary weight increments are found. This process is repeated till the end of an epoch. At the end of epoch, the weight increments are added to find the total increment, and added to the weights used within the epoch. This is sometimes called as **off-line learning**.

• **Supervised Learning** We may think learning as an approximation problem. Suppose that we have a function $h : \mathbf{R^n} \rightarrow \mathbf{R^m}$. That is :

• $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad h(x) = \begin{pmatrix} h_1 \\ \vdots \\ h_m \end{pmatrix}$

• Suppose that we have a **training set** $\mathcal{T}_r = \{x^1,\ x^2,\ \ldots, x^N\}$ and we know the training values $\{h(x^1),\ h(x^2),\ \ldots, h(x^N)\}$.

• Suppose that we choose a Neural Network structure with **unknown** weights. The output of such a NN will depend on both weights and the inputs. $\Rightarrow o(x) = H(W, x)$.

• The learning problem is : Find a weight W_* such that $H(W_*, x)$ is as close to the $h(x)$ over the training set as possible.

• Problem is how to define **closeness** $\rightarrow$ a metric.

• Given vectors $x, y \in \mathbf{R^n}$, a metric ρ is a function if it satisfies the following 3 conditions :

• **1 :** $\rho(x,y) = \rho(y,x) \geq 0$, and $\rho(x,y) = 0 \Leftrightarrow x = y$

• **2 :** $\rho(\alpha x, \alpha y) = \mid \alpha \mid \rho(x,y) \qquad \alpha \in \mathbf{R}$

• **3 :** $\rho(x,z) \leq \rho(x,y) + \rho(y,z) \qquad$ (Triangle inequality).

• We may treat $\rho(x,y)$ as the distance between the vectors x and y. Typical examples :

• $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$

• Euclidean distance : $\rho(x,y) = \sqrt{(x_1 - y_1)^2 + \ldots + (x_n - y_n)^2} = \|x - y\|$

• $\rho(x,y) = \mid x_1 - y_1 \mid + \ldots + \mid x_n - y_n \mid$

• $\rho(x,y) = \max\{\mid x_1 - y_1 \mid, \ldots, \mid x_n - y_n \mid\}$

• We will prefer **Euclidean distance**, since it is differentiable with respect to x and y. Note that $\|x - y\|^2 = (x - y)^T (x - y)$

• So, the learning problem is : given a metric ρ, and a training set $\mathcal{T}_r$, find a weight vector W_* such that

• $\rho(H(W_*, x), h(x)) \leq \rho(H(W, x), h(x))$ for any $x \in \mathcal{T}_r$ and for any other weight vector W.

