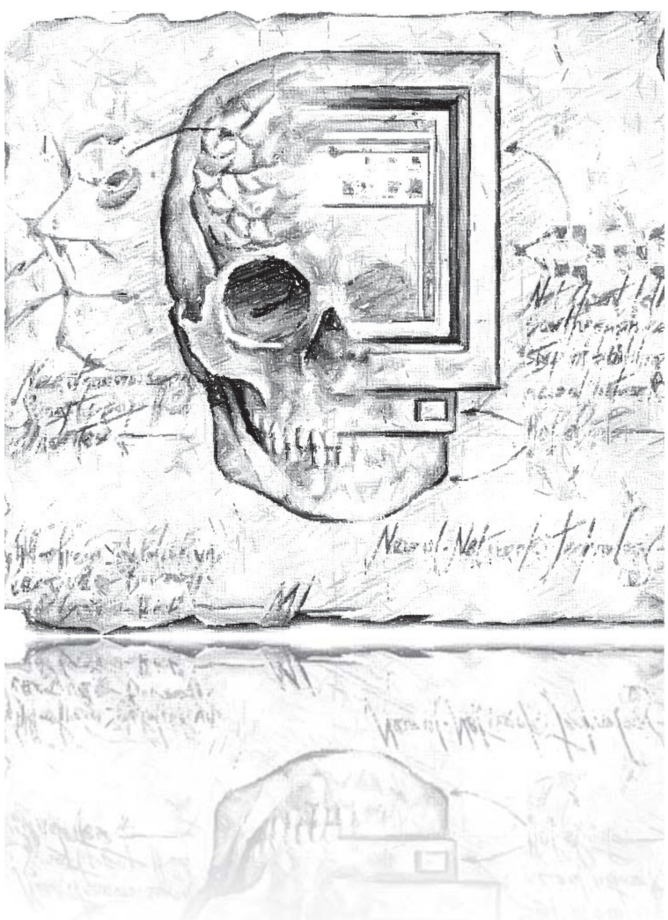
# Computational Intelligence I: Neural Networks

# Chapter 4: Single Layer Perceptrons



# Adaptive Filtering

⌘ Assume we have a dynamical system which can generate data at discrete time instances. Dataset D is composed of input-output samples:

$$D = \{(\mathbf{x}(i), d(i))\}, \quad i = 1, 2, \ldots \quad \text{where}$$

$$\mathbf{x}(i) = \left(x_1(i), x_2(i), \ldots, x_p(i)\right)^T \in \Re^p, \quad d(i) \in \Re$$

⌘ There are two ways to perceive the stimulus **x**(i), which is applied across all p nodes of the system to produce d(i):

- ⊚ **Spatial**: The p components correspond to different areas in space.
- ⊚ **Temporal**: The p components represent one present and p-1 past values.

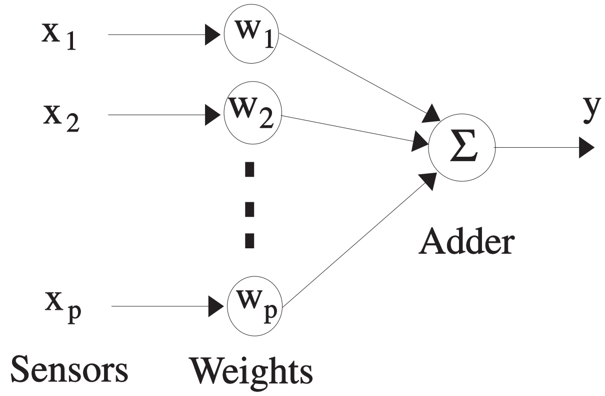
⌘ Adaptive Filtering problem (also System Identification):

- ⊚ **Filtering**: Given a stimulus **x**(i), a response y(i) is produced first, and then an error e(i)=d(i)-y(i) is calculated from the response comparison with the ideal signal d(i).
- ⊚ **Adaptation**: The synaptic weights w are adjusted accordingly.

