DD2437 – Artificial Neural Networks and Deep Architectures (annda)

Lecture 2: **From perceptron learning rules to backpropagation – supervised learning**

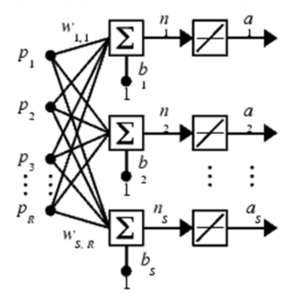
Pawel Herman

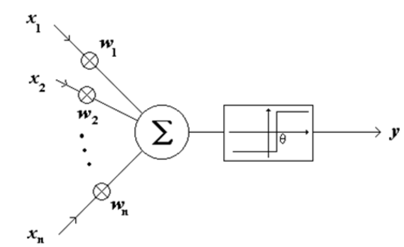
Computational Science and Technology (CST)

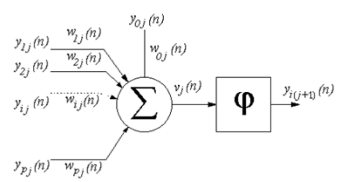
KTH Royal Institute of Technology

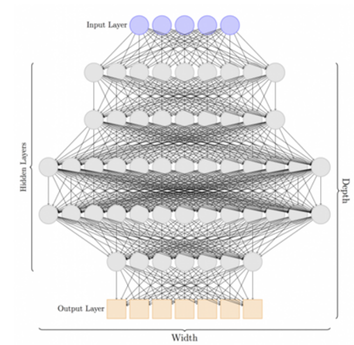
Lecture overview

- A quick recap
- Linear feed-forward networks
- Thresholded single-layer networks
- Perceptron learning, delta rule
- Multi-layer perceptron
- Backpropagation

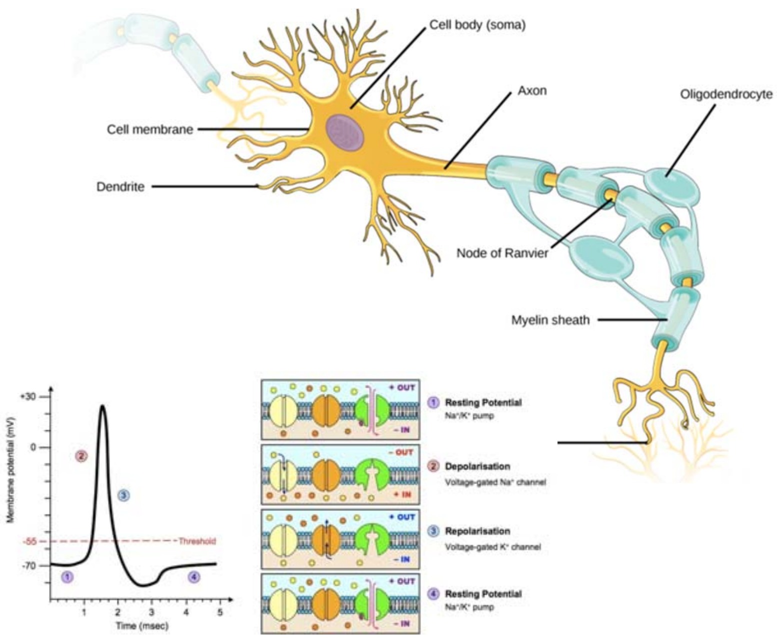


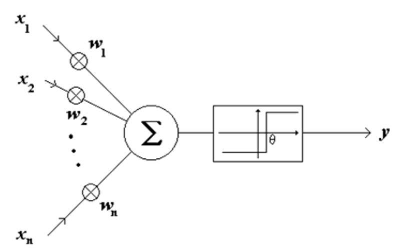






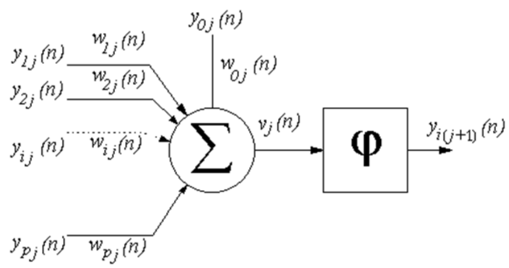
From biological inspirations to ANNs





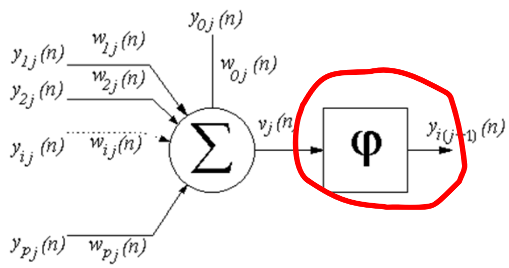
Fundamental aspects

nodes

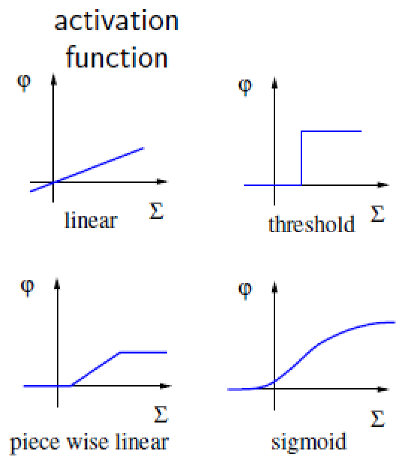


Fundamental aspects

nodes

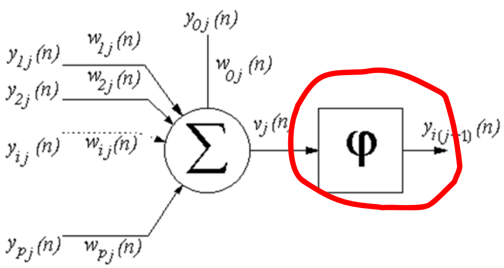


activation function

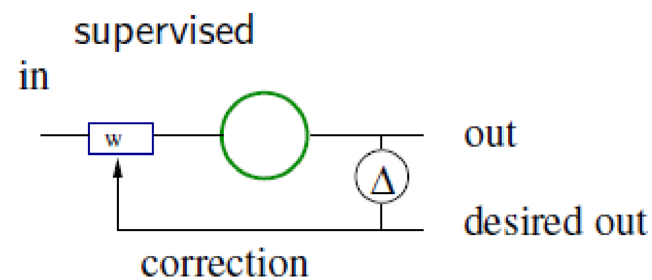


Fundamental aspects

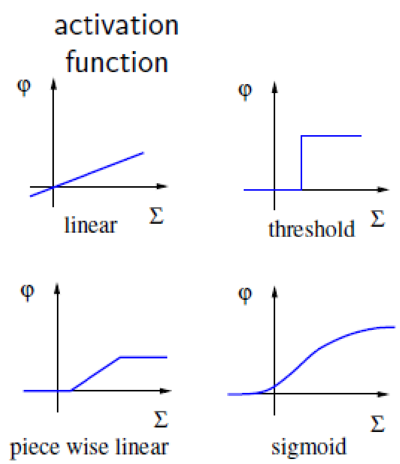
nodes



learning rule

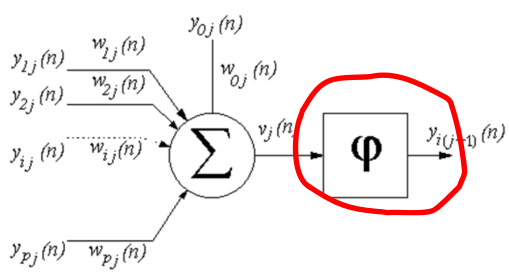


activation function

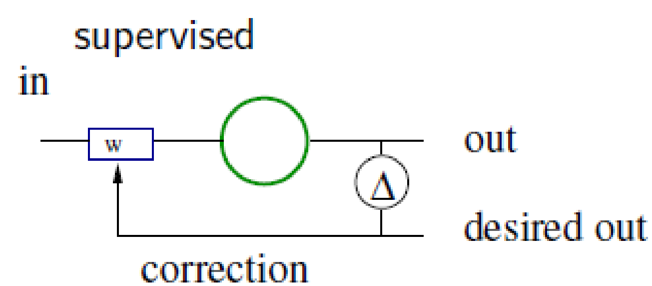


Fundamental aspects

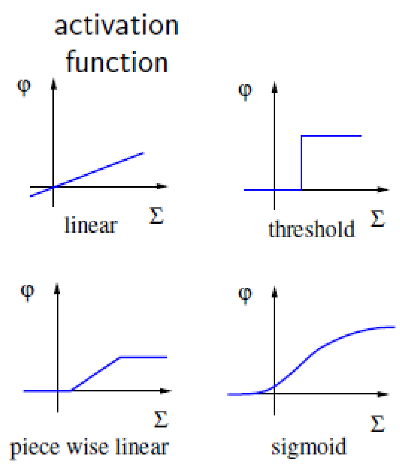
nodes



learning rule

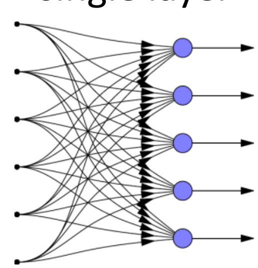


activation function

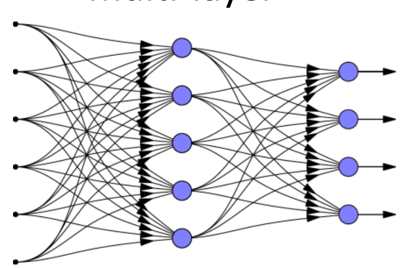


topologies, architectures

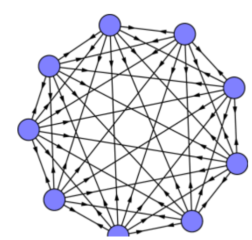
single layer



multi-layer

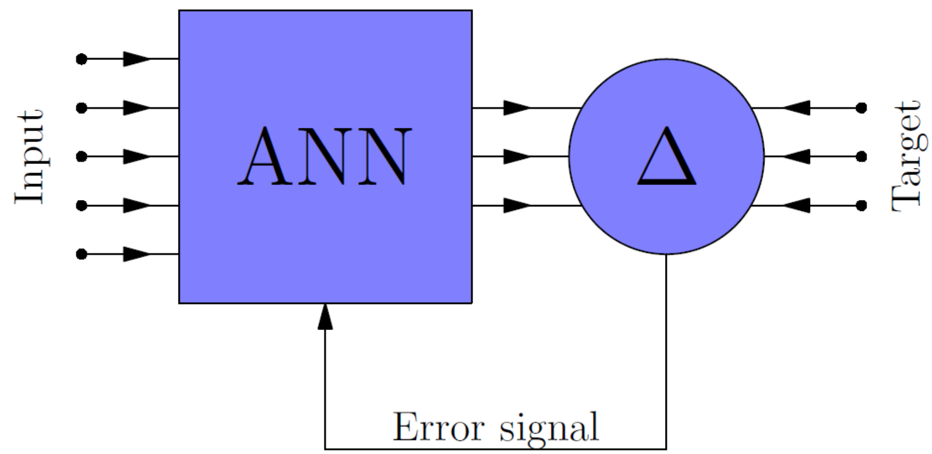


recurrent



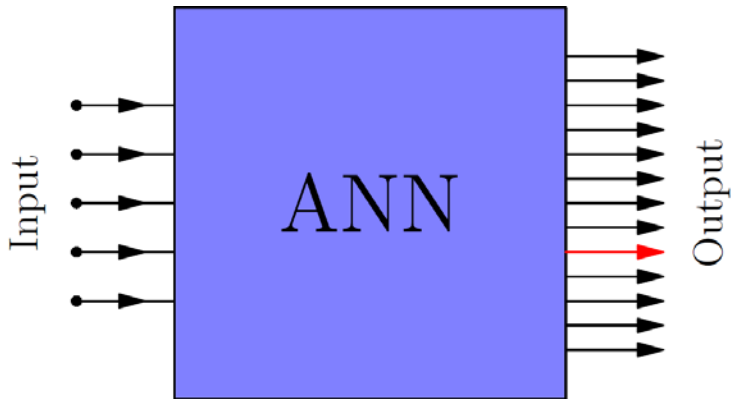
Learning principles

- **Error correction**
- Competitive learning
- Coincidence detection



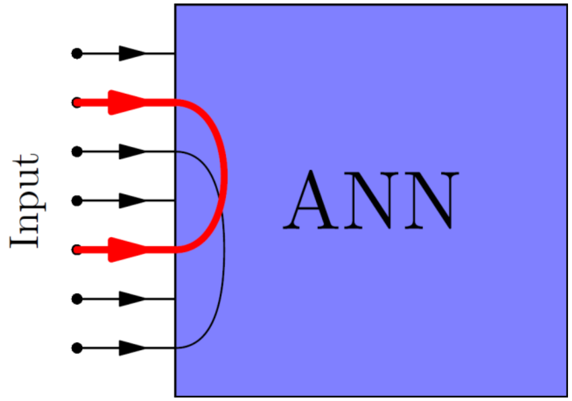
Learning principles

- Error correction
- **Competitive learning**
- Coincidence detection



Learning principles

- Error correction
- Competitive learning
- **Coincidence detection**



Learning principles

Learning approaches

- supervised
  - with a teacher that provides a correct answer
  - error correction paradigm