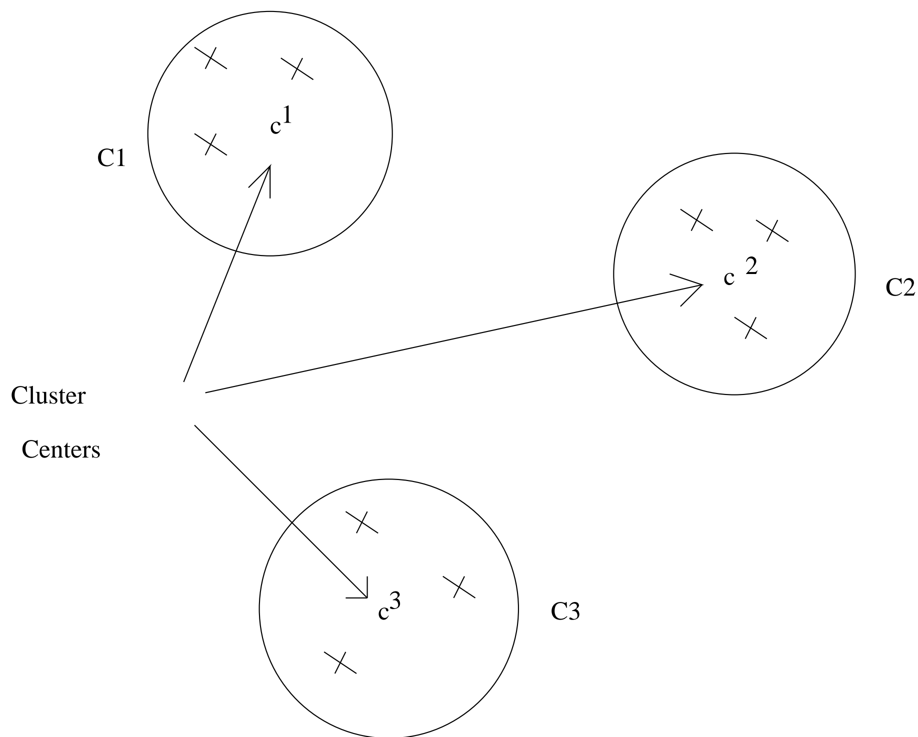
- Consider the previous function approximation problem :
- For pattern x^i, the approximation error : $E_i = \|H(W, x^i) - h(x^i)\|^2$
- Note that for $h(x^i)$, we may also use **desired output** d^i
- Average error made over the training set :
- $E(W) = \frac{1}{N} \Sigma_{i=1}^{N} E_i = \frac{1}{N} \Sigma_{i=1}^{N} \|H(W, x^i) - h(x^i)\|^2$
- So the problem is now : Find a weight W_* such that :
- $E(W_*) \leq E(W)$ for any $W \neq W_*$
- How to update the weights? $\rightarrow$ gradient descent ....
- $E(W + \Delta W) = E(W) + \frac{\partial E}{\partial W} \Delta W +$ h.o.t.
- Choose $\Delta W = -\eta \frac{\partial E}{\partial W}$
- $E(W + \Delta W) = E(W) - \eta \|\frac{\partial E}{\partial W}\|^2 +$ h.o.t.
- If we neglect the higher order terms : $\rightarrow E(W + \Delta W) \leq E(W)$
- This process will stop when $\frac{\partial E}{\partial W} = 0 \rightarrow$ **local minimum**.

• If $\eta > 0$ is sufficiently small, and if initial weight $W(1)$ is sufficiently close to a local minimum W_*, the above update law will always converge to the local minimum.

• This idea is the basic principle behind many optimization techniques, and also for the back propagation algorithm.

• There may be many local minimums. Global minimum?

• **Unsupervised Learning** This is also called self organization. A training set $\mathcal{T}_r$ is also given.But for each pattern, we do not know the **correct output**. The NN is required to organize itself, by possibly detecting the similarities between the patterns (statistical properties) and arranges its outputs accordingly. A typical case is **clustering**, which may be considered as a form of **classification**.



• Here we have inputs, and assume that they form separable clusters. Usually each cluster $\mathcal{C}_i$ will have a **cluster center** c^i

• If we know the cluster centers, we can distinguish the clusters by distance

• $\|x^s - c^i\| < \|x^s - c^j\|$ for any $j \quad \Leftrightarrow \quad x^s \in \mathcal{C}_i$

• Hence one way of looking at unsupervised learning is to find appropriate **cluster centers** for each clusters, hence detect the clusters in the given training set.

• Here, c^i are also called **quantization vectors**.

• Unsupervised learning tries to minimize **quantization error** by finding appropriate cluster centers :

• Suppose we have m clusters, and total of N patterns.

• $E = \sum_{i=1}^{N} \| x^i - \text{argmin}_{c^j} \; \|x^i - c^j\| \; \|$

• E is the total quantization error. Note that E depends on the chosen cluster centers $c^1, \ldots, c^m$, i.e. $E(c^1, \ldots, c^m)$.

