DD2437 – Artificial Neural Networks and Deep Architectures (annda)

Introduction to Lecture 2a

Perceptron

Pawel Herman

Computational Science and Technology (CST)

KTH Royal Institute of Technology

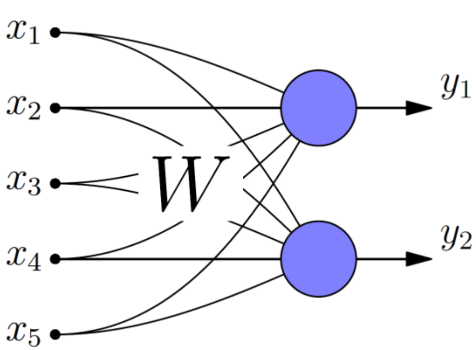
Outline

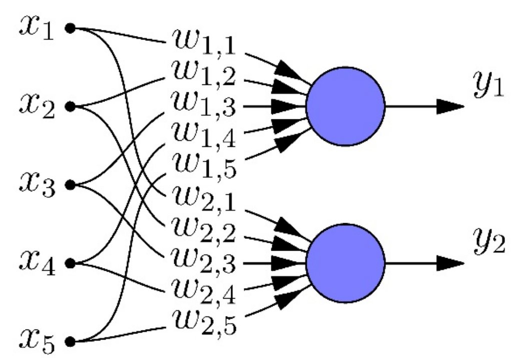
- Linear networks
- Hebbian learning and correlational memory
- Threshold logic unit (TLU)
- Perceptron vs delta rule learning

Linear networks

First, let's adopt a specific convention

W - weight matrix



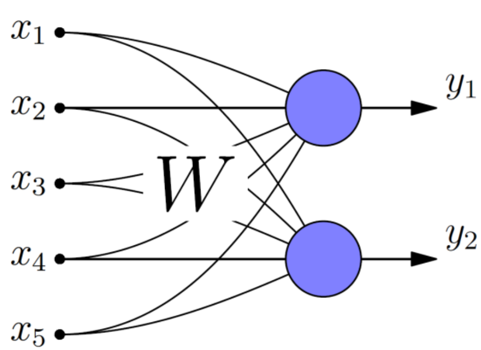


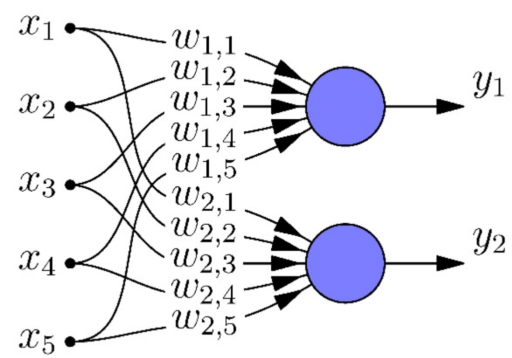
$$\mathrm{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{bmatrix}$$

Linear networks

First, let's adopt a specific convention

W - weight matrix





$\vec{w}_1^{\,\mathrm{T}}$

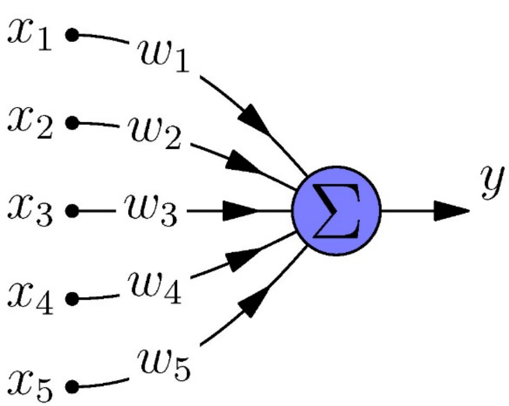
$$\mathrm{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{bmatrix}$$

$\vec{w}_2^{\,\mathrm{T}}$ - weight vector

Linear networks

First, let's adopt a specific convention

If there is a single output y, we just use a weight vector, $\vec{w}$: $\boxed{y = \vec{w}^{\mathrm{T}} \cdot \vec{x}}$

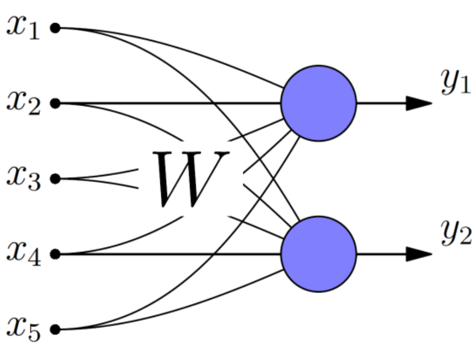


$$\mathrm{W} = \left[\overbrace{w_{11} \quad w_{12} \quad w_{13} \quad w_{14} \quad w_{15}}^{\vec{w}^{\mathrm{T}}}\right]$$

Linear networks

What can be computed?

W - weight matrix

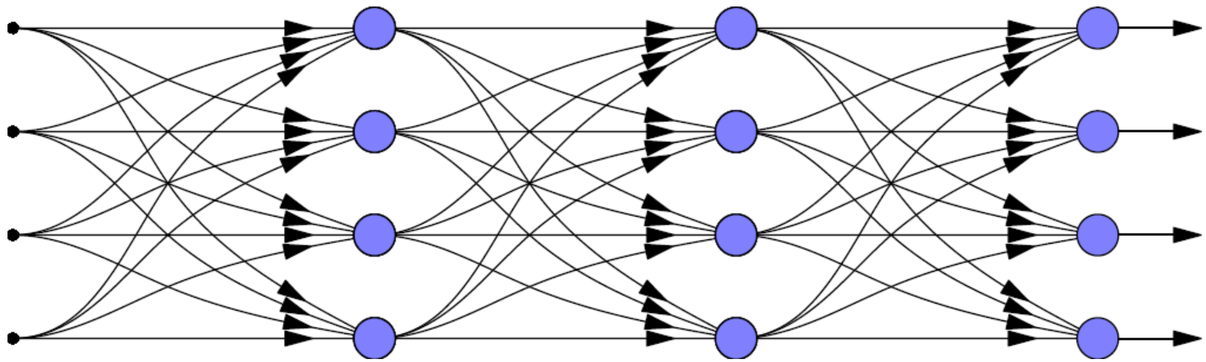


$$\mathrm{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \end{bmatrix}$$

$$y_1 = \vec{w}_1^{\mathrm{T}} \cdot \vec{x} \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \vec{w}_1^{\mathrm{T}} \\ \vec{w}_2^{\mathrm{T}} \end{bmatrix} \vec{x} = \mathrm{W}\,\vec{x}$$

Linear networks

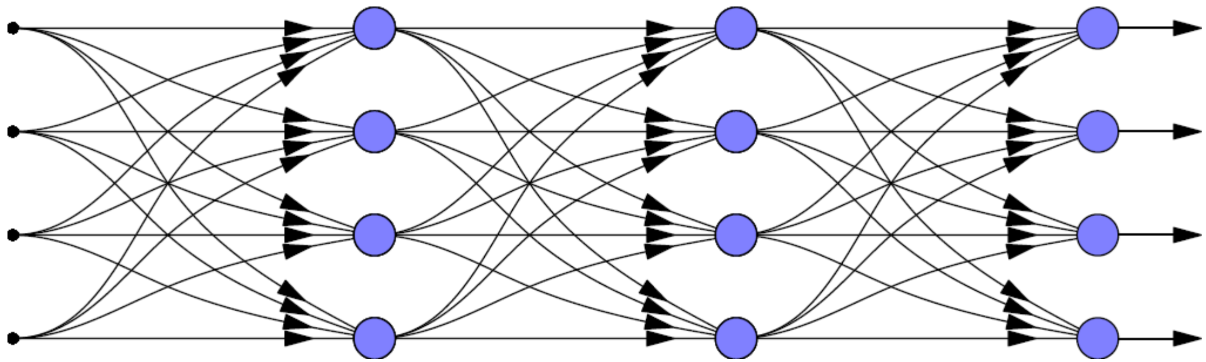
What happens when we concatenate several linear networks?