Learning principles

Learning approaches

- supervised
- unsupervised (input data only)
  - only input data is available
  - ability to organise information without any error signal to evaluate a potential solution – an explorative approach
  - detecting statistical regularities of the input data and forming internal representations that encode features of the input data

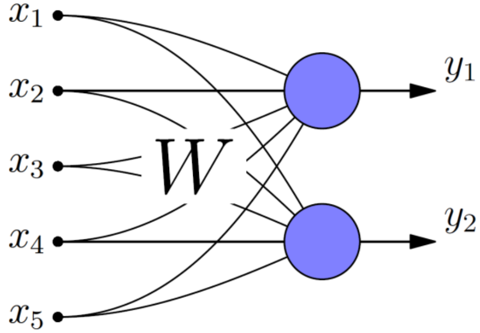
Learning principles

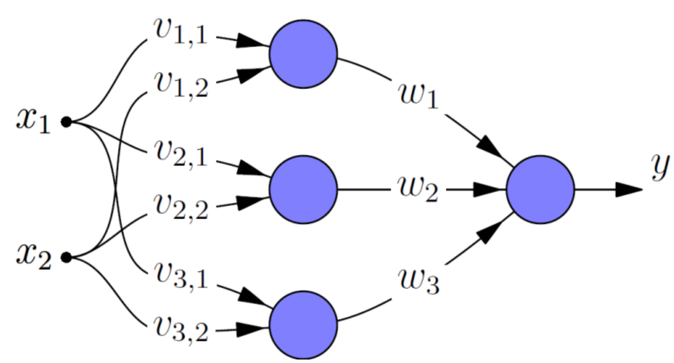
Learning approaches

- supervised
- unsupervised (input data only)
- reinforcement
  - simple scalar “reward” signal gives feedback on success

Linear networks

What can be computed?





$$y = \vec{w}^{\mathrm{T}} \cdot \vec{x}$$

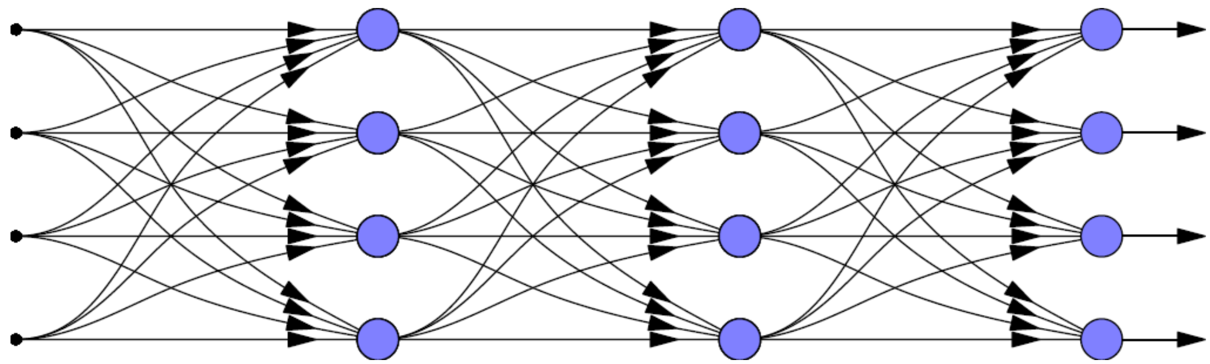
$$y = \mathrm{W} \cdot \vec{x}$$

$\vec{w}$ - weight vector

W - weight matrix

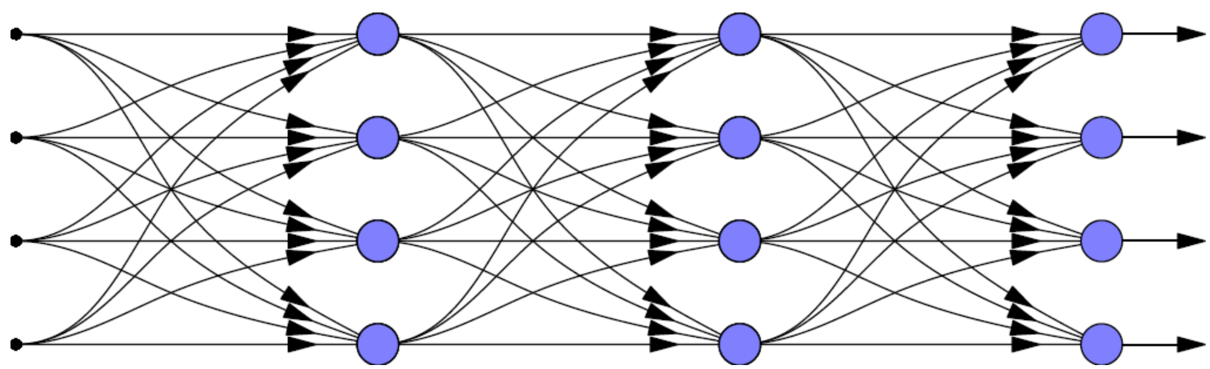
Linear networks

What happens when we concatenate several linear networks?



Linear networks

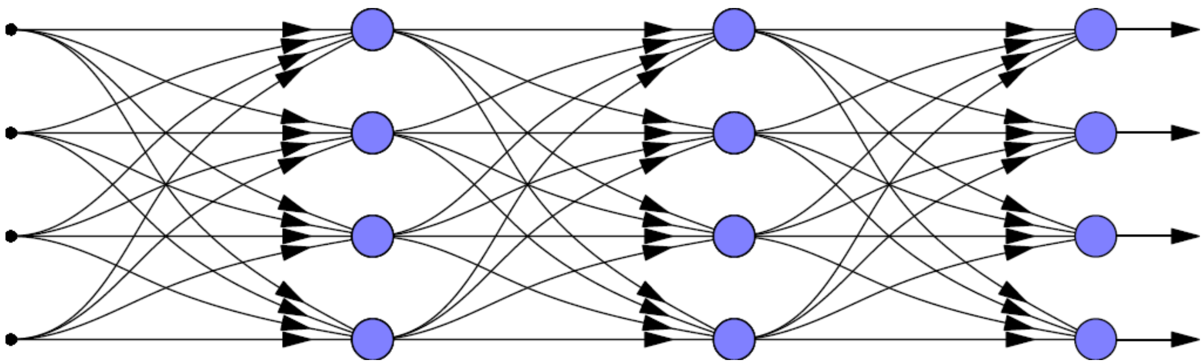
What happens when we concatenate several linear networks?



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

Linear networks

What happens when we concatenate several linear networks?

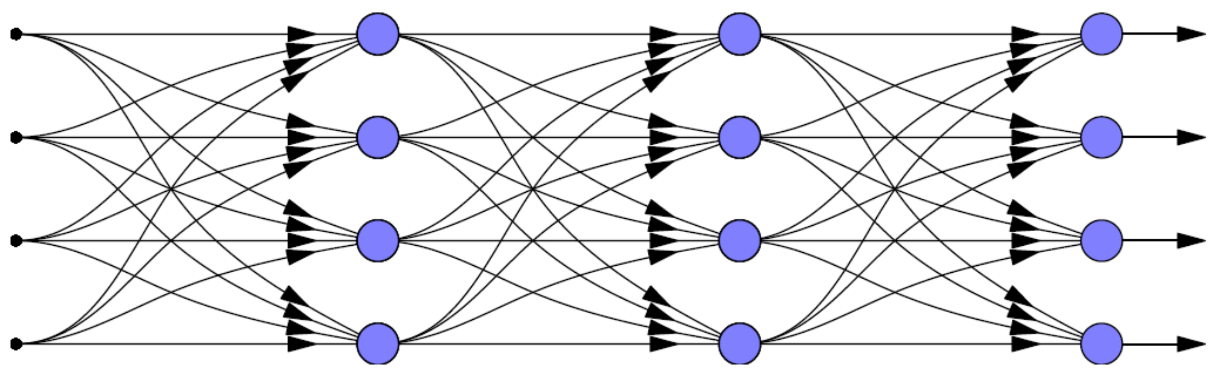


$$\vec{y} = W_3\left(W_2\left(W_1\vec{x}\right)\right) = \left(W_3 W_2 W_1\right)\vec{x}$$

$$\text{Let } \ W = W_3 W_2 W_1 \quad \Rightarrow \quad \vec{y} = W\vec{x}$$

Linear networks

What happens when we concatenate several linear networks?



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

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

It is still a linear mapping !

Storing mappings (memorising)

The program "resides" in weights

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

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

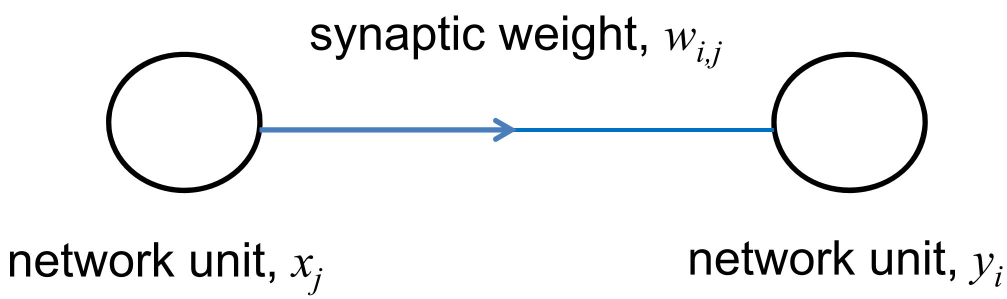
Simultaneous activation of two neurons strengthens their synaptic inter-connection

Common interpretation:

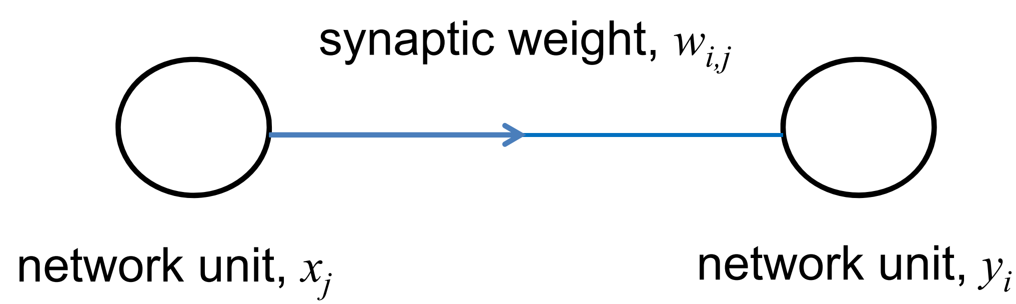
covariance rule

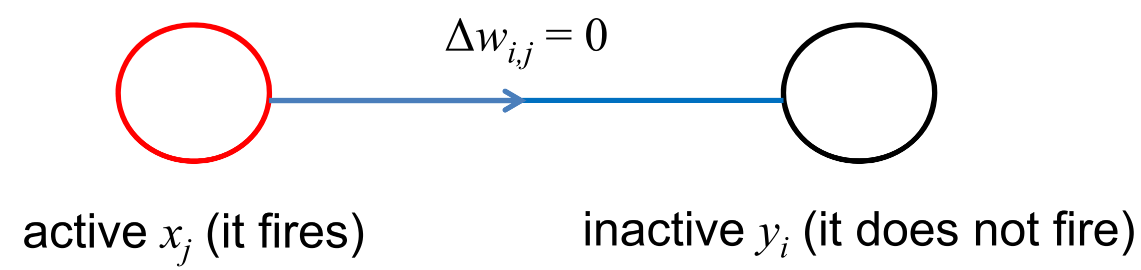
$$\Delta w_{ij} = x_j y_i \qquad \text{or} \ldots \quad \Delta w_{ij} = (x_j - \bar{x})(y_i - \bar{y})$$