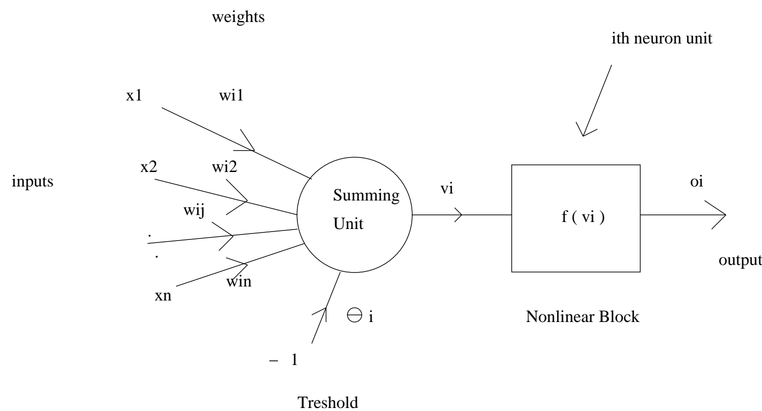
• E The problem is to find appropriate cluster centers such that E is minimized

• $$\min_{\{c^j\}} E$$

- **General Form of Learning rules**

- Consider a general neuron unit, say neuron i. Let us we are at the nth step of training and the last weight is $W_i(n)$, the next training input is x, and the corresponding desired output is d (if available). (We may also use extended notation).

- Consider the ith neuron as given below



- A typical update law which gives the next step is :

- $W_i(n+1) = W_i(n) + \Delta W_i(n), \qquad \Delta W_i(n) = crx$

- $c > 0$ is a learning constant.
- $r = r(W_i(n), d, x)$ is a **scalar** learning signal.

- Obviously, the desired output d only exists in supervised learning. It does not exist in the unsupervised case.

- Or componentwise :

- $W_{ij}(n+1) = W_{ij}(n) + \Delta W_{ij}(n), \qquad \Delta W_{ij}(n) = crx_j$

- **Hebbian Learning Rule**

- This learning rule was first proposed by D. Webb in 1940's. Hebb's idea was (1949):

- *When an axon of a cell A is near enough to excite a cell B and repeatedly or persistently fires it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*

- Mathematical expression is the following :

- $\Delta W_i(n) = cg(o_i(n))x$

- Here $g : \mathbf{R} \to \mathbf{R}$ is an appropriate function, x is the current input, and $o(n)$ is the output of the neuron, i.e. $o(n) = f(v(n)), \quad v(n) = W_i^T(n)x$

- Since there is no desired output d, this is an unsupervised learning rule.

- Typically g is identity, i.e. we have :

- $\Delta W_i(n) = co_i(n)x$

- $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad W_i = \begin{pmatrix} w_{i1} \\ \vdots \\ w_{in} \end{pmatrix} \qquad x_e = \begin{pmatrix} x_1 \\ \vdots \\ x_n \\ -1 \end{pmatrix} \qquad W_{ie} = \begin{pmatrix} w_{i1} \\ \vdots \\ w_{in} \\ \theta_i \end{pmatrix}$

- Each weight is updated as follows : ($j = 1, 2, \ldots, n$)

- $\Delta W_{ij}(n) = co_i(n)x_j$

• As a result, if $o_i(n)x_j > 0$, the corresponding weight is increased, otherwise decreased. This might cause saturation problem for the weights. Sometimes to avoid it, the following modification may be used :

• $\Delta W_{ij}(n) = co_i(n)x_j - \alpha W_{ij}(n) \qquad \alpha > 0$

• This is called generalized activity product rule. This prevents $W_{ij}(n)$ become very large.

• $\mathcal{C}_1 = \{x^1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} , \; x^2 = \begin{pmatrix} 0.9 \\ 1 \end{pmatrix} , \; x^3 = \begin{pmatrix} 1 \\ 1.1 \end{pmatrix}\}$

• $\mathcal{C}_2 = \{x^4 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} , \; x^5 = \begin{pmatrix} 1.1 \\ -1 \end{pmatrix} , \; x^6 = \begin{pmatrix} 1 \\ -1.1 \end{pmatrix}\}$

• **Question :** Find a single neuron which distinguishes these classes.

