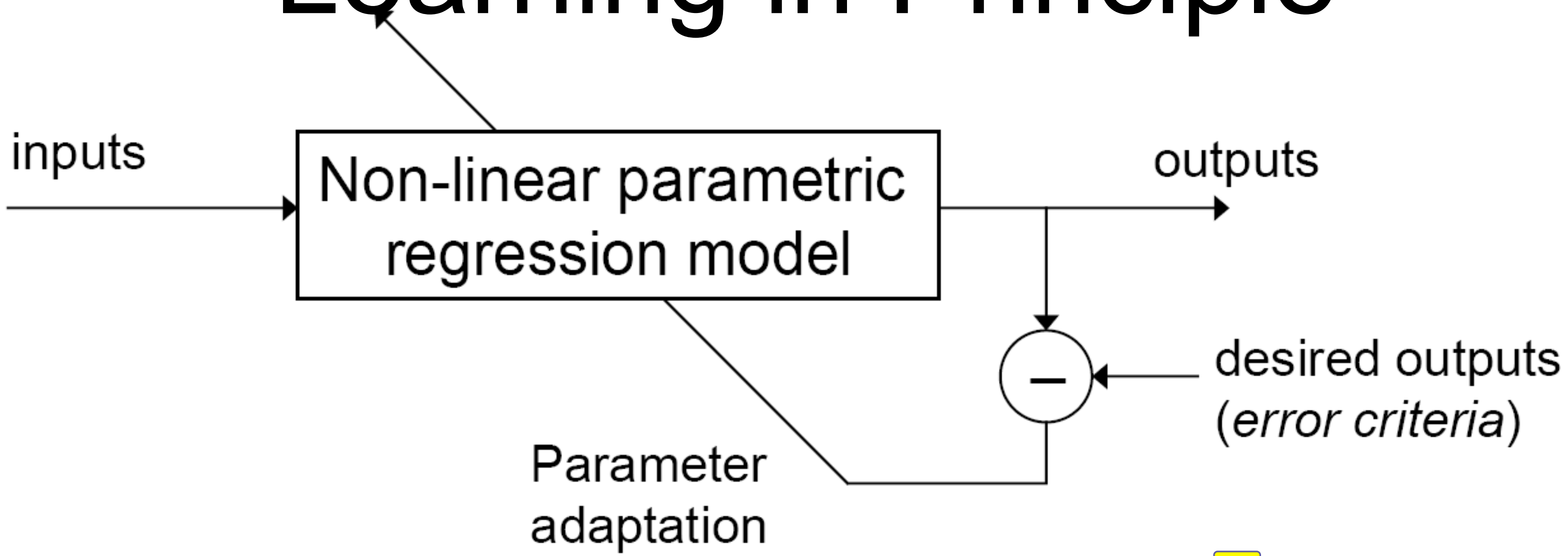
# Outline

- First Example for Learning: Classification by a Perceptron
- Second Example: Regression by a Perceptron
- Perceptron Convergence Theorem
- Definition: ANN learning
- 1st Learning rule
- … and four more Learning rules

# Learning in Principle




- ***Fix a Model => restrictions about the possible relation***
- ***Be parametric => fix the learning parameters to adjust (i.e. they "contain" the information)***
- ***Choose non-linear => more powerful == expressive than linear ones***
- ***NN's are a Universal approximations***

# Hands-on Example I: Perceptron learning

(1)Task : classification

(2)Network : perceptron (stepfunction)

(3)Learning: error correction

# (1) Task: Classifiy

- The perceptron should classify correctly a set of examples into one of the two classes C1, C2.

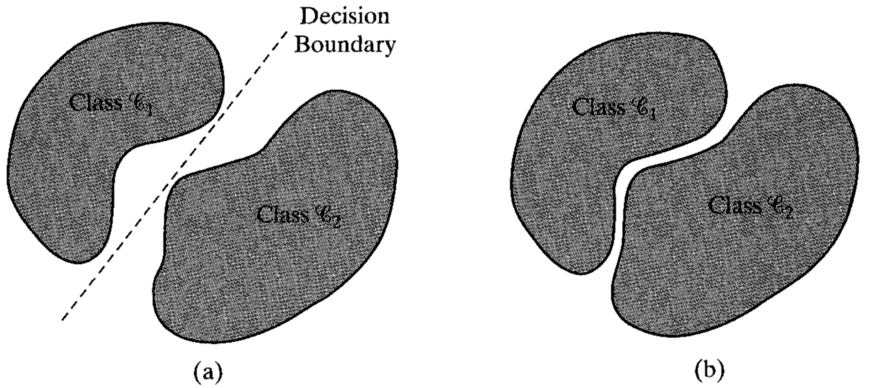
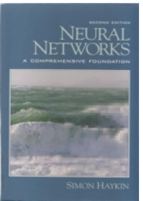



FIGURE 1.4 (a) A pair of linearly separable patterns. (b) A pair of non-linearly separable patterns.

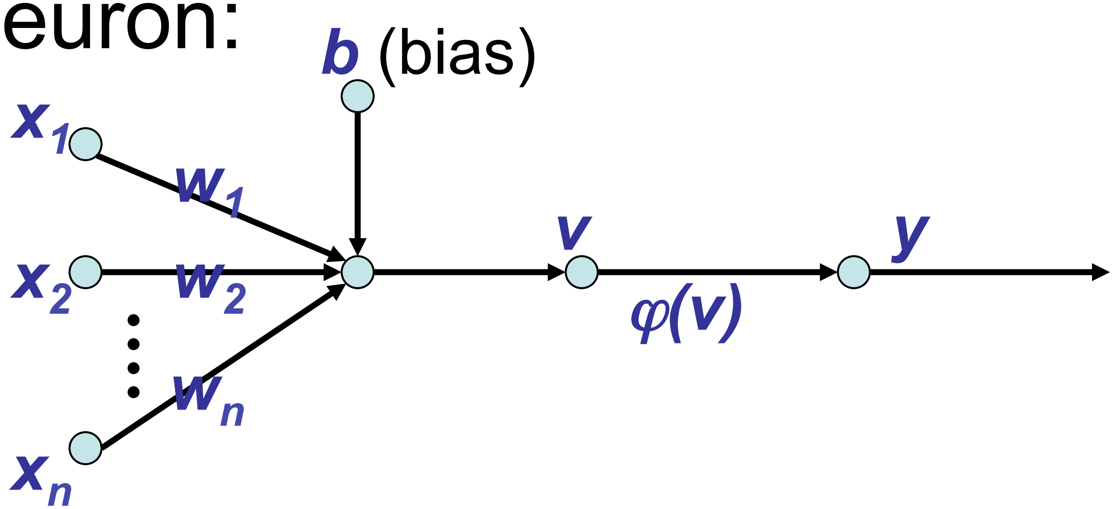
***If the output of the perceptron is +1 then the input is assigned to class $C_1$***

***If the output is 0 then the input is assigned to $C_2$***



# (2) Network: a single Perceptron

- Use a non-linear (McCulloch-Pitts) model of neuron:



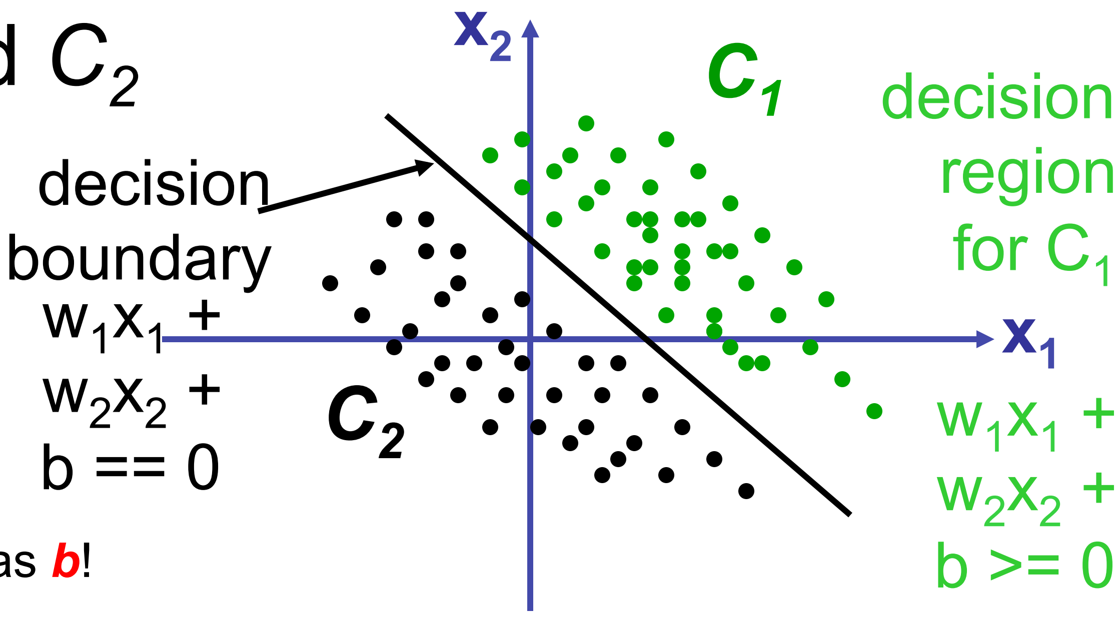

- φ is the step function :

$$\varphi(v) := \begin{cases} +1 & \text{if } v >= 0 \\ 0 & \text{if } v < 0 \end{cases}$$

# (2)Network: Output of Perceptron

- The equation below describes a hyperplane in the input space. This hyperplane is used to separate the two classes $C_1$ and $C_2$

$$\sum_{i=1}^{m} w_i x_i + b = 0$$




Wanted are: vector ***w*** and the bias ***b***!

# (3) Learning: Perceptron error

Test problem:

Let the set of training examples be

$x_1 = [1 ; 2]; d_1 = 1$

$x_2 = [-1; 2]; d_2 = 0$

$x_3 = [0 ;-1]; d_3 = 0$

"d"== desired

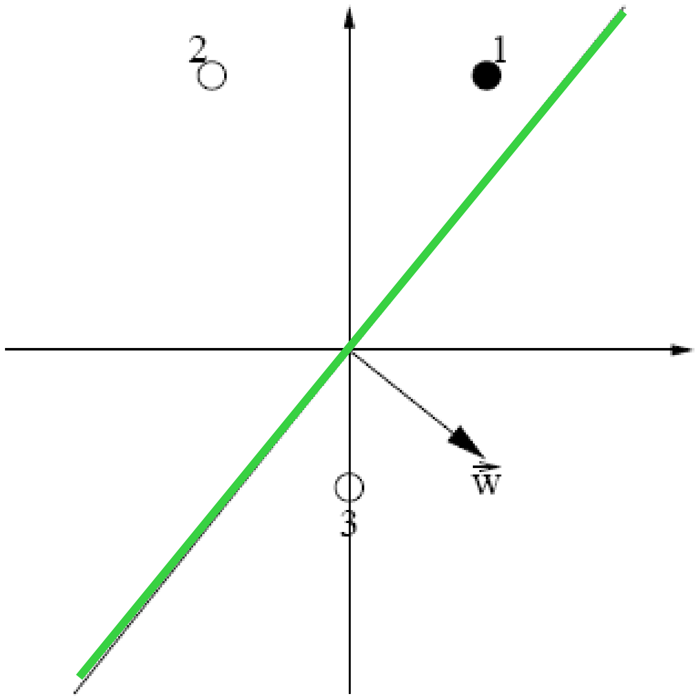
error = d - $y_{current}$

Let bias be $b$ = 0,learning rate $\eta$=1.