⌘ A <u>linear neuron</u> is employed here, i.e. y(i)= ϕ(u(i))= u(i)= **ILF**:

$$y(i) = \mathbf{x}(i)^T \mathbf{w}(i), \quad \text{where}$$

$$\mathbf{w}(i) = \left(w_1(i), w_2(i), \ldots, w_p(i)\right)^T$$



## ⌘ **Linear-Least-Squares** (**LLS**) filtering: analysis & derivation.

- Optimisation theory is used to produce the learning algorithm. This utilises the errors **e**(n) till n[th] instance in a cost function which identifies the filter's parameters (adjusts synaptic weights).

$$\underbrace{\begin{bmatrix} e(1) \\ e(2) \\ \vdots \\ e(n) \end{bmatrix}}_{\mathbf{e}(n)} = \underbrace{\begin{bmatrix} d(1) \\ d(2) \\ \vdots \\ d(n) \end{bmatrix}}_{\mathbf{d}(n)} - \underbrace{\begin{bmatrix} \mathbf{x}(1)^T \\ \mathbf{x}(2)^T \\ \vdots \\ \mathbf{x}(n)^T \end{bmatrix}}_{\mathbf{X}(n)} \mathbf{w}(n) \quad \text{or:} \quad \boxed{\mathbf{e}(n) = \mathbf{d}(n) - \mathbf{X}(n)\mathbf{w}(n)}$$

- We now define the least squares error function:

$$E(\mathbf{w}) \equiv E(\mathbf{w}; n, D) = \frac{1}{2}\sum_{i=1}^{n} e(i)^2 = \frac{1}{2}\mathbf{e}(n)^T \mathbf{e}(n) = \frac{1}{2}\left\|\mathbf{e}(n)\right\|_2^2$$

- In this case, the optimisation will produce the optimal weights:

$$\mathbf{w}(n+1) = \underset{\mathbf{w}\in\Re^p}{\operatorname{argmin}} E(\mathbf{w})$$

- Since the cost function is expressed as a sum of least squares, one way of doing this is to employ a Gauss-Newton method, where weights are updated via:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \left( \underbrace{\mathbf{J}(n)^T \mathbf{J}(n)}_{\sim\nabla^2 E(\mathbf{w})} + \underbrace{\delta I_{p\times p}}_{\text{regulariser}} \right)^{-1} \underbrace{\mathbf{J}(n)^T \mathbf{e}(n)}_{\nabla E(\mathbf{w}) = \sum_i \mathbf{e}(i)\nabla \mathbf{e}(i)}$$

$$\mathbf{J}(n) = \begin{bmatrix} \frac{\partial e(1)}{\partial w_1} & \cdots & \frac{\partial e(1)}{\partial w_p} \\ \vdots & \cdots & \vdots \\ \frac{\partial e(n)}{\partial w_1} & \cdots & \frac{\partial e(n)}{\partial w_p} \end{bmatrix}_{\mathbf{w}=\mathbf{w}(n)} \qquad \text{(the } n\times p \text{ Jacobian matrix of } \mathbf{e}(n)\text{ )}$$

## ⌘ Continued…

- Nevertheless, in our case, the Gauss-Newton method allows the learning algorithm to converge in a single iteration. This is because of the linear relationship between weights and error.
- It can be seen that the Jacobian **J**(n) is -**X**(n); thus, we have:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \left(\mathbf{X}(n)^T \mathbf{X}(n)\right)^{-1} \mathbf{X}(n)^T \left(\underbrace{\mathbf{d}(n) - \mathbf{X}(n)\mathbf{w}(n)}_{\mathbf{e}(n)}\right) =$$

$$= \left(\mathbf{X}(n)^T \mathbf{X}(n)\right)^{-1} \mathbf{X}(n)^T \mathbf{d}(n)$$

- Or, shortly, through the notation of **pseudo-inverse**:

$$\boxed{\mathbf{w}(n+1) = \mathbf{X}(n)^{+} \mathbf{d}(n)}$$

- The above solves the Linear Least-Squares (LLS) problem, over the observation interval of n samples.

## ⌘ **Least-Mean-Square** algorithm (**LMS**): analysis & derivation.

- This learning algorithm is based on instantaneous cost function values (just sample-by-sample learning; not entire past history in one go):

$$E(n) = \frac{1}{2} e(n)^2$$

- Similar to our previous analysis, we have:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = e(n) \frac{\partial e(n)}{\partial \mathbf{w}} \quad \overset{e(n)=d(n)-\mathbf{x}(n)^T \mathbf{w}(n)}{\Longrightarrow} \quad \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}(n)} = -e(n)\mathbf{x}(n)$$

- Therefore, using a simple steepest descent update, the LMS update (also stochastic gradient algorithm) is given as:

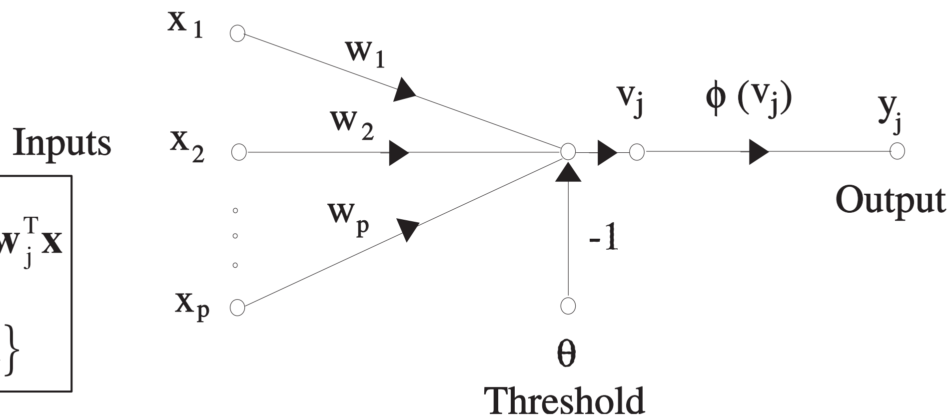
$$\boxed{\mathbf{w}(n+1) = \mathbf{w}(n) + \eta e(n)\mathbf{x}(n)}$$

- The user-defined parameter η is the <u>learning rate</u>, which regulates the strength of the low-pass filter achieved by the feedback around the weight vector **w**(n).

- The inverse of this parameter is a measure of the LMS algorithm <u>memory</u>, since lower values allow for a smoother filtering action where more past data are ‘remembered’.

# Perceptrons

⌘ Rosenblatt's Perceptrons are based on the McCulloch-Pitts neuronal model. The are similar to LMS, but they have <u>nonlinear activations</u> based on thresholding activations.

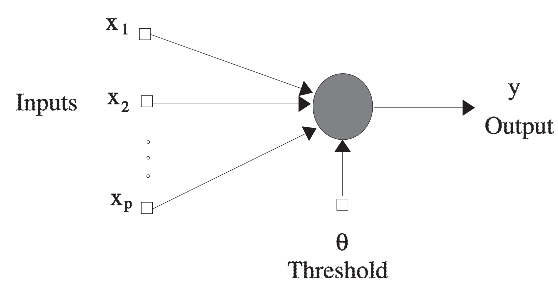



$$v_j = \sum_{i=1}^{p} w_{ji} x_i - \theta_j = \mathbf{w}_j^T \mathbf{x}$$

$$y_j = \phi(v_j) \in \{-1, +1\}$$

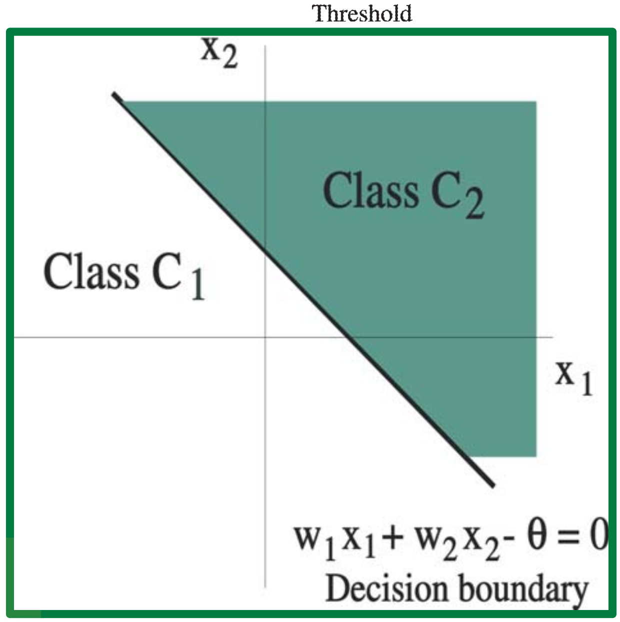
$$\text{e.g.}, \phi(v_j) = \text{signum}(v_j) = \begin{cases} +1 & \text{if } v_j > 0 \\ -1 & \text{if } v_j < 0 \end{cases}$$

⌘ The goal of the Perceptron is to correctly classify a set of external stimuli to one of two given classes $C_1$ or $C_2$.