• **Problem :** Given a pattern x, we do not know to which class it belongs. Hence, we do not have **correct output** information. If we had, we could used perceptron training algorithm to solve this problem.

• Let us choose two weights and no threshold $\Rightarrow o = sgn(w_1x_1 + w_2x_2)$.

• Let us choose $c = 1$ and initial weight $W(1) = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}$

• $x^1 \rightarrow o = sgn(0.1 + 0.1) = 1$

• $x^5 \rightarrow o = sgn(0.11 - 0.1) = 1$

• Hence initial weights do not distinguish two classes.

- **begin epoch 1 :**
- $x^1 \to o(1) = sgn(0.1 + 0.1) = 1 \to W(2) = W(1) + 1 \times 1 \times x^1 = \begin{pmatrix} 1.1 \\ 1.1 \end{pmatrix}$
- $x^2 \to o(2) = sgn(W(2)^T x^2) = 1 \to W(3) = W(2) + 1 \times 1 \times x^2 = \begin{pmatrix} 2 \\ 2.1 \end{pmatrix}$
- $x^3 \to o(3) = sgn(W(3)^T x^3) = 1 \to W(4) = W(3) + 1 \times 1 \times x^3 = \begin{pmatrix} 3 \\ 3.2 \end{pmatrix}$
- $x^4 \to o(4) = sgn(W(4)^T x^4) = -1 \to W(5) = W(4) - 1 \times 1 \times x^4 = \begin{pmatrix} 2 \\ 4.2 \end{pmatrix}$
- $x^5 \to o(5) = sgn(W(5)^T x^5) = -1 \to W(6) = W(5) - x^5 = \begin{pmatrix} 0.9 \\ 5.2 \end{pmatrix}$
- $x^6 \to o(6) = sgn(W(6)^T x^6) = -1 \to W(7) = W(6) - x^6 = \begin{pmatrix} -0.1 \\ 6.3 \end{pmatrix}$
- **end of epoch 1**
- We can keep on updating like this. Let us stop and call $W_* = W(7)$.
- $x^1 \to o = sgn(W_*^T x^1) = 1$
- $x^2 \to o = sgn(W_*^T x^2) = 1$
- $x^3 \to o = sgn(W_*^T x^3) = 1$
- $x^4 \to o = sgn(W_*^T x^4) = -1$
- $x^5 \to o = sgn(W_*^T x^5) = -1$
- $x^6 \to o = sgn(W_*^T x^6) = -1$

- $o = sgn(W_*^T x) = sgn(-0.1x_1 + 6.3x_2)$
- $o = \begin{cases} 1 & x \in \mathcal{C}_1 \\ -1 & x \in \mathcal{C}_2 \end{cases}$

- **Winner-Take-All Learning Rule**

- Also known as **competitive learning**

- Can be applied to a layer of neurons. Output neurons compete around themselves to be active.

- In this method, for a particular input vector x from the training set, is applied at the input of the layer. We have p total neurons, and hence compute p weighted sums $v_1, \ \ldots, \ v_p$ and p outputs $o_1 = f(v_1), \ \ldots, \ o_p = f(v_p)$. Note that v_m can be computed as an inner product :

- $v_m = w_m^T x$ where $m = 1, \ 2, \ \ldots, \ p$

- $x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad w_m = \begin{pmatrix} w_{m1} \\ w_{m2} \\ \vdots \\ w_{mn} \end{pmatrix} \qquad x_e = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ -1 \end{pmatrix} \qquad w_{me} = \begin{pmatrix} w_{m1} \\ w_{m2} \\ \vdots \\ w_{mn} \\ \theta_m \end{pmatrix}$

- After this computation, we look at the neuron which has the maximum output. Since $f(\cdot)$ is nondecreasing, this is the same as maximum weighted sums v_i. Let us say that the maximum output occurs as mth neuron, i.e.