$$\vec{y} = \mathrm{W}_3\left(\mathrm{W}_2\left(\mathrm{W}_1\vec{x}\right)\right) = \left(\mathrm{W}_3\mathrm{W}_2\mathrm{W}_1\right)\vec{x}$$

Linear networks

What happens when we concatenate several linear networks?



$$\vec{y} = W_3\left(W_2\left(W_1\vec{x}\right)\right) = \left(W_3W_2W_1\right)\vec{x}$$

$$\text{Let } \; W = W_3W_2W_1 \quad \Rightarrow \quad \vec{y} = W\vec{x}$$

It is still a linear mapping !

Storing mappings (memorising)

The program “resides” in weights

But how do we find suitable weights?

Storing mappings (memorising)

The program "resides" in weights

But how do we find suitable weights?

- **Learning** corresponds to adapting weights, often *iteratively*, to achieve better performance

$$w^{(new)} = w^{(old)} + \Delta w_{ij}$$

Storing mappings (memorising)

The program "resides" in weights

But how do we find suitable weights?

Learning corresponds to adapting weights, often *iteratively*, to achieve better performance

Hebb's learning hypothesis

Simultaneous activation of two neurons strengthens their synaptic inter-connection

Storing mappings (memorising)

The program "resides" in weights

But how do we find suitable weights?

Learning corresponds to adapting weights, often *iteratively*, to achieve better performance

Hebb's learning hypothesis

Simultaneous activation of two neurons strengthens their synaptic inter-connection

Common interpretation:

$$\Delta w_{ij} = x_j y_i$$

Storing mappings (memorising)

The program "resides" in weights

But how do we find suitable weights?

Learning corresponds to adapting weights, often *iteratively*, to achieve better performance

Hebb's learning hypothesis

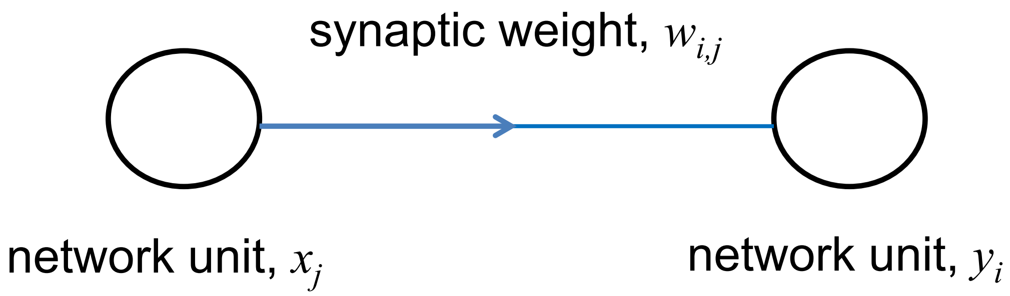
Simultaneous activation of two neurons strengthens their synaptic inter-connection

Common interpretation:

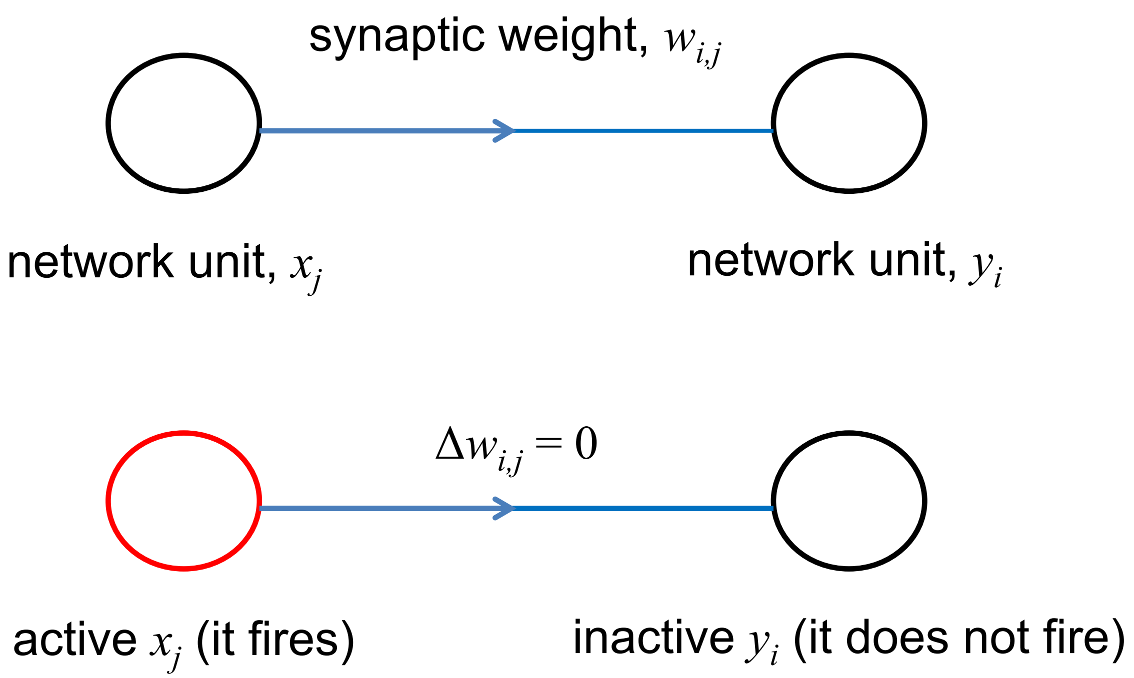
covariance rule

$$\Delta w_{ij} = x_j y_i \qquad \text{or } \ldots \; \Delta w_{ij} = (x_j - \bar{x})\,(y_i - \bar{y})$$