Hebbian learning rule

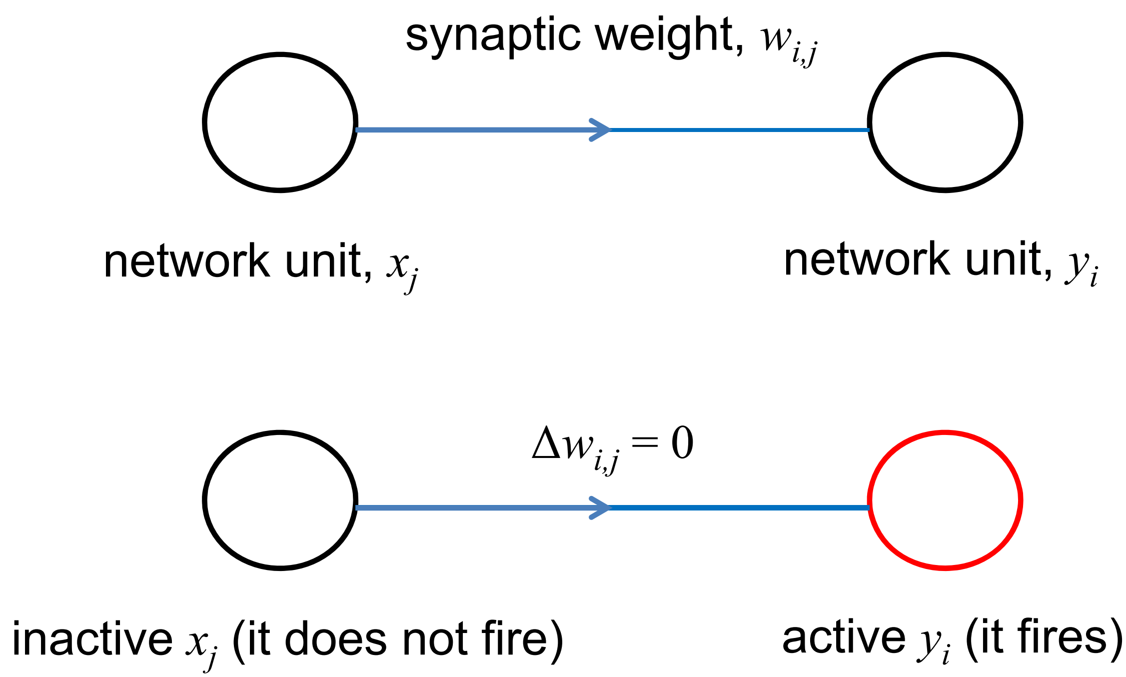


Hebbian learning rule

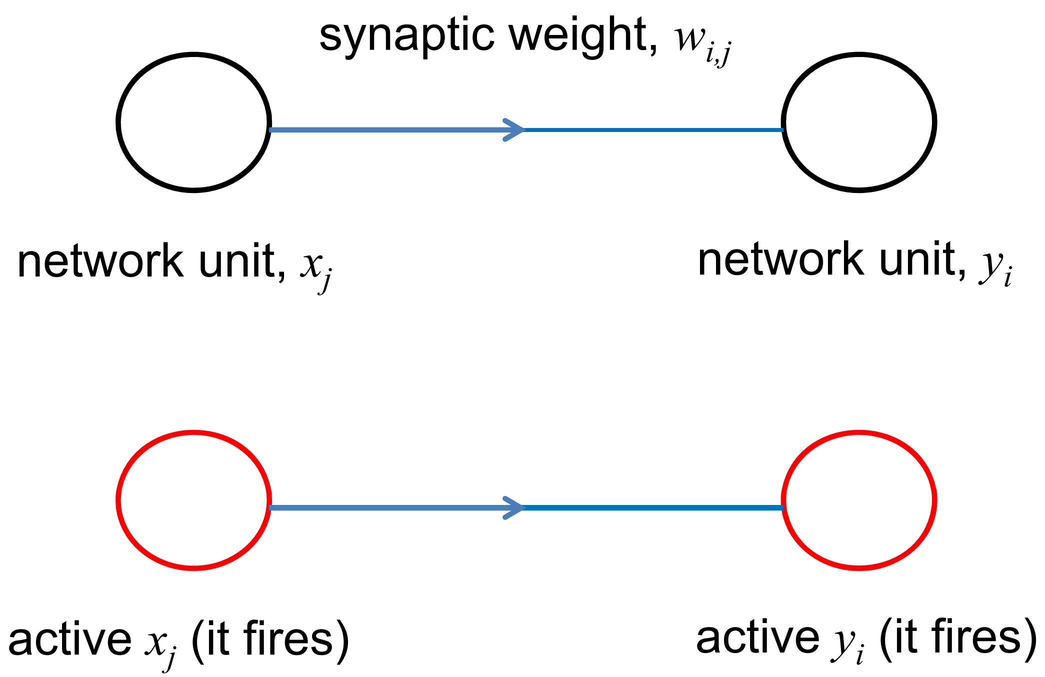




Hebbian learning rule



Hebbian learning rule

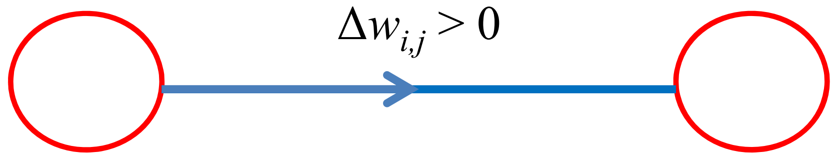


Hebbian learning rule

synaptic weight, $w_{i,j}$

network unit, x_j network unit, y_i

$\Delta w_{i,j} > 0$



active x_j (it fires) active y_i (it fires)

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

“Fire together, wire together”

Storing mappings (memorising)

Storing a mapping using Hebb’s rule

$$\vec{x}_1 \to \vec{y}_1 \qquad \vec{x}_2 \to \vec{y}_2 \qquad \vec{x}_3 \to \vec{y}_3 \quad \ldots \quad \vec{x}_n \to \vec{y}_n$$

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 $\Delta w_{ij} = x_i y_i$

Storing mappings (memorising)

Storing a mapping using Hebb's rule

$$\vec{x}_1 \to \vec{y}_1 \qquad \vec{x}_2 \to \vec{y}_2 \qquad \vec{x}_3 \to \vec{y}_3 \quad \dots \quad \vec{x}_n \to \vec{y}_n$$

Hebb's rule $\Delta w_{ij} = x_i y_j$

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

(outer product of vector patterns)

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 $$\Delta w_{ij} = x_i y_j$$

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

Correlational memory!

Storing mappings (memorising)

Retrieving a memory trace

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

$$\vec{x}_k \rightarrow \; ?$$

Storing mappings (memorising)

Retrieving a memory trace

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

$$\vec{x}_k \rightarrow \; ?$$

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

Storing mappings (memorising)

Retrieving a memory trace

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

$$\vec{x}_k \rightarrow \ ?$$

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

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

Storing mappings (memorising)

Retrieving a memory trace

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

$$\vec{x}_k \rightarrow \ ?$$

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

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

Storing mappings (memorising)

Retrieving a memory trace

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

$$\vec{x}_k \rightarrow \ ?$$

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

$$= \vec{y}_k \underbrace{(\vec{x}_k^{\mathrm{T}} \vec{x}_k)}_{\alpha} + \sum_{p \neq k}^{n} \vec{y}_p \underbrace{(\vec{x}_p^{\mathrm{T}} \vec{x}_k)}_{\approx \mathbf{0}} \approx \alpha \vec{y}_k$$

Storing mappings (memorising)

Retrieving a memory trace

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

$$\vec{x}_k \rightarrow ?$$

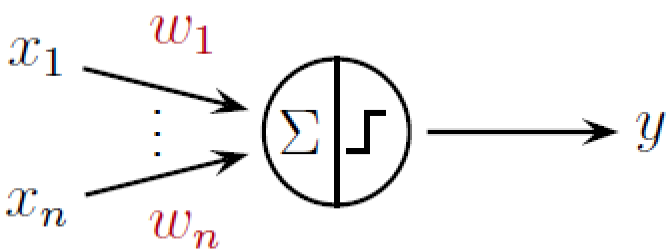
$$\vec{y}_{out} = \mathrm{W}\vec{x}_k = \sum_{p=1}^{n} (\vec{y}_p \vec{x}_p^{\mathrm{T}}) \vec{x}_k = \sum_{p=1}^{n} \vec{y}_p (\vec{x}_p^{\mathrm{T}} \vec{x}_k) =$$

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

Perfect memory only if the patterns $\vec{x}_p$ are orthogonal

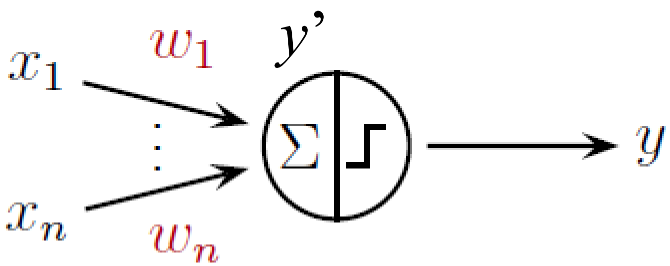
TLU – how it all started….

Threshold logic unit – McCulloch Pitts neuron (1942)



TLU – McCulloch Pitts

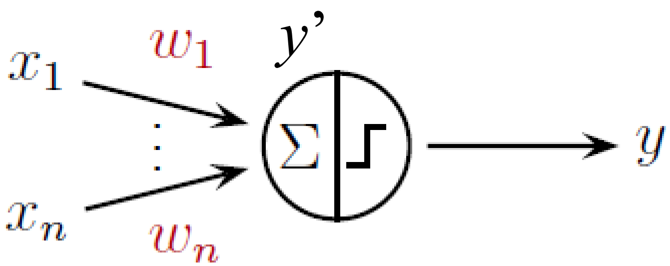
Threshold logic unit – McCulloch Pitts neuron (1942)



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

TLU – McCulloch Pitts

Threshold logic unit – McCulloch Pitts neuron (1942)



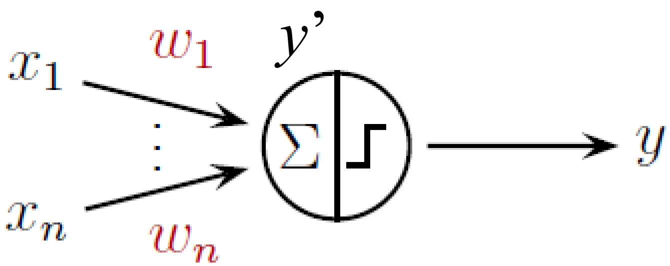
$$y' = w_1 x_1 + w_2 x_2 \qquad y = f_{step}(y')$$

If threshold is 0, then:

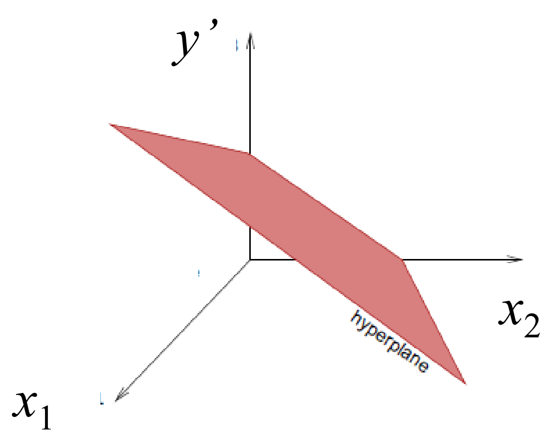
$$w_1 x_1 + w_2 x_2 > 0 \rightarrow y' > 0 \rightarrow y = 1$$

$$w_1 x_1 + w_2 x_2 <= 0 \rightarrow y' <= 0 \rightarrow y = 0$$

Geometrical interpretation

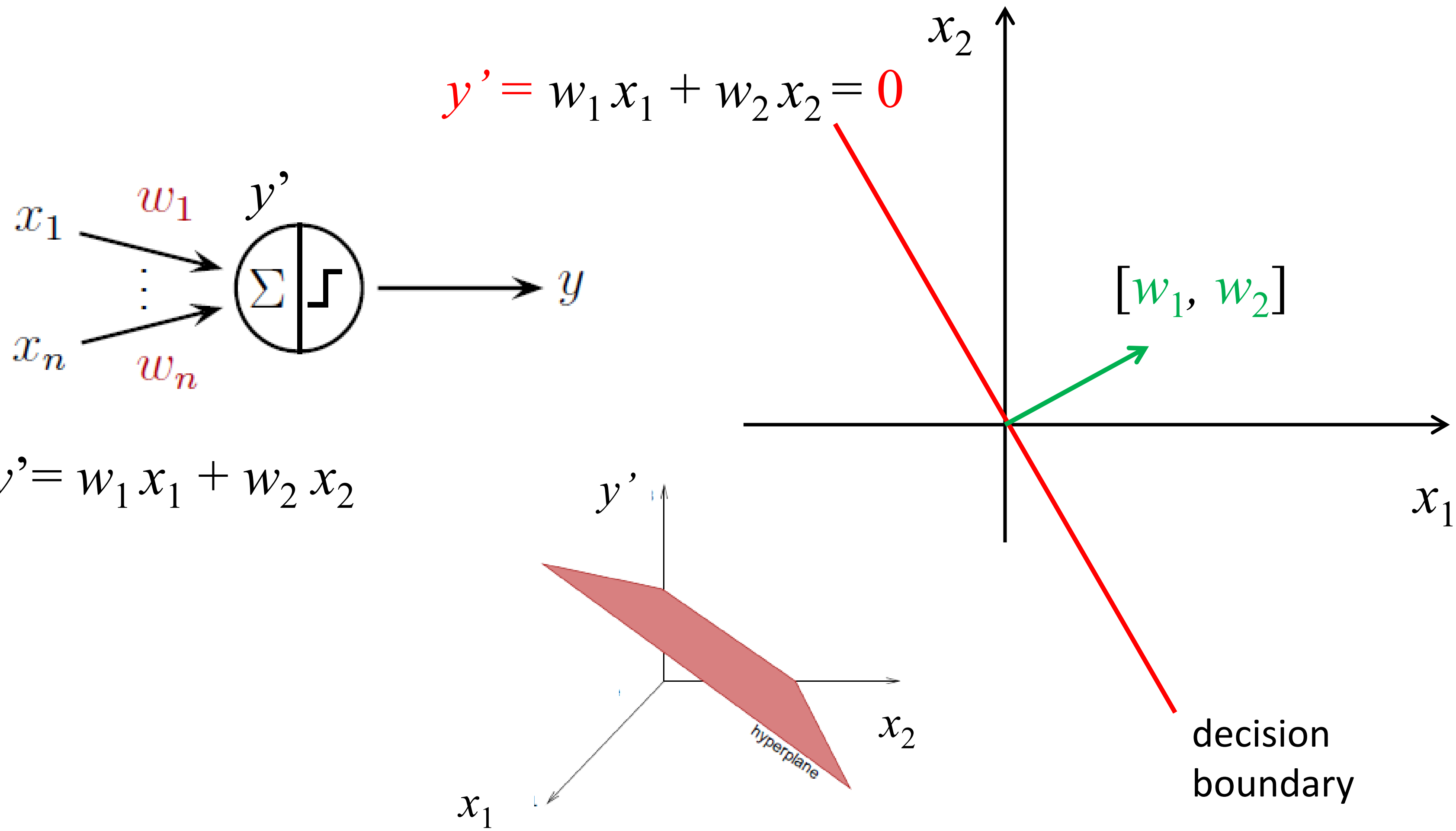


$$y' = w_1 x_1 + w_2 x_2$$



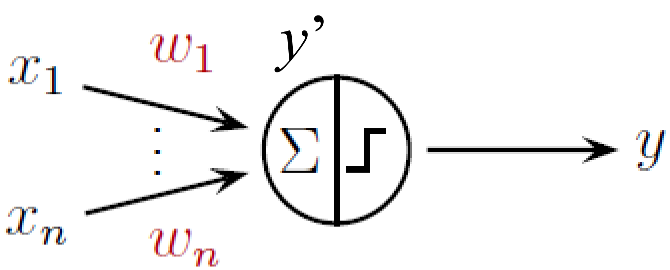
hyperplane

Geometrical interpretation

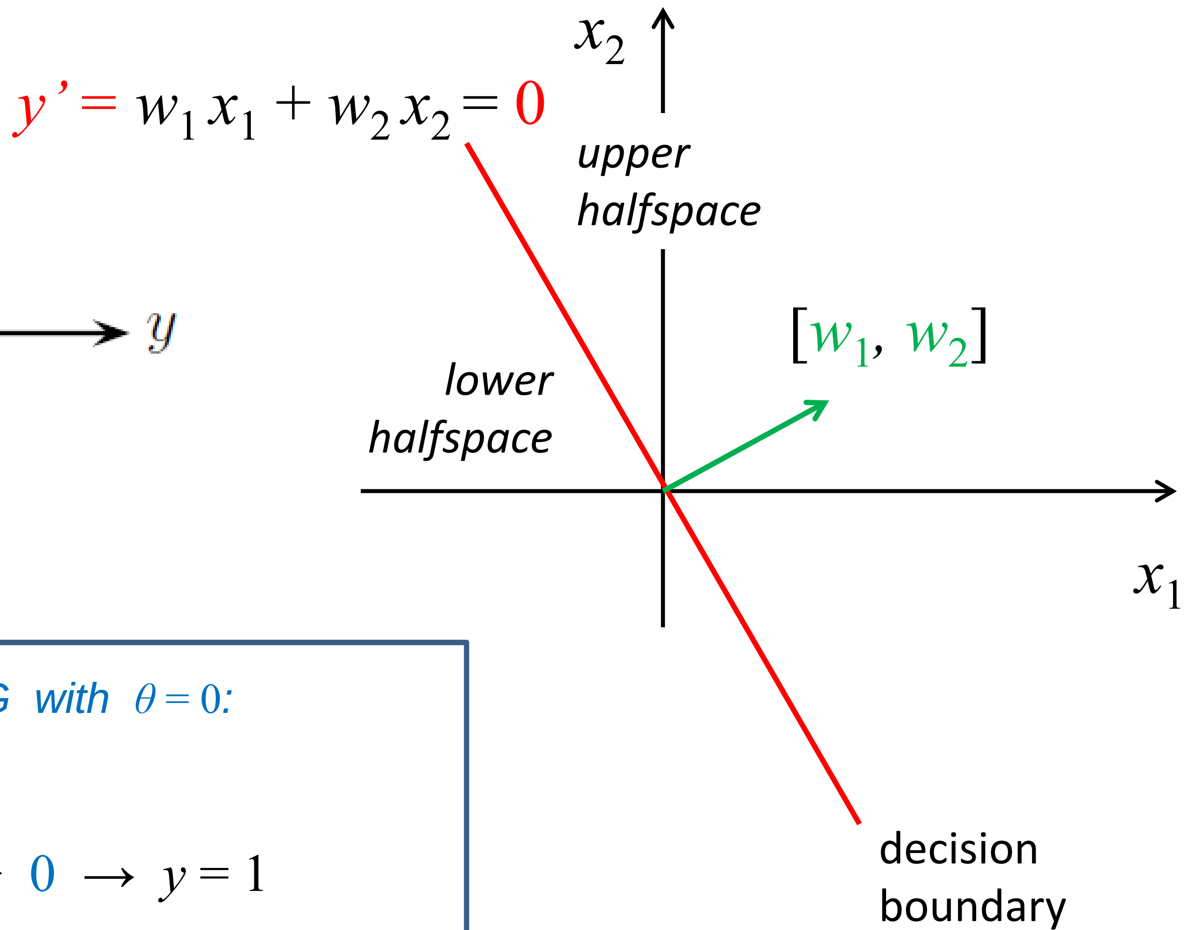


$$y' = w_1 x_1 + w_2 x_2$$

Threshold in TLU



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



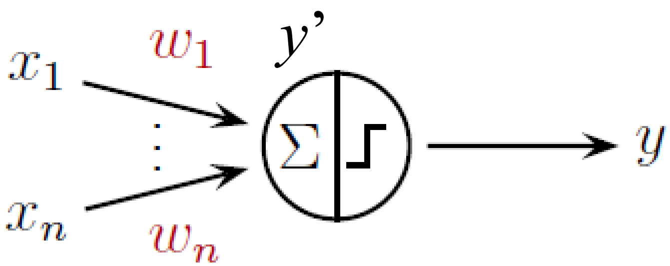
THRESHOLDING with $\theta = 0$:

$$y = f_{step}(y')$$

$$w_1 x_1 + w_2 x_2 > 0 \rightarrow y' > 0 \rightarrow y = 1$$

$$w_1 x_1 + w_2 x_2 <= 0 \rightarrow y' <= 0 \rightarrow y = 0$$

Threshold in TLU



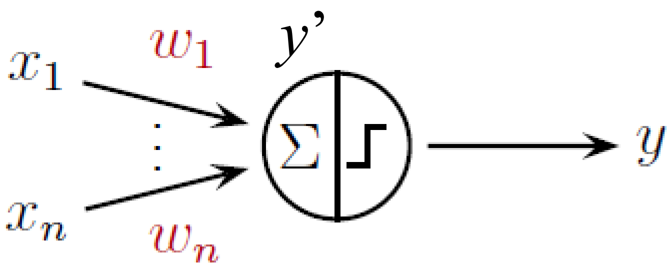
$$y' = w_1 x_1 + w_2 x_2$$

THRESHOLDING with $\theta \sim= 0$*:*

$$y' > \theta \rightarrow y = 1$$

$$y' <= \theta \rightarrow y = 0$$

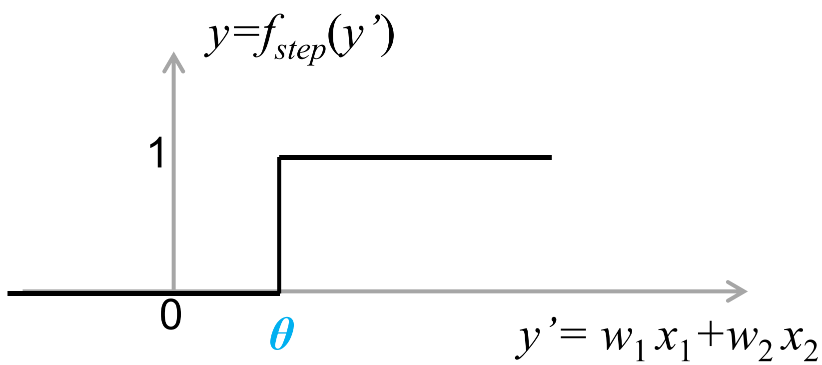
Threshold in TLU



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

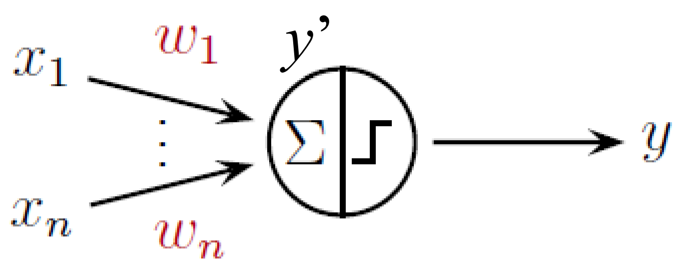
THRESHOLDING with $\theta \sim= 0$:

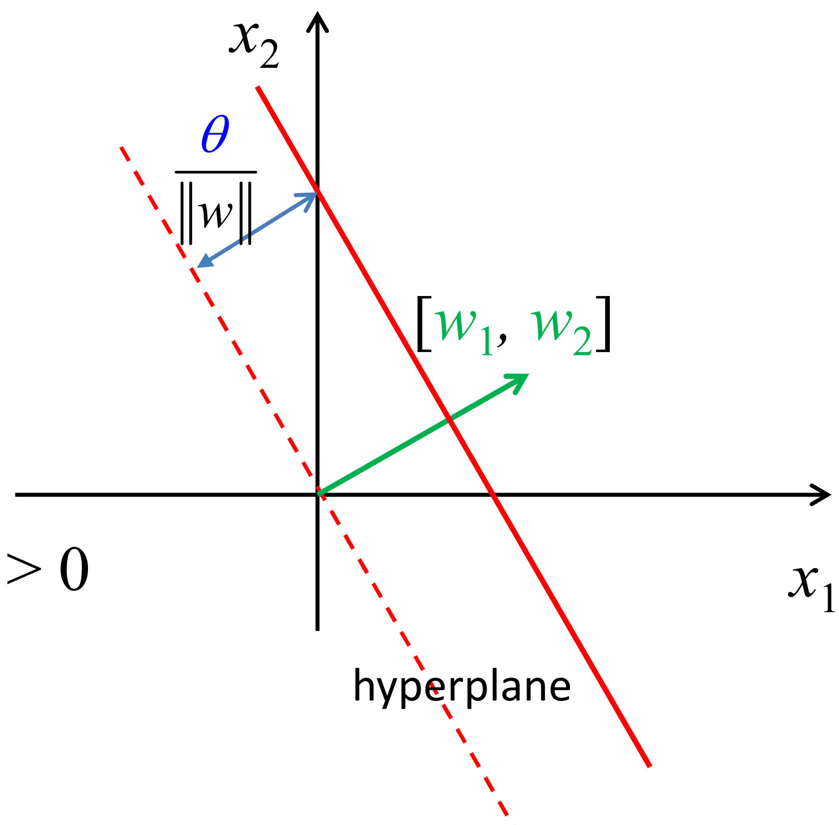
$$y' > \theta \rightarrow y = 1$$
$$y' <= \theta \rightarrow y = 0$$



$$w_1 x_1 + w_2 x_2 > \theta \rightarrow y = 1$$

Threshold in TLU – geometrical interpretation

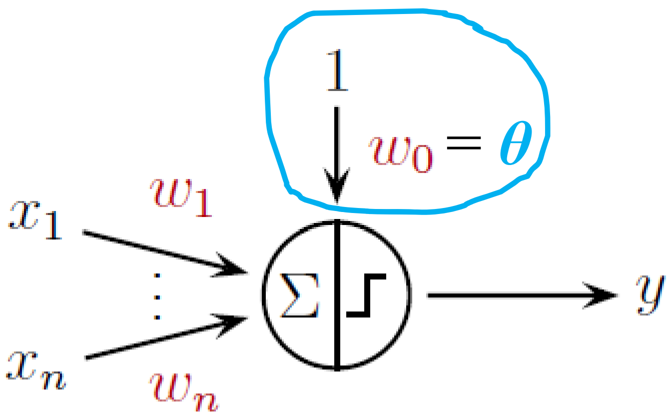


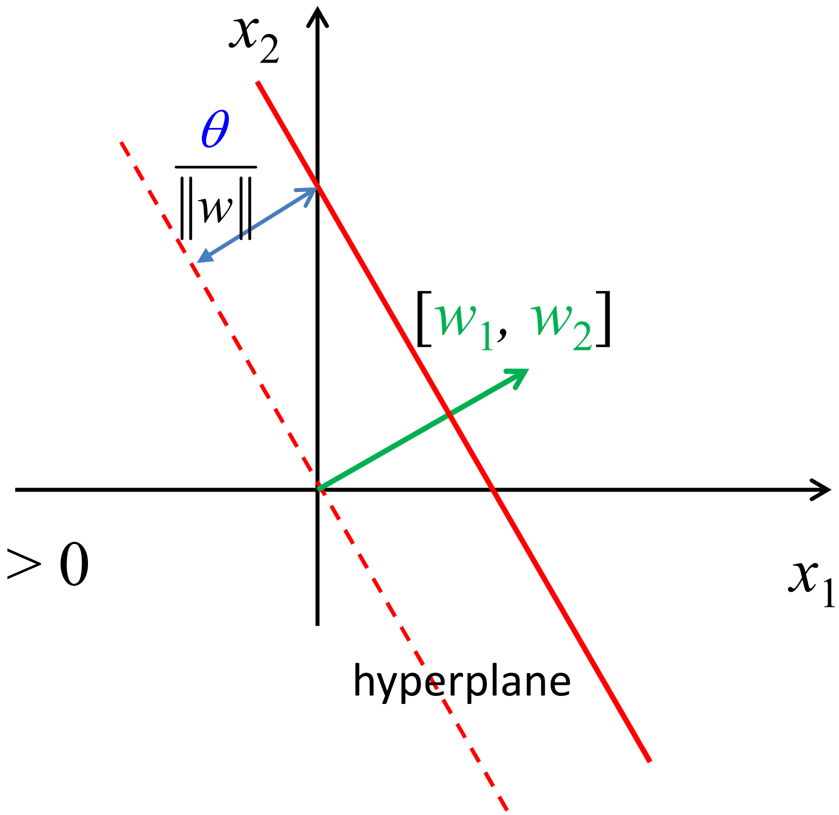


$$w_1 x_1 + w_2 x_2 > \theta \rightarrow w_1 x_1 + w_2 x_2 - \theta > 0$$

$$y' = w_1 x_1 + w_2 x_2 - \theta$$

Threshold in TLU – bias trick





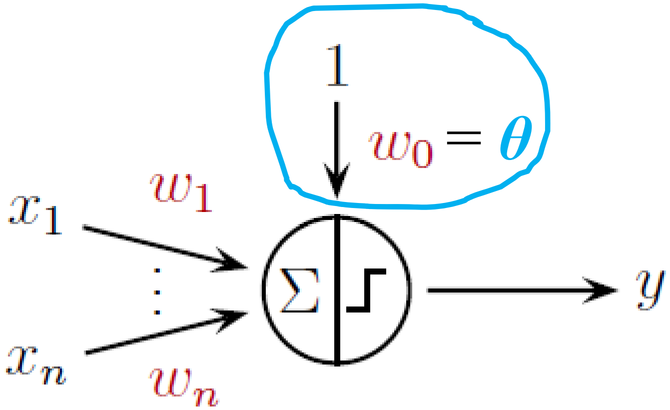
$$w_1 x_1 + w_2 x_2 > \theta \rightarrow w_1 x_1 + w_2 x_2 - \theta > 0$$

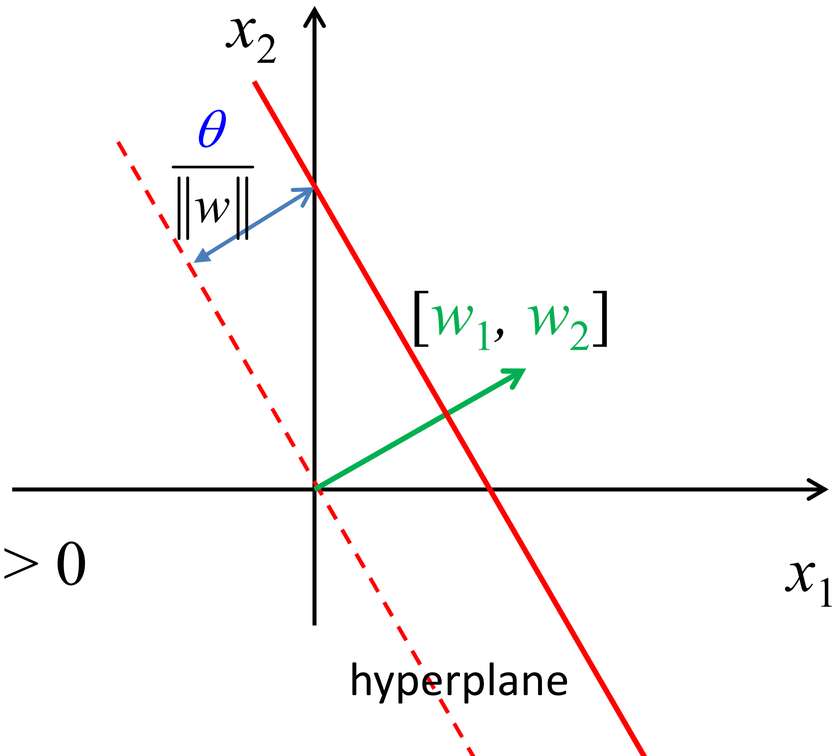
$$y' = w_1 x_1 + w_2 x_2 - \theta$$

$$y' = w_1 x_1 + w_2 x_2 + w_0 1$$

where: $bias \quad w_0 = -\theta$

Threshold in TLU – bias trick





$$w_1 x_1 + w_2 x_2 > \theta \rightarrow w_1 x_1 + w_2 x_2 - \theta > 0$$

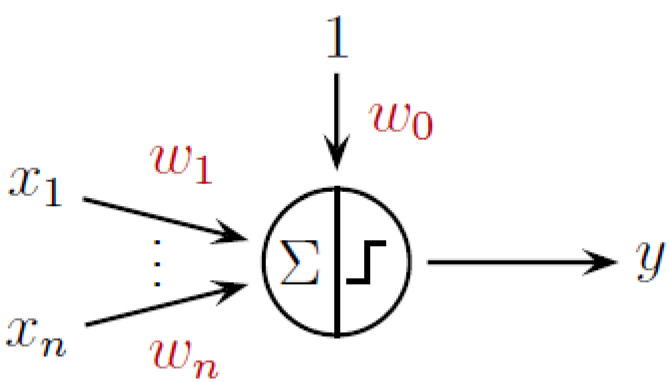
$$y' = w_1 x_1 + w_2 x_2 - \theta$$

$$y' = w_1 x_1 + w_2 x_2 + w_0 1$$

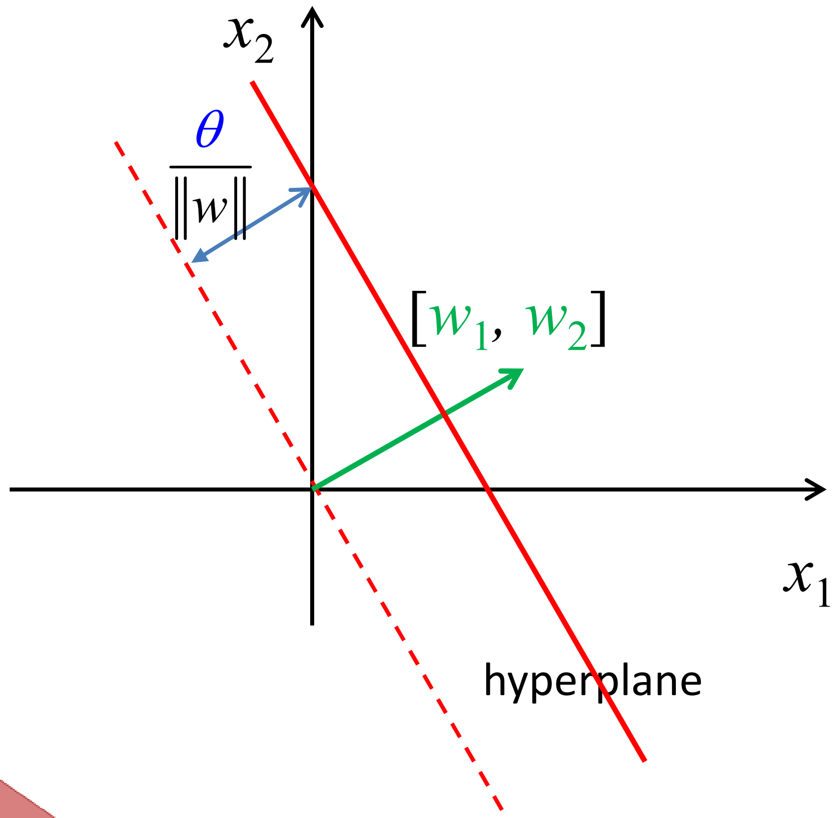
where: $bias \quad w_0 = -\theta$

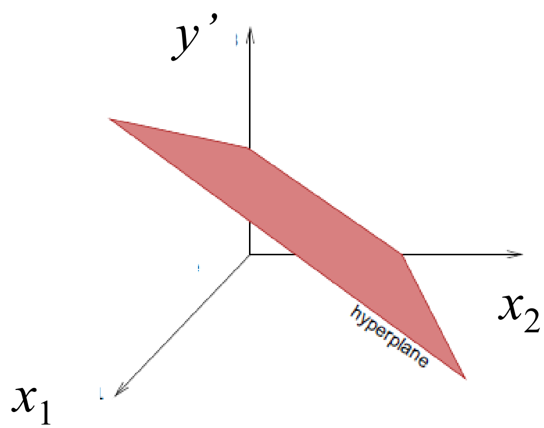
$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ -\theta \end{bmatrix} \qquad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

Linear separability with TLU – geometrical interpret.