- $v_m = w_m^T x = \max_{(1 \leq j \leq p)} \quad v_j = w_j^T x \longrightarrow o_m = \max_{(1 \leq j \leq p)} \quad o_j$

- Then mth neuron is declared as the **winning neuron** (wins the competition)

- Then we update only the weights of the winning neuron as follows :

- $w_m \longleftarrow w_m + \Delta w_m$, $\qquad \Delta w_m = \alpha(x - w_m)$

- Here $\alpha > 0$ is the learning constant, and is usually decreased as the learning evolves.

- Componentwise we have :

- $w_{mj} \longleftarrow w_{mj} + \Delta w_{mj}$, $\qquad \Delta w_{mj} = \alpha(x_j - w_{mj})$

- Other neuron weights (i.e. the neurons losing the competition) are not changed.

- This learning rule has the effect of moving the winning neuron weight w_m towards the input vector x.

- $w_m^T x = \|w_m\| \; \|x\| \; \cos(w_m \, , \, x)$

- For the success of the algorithm, most of the time input vectors and weights are normalized.

- **Example** suppose that the same pattern x is repeatedly applied and at each time, the neuron m wins. Then we have :

- $w_m(2) = w_m(1) + \alpha(x - w_m(1)) = (1 - \alpha)w_m(1) + \alpha x$

- $w_m(3) = w_m(2) + \alpha(x - w_m(2)) = (1 - \alpha)^2 w_m(1) + \alpha[1 + (1 - \alpha)]x$

- $w_m(n+1) = w_m(n) + \alpha(x - w_m(n))$

$$= (1-\alpha)^n w_m(1) + \alpha[1 + (1-\alpha) + \ldots + (1-\alpha)^{n-1}]x$$

- For stability, we need $\quad | 1 - \alpha | < 1$

- $1 + (1-\alpha) + \ldots + (1-\alpha)^{n-1} = \frac{1-(1-\alpha)^n}{1-(1-\alpha)}$

- As $n \to \infty$, $\quad | 1 - \alpha | \to 0, \quad \alpha \, \frac{1-(1-\alpha)^n}{\alpha} \to 1$

- $w_m \to x$

- If we apply patterns from a cluster $\mathcal{C}$, we expect that the same neuron wins the competition, and its weight converges to the **cluster center** of $\mathcal{C}$

- **previous example**

- $\mathcal{C}_1 = \{x^1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \ x^2 = \begin{pmatrix} 0.9 \\ 1 \end{pmatrix}, \ x^3 = \begin{pmatrix} 1 \\ 1.1 \end{pmatrix}\}$

- $\mathcal{C}_2 = \{x^4 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \ x^5 = \begin{pmatrix} 1.1 \\ -1 \end{pmatrix}, \ x^6 = \begin{pmatrix} 1 \\ -1.1 \end{pmatrix}\}$

- **Question :** Find a two neuron single layer which distinguishes these classes.

- **Problem :** Given a pattern x, we do not know to which class it belongs. Hence, we do not have **correct output** information. If we had, we could used perceptron training algorithm to solve this problem.

- Let us choose two neurons and no threshold
- $\Rightarrow v_1 = (w_{11}x_1 + w_{12}x_2), \qquad v_2 = (w_{21}x_1 + w_{22}x_2)$
- choose $\alpha = 0.5$ and $W_1(1) = \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix} \qquad W_2(1) = \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix}$

- **begin epoch 1 :**

- $x^1 \rightarrow v_1(1) = (0.1 + 0.1) = 0.2, \;\; v_2(1) = 0.1 - 0.1 = 0$
- $\longrightarrow$ First neuron wins...
- $W_1(2) = W_1(1) + 0.5(x^1 - W_1(1)) = \begin{pmatrix} 0.55 \\ 0.55 \end{pmatrix} \quad , \quad W_2(2) = W_2(1)$