- ◎ This means that the internal representation is a <u>hyperplane</u> in the p-dimensional space which separates the space into two class coding regions.
- ◎ This hyperplane is the **decision boundary** (defined as $\sum_{i=1}^{p} w_i x_i = \theta$) of the system's intelligence.

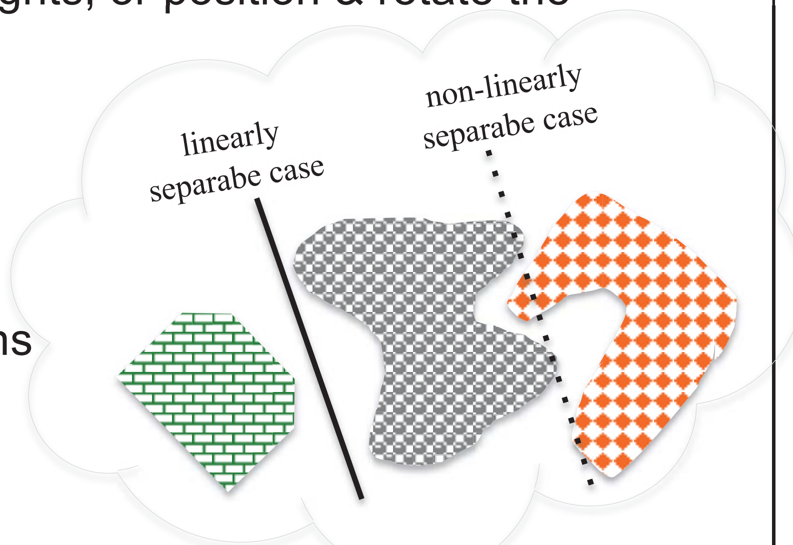
2D example



# Perceptron Training

- An error-correction learning algorithm is used to train the perceptron (to adjust weights, or position & rotate the decision boundary).

- Since the decision boundary is linear, only linearly separable patterns can be fully classified by the perceptron correctly.




- Assume that the dataset $D_{train}$ is split to two subsets $D_1$ and $D_2$, containing the samples from classes $C_1$ and $C_2$, respectively.
  - Then if we can find some vector **w** that satisfies both of the following inequalities, the samples in D are linearly separable.

$$\left\{ \begin{array}{ll} \mathbf{w}^T\mathbf{x} > 0 & \forall \mathbf{x} \in D_1 \\ \mathbf{w}^T\mathbf{x} \leq 0 & \forall \mathbf{x} \in D_2 \end{array} \right\} \quad \text{where: } D_1 \cap D_2 = \varnothing, D_1 \cup D_2 = D_{train}$$

The problem of training the Perceptron, now simply becomes a problem of finding one possible weight vector **w**, which satisfies the above inequalities.

## ⌘ The Perceptron Training Algorithm:

- ◎ Step 1 (Initialisation): Set t=1, **w**(1)=**0**, and learning rate η in (0,1]. (*Note: small* η yields averaging of past inputs and stable estimates, while large η yields fast learning and fast adjustment to environmental changes).

- ◎ Step 2 (Activation): Apply the sample input **x**(t) to the neuron.

- ◎ Step 3 (Response): Compute its response:

$$y(t) = \text{signum}\left[\mathbf{w}(t)^T \mathbf{x}(t)\right]$$

- ◎ Step 4 (Adaptation): Update the current weight vector via:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \underbrace{\left[d(t) - y(t)\right]}_{\text{error signal} \in \{0, \pm 2\}} \mathbf{x}(t)$$

$$d(t) = \begin{cases} +1 & \text{if } \mathbf{x}(t) \text{ belongs to } C_1 \\ -1 & \text{if } \mathbf{x}(t) \text{ belongs to } C_2 \end{cases}$$

- ◎ Step 5 (Continuation): Unless all samples are classified correctly, set t=t+1 and go to step 2.

⌘ The Perceptron Convergence Theorem guarantees that **if** the dataset is linearly separable, the learning rule **will** terminate.