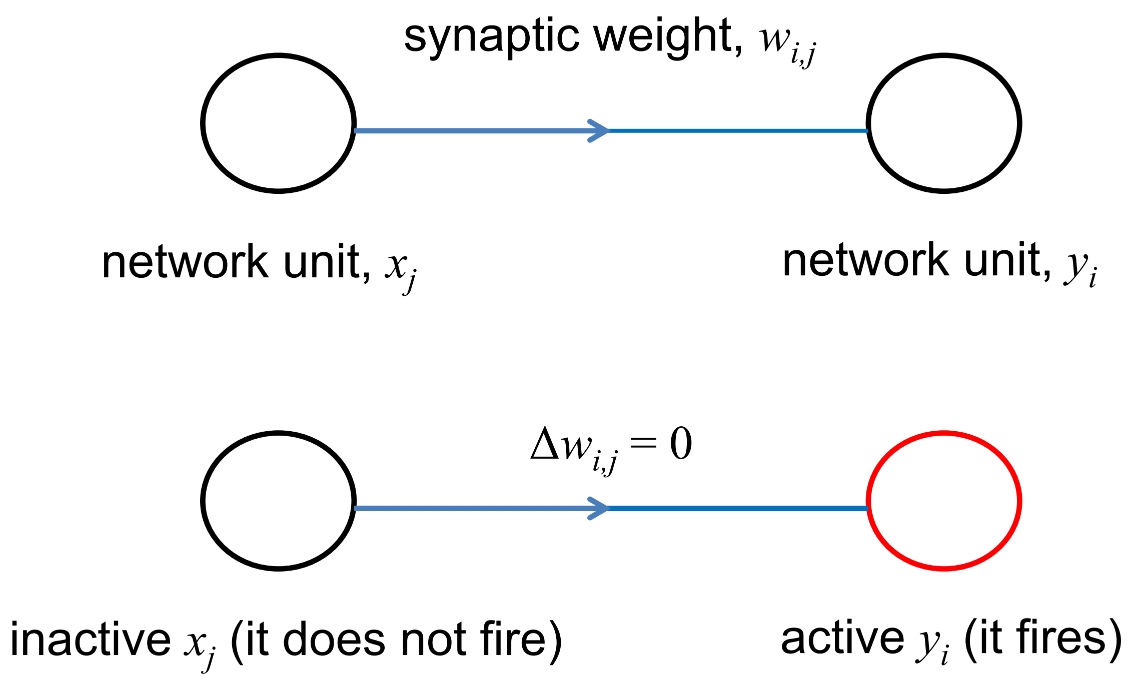
Hebbian learning rule



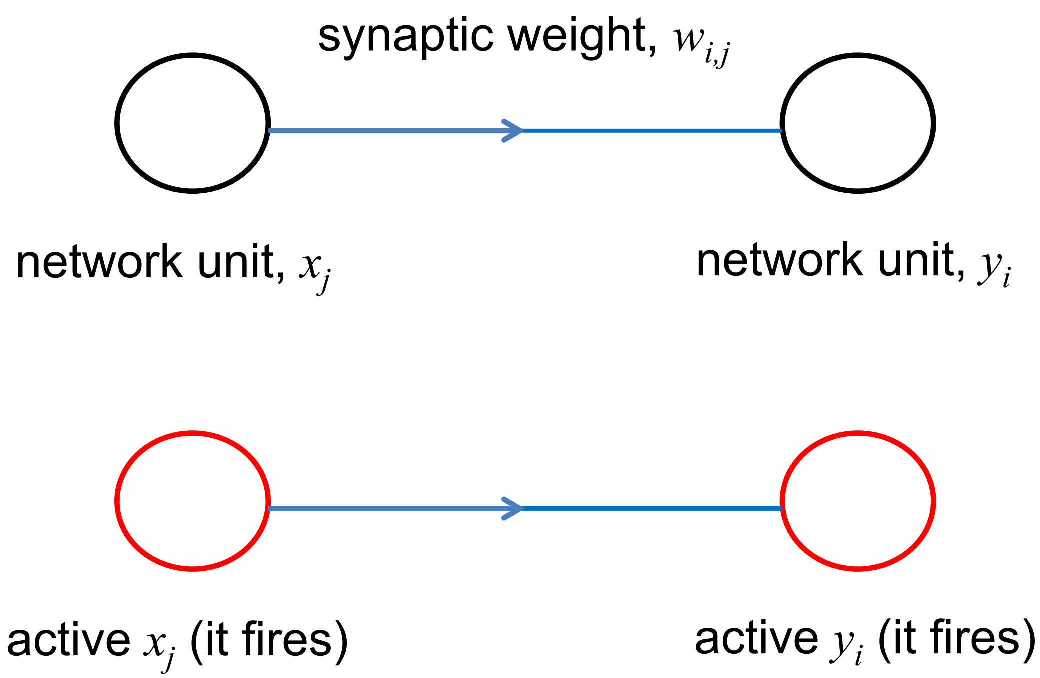
Hebbian learning rule



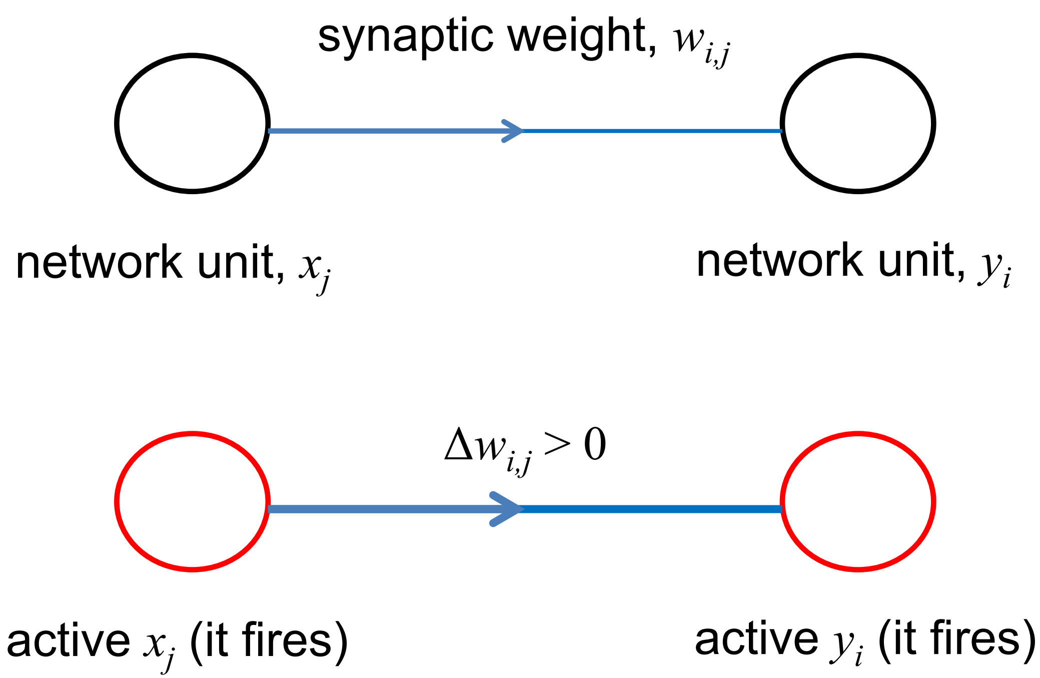
Hebbian learning rule



Hebbian learning rule



Hebbian learning rule



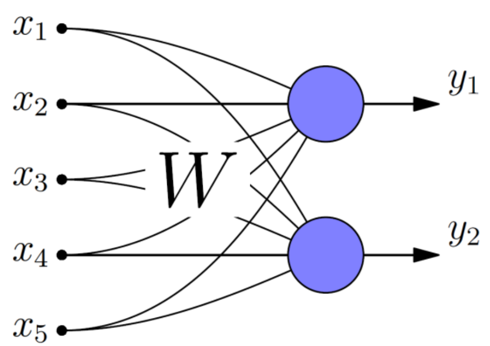
$$\Delta w_{i,j} = x_j \, y_i$$

"Fire together, wire together"

Storing a mapping using Hebb's rule

$$\vec{x}^{(1)} \rightarrow \vec{y}^{(1)} \qquad \vec{x}^{(2)} \rightarrow \vec{y}^{(2)} \qquad \vec{x}^{(3)} \rightarrow \vec{y}^{(3)} \quad \ldots \quad \vec{x}^{(n)} \rightarrow \vec{y}^{(n)}$$