- $x^2 \rightarrow v_1(2) = (W_1(2)^T x^2) = 1.045, \;\; v_2(2) = (W_2(2)^T x^2) = -0.01$
- $\longrightarrow$ First neuron wins...
- $\rightarrow W_1(3) = W_1(2) + 0.5(x^2 - W_1(2)) = \begin{pmatrix} 0.725 \\ 0.775 \end{pmatrix} \quad , \quad W_2(3) = W_2(2)$

- $x^3 \rightarrow v_1(3) = (W_1(3)^T x^3) = 1.5775, \;\; v_2(3) = (W_2(3)^T x^3) = -0.01$
- $\longrightarrow$ First neuron wins...
- $\rightarrow W_1(4) = W_1(3) + 0.5(x^3 - W_1(3)) = \begin{pmatrix} 0.8625 \\ 0.9375 \end{pmatrix} \quad , \quad W_2(4) = W_2(3)$

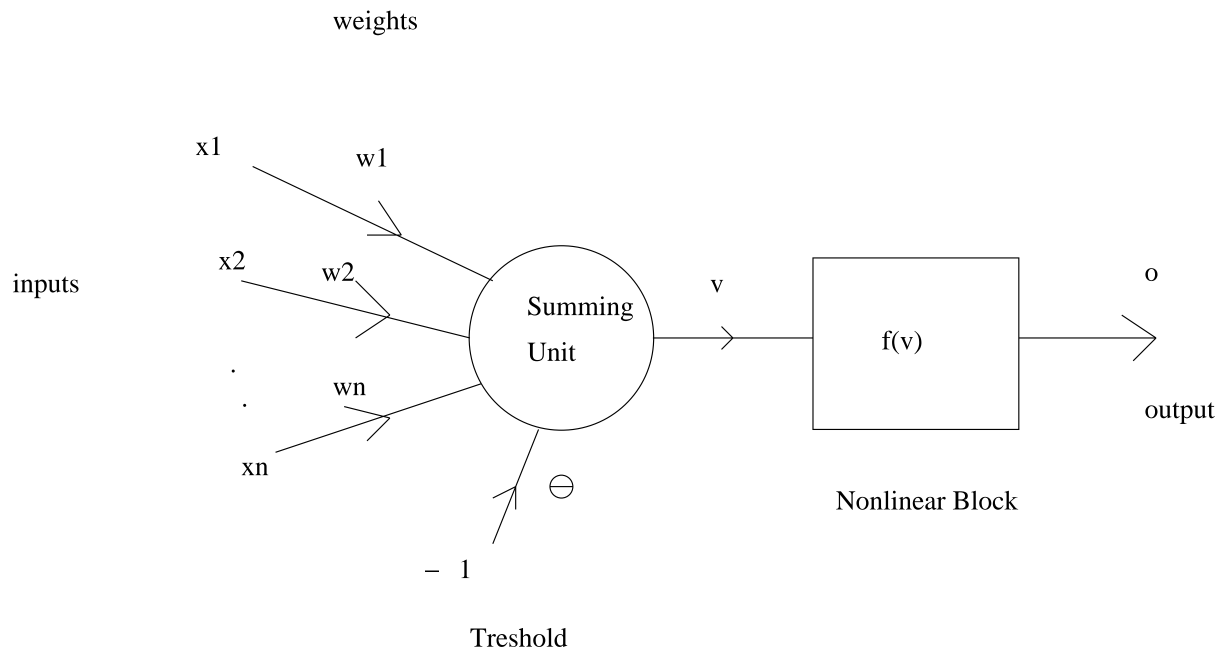
- $x^4 \rightarrow v_1(4) = (W_1(4)^T x^4) = -0.075, \;\; v_2(4) = W_2(4)^T x^4 = 0.2$
- $\longrightarrow$ Second neuron wins...
- $\rightarrow W_2(5) = W_2(4) + 0.5(x^4 - W_2(4)) = \begin{pmatrix} 0.55 \\ -0.55 \end{pmatrix} \quad , \quad W_1(5) = W_1(4)$

- $x^5 \rightarrow v_1(5) = (W_1(5)^T x^5) = 0.011, \;\; v_2(5) = W_2(5)^T x^5 = 1.155$
- $\longrightarrow$ Second neuron wins...