Let initial weight vector be

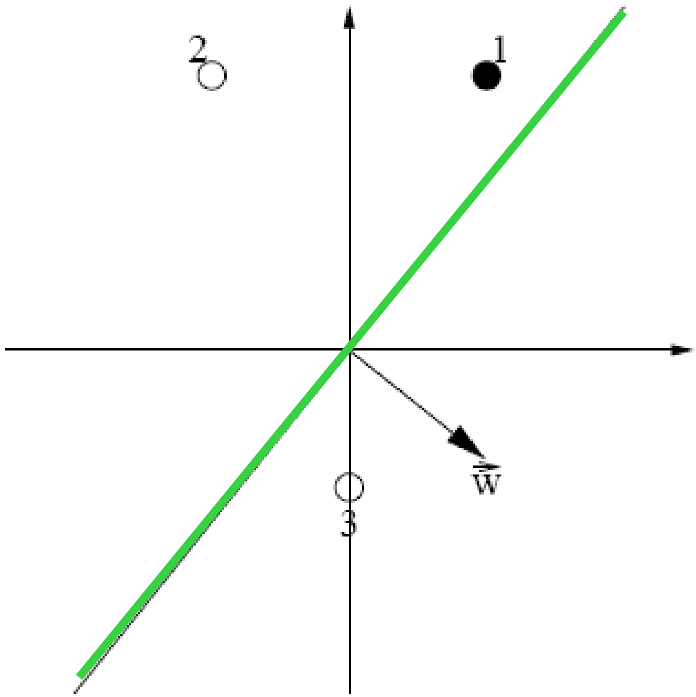
**w = [1; -0.8]**

We want to obtain a learning algorithm that finds a weight vector *w* which will correctly classify (separate) the given examples.



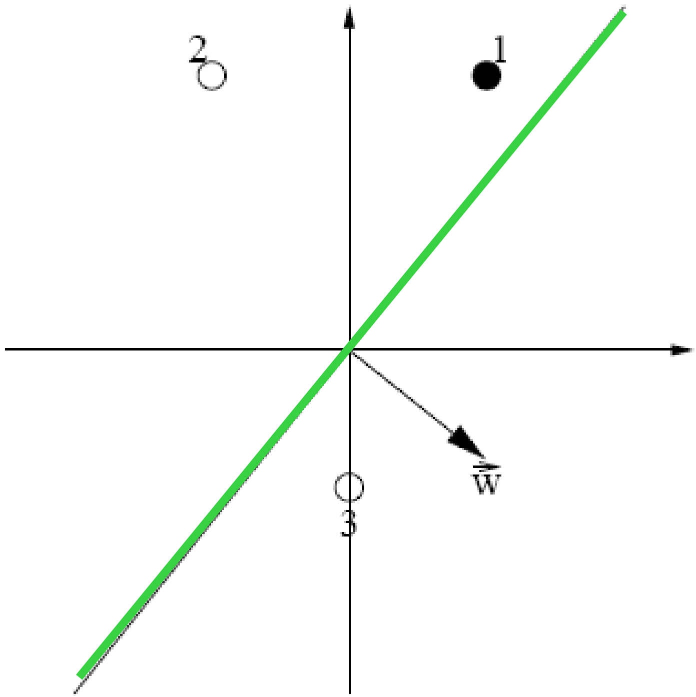
# (3) Learning: Perceptron error …

- First input $x_1$ is misclassified with positive error. What to do?



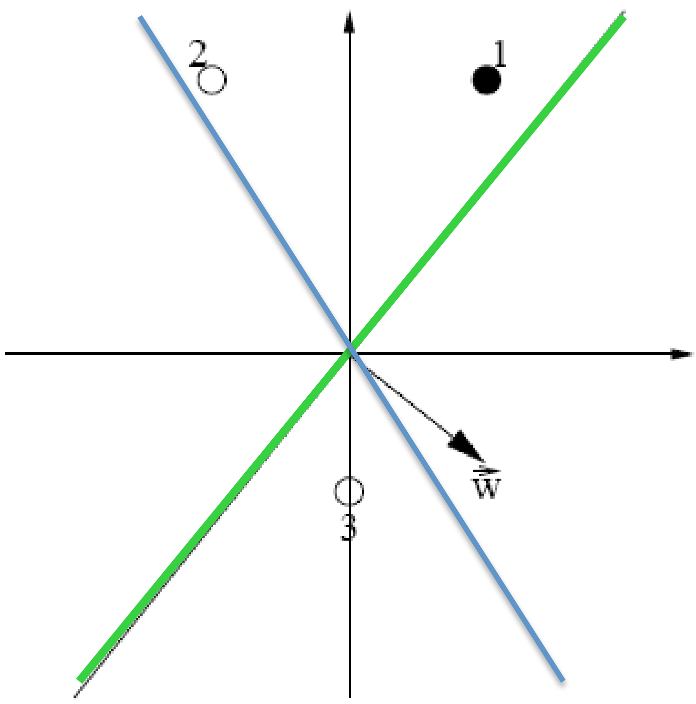
# (3) Learning: Perceptron error …

- First input $x_1$ is misclassified with positive error. What to do?
- Idea: move hyperplane towards separating position



# (3) Learning: Perceptron error …

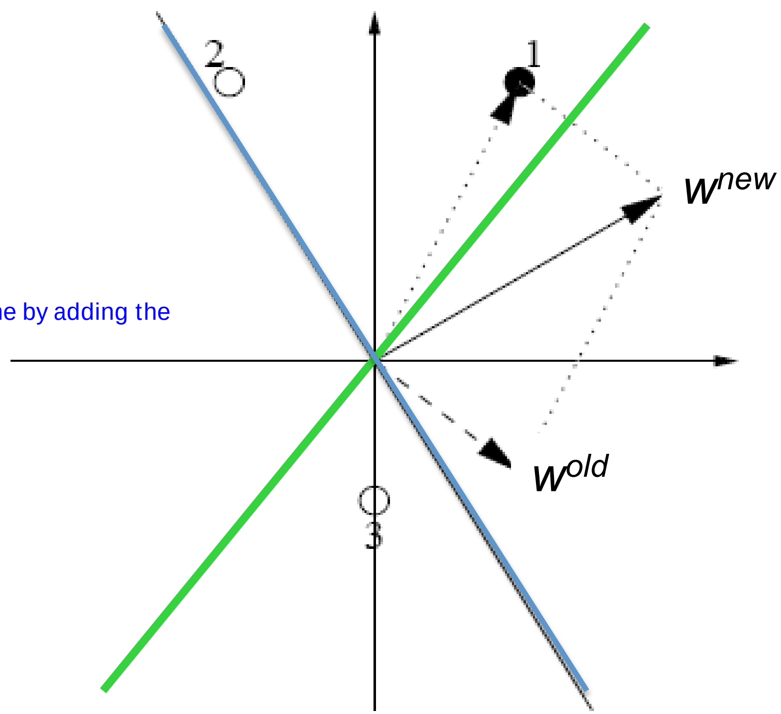
- First input $x_1$ is misclassified with positive error. What to do?
- Idea: move hyperplane towards separating position



# (3) Learning: Perceptron error …

- First input $x_1$ is misclassified with positive error. What to do?
- Idea: move hyperplane towards separating position

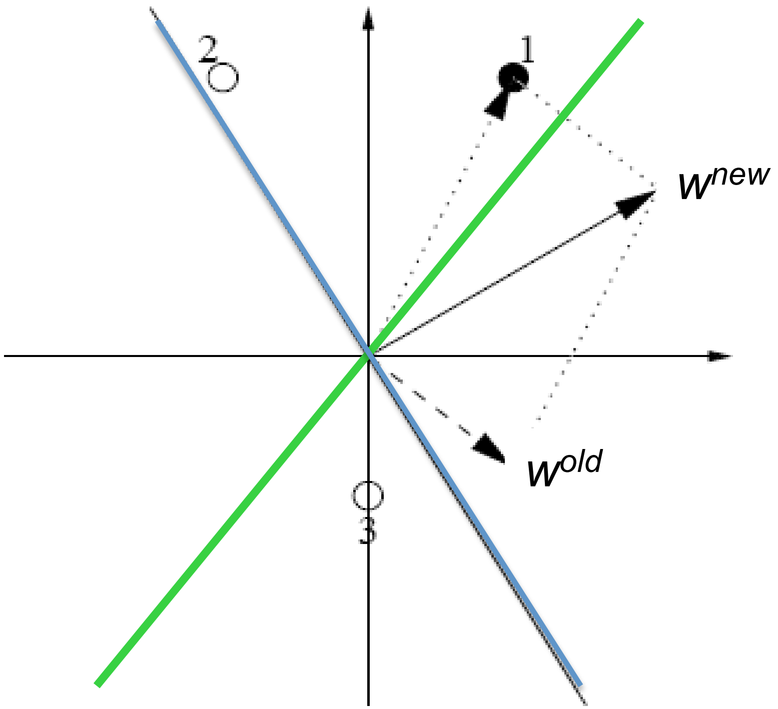
we get to the blue line by adding the new data



# (3) Learning: Perceptron error …

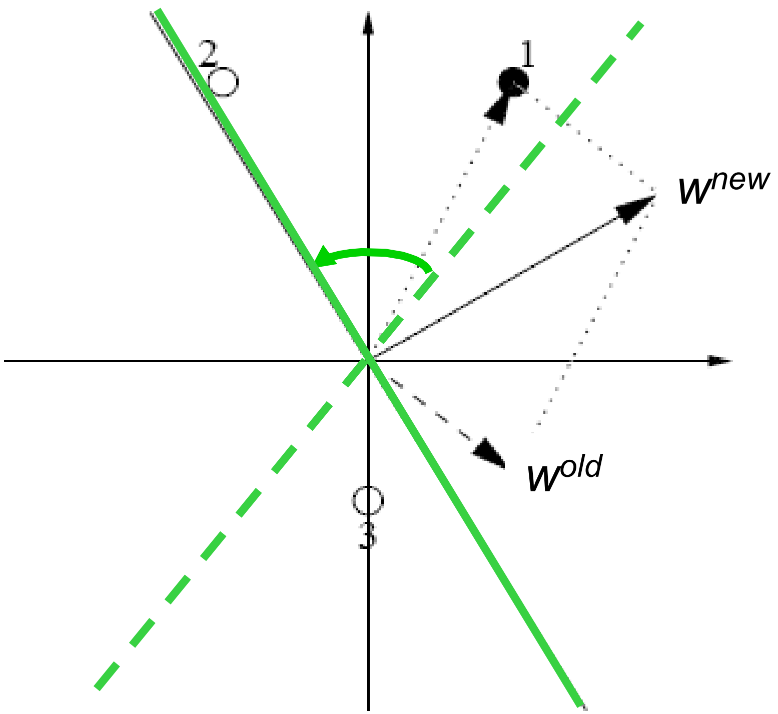
- First input $x_1$ is misclassified with positive error. What to do?
- Idea: move hyperplane towards separating position
- So:

- To move $w$ closer to $x_1$:
  add $x_1$ to $w$
  $w^{new} = w + x_1$



# (3) Learning: Perceptron error …

- First input $x_1$ is misclassified with positive error. What to do?
- Idea: move hyperplane towards separating position
- So:

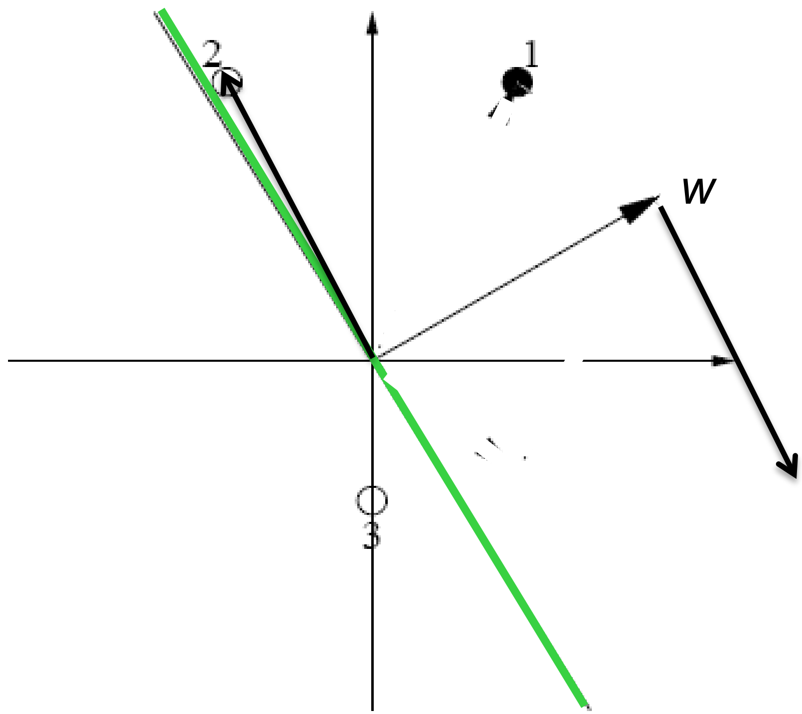



- To move $w$ closer to $x_1$:
  add $x_1$ to $w$
  $w^{new} = w + x_1$
- **First rule:**
  ***positive error rule (err=(d-y)>0)***
  If $d = 1$ and $y = 0$ then
  $w^{new} = w^{old} + x$

# (3) Learning: Perceptron error …

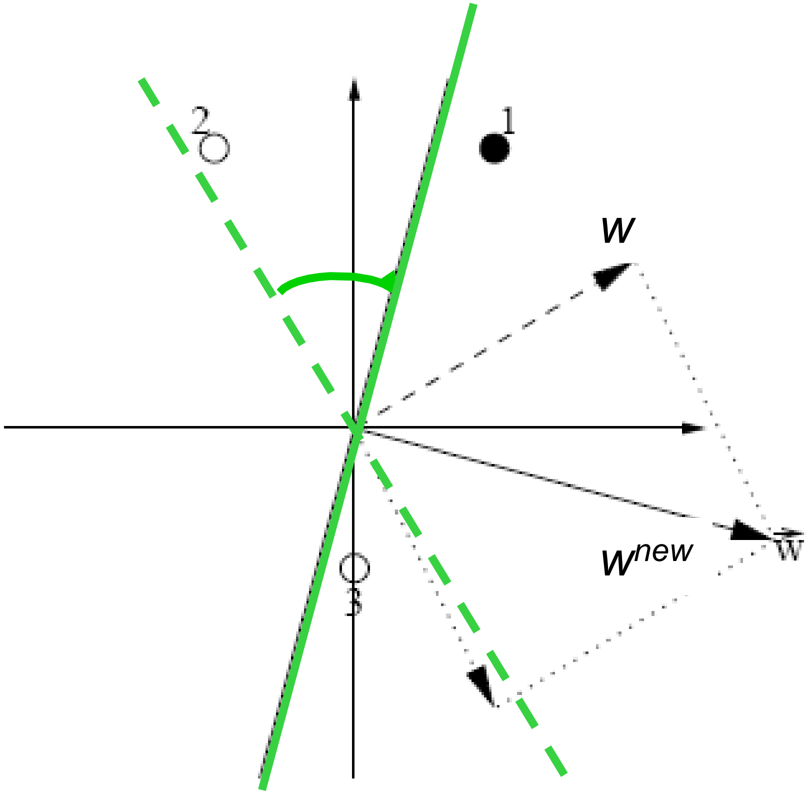
- Now input $x_2$ is misclassified with negative error. What to do?
- Idea: move hyperplane away from separating position
- So:

- To move $w$ away from $x_2$: subtract $x_2$ to $w$
  $w^{new} = w - x_2$



# (3) Learning: Perceptron error …

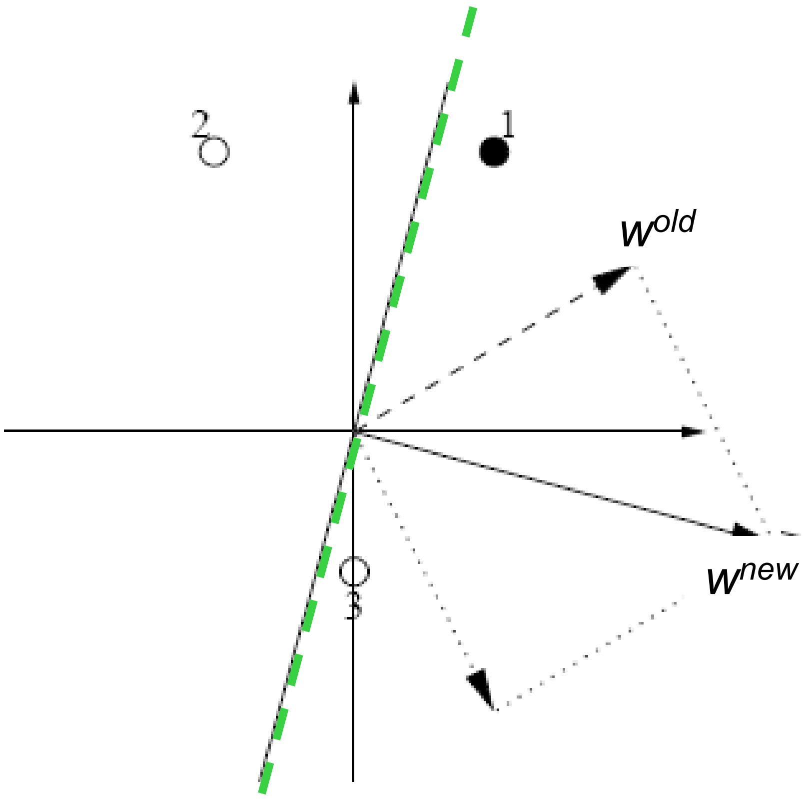
- Now input $x_2$ is misclassified with negative error. What to do?
- Idea: move hyperplane away from separating position
- So:




- To move $w$ away from $x_2$: subtract $x_2$ to $w$
  $w^{new} = w - x_2$
- **Second rule:**
  ***negative error rule (d-y)<0:***
  If $d = 0$ and $y = 1$ then
  $w^{new} = w^{old} - x$

# (3) Learning: Perceptron error …

- Third input $x_3$ is misclassified with negative error
- So




- To move w away from $x_3$: subtract $x_3$ from w

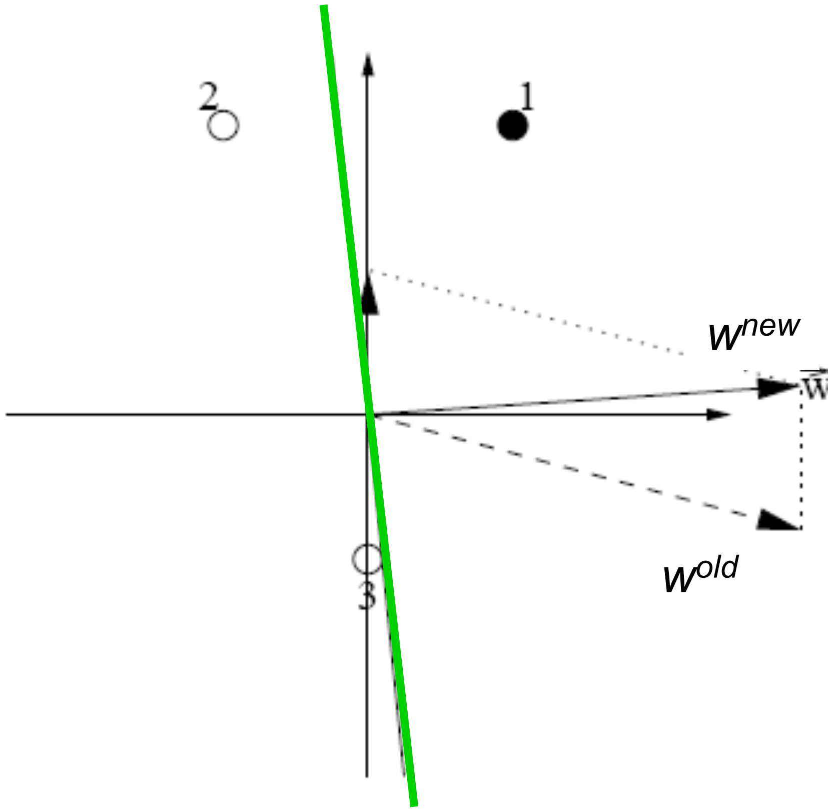
  $w^{new} = w - x_3$

- **Second rule:**
  **negative error rule: (d-y)<0**

  If d = 0 and y = 1 then
  $w^{new} = w^{old} - x$

# (3) Learning: Perceptron error …

- Finally: no data misclassified




- The perceptron will now correctly classify all inputs $x_1$, $x_2$, $x_3$ if presented to it again. There are no errors
- **Third rule: no error rule**
- If d = y then

  $w^{new} = w^{old}$ .