$\vec{x}^{(k)}$



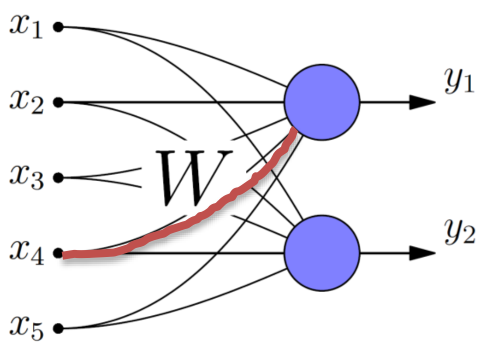
$\vec{y}^{(k)}$

Storing mappings (memorising)

Storing a mapping using Hebb's rule

$$\vec{x}^{(1)} \rightarrow \vec{y}^{(1)} \quad \vec{x}^{(2)} \rightarrow \vec{y}^{(2)} \quad \vec{x}^{(3)} \rightarrow \vec{y}^{(3)} \quad \ldots \quad \vec{x}^{(n)} \rightarrow \vec{y}^{(n)}$$

Hebb's rule $\Delta w_{ij} = x_j y_i$



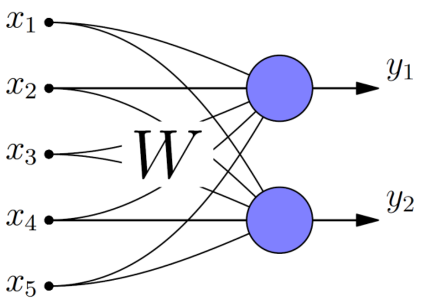
$$\Delta w_{1,4} = x_4 y_1$$

Storing mappings (memorising)

Storing a mapping using Hebb's rule

$$\vec{x}^{(1)} \rightarrow \vec{y}^{(1)} \qquad \vec{x}^{(2)} \rightarrow \vec{y}^{(2)} \qquad \vec{x}^{(3)} \rightarrow \vec{y}^{(3)} \quad \ldots \quad \vec{x}^{(n)} \rightarrow \vec{y}^{(n)}$$

Hebb's rule $\qquad \Delta w_{ij} = x_j y_i$



$$\Delta \mathrm{W} = \vec{y} \times \vec{x} = \vec{y}\,\vec{x}^{\mathrm{T}} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \begin{bmatrix} x_1, & x_2, x_3, & x_4, x_5 \end{bmatrix} =$$

$$\begin{bmatrix} x_1 y_1 & x_2 y_1 & x_3 y_1 & x_4 y_1 & x_5 y_1 \\ x_1 y_2 & x_2 y_2 & x_3 y_2 & x_4 y_2 & x_5 y_2 \end{bmatrix} = \begin{bmatrix} \Delta w_{11} & \Delta w_{12} & \Delta w_{13} & \Delta w_{14} & \Delta w_{15} \\ \Delta w_{21} & \Delta w_{22} & \Delta w_{23} & \Delta w_{24} & \Delta w_{25} \end{bmatrix}$$

Storing mappings (memorising)

Storing a mapping using Hebb's rule

$$\vec{x}^{(1)} \to \vec{y}^{(1)} \qquad \vec{x}^{(2)} \to \vec{y}^{(2)} \qquad \vec{x}^{(3)} \to \vec{y}^{(3)} \quad \ldots \quad \vec{x}^{(n)} \to \vec{y}^{(n)}$$

Hebb's rule $\Delta w_{ij} = x_j y_i$

Result

$$\mathrm{W} = \sum_{p=1}^{n} \vec{y}^{(p)} \, \vec{x}^{(p)^{\mathrm{T}}}$$

(outer product "x" of vector patterns)

$$\vec{y}^{(p)} \times \vec{x}^{(p)}$$

Correlational memory!

Storing mappings (memorising)

Retrieving a memory trace

$$\mathrm{W} = \sum_{p=1}^{n} \vec{y}^{(p)} \, \vec{x}^{(p)^{\mathrm{T}}}$$

$$\vec{x}^{(k)} \rightarrow \; ?$$

We expect to get $\vec{y}^{(k)}$

Storing mappings (memorising)

Retrieving a memory trace

$$\mathrm{W} = \sum_{p=1}^{n} \vec{y}^{(p)} \, \vec{x}^{(p)^{\mathrm{T}}}$$

$$\vec{x}^{(k)} \rightarrow \; ?$$

$$\vec{y}_{out} = \mathrm{W}\,\vec{x}^{(k)} = \sum_{p=1}^{n} \left(\vec{y}^{(p)} \vec{x}^{(p)^{\mathrm{T}}}\right) \vec{x}^{(k)} = \sum_{p=1}^{n} \vec{y}^{(p)} \left(\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(p)}\right)$$

Storing mappings (memorising)

Retrieving a memory trace

$$\mathrm{W} = \sum_{p=1}^{n} \vec{y}^{(p)} \, \vec{x}^{(p)^{\mathrm{T}}}$$

$$\vec{x}^{(k)} \rightarrow \; ?$$

$$\vec{y}_{out} = \mathrm{W}\,\vec{x}^{(k)} = \sum_{p=1}^{n} (\vec{y}^{(p)} \vec{x}^{(p)^{\mathrm{T}}})\,\vec{x}^{(k)} = \sum_{p=1}^{n} \vec{y}^{(p)} (\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(p)}) =$$

$$= \vec{y}^{(k)} \underbrace{(\vec{x}^{(k)^{\mathrm{T}}} \vec{x}^{(k)})}_{\alpha} + \sum_{p \neq k}^{n} \vec{y}^{(p)} (\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(k)})$$

Storing mappings (memorising)

Retrieving a memory trace

$$\mathrm{W} = \sum_{p=1}^{n} \vec{y}^{(p)} \cdot \vec{x}^{(p)^{\mathrm{T}}}$$

$$\vec{x}^{(k)} \rightarrow ?$$

$$\vec{y}_{out} = \mathrm{W}\,\vec{x}^{(k)} = \sum_{p=1}^{n} (\vec{y}^{(p)} \vec{x}^{(p)^{\mathrm{T}}})\,\vec{x}^{(k)} = \sum_{p=1}^{n} \vec{y}^{(p)} (\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(p)}) =$$

$$= \vec{y}^{(k)} (\underbrace{\vec{x}^{(k)^{\mathrm{T}}} \vec{x}^{(k)}}_{\alpha}) + \sum_{p \neq k}^{n} \vec{y}^{(p)} (\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(k)}) \approx \alpha \vec{y}^{(k)}$$

$$\approx \mathbf{0}$$

Storing mappings (memorising)

Retrieving a memory trace

$$\mathrm{W} = \sum_{p=1}^{n} \vec{y}^{(p)} \cdot \vec{x}^{(p)^{\mathrm{T}}}$$

$$\vec{x}^{(k)} \rightarrow \ ?$$

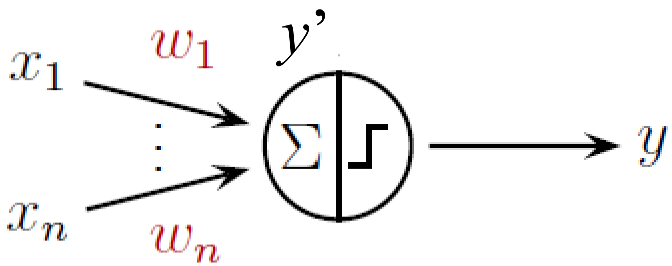
$$\vec{y}_{out} = \mathrm{W}\,\vec{x}^{(k)} = \sum_{p=1}^{n} (\vec{y}^{(p)} \vec{x}^{(p)^{\mathrm{T}}})\,\vec{x}^{(k)} = \sum_{p=1}^{n} \vec{y}^{(p)} (\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(p)}) =$$

$$= \vec{y}^{(k)} (\vec{x}^{(k)^{\mathrm{T}}} \vec{x}^{(k)}) + \sum_{p \neq k}^{n} \vec{y}^{(p)} (\vec{x}^{(p)^{\mathrm{T}}} \vec{x}^{(k)}) \approx \alpha \vec{y}^{(k)}$$