- $\rightarrow W_2(6) = W_2(5) + 0.5(x^5 - W_2(5)) = \begin{pmatrix} 0.825 \\ -0.775 \end{pmatrix}$, $W_1(6) = W_1(5)$
- $x^6 \rightarrow v_1(6) = (W_1(6)^T x^6) = -0.168, \quad v_2(6) = (W_2(6)^T x^6) = 1.6775$
- $\longrightarrow$ Second neuron wins...
- $\rightarrow W_2(7) = W_2(6) + 0.5(x^6 - W_2(6)) = \begin{pmatrix} 0.9125 \\ -0.9375 \end{pmatrix}$, $W_1(7) = W_1(6)$
- **end of epoch 1**

• Last weights separates two clusters. Moreover, these weights are close to the corresponding cluster centers.

- **Supervised learning rules**
- **perceptron** learning rule was one of such rules.
- **Delta** Learning Rule



- $v = w_1x_1 + w_2x_2 + \ldots + w_nx_n - \theta = w^Tx - \theta = w_e^Tx_e \rightarrow o = f(v)$
- $w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad w_e = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ \theta \end{pmatrix} \qquad x_e = y = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ -1 \end{pmatrix}$

- Assume that a training set input x is applied, and the **correct=desired** output is d. Then the update rule for weights is as follows :

- $\Delta W = crx = c[d-o]f'(v)x \longrightarrow r = [d-o]f'(v)$

- r is called the **learning signal**. Componentwise :

- $\Delta w_i = c[d-o]f'(v)x_i$

- **Fact :** This rule tries to minimize $e = 0.5(d-o)^2$.

- $o = f(v), \quad v = w_1x_1 + w_2x_2 + \ldots + w_nx_n - \theta \longrightarrow e = F(w_1, \ldots, w_n, \theta)$

- **Proof :** Remember the gradient descent idea : update the weights in the negative gradient direction :

- $\Delta w_i = -\eta \, \frac{\partial e}{\partial w_i}$

- By Leibnitz Rule : $\frac{\partial e}{\partial w_i} = \frac{\partial e}{\partial o} \, \frac{\partial o}{\partial v} \, \frac{\partial v}{\partial w_i}$

- $\frac{\partial e}{\partial o} = -(d-o)$

- $o = f(v) \longrightarrow \frac{\partial o}{\partial v} = f'(v)$

- $v = w_1x_1 + w_2x_2 + \ldots + w_nx_n - \theta = w^Tx - \theta = w_e^Tx_e \longrightarrow \frac{\partial v}{\partial w_i} = x_i$

- $\Delta w_i = -\eta \, \frac{\partial e}{\partial w_i} = -\eta \, \frac{\partial e}{\partial o} \, \frac{\partial o}{\partial v} \, \frac{\partial v}{\partial w_i} = \eta[d-o]f'(v)x_i$

- $\Delta W = -\eta \nabla e = -\eta \frac{\partial e}{\partial W} \longrightarrow \nabla e = \begin{pmatrix} \frac{\partial e}{\partial w_1} \\ \vdots \\ \frac{\partial e}{\partial w_n} \end{pmatrix}$ **error gradient**

- Computation of $f'(v)$ may be a problem. But if f is of sigmoidal type, then a simplification is possible :

- Bipolar sigmoidal :