# (3) Learning: Perceptron error …

- Unified learning rule :
  $w^{new} = w^{old} + (d - y)\, x$
- Using a learning rate $\eta$ :
  $w^{new} = w^{old} + \eta\, (d - y)\, x$

we do not completely add or remove the data according to the error, we use the learning rate to determine how much to shift the line.
learning rate implicitly normalizes the weights. this affects by how much the line shifts

- Choice of learning rate $\eta$

  too large => learning oscillates
  too small => very slow learning
- $0 < \eta <= 1$, popular choices:
  $\eta = 0.5$
  $\eta = 1$
- Variable learning rate
  $\eta = |w.*x| / |x.*x|$
- Adaptive learning rate

- Perceptron convergence theorem: [M. Minsky and S. Papert, 1969]
- The perceptron learning algorithm terminates if and only if the task is linearly separable
- Cannot learn non-linearly separable functions
- Observe: the learning rules adapted the weight vectors in a way derived by „human“ insight

# Hands-on Example II: Adaline learning

- Task : regression
- Network : **ADA**ptive **LIN**ear **E**lement

  (linear output activation function,
  “perceptron” with a linear output element) (ADALINE)
- Learning: error correction
  will in contrast be **derived (!)**
  from steepest gradient (=> LMS)

# Adaline: Problem Description

- Data fitting
  (or linear regression)
  - Given a set of data
    (e.g. from measurements)
    $\{ [x_i, d_i] \}$
  - Find ***w*** and ***b*** such that
    $d_i \sim\sim wx_i + b$
  - or more precisely
  - $d_i = wx_i + b + \varepsilon_i = y_i + \varepsilon_i$

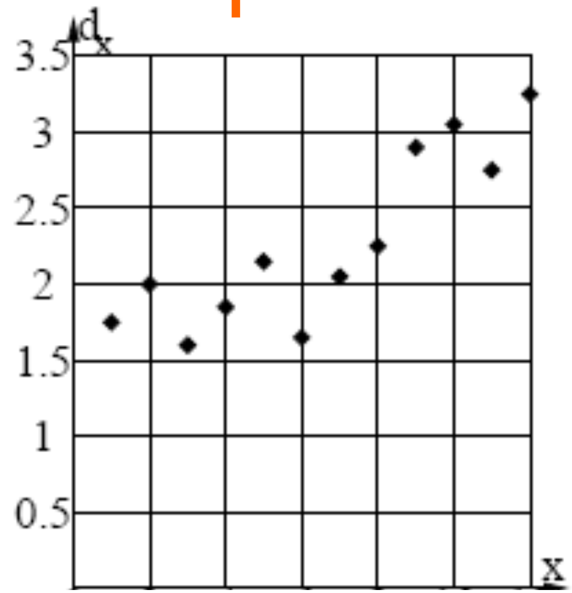
  where

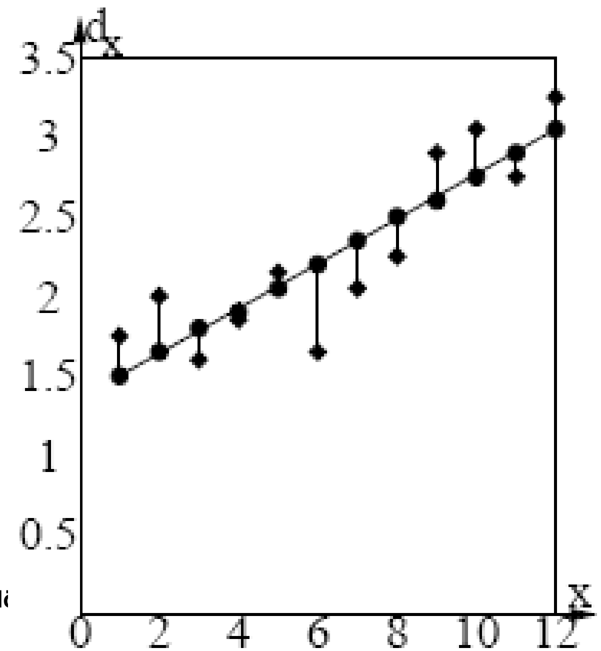
  $\varepsilon_i$ = instantaneous error

  $y_i$ = linearly fitted value

  ***w*** = line slope

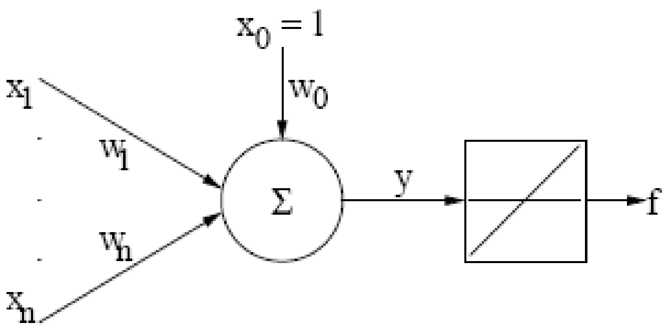
  ***b*** = d-axis intercept (or bias)
- Best fit problem:
  find the best choice of
  (***w***,***b***) such that the fitted line
  passes closest to all points
- Solution: Least mean square



# Adaline: Architecture

- Adaline: uses a linear neuron model and the Least-Mean-Square (LMS) learning algorithm
- The idea: try to minimize the mean square error, which is a function of the weights
- We may determine the minimum of the error function ***E*** by means of the steepest descent method




- Architecture:
  - Input: $\vec{x} = (x_0 = 1, x_1, \ldots, x_n)$
  - Weights (with bias $\theta$): $\vec{w} = (w_0 = -\theta, w_1, \ldots, w_n)$
  - Net input: $y = \vec{x}\vec{w} = \sum_{i=0}^{n} x_i w_i$
  - Output: $f(\vec{x}) = \vec{x}\vec{w}$
- Learning Task: regression
- Learning Type: Error-correction learning (Supervised learning)

# Idea ADALINE learning

- Uses a linear output neuron model
- The idea: look at the current squared error
- viewed as a function of the weights
- Minimize now by building derivative (possible!)
- Search miminimum of the error function ***E(w(n))*** by means of the steepest descent
- Least-Mean-Square (LMS) based learning algorithm

- ***y= φ(x) = w*x + b***

$$\boxed{E(\mathrm{w}(\mathrm{n})) = \frac{1}{2}\varepsilon^2(\mathrm{n})}$$

$$\boxed{\varepsilon(\mathrm{n}) = \mathrm{d}(\mathrm{n}) - \sum_{\mathrm{j}=0}^{\mathrm{m}} \mathrm{x}_{\mathrm{j}}(\mathrm{n})\mathrm{w}_{\mathrm{j}}(\mathrm{n})}$$

the cost function helps build the error function.

# ADALine: Learning Algorithm by Gradient descent

$$given\ a\ learning\ example: [x_i, d_i]$$

$$an\ observed\ output: y_i = \ldots$$

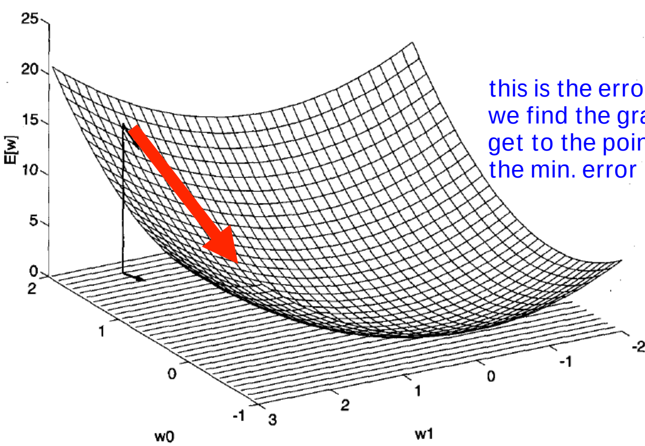
$$squared\ error: \varepsilon_i = (d_i - y_i)^2$$

$$gradient\ of\ \varepsilon_i:$$

$$\nabla \varepsilon_i = \frac{\partial \varepsilon_i}{\partial \vec{w}} =$$

$$\left( \frac{\partial \varepsilon_i}{\partial w_0}, \frac{\partial \varepsilon_i}{\partial w_1}, \ldots, \frac{\partial \varepsilon_i}{\partial w_n}, \right)$$

$$if\ \varepsilon_i\ minimal \Rightarrow \nabla \varepsilon_i = 0$$




this is the error function,
we find the gradient of this function to
get to the point where the weights have
the min. error

ereich
iatik

- Negative gradient of $\boldsymbol{\varepsilon_i}$ gives direction of steepest descent to the minimum $\boldsymbol{-\nabla\varepsilon_i}$
- Then do gradient descent

$$\Delta \vec{w} = -\eta \nabla \varepsilon_i = -\eta \frac{\partial \varepsilon_i}{\partial \vec{w}}$$

the gradient usually points in the upward direction that is why we have a -ve
sign in the formula to move in the other direction.

$$Widrow-Hoff\ delta\ rule$$

$$(for\ component\ k):$$

$$\frac{\partial \varepsilon_i}{\partial w_k} = \frac{\partial (d_i - y_i)^2}{\partial w_k} = 2(d_i - y_i)\frac{\partial(-y_i)}{\partial w_k}$$

$$= 2(d_i - y_i)\frac{\partial(-\sum_{l=0}^{n} w_l x_l)}{\partial w_k} = -2(d_i - y_i)x_k$$

$$Final\ overall\ learning\ rule:$$

$$\vec{w}^{new} = \vec{w}^{old} - (-\eta(d_i - y_i)\vec{x})$$

Example

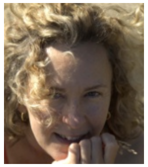
# Steepest Descent Method

- start with an arbitrary point
- find a direction in which E is decreasing most rapidly

$$\boxed{-(\text{gradient of } E(\mathrm{w})) = -\left[\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m}\right]}$$

- make a small step in that direction

$$\boxed{\mathrm{w}(\mathrm{n}+1) = \mathrm{w}(\mathrm{n}) + \eta(\text{gradient of } \mathrm{E}(\mathrm{n}))}$$



Example

# *L*east-*M*ean-*S*quare algorithm (Widrow-Hoff algorithm)

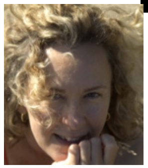
- Approximation of gradient(E)

$$\frac{\partial E(\mathrm{w}(\mathrm{n}))}{\partial \mathrm{w}(\mathrm{n})} = \varepsilon(\mathrm{n}) \frac{\partial \varepsilon(\mathrm{n})}{\partial \mathrm{w}(\mathrm{n})}$$

$$= \varepsilon(\mathrm{n})[-\mathrm{x}(\mathrm{n})^{\mathrm{T}}]$$

- Update rule for the weights becomes:

$$\mathrm{w}(\mathrm{n}+1) = \mathrm{w}(\mathrm{n}) + \eta \mathrm{x}(\mathrm{n}) \varepsilon(\mathrm{n})$$



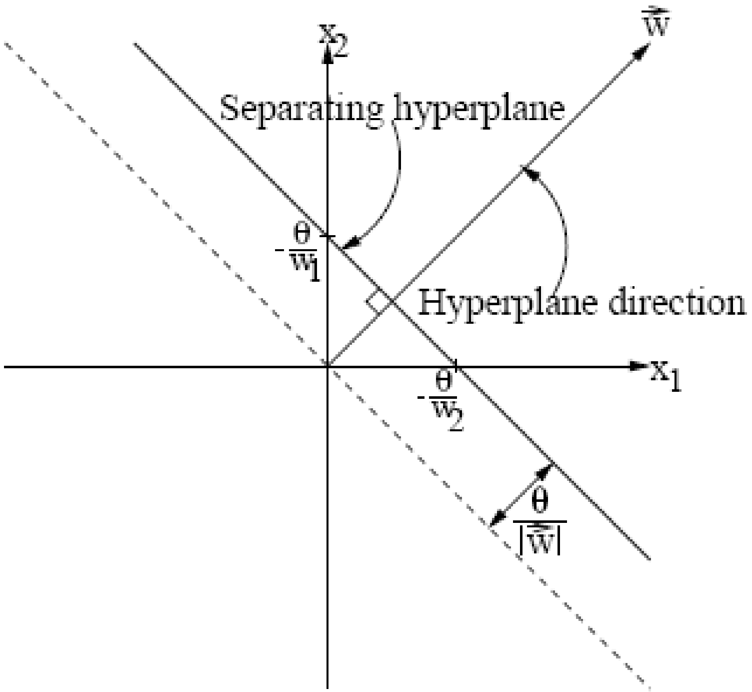
Example

# Decision boundary

- Adaline's decision boundary

  $$w_0 x_0 + w_1 x_1 + \ldots + w_n x_n = 0$$

- The Adaline:
  - has a decision boundary like the perceptron
  - can also be used to classify objects into two categories
  - has same limitation as the perceptron
  - **But**
    the learning can be derived from analysis (gradient descent)



# ADALine: general Learning Principle

- Minimize sum of squared errors (SSE) or the mean of squared errors (MSE)

$$\varepsilon_i = d_i - \tilde{d}_i$$

$$where$$

$$\tilde{d}_i = \vec{w}\vec{x}_i + b$$

$$MSE : E = \frac{1}{N}\sum_{i=1}^{N}\varepsilon_i^2$$

using mse is batch method as we take the mean of all the data points at once

the other one is stochastic weherein we take each data point individually and then update the weights based on error.