Perfect memory only if the patterns $\vec{x}^{(p)}$ are orthogonal

TLU – McCulloch Pitts

Threshold logic unit – McCulloch Pitts neuron (1942)

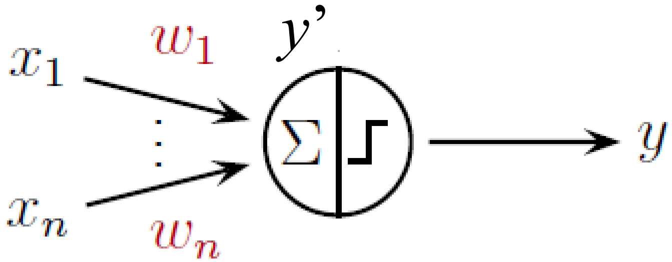


$y' = w_1 x_1 + w_2 x_2$ $\qquad$ $y = f_{step}(y')$

y': *before* thresholding $\qquad$ y: *after* thresholding

TLU – McCulloch Pitts

Threshold logic unit – McCulloch Pitts neuron (1942)



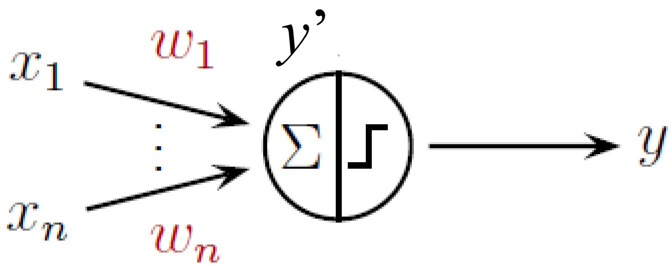
$$y' = w_1 x_1 + w_2 x_2 \qquad\qquad y = f_{step}(y')$$

In the simplest case: if threshold is **0**, then:

$$w_1 x_1 + w_2 x_2 > \mathbf{0} \rightarrow y' > \mathbf{0} \rightarrow y = 1$$
$$w_1 x_1 + w_2 x_2 <= \mathbf{0} \rightarrow y' <= \mathbf{0} \rightarrow y = 0$$

Geometrical interpretation

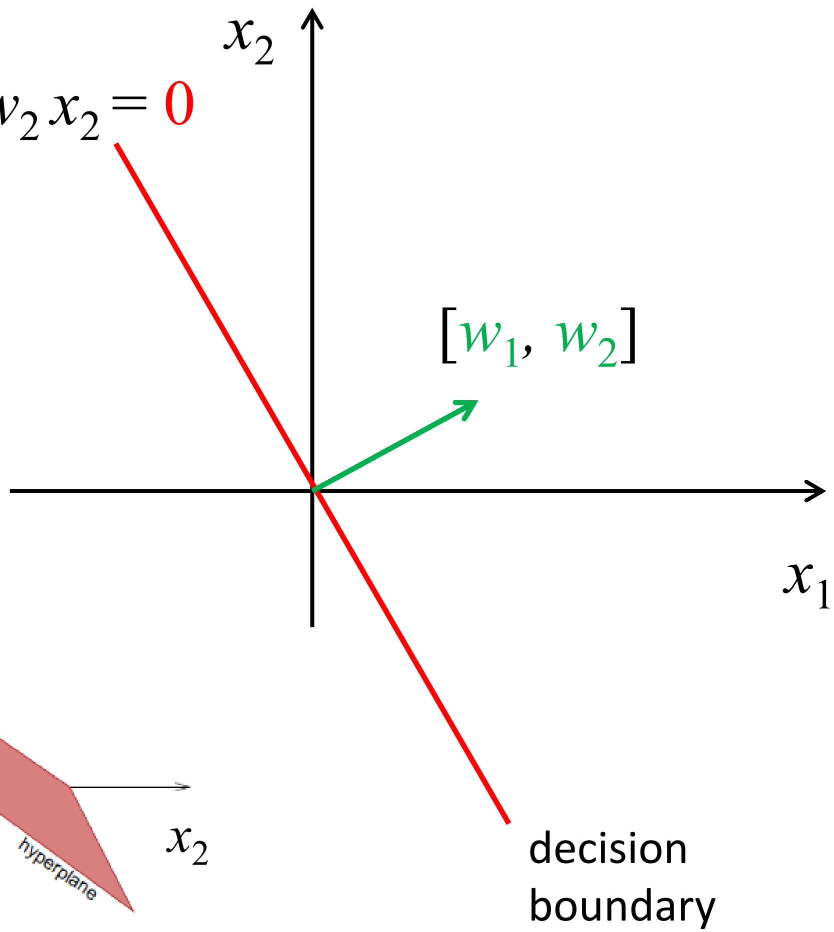
If threshold is **0**, then:

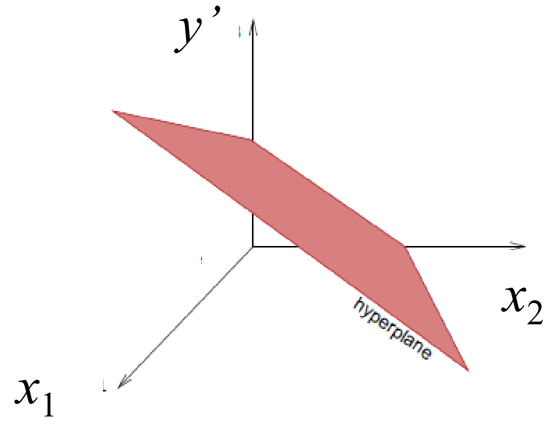


$$y' = w_1 x_1 + w_2 x_2$$

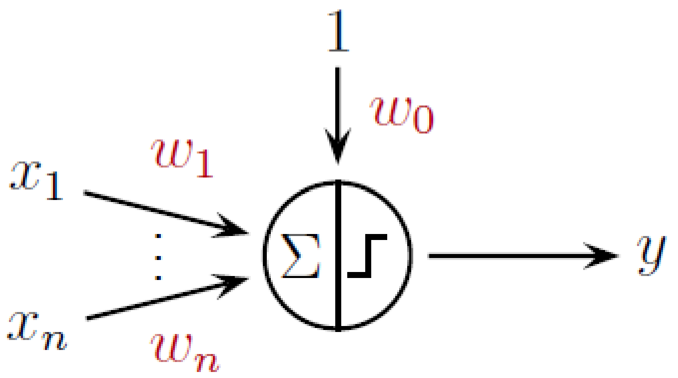
$$(a x + by + cz + d = 0)$$

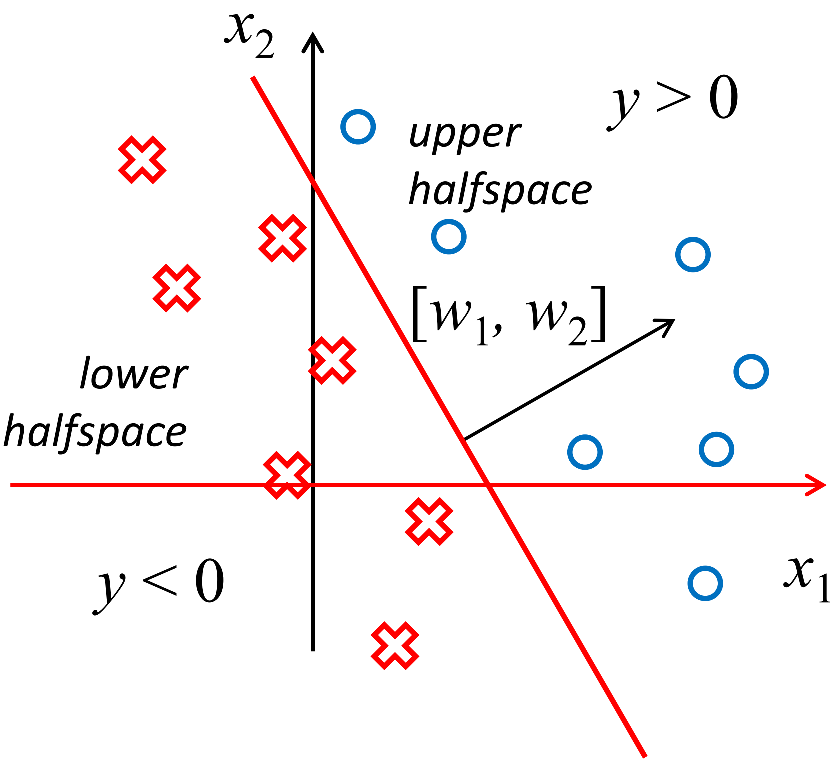
$$y' = w_1 x_1 + w_2 x_2 = 0$$





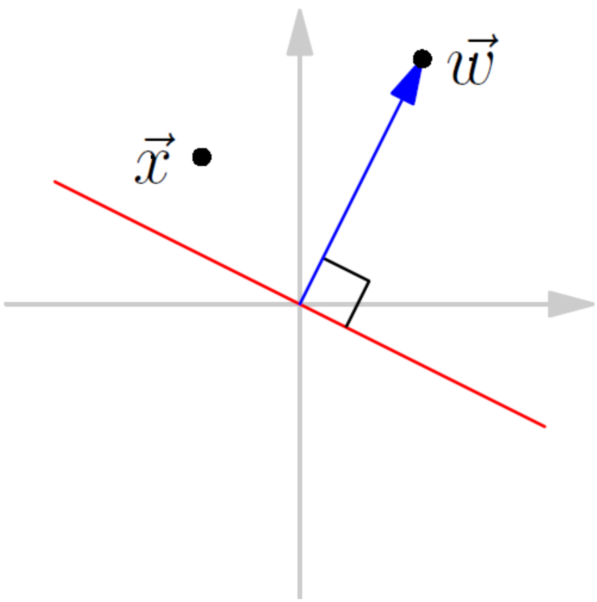
Binary classification with perceptron

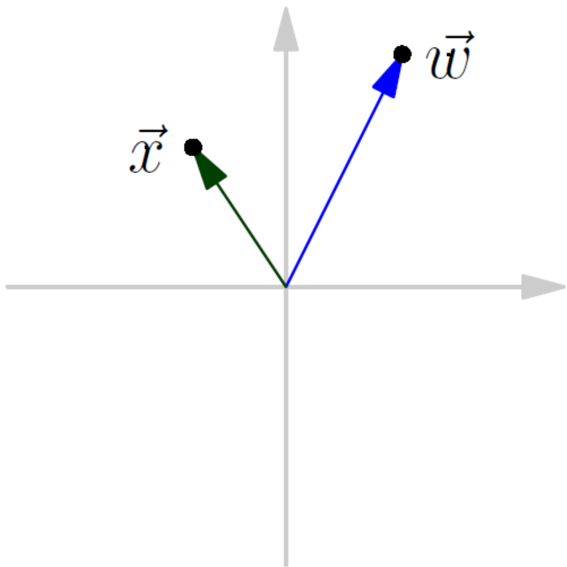




Space of weights and inputs - perceptron

Dual space for data and weights



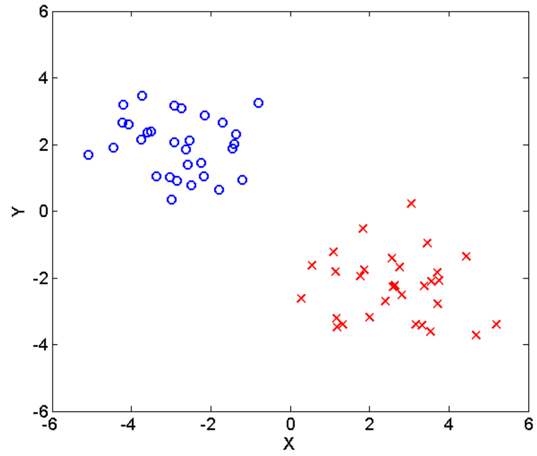


$$y' = w_1 x_1 + w_2 x_2 = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}$$

Principle of perceptron learning

How do we go about fixing weights, $\boldsymbol{w}$, for a given task?

Aim: classify all data samples (*binary* classification of *training* data)



So, how do we go about iteratively adjusting weights, $\boldsymbol{w}$?

$$\boldsymbol{w}^{(new)} = \boldsymbol{w}^{(old)} + \Delta\boldsymbol{w}$$

Principle of perceptron learning

How do we go about fixing weights, $\boldsymbol{w}$, for a given task?

Aim: classify all data samples (*binary* classification of *training* data)

$$\boldsymbol{w}^{(new)} = \boldsymbol{w}^{(old)} + \Delta\boldsymbol{w}$$

<u>What is the *intuition*?</u>

Principle of perceptron learning

How do we go about fixing weights, $\boldsymbol{w}$, for a given task?

Aim: classify all data samples (*binary* classification of *training* data)

$$\boldsymbol{w}^{(new)} = \boldsymbol{w}^{(old)} + \Delta \boldsymbol{w}$$

<u>What is the *intuition*?</u>

1. If a data sample is correctly classified, do nothing.

Principle of perceptron learning

How do we go about fixing weights, $\boldsymbol{w}$, for a given task?

Aim: classify all data samples (*binary* classification of *training* data)

$$\boldsymbol{w}^{(new)} = \boldsymbol{w}^{(old)} + \Delta\boldsymbol{w}$$

<u>What is the *intuition*?</u>

1. If a data sample is correctly classified, do nothing.
2. If a data sample belongs to "positive" class but the perceptron's output is 0 ($y' < 0 \Rightarrow y = 0$), modify $\boldsymbol{w}$ in the positive class direction, e.g.

$$\Delta\boldsymbol{w} = \boldsymbol{x}$$

Principle of perceptron learning

How do we go about fixing weights, $\boldsymbol{w}$, for a given task?