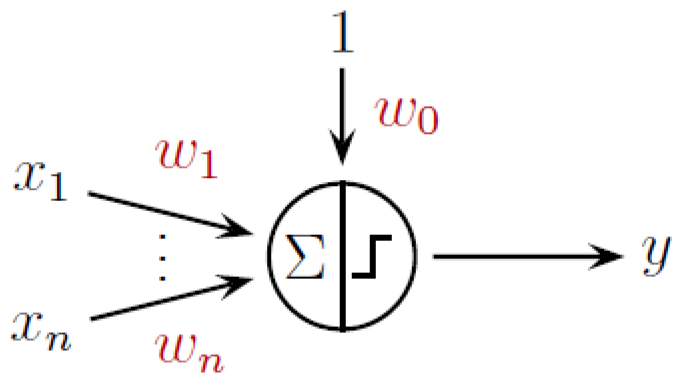


$$y' = w_1 x_1 + w_2 x_2 - \theta$$

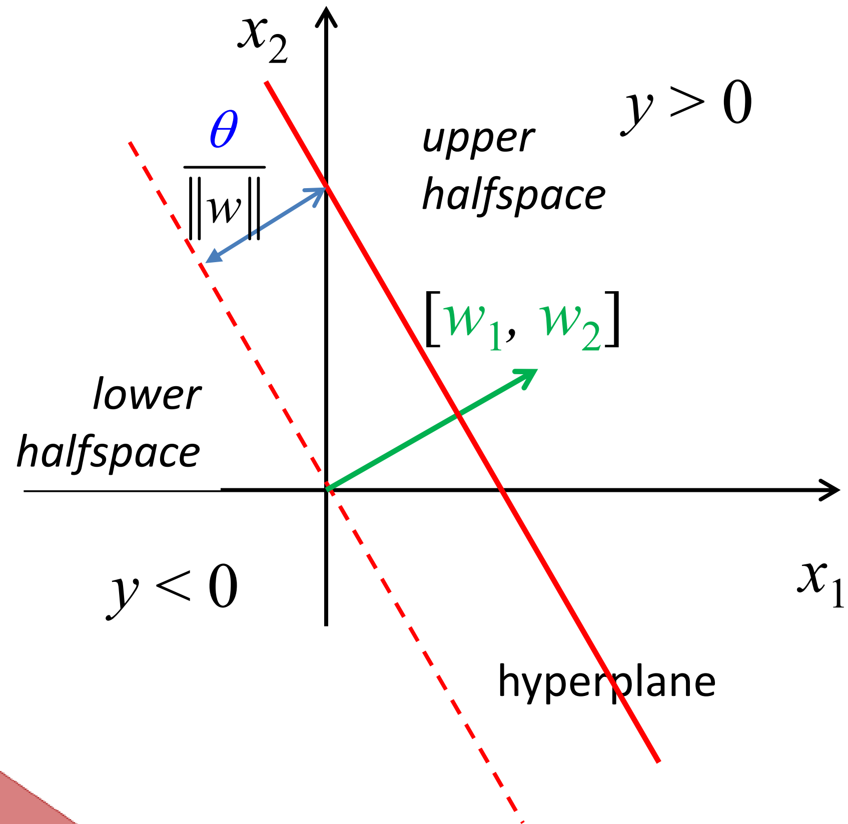


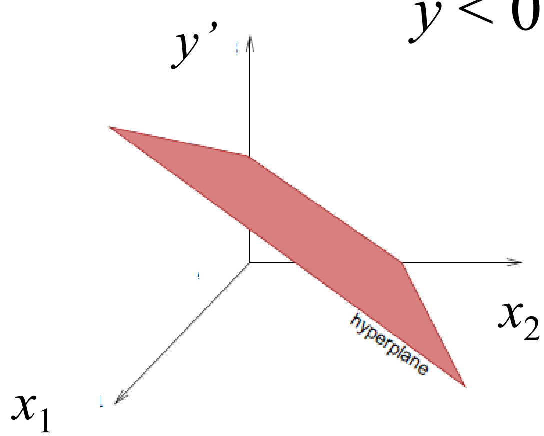


Linear separability with TLU – geometrical interpret.