- The minimum MSE is called the least mean square (LMS), which can be obtained analytically by building:

$$\frac{\partial E}{\partial \vec{w}} = 0$$ wrt actual weights

we split the derivative into two as we have to differentiate wrt two terms

$$\frac{\partial E}{\partial b} = 0$$ wrt bias

- then solve for ***w*** and ***b***.
- LMS is difficult to obtain for larger dimensions (complex formula) and larger data sets
- ADALine:
  - learns by minimizing the MSE
  - not sensitive to noise
  - powerful and robust learning

Example

# Summary of Adaline: Learning algorithm (LMS)

Training sample:

input signal vector **x(t)**

desired response **d(t)**

User selected parameter:

$\eta > 0$

Computation:

**for t = 1, 2, …**

**compute**

$\delta(t) = d(t) - \hat{w}^T(t)x(t)$

$\hat{w}(t+1) = \hat{w}(t) + \eta x(t)\delta(t)$

Initialization: $\vec{w}_0 = \vec{0}$;

$t = 0$;

Repeat

$t = t + 1$;

For each training example $[\vec{x}, d_{\vec{x}}]$ do

$net_{\vec{x}} = \vec{w} \cdot \vec{x}$;

$a_{\vec{x}} = g(net_{\vec{x}}) = net_{\vec{x}}$;

$\delta_{\vec{x}} = d_{\vec{x}} - a_{\vec{x}}$;

$\vec{w}_{t+1} = \vec{w}_t + \eta \cdot \delta_{\vec{x}} \cdot \vec{x}$;

{

or equivalently,

For $0 \le i \le n$

$w_{i,t+1} = w_{i,t} + \eta \cdot \delta_{\vec{x}} \cdot x_i$;

}

Until $MSE(\vec{w})$ is minimal;

Save last weight vector;

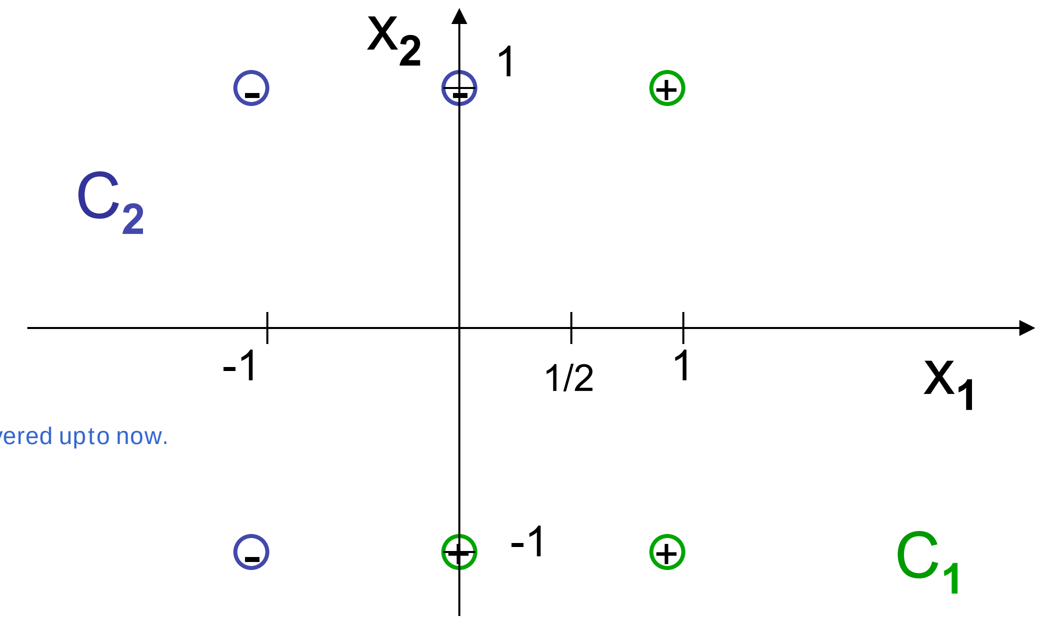
ADAline END

# New Classification Example (now *with* bias)

Consider 2D training set
$C_1 \cup C_2$, where:

$C_1 = \{(1,1), (1, -1), (0, -1)\}$
elements of class 1

$C_2 = \{(-1,-1), (-1,1), (0,1)\}$
elements of class -1




- Use the perceptron learning algorithm to classify these examples.

what we covered upto now.

$\mathbf{w}(1) = [1, 0, 0]^T$
$\eta = 1$

# Trick for bias and learning rule

Consider the *augmented* training set $C'_1 \cup C'_2$, with first entry fixed to 1 (to deal with the *bias as extra weight*):

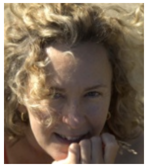
$C'_1 = \{ (1, 1, 1), (1, 1, -1), (1, 0, -1)\}$

$C'_2 = \{ (1,-1, -1), (1,-1, 1), (1,0, 1) \}$

Replace tuples (a,b,c) from $\in C_2'$ by (-a,-b,-c) and use the following simpler update rule (Note: NOT so obvious…):

we change if the product is -ve

$$\mathbf{w}(n+1) = \begin{cases} \mathbf{w}(n) + \eta\, \mathbf{x}(n) & \text{if } \mathbf{w}^T(n)\, \mathbf{x}(n) \leq 0 \\ \mathbf{w}(n) & \text{otherwise} \end{cases}$$

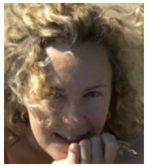


# Cont…

- Training set after application of trick:
(1, 1, 1), (1, 1, -1), (1,0, -1), (-1,1, 1), (-1,1, -1), (-1,0, -1)
- Application of perceptron learning algorithm:

| Adjusted pattern | Weight applied | w(n) x(n) | Update? | New weight |
|---|---|---|---|---|
| (1, 1, 1) | (1, 0, 0) | 1 | No | (1, 0, 0) |
| (1, 1, -1) | (1, 0, 0) | 1 | No | (1, 0, 0) |
| (1,0, -1) | (1, 0, 0) | 1 | No | (1, 0, 0) |
| (-1,1, 1) | (1, 0, 0) | -1 | Yes | (0, 1, 1) |
| (-1,1, -1) | (0, 1, 1) | 0 | Yes | (-1, 2, 0) |
| (-1,0, -1) | (-1, 2, 0) | 1 | No | (-1, 2, 0) |

End epoch 1



# Example

| Adjusted pattern | Weight applied | $w^T(n)x(n)$ | Update? | New weight |
|---|---|---|---|---|
| (1, 1, 1) | (-1, 2, 0) | 1 | No | (-1, 2, 0) |
| (1, 1, -1) | (-1, 2, 0) | 1 | No | (-1, 2, 0) |
| (1,0, -1) | (-1, 2, 0) | -1 | Yes | (0, 2, -1) |
| (-1, 1, 1) | (0, 2, -1) | 1 | No | (0, 2, -1) |
| (-1, 1, -1) | (0, 2, -1) | 3 | No | (0, 2, -1) |
| (-1,0, -1) | (0,2, -1) | 1 | No | (0, 2, -1) |

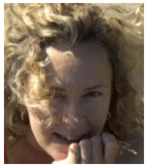
End epoch 2

At Epoch 3 no updates are performed. (check!)

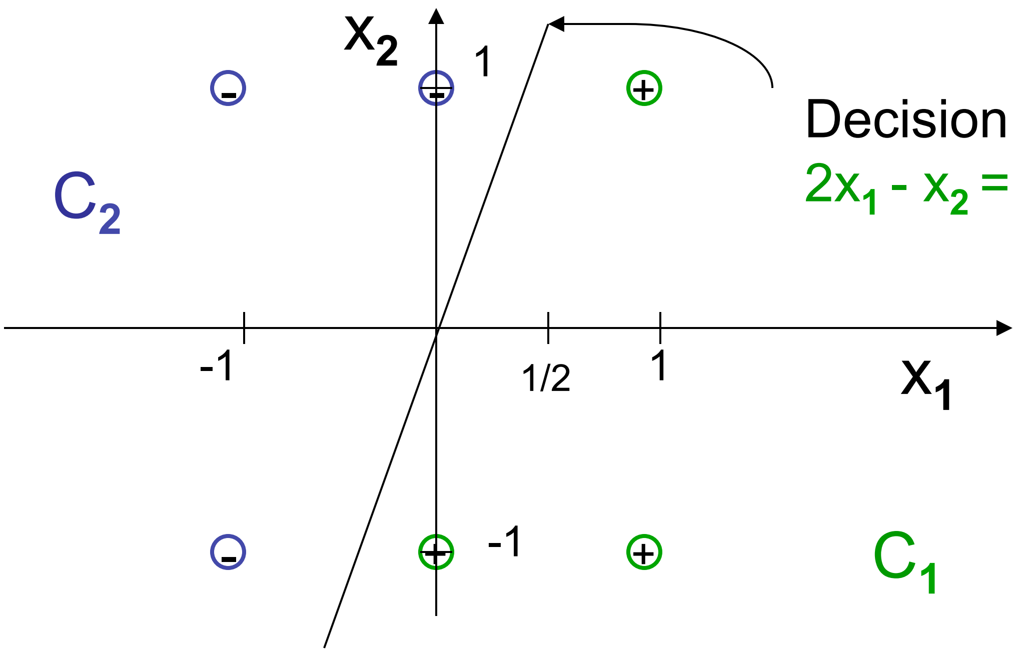
⇒ stop execution of algorithm.