Aim: classify all data samples (*binary* classification of *training* data)

$$\boldsymbol{w}^{(new)} = \boldsymbol{w}^{(old)} + \Delta\boldsymbol{w}$$

<u>What is the *intuition*?</u>

1. If a data sample is correctly classified, do nothing.
2. If a data sample belongs to "positive" class but the perceptron's output is 0 ($y' < 0 \Rightarrow y = 0$), modify $\boldsymbol{w}$ in the positive class direction, e.g.

$$\Delta\boldsymbol{w} = \boldsymbol{x}$$

3. If a data sample belongs to "negative" class but the perceptron's output is 1 ($y' > 0 \Rightarrow y = 1$), modify $\boldsymbol{w}$ in the negative class direction, e.g.

$$\Delta\boldsymbol{w} = -\boldsymbol{x}$$

Perceptron learning rule

Training of a Thresholded Network: Perceptron Learning
Basic Principle: Weights are changed whenever a pattern is erroneously classified

When the result $= 0$, should be $= 1$

$$\Delta\vec{w} = \eta\vec{x}$$

When the result $= 1$, should be $= 0$

$$\Delta\vec{w} = -\eta\vec{x}$$

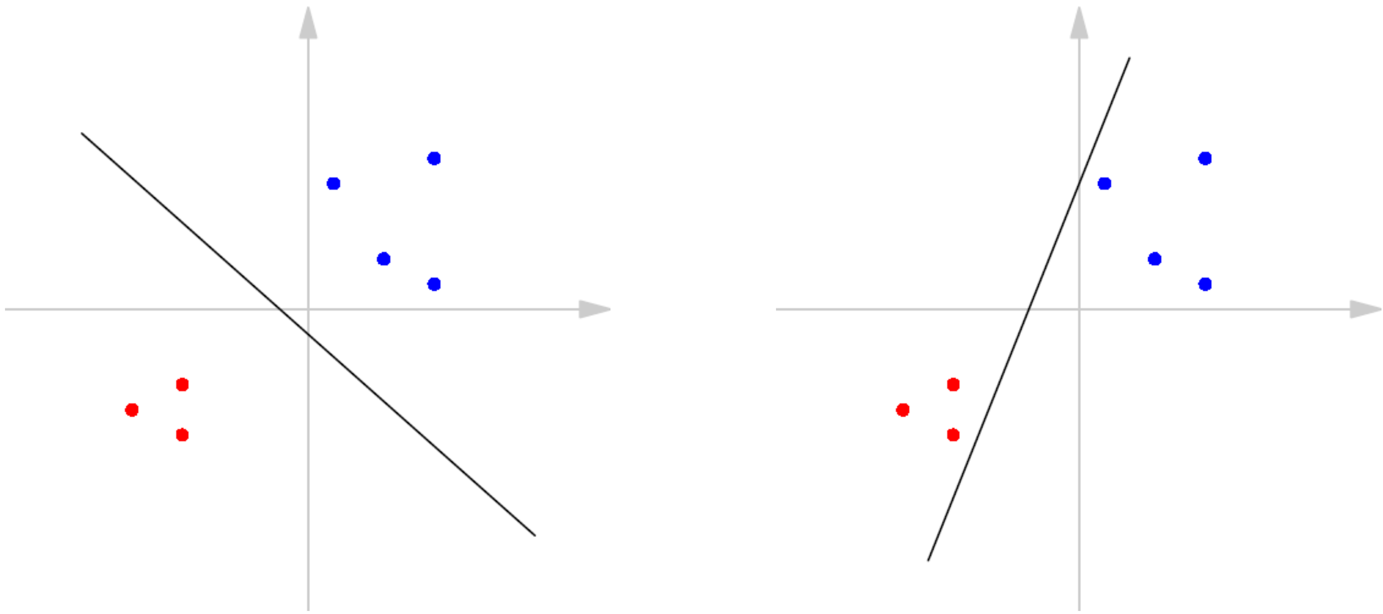
Perceptron learning – convergence theorem

Convergence theorem

If a solution exists for a finite training dataset then perceptron learning always converges after a finite number of steps (independent of step size/learning rate, η)

Perceptron learning

Problem: learning terminates prematurely.



Delta rule

Delta rule (Widrow-Hoff rule, ADALINE)

1. Symmetric target values: {-1, 1}
2. Error is measured before thresholding

$$e = t - \vec{w}^{\mathrm{T}} \vec{x}$$

3. Find weights that minimise the error cost function

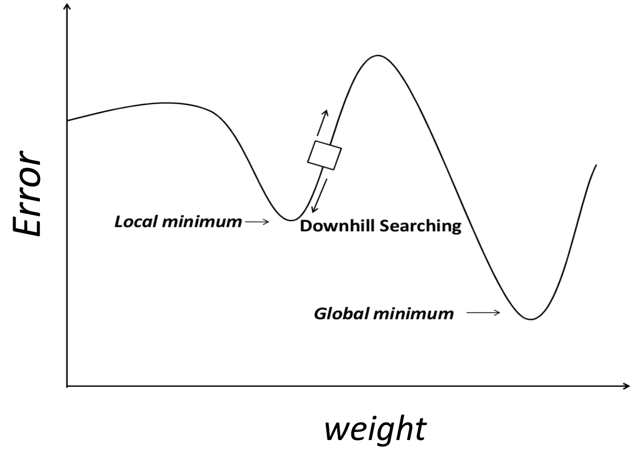
$$\varepsilon = \frac{e^2}{2}$$

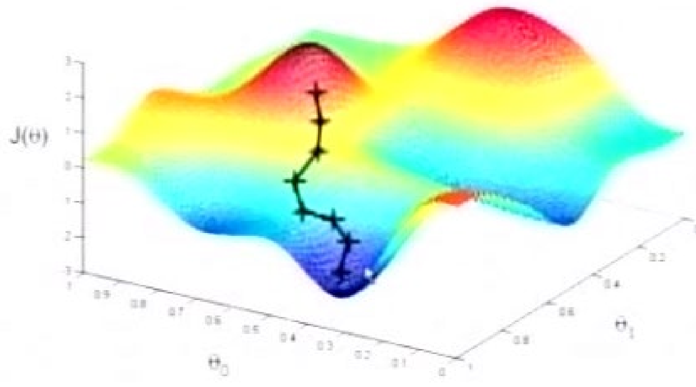
Delta rule

The task is to minimise the cost function $\varepsilon = \frac{e^2}{2}$

Simple algorithm: **steepest descent**

- Gradient defines the direction in which the error increases most
- *Steepest* gradient *descent* implies that the move in the opposite direction in the weight space should be taken $\Delta\vec{w} = -\eta \frac{\partial \varepsilon}{\partial \vec{w}}$





Delta rule

The task is to minimise the cost function $\varepsilon = \frac{e^2}{2}$

Simple algorithm: **steepest descent**

- Gradient defines the direction in which the error increases most
- *Steepest* gradient *descent* implies that the move in the opposite direction in the weight space should be taken $\Delta\vec{w} = -\eta\frac{\partial\varepsilon}{\partial\vec{w}}$

- Gradient is calculated as follows (*chain rule*):

$$\frac{\partial}{\partial\vec{w}}\varepsilon\left(e(\vec{w})\right) = \frac{d\varepsilon}{de}\,\frac{\partial e(\vec{w})}{\partial\vec{w}} = e\frac{\partial e}{\partial\vec{w}} = e\frac{\partial(t-\vec{w}^{\mathrm{T}}\vec{x})}{\partial\vec{w}} = -e\vec{x}$$

Delta rule

The task is to minimise the cost function $\varepsilon = \dfrac{e^2}{2}$

Simple algorithm: **steepest descent**

- Gradient defines the direction in which the error increases most
- *Steepest* gradient *descent* implies that the move in the opposite direction in the weight space should be taken $\Delta\vec{w} = -\eta \dfrac{\partial \varepsilon}{\partial \vec{w}}$