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

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

- $1 - f^2(v) = 1 - \frac{(1-e^{-\lambda v})^2}{(1+e^{-\lambda v})^2} = \frac{(1+e^{-\lambda v})^2-(1-e^{-\lambda v})^2}{(1+e^{-\lambda v})^2} = \frac{4e^{-\lambda v}}{(1+e^{-\lambda v})^2}$

- $f'(v) = \frac{\lambda}{2}\ (1 - f^2(v)) = \frac{\lambda}{2}\ (1 - o^2)$

- Unipolar sigmoidal :

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

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

- $1 - f(v) = (1 - o) = \frac{e^{-\lambda v}}{1+e^{-\lambda v}}$

- $f'(v) = \lambda o(1 - o)$

- In both cases, the update rule is simplified :

- $\Delta W = crx = c[d - o]f'(v)x \longrightarrow r = [d - o]f'(v)$

- r is called the **learning signal**. Componentwise :

- $\Delta w_i = c[d - o]f'(v)x_i$

• Usually we have a training set set $\mathcal{S}_{Tr} = \{x^1, x^2, \ldots x^N\}$ and we have a set of desired outputs $d^1, \ldots, d^N$.

• For a fixed weight vector w, the error made at ith pattern e_i can be defined as $e_i = 0.5(d^i - o(i))^2$. Here $o(i) = f(w^T x^i)$. The cumulative and average errors made over the training set are :

• $E_{cum} = \Sigma_{i=1}^N e_i$ $\qquad$ $E_{ave} = \frac{1}{N} E_{cum}$

• From pure theoretical point of view, we should minimize E_{cum} or E_{ave}. Hence we should update the weight w by :

• $\Delta w = -\eta \frac{\partial E_{cum}}{\partial w} = -\eta \Sigma_{i=1}^N \frac{\partial e_i}{\partial w}$

• This could be achieved if we use **batch mode** of learning, i.e. :

• initialize weight randomly by w,

• apply a pattern in the training set, say x^1, compute the corresponding error gradient, say $\frac{\partial e_1}{\partial w}$, but **do not update** w.

• apply another pattern in the training set, say x^2, compute the corresponding error gradient, say $\frac{\partial e_2}{\partial w}$, but **do not update** w.

• repeat this till all of the patterns are used. At the end of the epoch, sum all of these error gradients, and find $\Sigma_{i=1}^N \frac{\partial e_i}{\partial w}$

• Now update the weight as $\qquad w \leftarrow w - \eta \Sigma_{i=1}^N \frac{\partial e_i}{\partial w}$

• Start a new epoch with the updated weight.

• This method is the correct way of applying the error gradient method and minimizes E_{cum}

• For practical reasons, we do not use the batch mode, and use **incremental learning** rule as a modification :

• initialize weight randomly by w,

• apply a pattern in the training set, say x^1, compute the corresponding error gradient, say $\frac{\partial e_1}{\partial w}$, update the weights as $w \leftarrow w - \eta \, \frac{\partial e_1}{\partial w}$

• apply another pattern in the training set, say x^2, use the last updated weight, compute the corresponding error gradient, say $\frac{\partial e_2}{\partial w}$, update the weights as $w \leftarrow w - \eta \frac{\partial e_2}{\partial w}$

• repeat this till all of the patterns are used.

• Usually, if $\eta > 0$ is sufficiently small, both methods yield similar results. The incremental version is used in **backpropagation** algorithm.

• **Widrow-Hoff Rule**

• Here we have the following rule :

• Assume that a training set input x is applied, and the **correct=desired** output is d. Then the update rule for weights is as follows :

• $\Delta W = crx = c[d - v]x \longrightarrow r = [d - v]$

• r is called the **learning signal**. Componentwise :

• $\Delta w_i = c[d - v]x_i$

• **Fact :** This rule tries to minimize $e = 0.5(d - v)^2$ and hence it is independent of f. Sometimes this rule is called **Least Mean Square Rule**. It can be considered as a special case of Delta Rule when $f(v) = v$.