**Final weight vector: (0, 2, -1).**

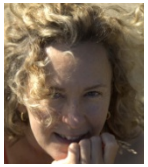
**⇒ decision hyperplane is 2x1 - x2 = 0.**



# Example visualized:




Decision boundary:
$2x_1 - x_2 = 0$



# Perceptron: fixed-increment learning algorithm

- Variables and parameters at iteration n of the learning algorithm:

  $\mathbf{x}(n)$ = input vector

  $= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$

  $\mathbf{w}(n)$ = weight vector

  $= [b(n), w_1(n), w_2(n), \ldots, w_m(n)]^T$

  $b(n)$ = bias

  $y(n)$ = actual response

  $d(n)$ = desired response

  $\eta$ = learning rate parameter

Theorem

# Perceptron: fixed-increment learning algorithm (cont)

- Initialization: n=1, w(n) = 0 (or random small val.)
- Activation: activate perceptron by applying input example (input = x (n), desired output = d(n) )
- Compute actual output of perceptron:

$$y(n) = \mathrm{step}[w^T(n)\, x(n)]$$

- Adapt weight vector: if $d(n) \neq y(n)$ then

$$w(n + 1) = w(n) + \eta\, e(n)\, x(n)$$

$$\text{where } e(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \in C_1 \\ -1 & \text{if } \mathbf{x}(n) \in C_2 \end{cases}$$

- Continuation: increment time step *n* and go to Activation:

# Perceptron Learning Convergence

Suppose datasets $C_1$, $C_2$ are linearly separable. The perceptron convergence algorithm converges after $n_0$ iterations, with $n_0 \leq n_{max}$ on training set $C_1 \cup C_2$.

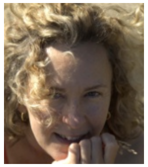
Proof:

- suppose $\mathbf{x} \in C_1 \Rightarrow$ output = 1 and $\mathbf{x} \in C_2 \Rightarrow$ output = -1.
- For simplicity assume $\mathbf{w}(1) = 0, \ \eta = 1$.
- Suppose perceptron incorrectly classifies $\mathbf{x}(1) \ldots \ \mathbf{x}(k) \ldots \in \mathbf{C_1}$.

Then $\mathbf{w}^T(k)\ \mathbf{x}(k) \leq 0$.

$\Rightarrow$ Error correction rule:

$$\left.\begin{aligned} \mathbf{w}(2) &= \mathbf{w}(1) + \mathbf{x}(1) \\ \mathbf{w}(3) &= \mathbf{w}(2) + \mathbf{x}(2) \\ &\ \vdots \\ \mathbf{w}(k+1) &= \mathbf{w}(k) + \mathbf{x}(k) \end{aligned}\right\} \Rightarrow \mathbf{w}(k+1) = \mathbf{x}(1) + \ldots + \mathbf{x}(k)$$



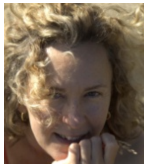
# Proof : Convergence theorem (cont.)

- Let $\mathbf{w}_0$ be such that $\mathbf{w}_0^T \mathbf{x}(n) > 0 \quad \forall\ \mathbf{x}(n) \in C_1$.
  $w_0$ exists because $C_1$ and $C_2$ are linearly separable.
- Let $\alpha = \min w_0^T x(n) \quad x(n) \in C_1$.
- Then $\mathbf{w}_0^T w(k+1) = \mathbf{w}_0^T \mathbf{x}(1) + \ldots + \mathbf{w}_0^T \mathbf{x}(k) \geq k\alpha$
- Cauchy-Schwarz inequality:

$$||\mathbf{w}_0||^2 \; ||\mathbf{w}(k+1)||^2 \geq [\mathbf{w}_0^T \mathbf{w}(k+1)]^2$$

replacing by the equatio above

$$||\mathbf{w}(k+1)||^2 \geq \frac{k^2 \alpha^2}{||w_0||^2} \qquad \textbf{(A)}$$



# Proof: Convergence theorem (cont.)

- Now we consider another route:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{x}(k)$$

$$\|\,\mathbf{w}(k+1)\|^2 = \|\,\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2 + \underbrace{2\,\mathbf{w}^{T}(k)\mathbf{x}(k)}_{\leq\, 0 \text{ because } \mathbf{x}(k) \text{ is misclassified}}$$

length of one vector is the sum of its component vectors and the corrective term(cosine distance).

euclidean norm

$$\Rightarrow \|\mathbf{w}(k+1)\|^2 \leq \|\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2$$

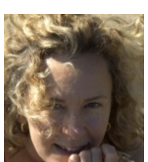
=0

$$\|\mathbf{w}(2)\|^2 \leq \|\mathbf{w}(1)\|^2 + \|\mathbf{x}(1)\|^2$$

$$\|\mathbf{w}(3)\|^2 \leq \|\mathbf{w}(2)\|^2 + \|\mathbf{x}(2)\|^2$$

$$\vdots$$

$$\Rightarrow \|\mathbf{w}(k+1)\|^2 \leq \sum_{i=1}^{k} \|\,\mathrm{x}(\mathrm{i})\,\|^2$$

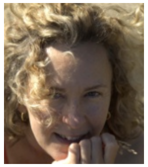


# Proof: Convergence theorem (end)

- Let $\beta = \max ||\mathbf{x}(n)||^2 \quad \mathbf{x}(n) \in C_1$
- $||\mathbf{w}(k+1)||^2 \leq k\,\beta$ **(B)**
- For sufficiently large values of k:
  **(B)** becomes in conflict with **(A)**. since in A the k is a squared term
  So k cannot be greater than some $k_{max}$, where **(A)** and **(B)** are both satisfied with the equality sign.

$$\frac{k_{max}^2 \alpha^2}{||w_0||^2} = k_{max}\beta \Rightarrow \boxed{k_{max} = \frac{||w_0||^2}{\alpha^2}\beta}$$

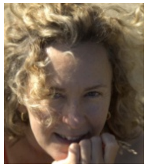
- Perceptron convergence algorithm terminates in at most
  $n_{max} = \dfrac{\beta\, ||\mathbf{w}_0||^2}{\alpha^2}$ iterations.



# Perceptron: Limitations

- The perceptron can only model linearly separable functions.
- The perceptron can be used to model the following Boolean functions:
- AND / OR / COMPLEMENT

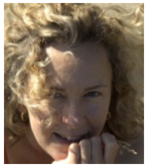
- But it cannot model the XOR. Why?



**Observation**

# Comparison Adaline and Perceptron

- Both represent different implementations of a single-layer perceptron based on error-correction learning.
- Model of a neuron
  - Adaline: Linear.
  - Perceptron: Non linear.
    Hard-Limiter activation function.
    McCulloch-Pitts model.
- Learning Process
  - Adaline: Continuous.
  - Perceptron: A finite number of iterations.



Observation

# Outline

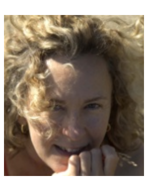
- First Example for Learning: Classification by a Perceptron
- Second Example: Regression by a Perceptron
- Perceptron Convergence Theorem
- Definition: ANN learning
- 1$^{st}$ Learning rule
- … **and four more Learning rules**

# Example

- How to train a perceptron to recognize this 3?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | +1 | +1 | +1 | +1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | +1 | -1 | -1 |
| -1 | -1 | -1 | +1 | +1 | +1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | +1 | -1 | -1 |
| -1 | -1 | -1 | -1 | - 1 | +1 | -1 | -1 |
| -1 | -1 | +1 | +1 | +1 | +1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

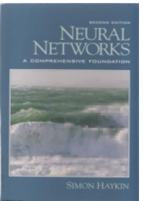
- Assign 1 to weights of input values that are equal to +1, and -1 to weights of input values that are equal to -1



# ANN Learning

- *Learning is a process by which the free parameters of a neural network are adapted in an desired way through a process of stimulation by the environment in which the network is embedded.*
  *The type of learning is determined by the manner in which the parameter change takes place.*

$$e.g. \quad w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}$$



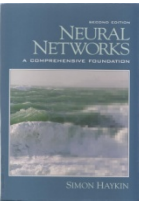
**Definition**

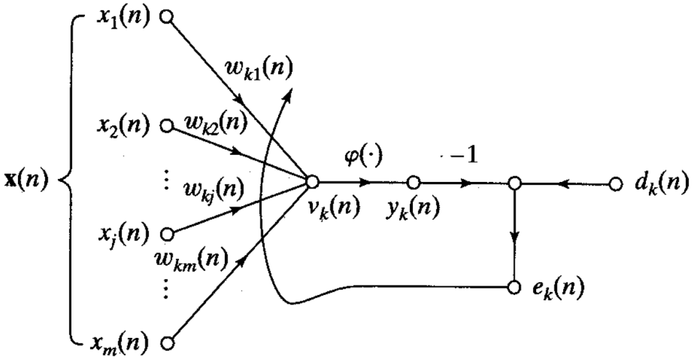
# 1st Learning Rule: Error Correction

- Error $\boldsymbol{e_k}$ actuates a controls mechanism

| | |
|---|---|
| n | time step |
| k | neuron number |
| $d_k(n)$ | desired signal |
| $y_k(n)$ | observed signal |

$$e_k(n) = d_k(n) - y_k(n)$$



Definition

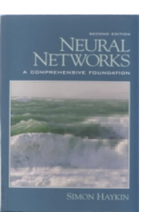
# 1st Learning Rule: Error Correction

- Error $e_k$ actuates a controls mechanism
- Cost function $E$ on top (here: instantaneous at time n, local at output node k)

| | |
|---|---|
| $n$ | time step |
| $k$ | neuron number |
| $d_k(n)$ | desired signal |
| $y_k(n)$ | observed signal |

$$e_k(n) = d_k(n) - y_k(n)$$

$$E(n) = \tfrac{1}{2} e_k^{\,2}(n)$$



# 1st Learning Rule: Error Correction

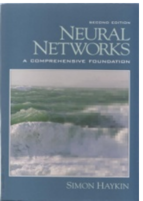
- Error $e_k$ actuates a controls mechanism
- Cost function $E$ on top (here: instantaneous at time n, local at output node k)
- Minimization of cost function: ***Widrow-Hoff rule (delta rule)***

| | |
|---|---|
| n | time step |
| k | neuron number |
| $d_k(n)$ | desired signal |
| $y_k(n)$ | observed signal |

$$e_k(n) = d_k(n) - y_k(n)$$

$$E(n) = \tfrac{1}{2} e_k^{\,2}(n)$$

$$\boxed{\Delta w_{kj}(n) = e_k(n) x_j(n)}$$



Definition

# 1st Learning Rule: Error Correction

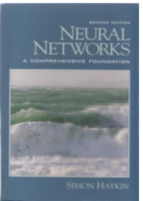
- Error $\boldsymbol{e_k}$ actuates a controls mechanism
- Cost function ***E*** on top (here: instantaneous at time n, local at output node k)
- Minimization of cost function: ***Widrow-Hoff rule (delta rule)***
- *The adjustment made to a synaptic weight of a neuron is proportional to the product of the error signal and the input signal of the synapse in question.*

| | |
|---|---|
| n | time step |
| k | neuron number |
| $d_k(n)$ | desired signal |
| $y_k(n)$ | observed signal |

$$e_k(n) = d_k(n) - y_k(n)$$

$$E(n) = \tfrac{1}{2} e_k^{\,2}(n)$$

$$\boxed{\Delta w_{kj}(n) = e_k(n) x_j(n)}$$



# 1st Learning Rule: Error Correction

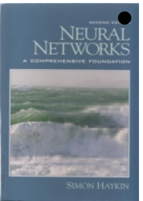
| | |
|---|---|
| n | time step |
| k | neuron number |
| $d_k(n)$ | desired signal |
| $y_k(n)$ | observed signal |

