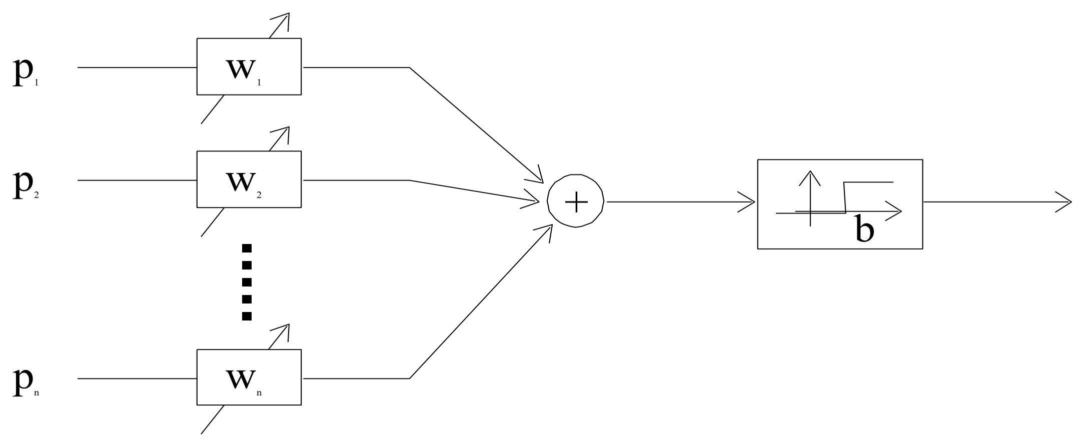
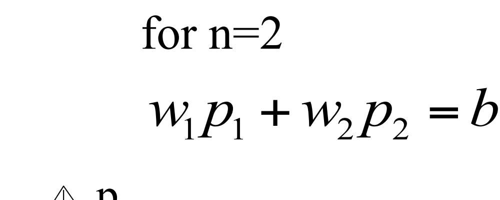
The McCulloch Neuron (1943)

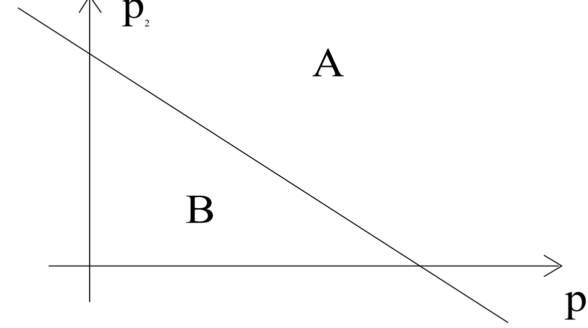


$$a = g\left(\sum_{i=1}^{n} w_i p_i - b\right) = g(\mathbf{w}^t \mathbf{p} - b) \rightarrow a \in [0;1]$$