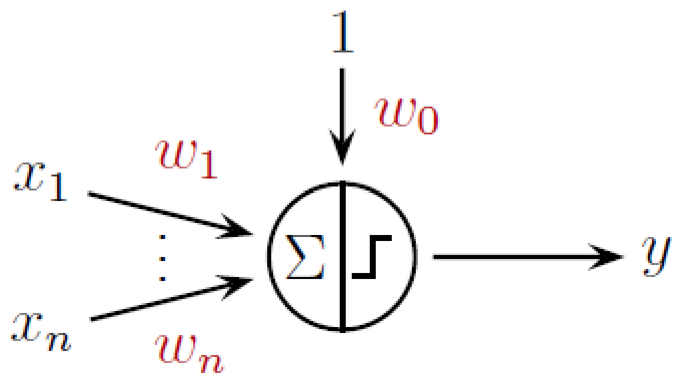


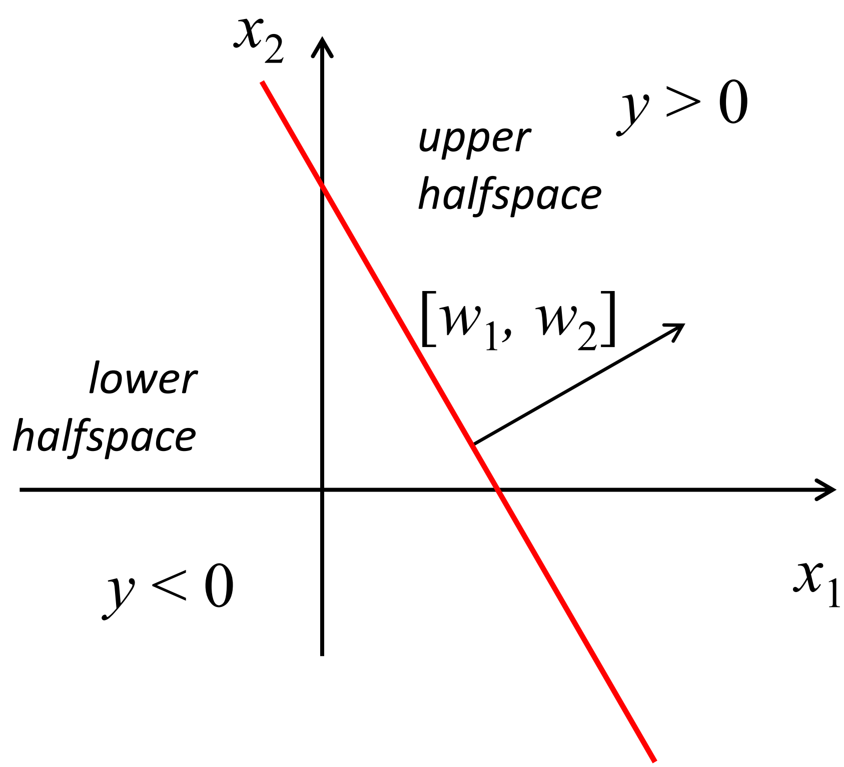
$$y' = w_1 x_1 + w_2 x_2 - \theta$$



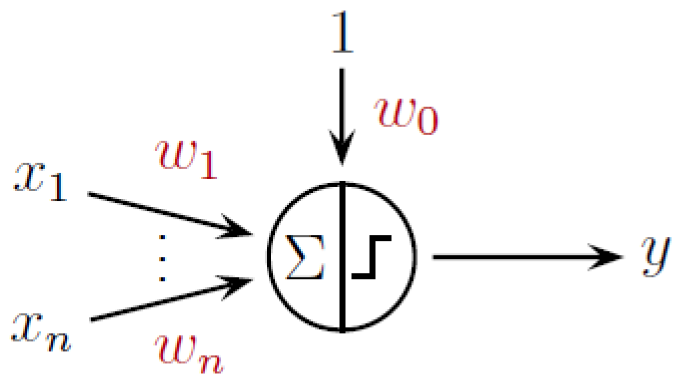


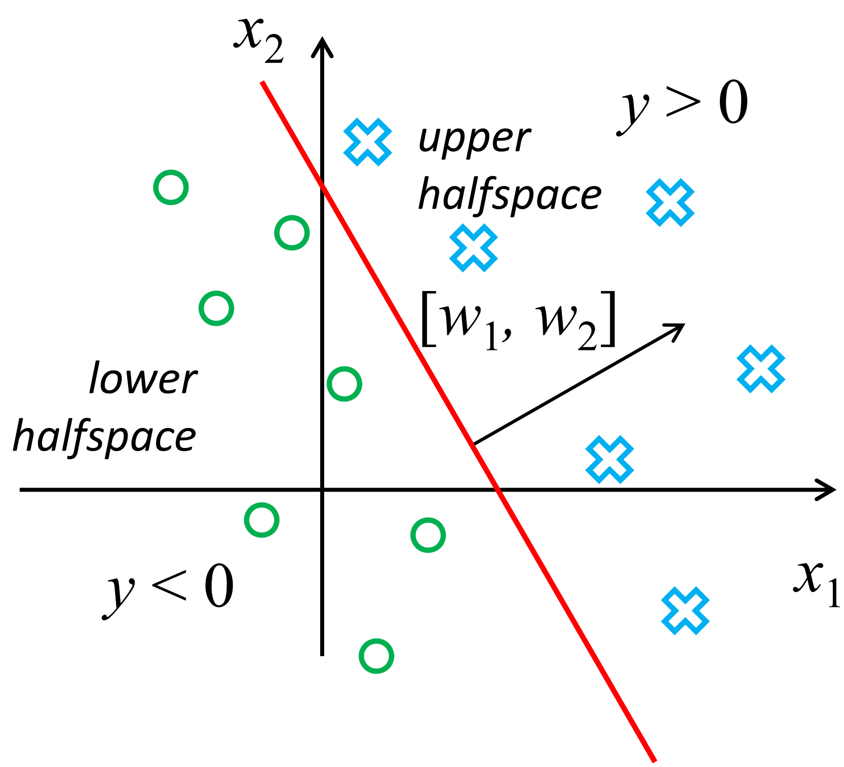
Binary classification with perceptron



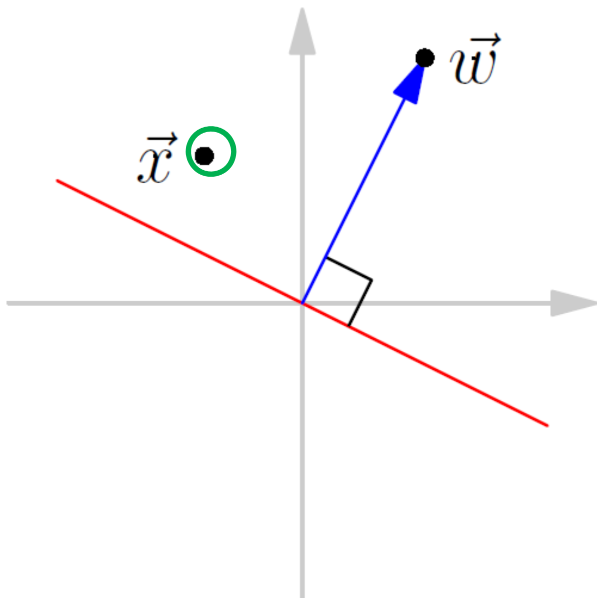


Binary classification with perceptron

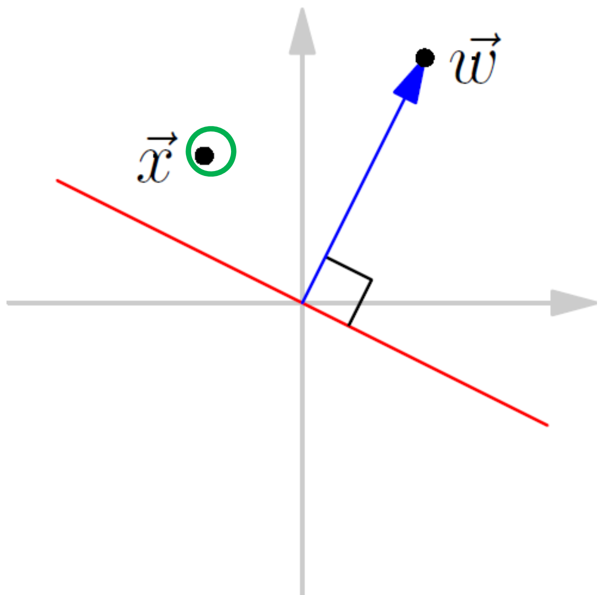


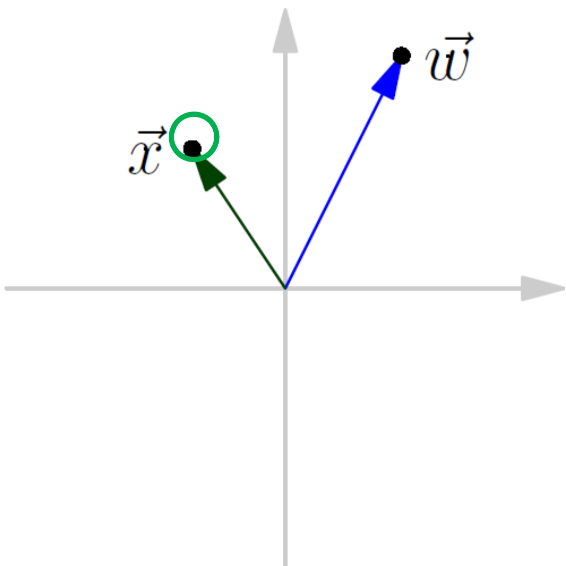


Space of weights and inputs - perceptron



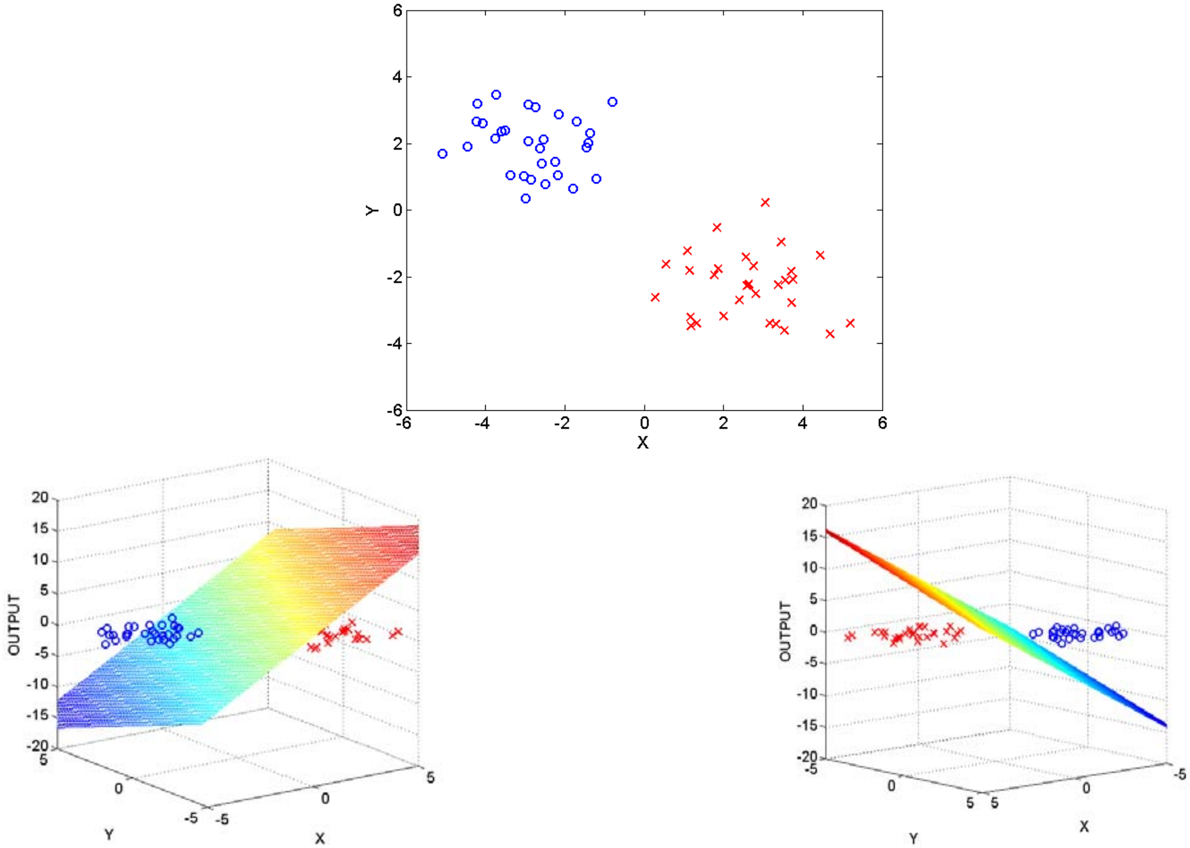
Space of weights and inputs - perceptron





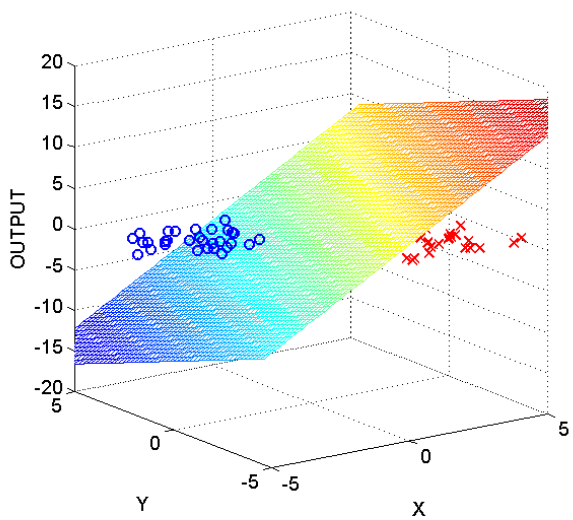
Classification with perceptron – how does it work?

2D input space and 3D network's linear output

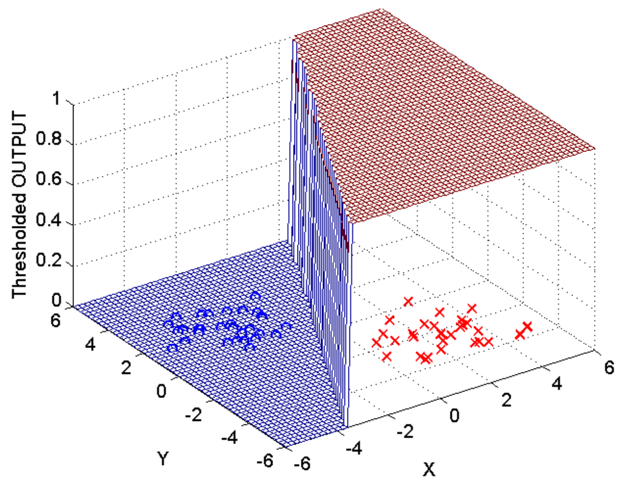


Classification with perceptron

Linear output and
perceptron's thresholded output



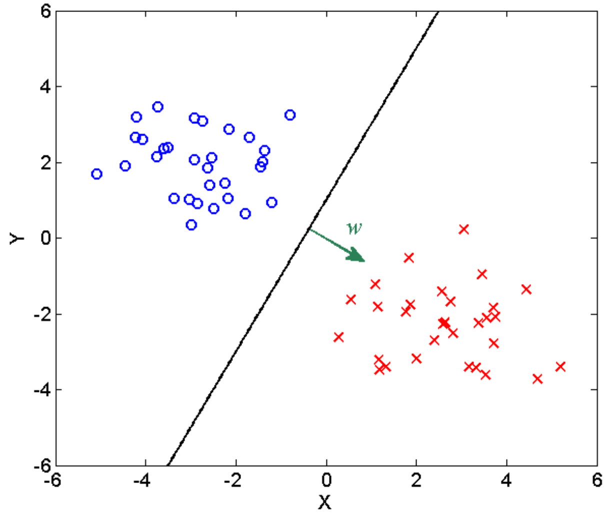
Separating hyperplane – network's linear output

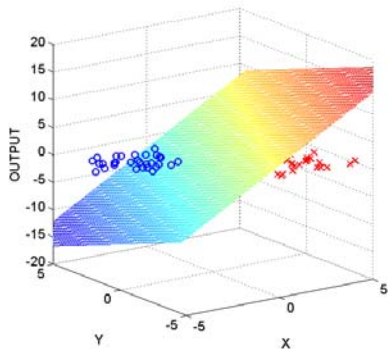


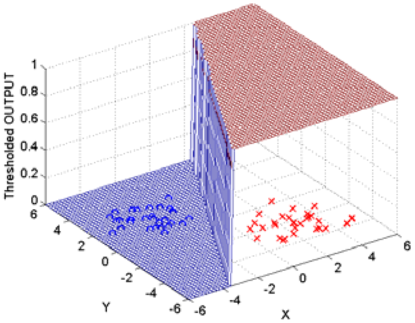
Output surface – percpetron's thresholded output

Classification with perceptron

Decision boundary in the input space







Perceptron learning for classification

Perceptron learning for thresholded single-layer networks

Basic principle: weights are modified if and only if a pattern is erroneously classified:

When the network *output* = 0 but it should be 1 (*target* = 1)

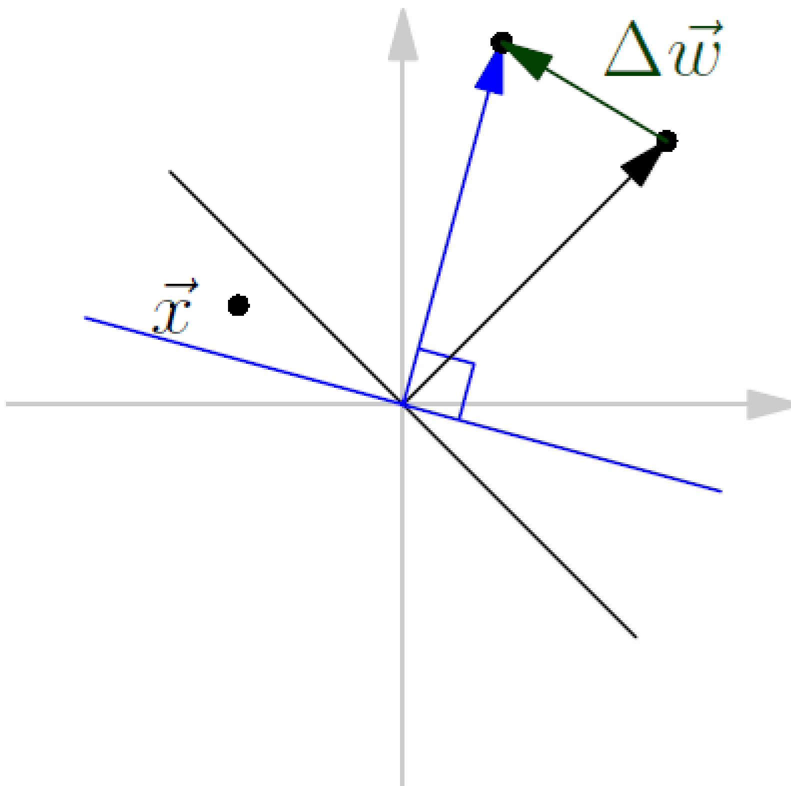
$$\Delta\vec{w} = \eta\vec{x}$$

When the network *output* = 1 but it should be 0 (*target* = 0)

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

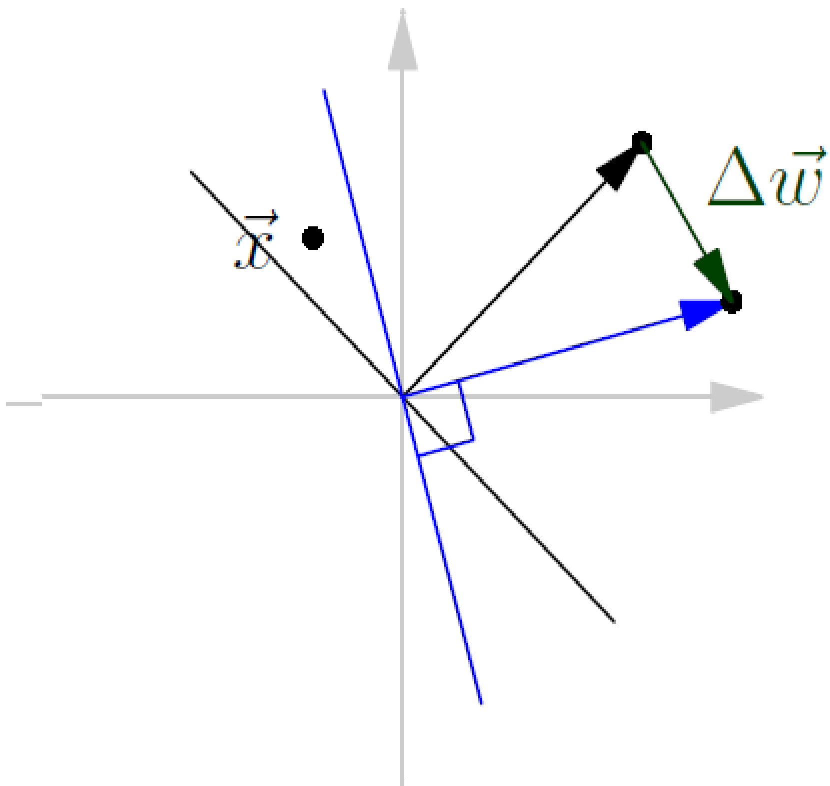
Perceptron learning – geometrical interpretation