- Error $\boldsymbol{e_k}$ actuates a control mechanism
- Cost function ***E*** on top (here: instantaneous at time n, local at output node k)
- Minimization of cost function: ***Widrow-Hoff rule (delta rule)***
- *The adjustment made to a synaptic weight of a neuron is proportional to the product of the error signal and the input signal of the synapse in question.*
- Learning rate $\eta$ and final eq. $w_{kj}(n+1) = w_{kj}(n) + \eta \, \Delta w_{kj}$

$$e_k(n) = d_k(n) - y_k(n)$$

$$E(n) = \tfrac{1}{2} e_k^{\,2}(n)$$

$$\Delta w_{kj}(n) = e_k(n) x_j(n)$$



# 2nd: Memory based learning

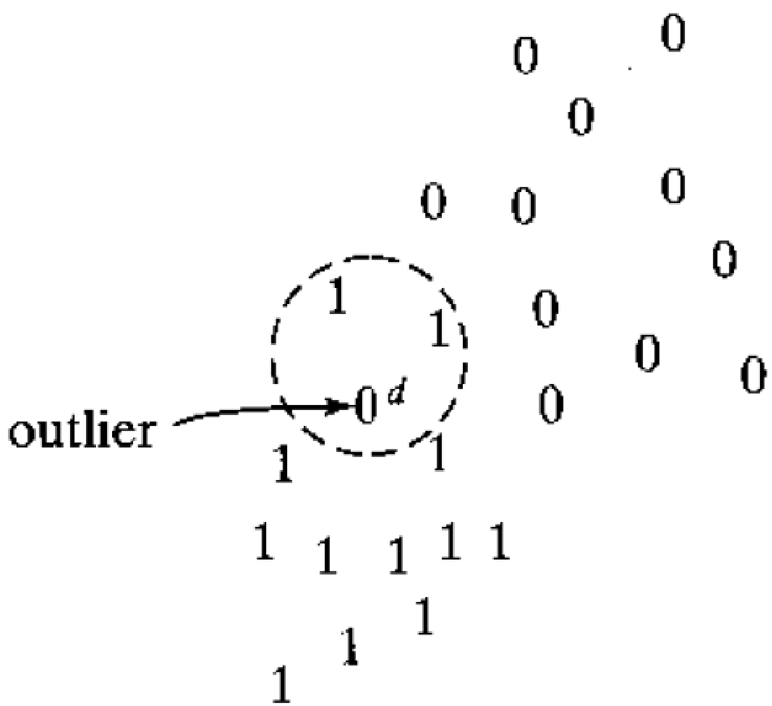
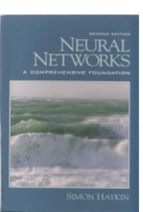
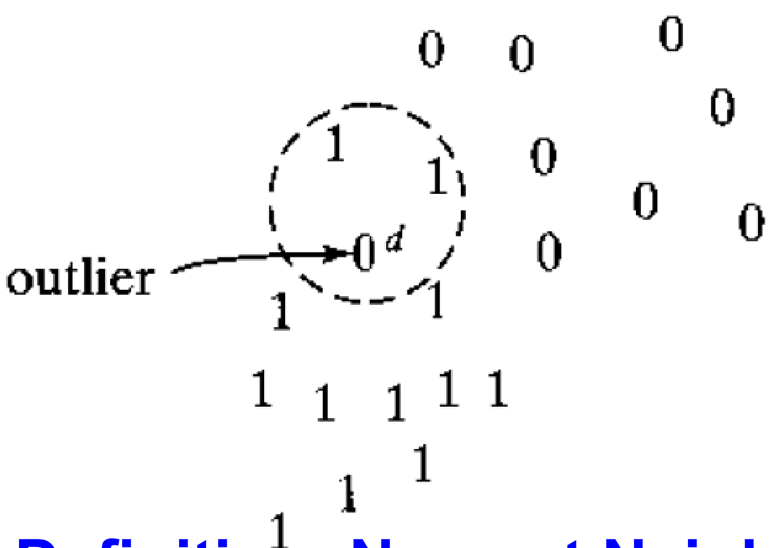



**FIGURE 2.2** The area lying inside the dashed circle includes two points pertaining to class 1 and an outlier from class 0. The point *d* corresponds to the test vector $\mathbf{x}_{test}$. With $k = 3$, the *k-nearest* neighbor *classifier* assigns class 1 to point *d* even though it lies closest to the *outlier.*

*(majority vote)*



# 2nd Memory based learning cont.




## Definition: Nearest Neighbor

$$\textit{Given } L = \{x_1, x_2, \ldots, x_N\} \textit{ and } x_{test} \notin L.$$

$$\textit{Then } x' \in L \textit{ is called}$$

$$\textit{nearest neighbor to } x_{test} \textit{ in } L :\Leftrightarrow$$

$$\min_i d(x_i, x_{test}) = d(x', x_{test})$$

## Algorithm: k-nearest neighbor learning

$$\textit{Given } L, x_{test} \notin L, k \in IN \textit{ fixed},$$

$$\textit{class function} : \textit{class\_of}() \textit{ on } L$$

$$\textit{Set } x' = \{\}, L_0 = L, \textit{Classf} = \textit{empty\_list}$$

$$\textit{for } j = 1 \ldots k \textit{ do } \{$$

$$L_j \leftarrow L_{j-1} \setminus x';$$

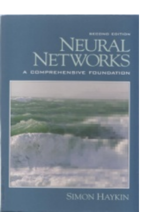
$$x' \leftarrow \textit{find NN to } x_{test} \textit{ in } L_j;$$

$$c \leftarrow \textit{class\_of}(x');$$

$$\textit{Classf} \leftarrow \textit{push}(c);$$

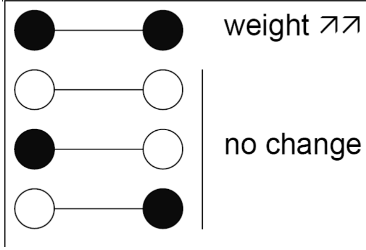
$$\}$$

$$\textit{set } c(x_{test}) := \textit{most frequent value in Classf}$$



# 3rd Hebbian learning

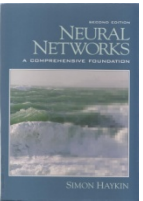
**Definition**




Donald Hebb, Canadian psychologist, wrote a revolutionary paper in 1949:

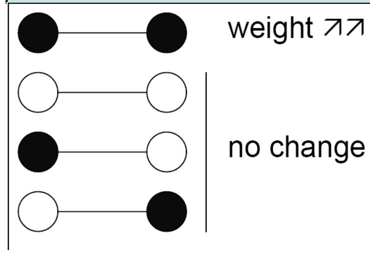
”Let us assume that the persistence or repetition of a reverberatory activity (or “trace”) tends to induce lasting cellular changes that add to its stability…. When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A’s efficiency, as one of the cells firing B, is increased.”

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.



# 3rd Hebbian learning

**Definition**

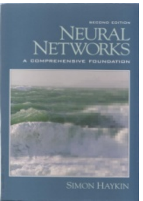



Donald Hebb, Canadian psychologist, wrote a revolutionary paper in 1949:

”Let us assume that the persistence or repetition of a reverberatory activity (or “trace”) tends to induce lasting cellular changes that add to its stability…. When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing 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.”

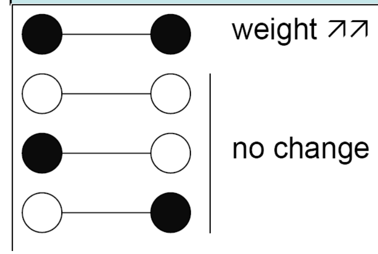
1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.



# 3rd Hebbian learning

**Definition**

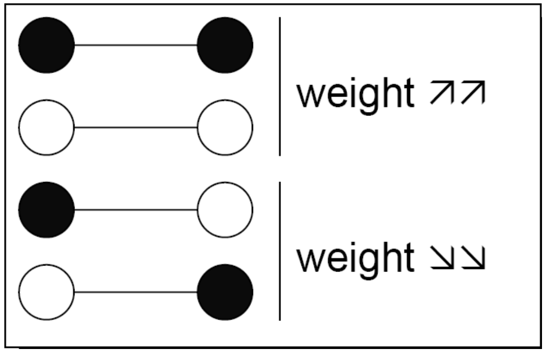


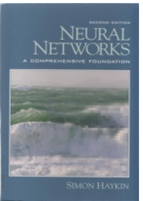

Donald Hebb, Canadian psychologist, wrote a revolutionary paper in 1949:

”Let us assume that the persistence or repetition of a reverberatory activity (or “trace”) tends to induce lasting cellular changes that add to its stability…. When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing 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.”

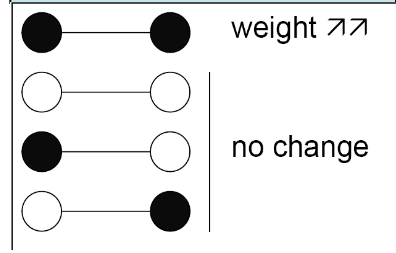
1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.



Definition

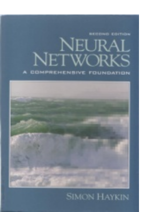



# 3rd Hebbian learning

- Time-dependent mechanism
- Local mechanism
- Interactive mechanism
- Conjuctional or correlational mechanism

- In general $$\Delta w_{kj}(n) = F(y_k(n), x_j(n))$$
- Most simple $$\Delta w_{kj}(n) = \eta y_k(n) x_j(n)$$
- Covariance hypothesis $$\Delta w_{kj}(n) = \eta (y_k - \bar{y})(x_j - \bar{x})$$

**summarized: cells that fire together, wire together.**



# 4th Competitive Learning

- In competitive learning the output neurons compete among themselves to become active (fired).

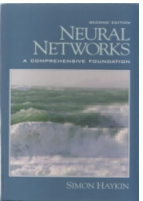
- Hebbian learning ⇔ several output neurons may be active simultaneously,

- competitive learning ⇔ only a **single output neuron** is active **at any one time**.

- A set of neurons that are all the same except for some randomly distributed synaptic weights, and which therefore respond differently to a given set of input patterns.
- A limit imposed on the "strength" of each neuron.
- A mechanism that permits the neurons to compete for the right to respond to a given subset of inputs, such that only one output neuron, or only one neuron per group, is active (i.e., "on") at a time.