- ◎ For example: assume that **x**(t) belongs to $C_1$ (i.e., d(t)=+1) but it is wrongly classified as $C_2$ (i.e., y(t)=-1). Then, we see that the ILF after setting **w**(t+1) becomes larger than before using **w**(t). This increases and 'pushes' the signum response towards the positive side, since:

$$\mathbf{w}(t+1)^T \mathbf{x}(t) = \left(\mathbf{w}(t)^T + 2\eta \cdot \mathbf{x}(t)^T\right)\mathbf{x}(t) = \mathbf{w}(t)^T \mathbf{x}(t) + 2\eta \cdot \left\|\mathbf{x}(t)\right\|_2^2$$

$$\Rightarrow$$

$$\mathbf{w}(t+1)^T \mathbf{x}(t) > \mathbf{w}(t)^T \mathbf{x}(t)$$

<END of Chapter 4>

# Extras: additional theory

(non examinable)

# Perceptron Convergence Theorem Proof:

⌘ We now give more details for the convergence of the aforementioned learning rule:

- Without loss of generality, assume η=½, and only consider the initial case **w**(1)=**0**. Assume that some inputs **x**(i) in $D_1$, i=1,2,…,n are incorrectly classified as $C_2$, i.e. $\mathbf{w}(i)^T\mathbf{x}(i) \le 0$
- Iteratively we get: $\mathbf{w}(n+1) = \mathbf{w}(1) + \mathbf{x}(1) + \cdots + \mathbf{x}(n) = \sum_{i=1}^{n} \mathbf{x}(i)$
- Since these points belong to $C_1$, there exist a vector **w*** which classifies all such points (i.e., positive inner product). We can now define a positive quantity: $\alpha = \min_{\mathbf{x}\in D_1} \mathbf{x}^T\mathbf{w}^*$
- Post-multiplying the previous equation with **w***, produces:

$$\mathbf{w}(n+1)^T \mathbf{w}^* = \sum_{i=1}^{n} \mathbf{x}(i)^T \mathbf{w}^* \ge n\alpha$$

- From C.B.S. inequality the get:

$$\left\|\mathbf{w}(n+1)\right\|_2^2 \cdot \left\|\mathbf{w}^*\right\|_2^2 \ge n^2\alpha^2 \Leftrightarrow \boxed{\left\|\mathbf{w}(n+1)\right\|_2^2 \ge \frac{n^2\alpha^2}{\left\|\mathbf{w}^*\right\|_2^2}}$$

1

- Now, we can also rewrite the first expression for i=1,…,n as:

$$\mathbf{w}(i+1) = \mathbf{w}(i) + \mathbf{x}(i) \quad \Leftrightarrow \quad \left\|\mathbf{w}(i+1)\right\|_2^2 = \left\|\mathbf{w}(i)\right\|_2^2 + \left\|\mathbf{x}(i)\right\|_2^2 + 2\mathbf{w}(i)^T\mathbf{x}(i)$$

- Using the original misclassification assumption, we simplify the above by dropping the negative terms and converting to inequality:

$$\left\|\mathbf{w}(i+1)\right\|_2^2 \le \left\|\mathbf{w}(i)\right\|_2^2 + \left\|\mathbf{x}(i)\right\|_2^2 \quad \text{or} \quad \left\|\mathbf{w}(i+1)\right\|_2^2 - \left\|\mathbf{w}(i)\right\|_2^2 \le \left\|\mathbf{x}(i)\right\|_2^2$$

## ⌘ Continued…

- Summing up the entire set of previous inequalities for all i=1,…n, and using the initial assumption that **w**(1)=0, we obtain:

$$\boxed{\left\|\mathbf{w}(n+1)\right\|_2^2 \le \sum_{i=1}^{n}\left\|\mathbf{x}(i)\right\|_2^2 \le n\beta} \quad (2)$$

$$\text{where } \beta = \max_{\mathbf{x}\in D_1}\left\|\mathbf{x}\right\|_2^2$$

- **Conclusions:**

- Eq(2) shows that the norm of **w**(i) grows at most linearly (no faster than √β) with an upper bound.
- Eq(1) shows that the norm of **w**(i) is lower bounded by a quantity linear in n.
- Thus, since **w*** and the training samples are fixed, for large values of n, Eqs(1&2) will conflict.
- Therefore, n cannot grow indefinitely; the algorithm <u>will terminate</u> after a fixed number of $\le n_{max}$ iterations.
- $n_{max}$ is a sufficiently small value to guarantee compatibility in the two bounds:

$$n_{max}\beta = \frac{\alpha^2 n_{max}^2}{\left\|\mathbf{w}^*\right\|_2^2} \Rightarrow n_{max} = \frac{\beta\left\|\mathbf{w}^*\right\|_2^2}{\alpha^2}$$

- Remember: **w*** and $n_{max}$ are not unique!
- Also, a different value for the initial **w**(1), simply results in altering the number of iterations necessary for convergence.