for n=2



$$w_1 p_1 + w_2 p_2 = b$$

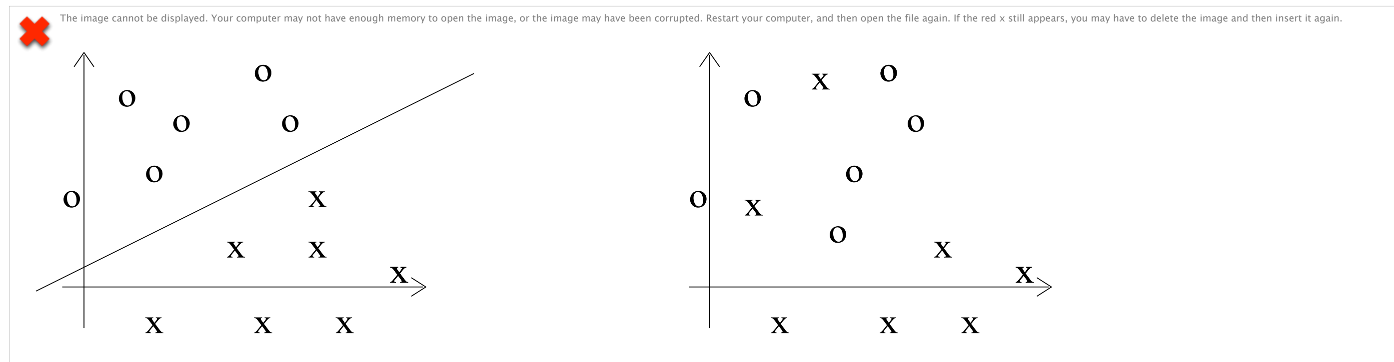


g = step function

The euclidian space $\Re^n$ is divided in two regions A and B

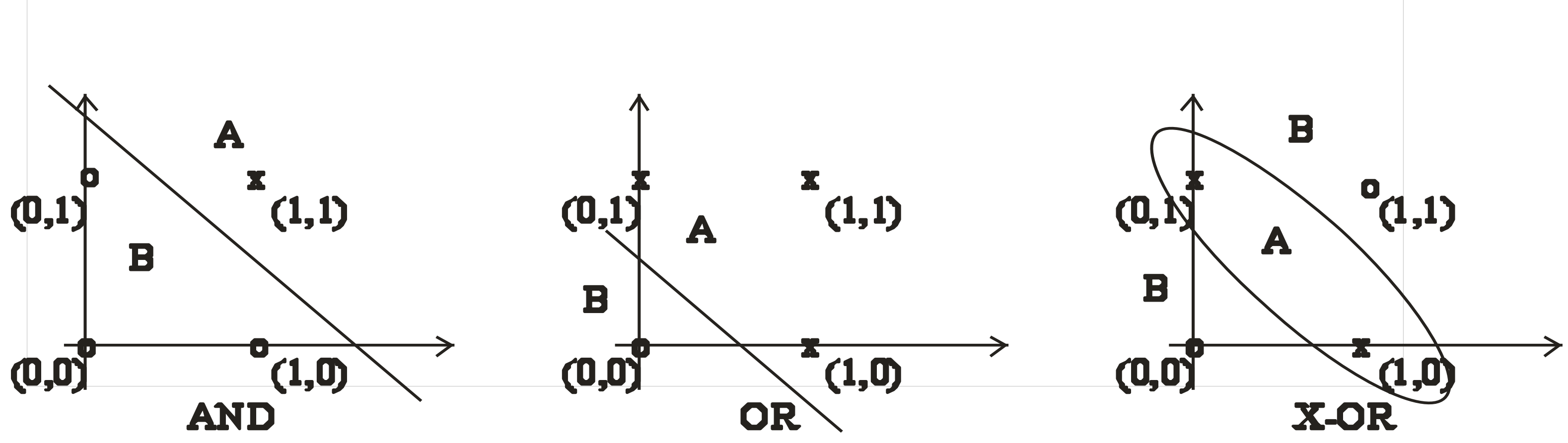
The McCulloch Neuron

– as patterns classifier



Linearly separable collections

Linearly dependent (non-separable) collections



Some Boolean functions of two variables represented in a binary plan.

Linear and Non-Linear Classifiers

There exist $2^m = 2^{2^n}$ possible logical functions connecting n inputs to one binary output.

n	# of binray patterns	# of logical functions	# linearly separable	% linearly separable
1	2	4	4	100
2	4	16	14	87,5
3	8	256	104	40,6
4	16	65536	1.772	2,9
5	32	$4{,}3 \times 10^{9}$	94.572	$2{,}2 \times 10^{-3}$
6	64	$1{,}8 \times 10^{19}$	5.028.134	$3{,}1 \times 10^{-13}$

The logical functions of one variable:

$A, \ \bar{A}, \ 0, \ 1$

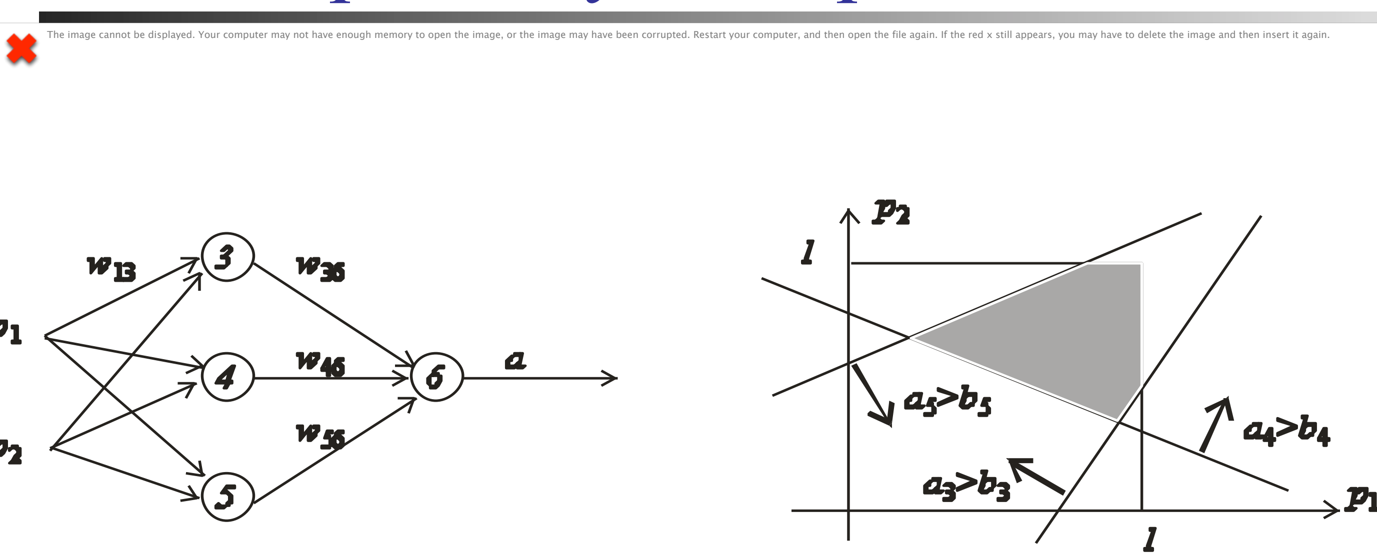
The logical functions of two variables:

$A, \ B, \bar{A}, \ \bar{B}, \ 0, \ 1$

$A \vee B, A \wedge B, \bar{A} \vee B, \bar{A} \wedge B,$

$A \vee \bar{B}, A \wedge \bar{B}, \bar{A} \vee \bar{B}, \bar{A} \wedge \bar{B}, \ A \oplus B, \overline{A \oplus B}$

Two Step Binary Perceptron



The neuron 6 implements a logical AND function by choosing

$$b_6 = \sum_{i=3}^{5} w_{i6} \, .$$

For example:

$$w_{36} = w_{46} = w_{56} = \frac{1}{3}; \quad b_6 = 1 \quad \Rightarrow a_6 = 1 \; if \;\; and \;\; only \;\; if \quad a_3 = a_4 = a_5 = 1$$

Three Step Binary Perceptron

w_{13}, w_{39}, w_{49}, w_{59}

p_1, p_2 → nodes 3, 4, 5, 6, 7, 8 → nodes 9, 10 → (A, B) → node 10 → a_{11}

$$a_{11} = A \wedge \overline{B}$$

Neurons and Artificial Neural Networks

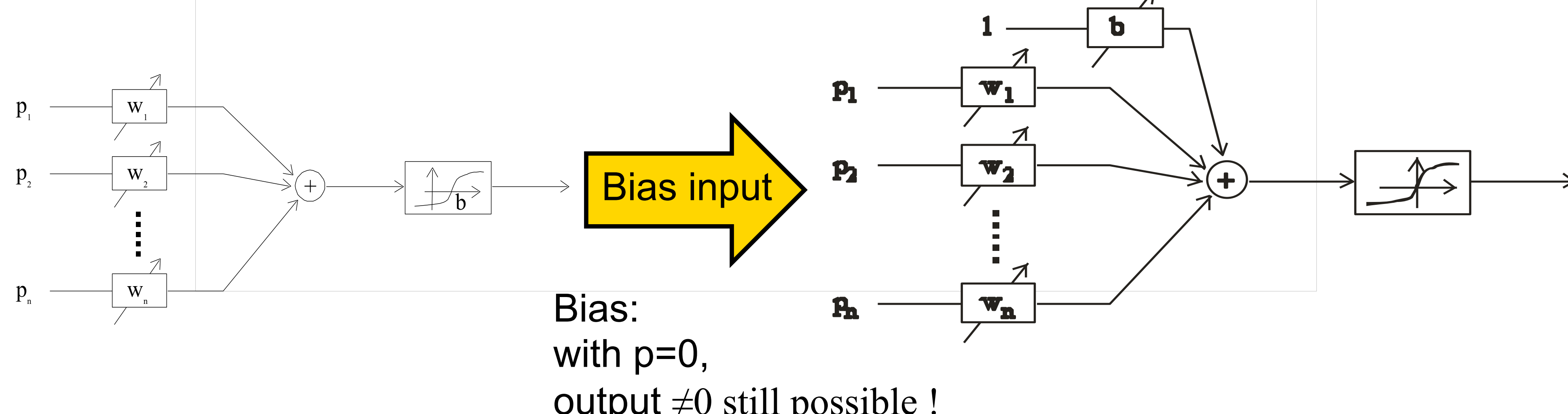
- Micro-structure

  The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

  characteristics of each neuron in the network
- Meso-Structure

  organization of the network
- Macro-Structure

  association of networks, eventually with some analytical processing approach for complex problems



Typical activation functions

Linear	$f(s) = s$	Hopfield BSB	purelin	$f(s)$, s
Signal	$f(s) = \begin{cases} +1 \;\; se \;\; s \geq 0 \\ -1 \;\; se \;\; s < 0 \end{cases}$	Perceptron	hardlims	$f(s)$, 1, -1, s
Step	$f(s) = \begin{cases} +1 \; se \; s \geq 0 \\ 0 \; se \; s < 0 \end{cases}$	Perceptron BAM	hardlim	$f(s)$, 1, s
Hopfield/ BAM	$f(s) = \begin{cases} +1 \; se \; s > 0 \\ -1 \; se \; s < 0 \\ unchanged \;\; if \;\; s = 0 \end{cases}$	Hopfield BAM		$f(s)$, 1, -1, s