When the result is 0 but should be 1: $\Delta\vec{w} = \eta\Delta\vec{x}$



Perceptron learning – geometrical interpretation

When the result is 1 but should be 0: $\Delta\vec{w} = -\eta\Delta\vec{x}$



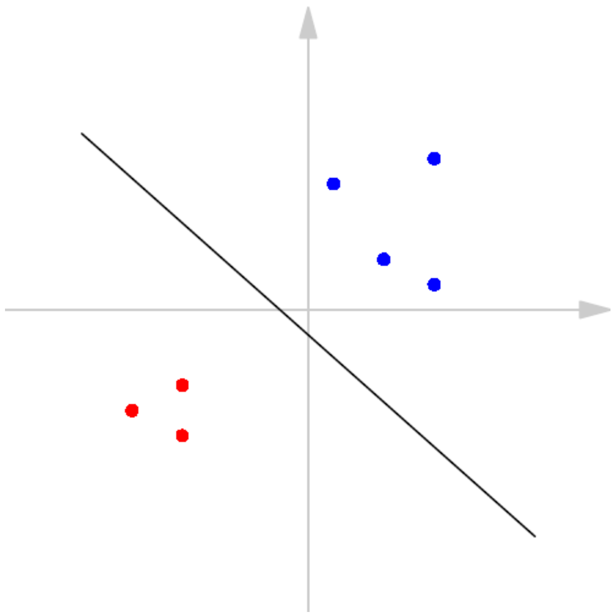
Perceptron learning – convergence theorem

Convergence theorem

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

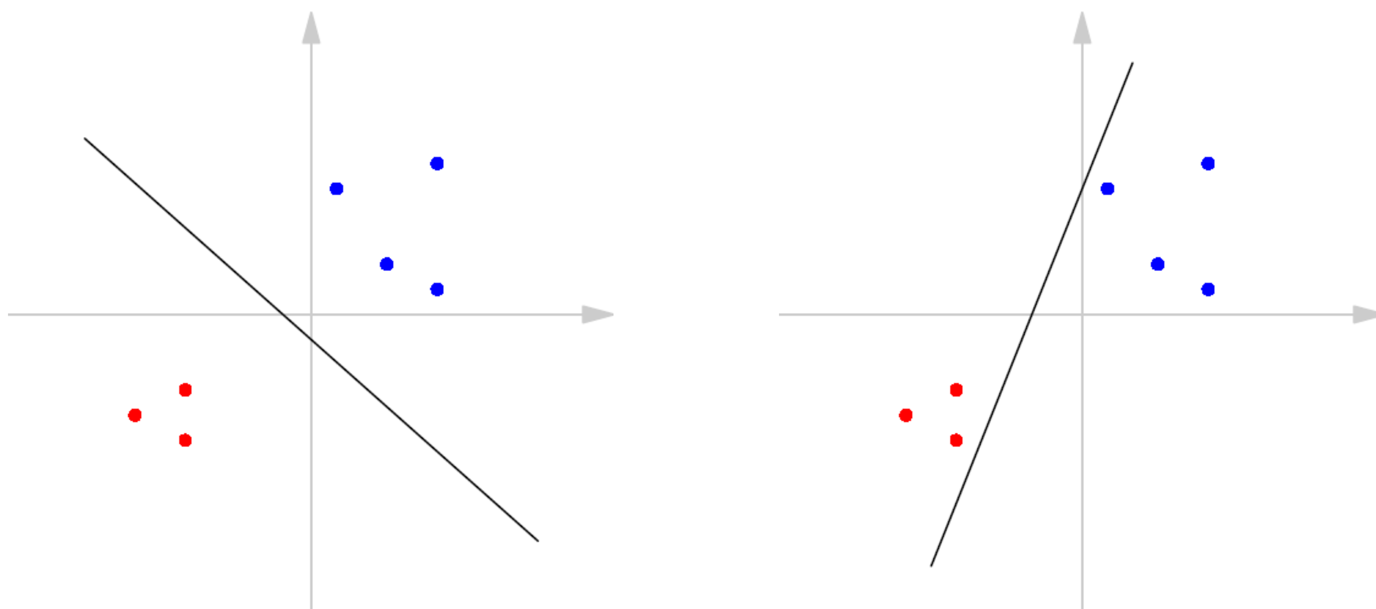
Perceptron learning

Problem: learning terminates prematurely.



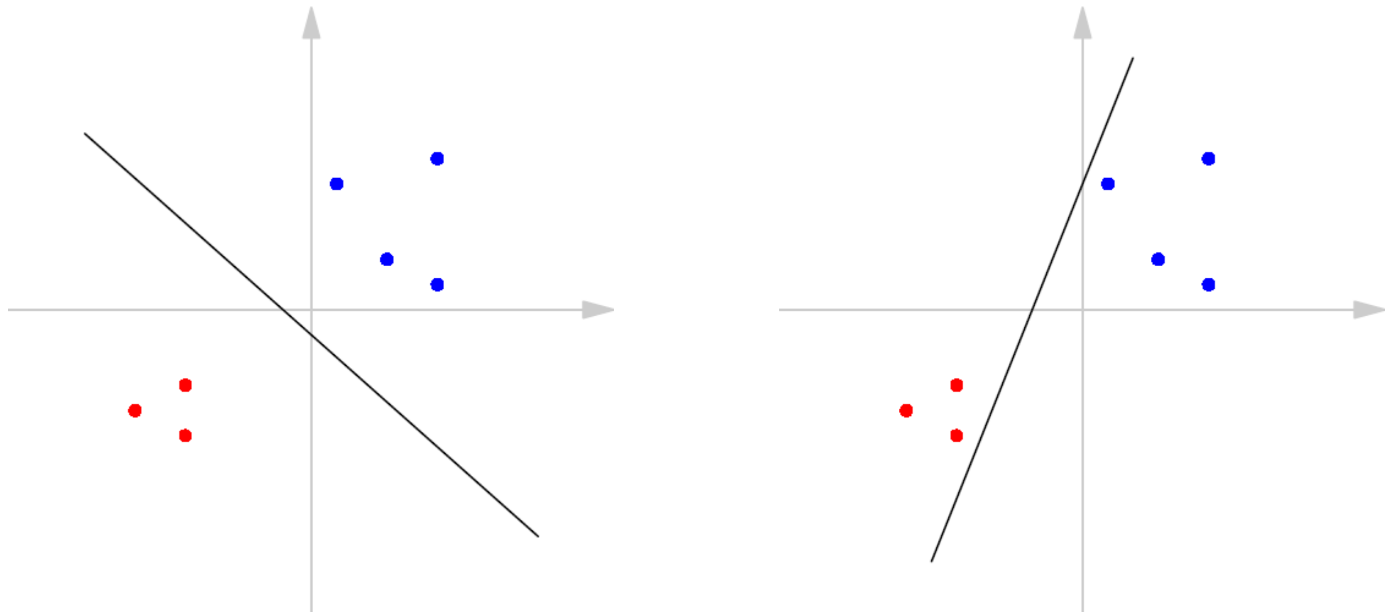
Perceptron learning

Problem: learning terminates prematurely.



Perceptron learning

Problem: learning terminates prematurely.



Negative consequences are likely when patterns are only *approximately similar* to those used for training

Delta rule

Delta rule (Widrow-Hoff rule, ADALINE)

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**

Delta rule

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

Simple algorithm: **steepest descent**

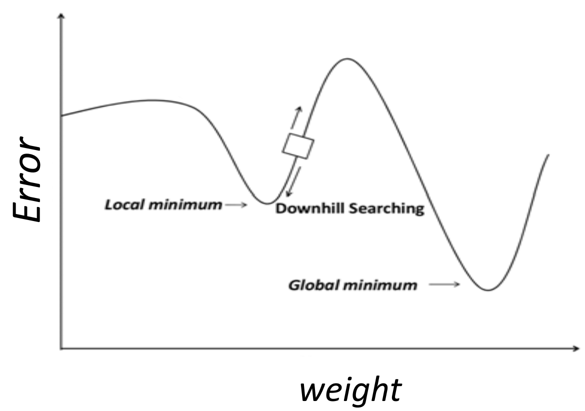
- Gradient defines the direction in which the error increases most
- *Steepest descent* implies that the move in the opposite direction in the weight space should be taken

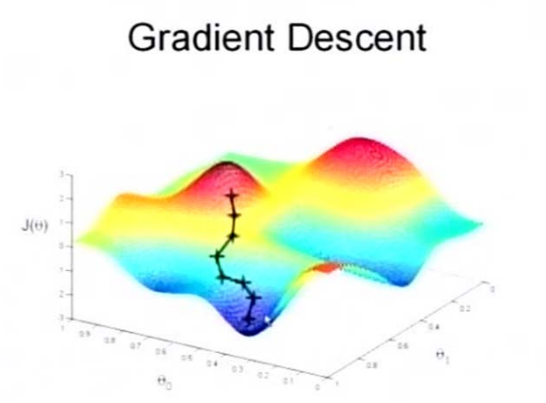
Delta rule

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 descent* implies that the move in the opposite direction in the weight space should be taken





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 descent* implies that the move in the opposite direction in the weight space should be taken
- Gradient is calculated as follows:

$$\frac{\partial \varepsilon}{\partial \vec{w}} = e\frac{\partial e}{\partial \vec{w}} = e\frac{\partial (t - \vec{w}^{\mathrm{T}}\vec{x})}{\partial \vec{w}} = -e\vec{x}$$

Delta rule

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

Simple algorithm: **steepest descent**

- Gradient defines the direction in which the error increases most
- *Steepest descent* implies that the move in the opposite direction in the weight space should be taken
- Gradient is calculated as follows:

$$\frac{\partial \varepsilon}{\partial \vec{w}} = e\frac{\partial e}{\partial \vec{w}} = e\frac{\partial (t - \vec{w}^{\mathrm{T}}\vec{x})}{\partial \vec{w}} = -e\vec{x}$$

Delta Rule:

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

Training of thresholded single-layer networks

Perceptron learning:

Delta rule:

Training of thresholded single-layer networks

Perceptron learning:

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

Delta rule:

Training of thresholded single-layer networks

Perceptron learning:

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

Delta rule:

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

Separability with TLU / perceptron

Can all sets of patterns be separated?

Separability with TLU / perceptron

Can all sets of patterns be separated?

Classical counter-example is Exclusive OR (XOR)

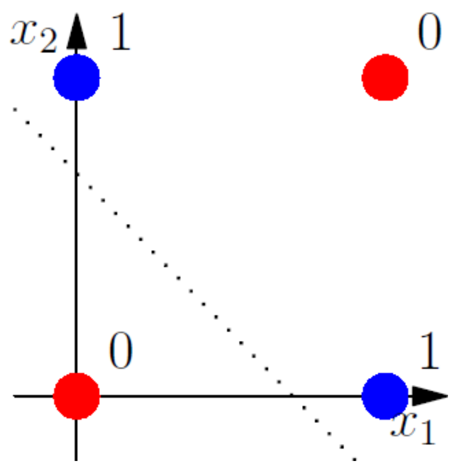
$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow 0 \qquad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow 1 \qquad \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow 1 \qquad \begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow 0$$

Separability with TLU / perceptron

Can all sets of patterns be separated?

Classical counter-example is Exclusive OR (XOR)

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

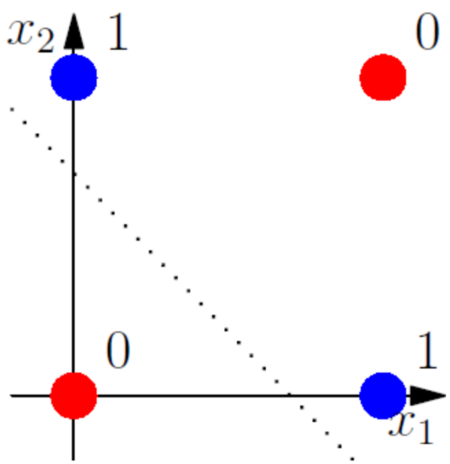


Separability with TLU / perceptron

Can all sets of patterns be separated?

Classical counter-example is Exclusive OR (XOR)

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



Not linearly separable!