The neuron that wins the competition is called a **winner-takes-all neuron.**

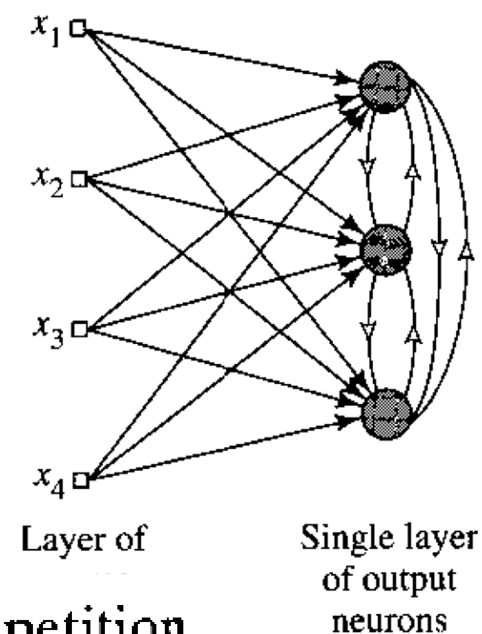
(Rumelhart and Zipser,1985)



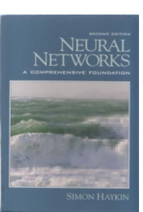
# 4th Competitive Learning (cont)

$$y_k = \begin{cases} 1 & \text{if } v_k > v_j \text{ for all } j, j \neq k \\ 0 & \text{otherwise} \end{cases}$$

$$\sum_j w_{kj} = 1 \qquad \text{for all } k$$




$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{if neuron } k \text{ wins the competition} \\ 0 & \text{if neuron } k \text{ loses the competition} \end{cases}$$



# 4th Competitive learning (cont)

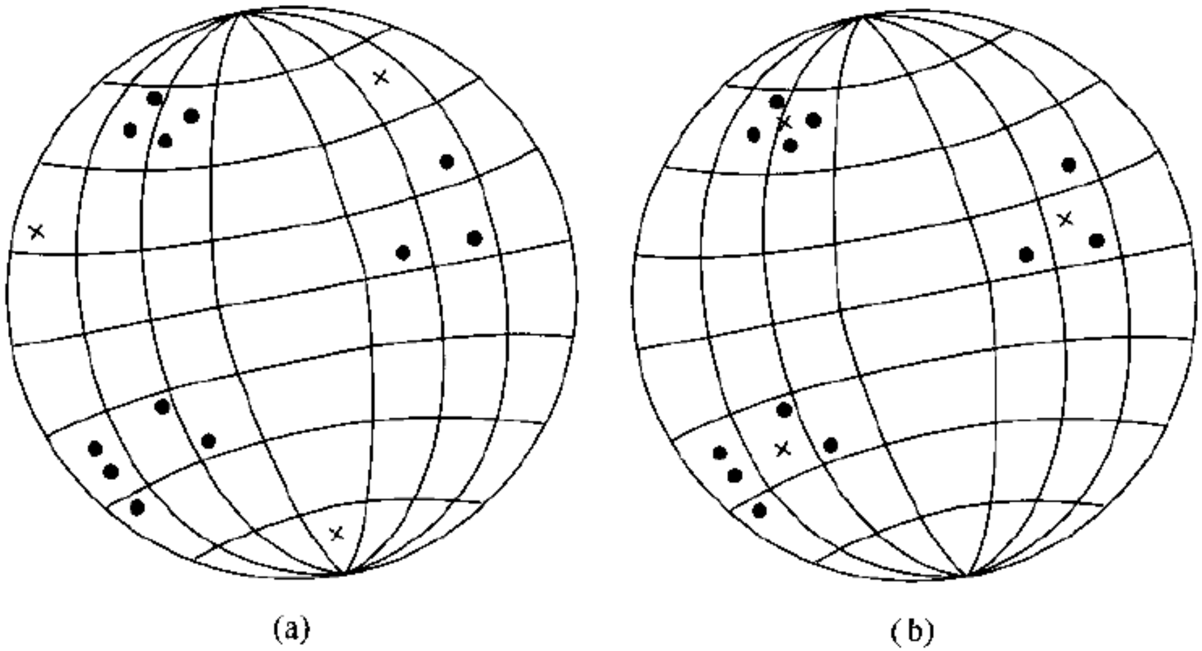
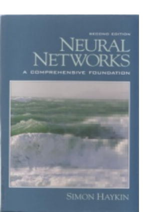
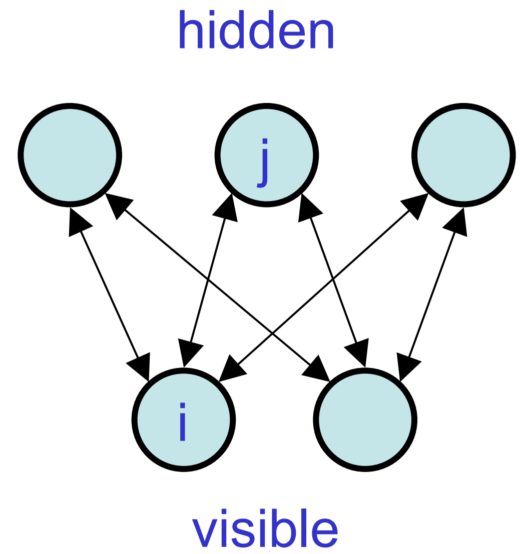


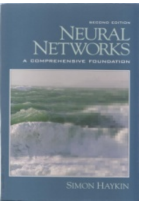

**FIGURE 2.5** Geometric interpretation of the competitive learning process. The dots represent the input vectors, and the crosses represent the synaptic weight vectors of three output neurons. (a) Initial state of the network. (b) Final state of the network.



# Sketch: 5th Boltzmann learning

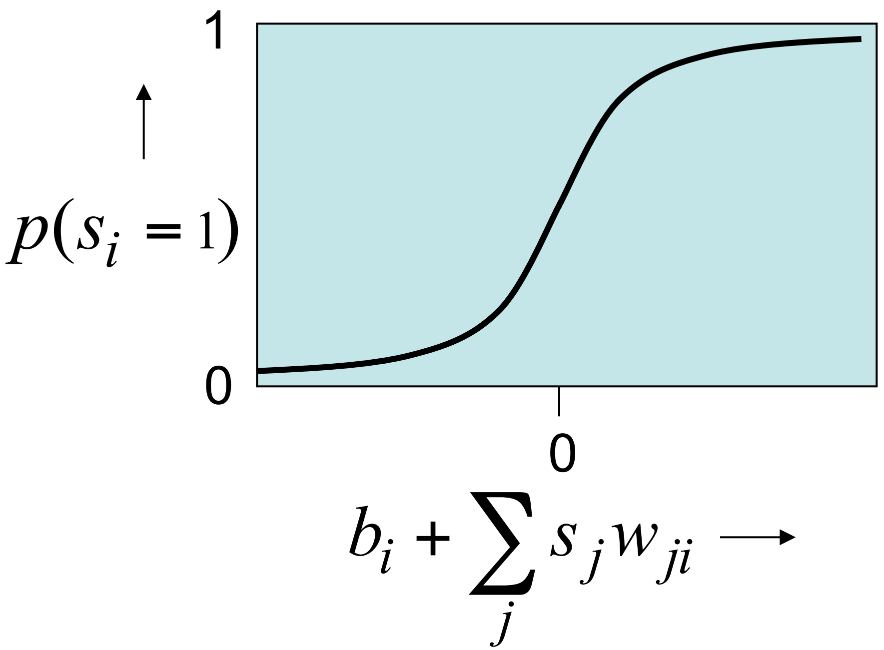
- Recurrent ANN
- Two groups: visible and hidden neurons
- Biparted graph, reduced Boltzmann Machine == RBM
- Binary neurons (+/- 1 as state)
- operate by flipping



# Stochastic binary units
## (Bernoulli variables)

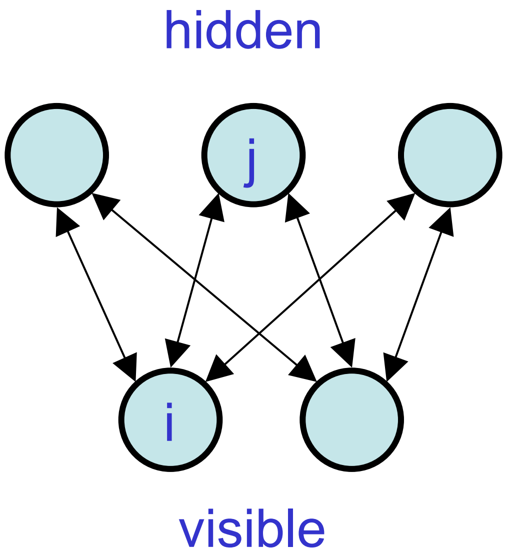
- These have a state of 1 or 0.
- The probability of turning on is determined by the weighted input from other units (plus a bias)




$$p(s_i = 1) \quad = \quad \frac{1}{1 + \exp(-b_i - \sum_j s_j w_{ji})}$$

Cited from:
2007 NIPS Tutorial on:
Deep Belief Nets
by Geoffrey Hinton

# Sketch: 5th Boltzmann learning

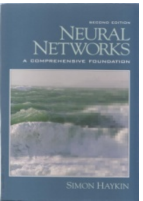



- Recurrent ANN
- Two groups: visible and hidden neurons
- Binary neurons (+/- 1 as state)
- operate by flipping
- Modes of operation: clamped / free running conditions

$\rho_{kj}^{+}$ *correlation in clamped state*

$\rho_{kj}^{-}$ *correlation in free running state*

$$\Delta w_{kj} = \eta(\rho_{kj}^{+} - \rho_{kj}^{-}) \quad j \neq k$$



# Multi-valued Perceptron

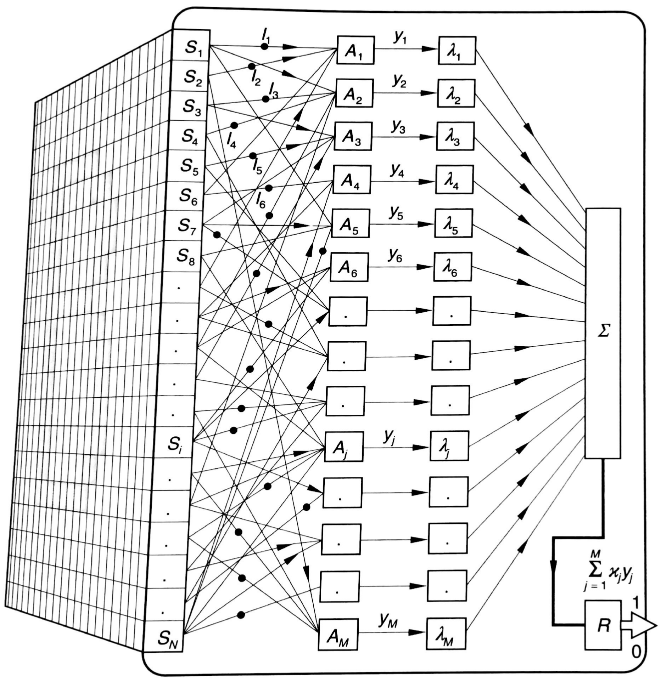



Abb. 3.14

- fixed receptive fields to connect a patch of sensor cells $S_i$ to $A_j$
- adjustable weights $\lambda_i$
- multiple linear combiners, one for each letter