- Gradient is calculated as follows (*chain rule*):

$$\frac{\partial}{\partial \vec{w}}\varepsilon\left(e(\vec{w})\right) = \frac{d\varepsilon}{de}\,\frac{\partial e(\vec{w})}{\partial \vec{w}} = e\frac{\partial e}{\partial \vec{w}} = e\frac{\partial (t - \vec{w}^{\mathrm{T}}\vec{x})}{\partial \vec{w}} = -e\vec{x}$$

Delta rule: $\Delta\vec{w} = \eta e\,\vec{x}$

Training of thresholded single-layer networks

Perceptron

Perceptron learning:

$$\Delta\vec{w} = \eta e \vec{x} \qquad \text{where} \quad e = t - y \qquad y = f_{step}\left(\vec{w}^{\mathrm{T}}\vec{x}\right)$$

Delta rule:

$$\Delta\vec{w} = \eta e \vec{x} \qquad \text{where} \quad e = t - \vec{w}^{\mathrm{T}}\vec{x}$$

Training/learning process

Choose data sample $\boldsymbol{x}^{(i)}; t^{(i)}$

Feed input $\boldsymbol{x}^{(i)}$ and estimate error $e^{(i)} = t^{(i)} - f(\boldsymbol{x}^{(i)}, \boldsymbol{w})$

Calculate update $\Delta \boldsymbol{w}^{(i)} = \eta e^{(i)} \boldsymbol{x}^{(i)}$

epoch

Apply update

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \Delta \boldsymbol{w}^{(i)}$$

Update the network after each sample

on-line, sample-by-sample learning

Training/learning process

Choose data sample $\boldsymbol{x}^{(i)}; t^{(i)}$

Feed input $\boldsymbol{x}^{(i)}$ and estimate error $e^{(i)} = t^{(i)} - f(\boldsymbol{x}^{(i)}, \boldsymbol{w})$

Calculate update $\Delta \boldsymbol{w}^{(i)} = \eta e^{(i)} \boldsymbol{x}^{(i)}$

epoch

Apply update

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \Delta \boldsymbol{w}^{(i)}$$

epoch

Accumulate $\Delta \boldsymbol{w}$

$$\Delta \boldsymbol{w}^{(accum)} = \sum_i \Delta \boldsymbol{w}^{(i)}$$

Apply update

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \Delta \boldsymbol{w}^{(accum)}$$

Update the network after an epoch

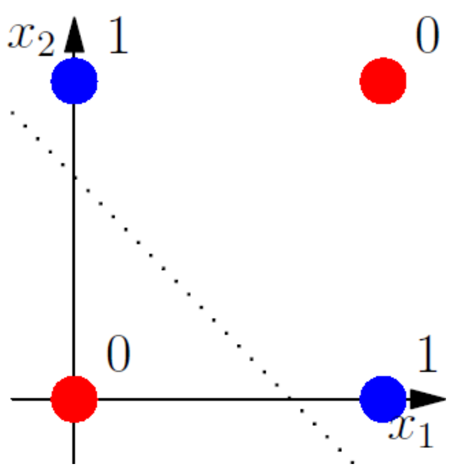
batch learning

Separability with TLU / perceptron

Can all sets of patterns be separated?

Classical counter-example is Exclusive OR (XOR)

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow 0 \qquad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow 1 \qquad \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow 1 \qquad \begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow 0$$

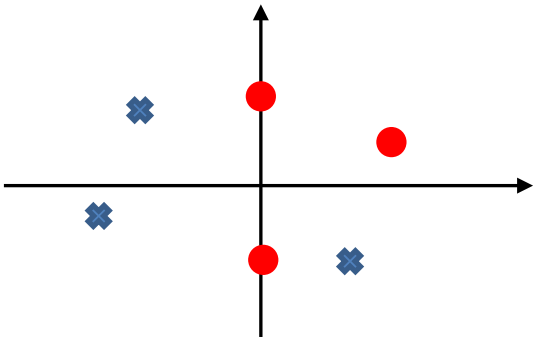


It is NOT linearly separable!

Discussion 1

1. What <u>parameter</u> (and how) determines *storage capacity* of the linear correlational memory with Hebbian learning? What do you think could help increase the capacity of networks with Hebbian learning?

2. Predict a type of behaviour (qualitatively) of the perceptron learning rule for the given problem

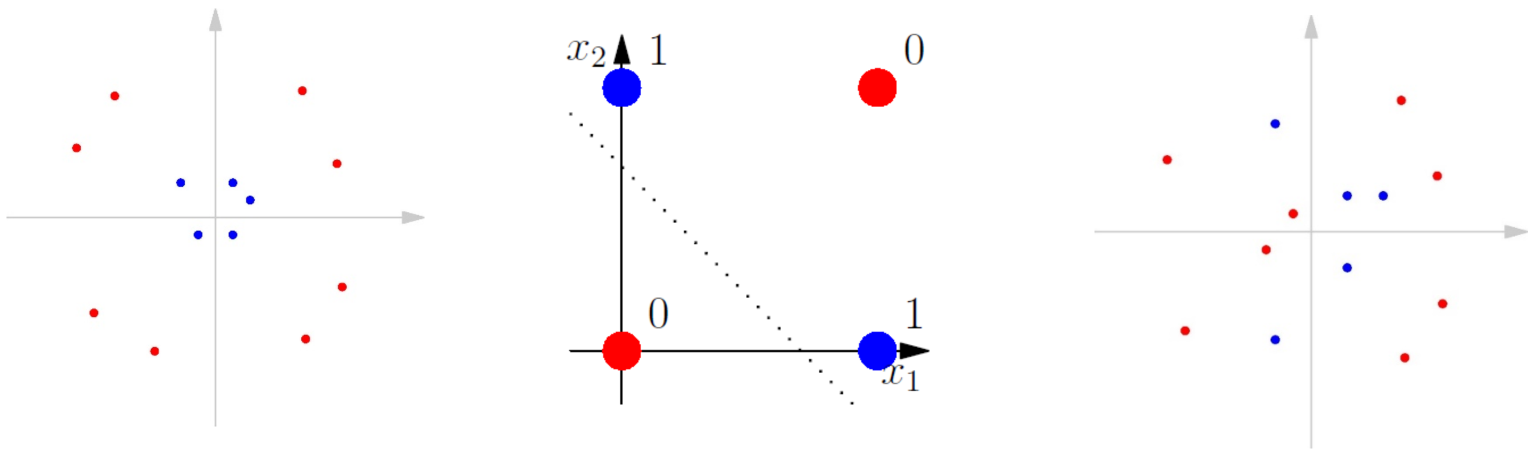


3. For problems that are not linearly separable, delta rule converges while perceptron learning does not terminate. Please modify perceptron learning to compute the linear separation with the highest classification accuracy?

4. What affects the process of learning with a classical perceptron rule for a given linearly separable dataset? List these factors, comment on their impact.

Discussion 2

1. Do you need an iterative delta rule to find the “best” linear separation?
2. What effect, if any, do initial conditions (initial weights, order of samples etc.) have on perceptron’s separating hyperplane found with an online delta rule?
3. In classical perceptron learning the weight vector could be normalized (its length) every few epochs – what effect would it have on the learning process?
4. How would you approach classifying the following datasets into two classes with 100% accuracy?



Some extra questions

- When does feed-forward cascading of layers of neurons offer extra computational advantages (over a single linear layer)?
- In what sense is Hebbian learning (biologically motivated) local? Is perceptron learning local too?
- What do we need a bias for in perceptrons?
- Does the delta rule (gradient descent) guarantee to find a separating hyperplane for linearly separable problems?
- What does the term “dual space” mean?
- How can we check linear separability in high-dim spaces?