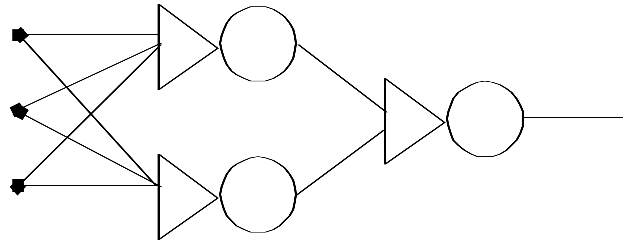
Typical activation functions

BSB or Logical Threshold	$f(s) = \begin{cases} -K & se\ s \leq -K \\ s & se\ -K < s < +K \\ +K & se\ s \geq +K \end{cases}$	BSB	satlin satlins	$f(s)$, K, $-K$, s
Logístics	$f(s) = \dfrac{1}{1+e^{-s}}$	Perceptron Hopfield BAM, BSB	logsig	$f(s)$, 1, s
Hiperbolic Tangent	$f(s) = \tanh(s) = \dfrac{1-e^{-2s}}{1+e^{-2s}}$	Perceptron Hopfield BAM, BSB	tansig	$f(s)$, 1, -1, s

Meso-Structure – Network Organization...

neurons per layer
network layers
connection type (forward, backward, lateral).

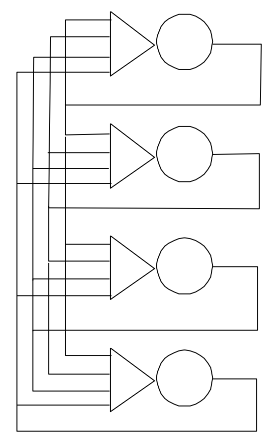
1- Multilayer *Feedforward*



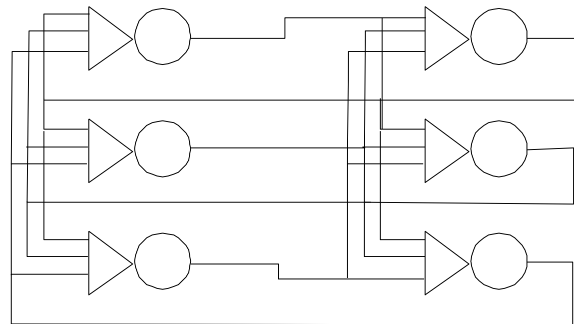
Multilayer Perceptron (MLP)

Meso-Structure – Network Organization...

2- Single Layer laterally connected (BSB (self-feedback), Hopfield)

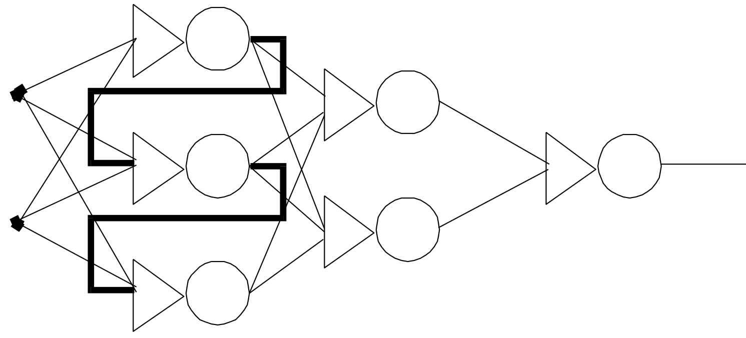


3 – Bilayers *Feedforward/Feedbackward*

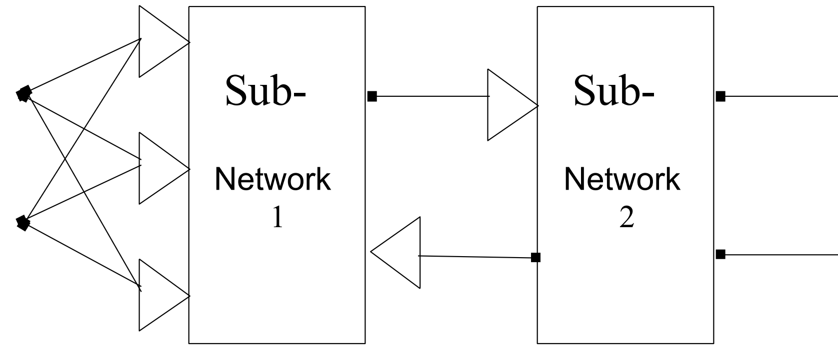


Meso-Structure – Network Organization

4 – Multilayer Cooperative/Comparative Network

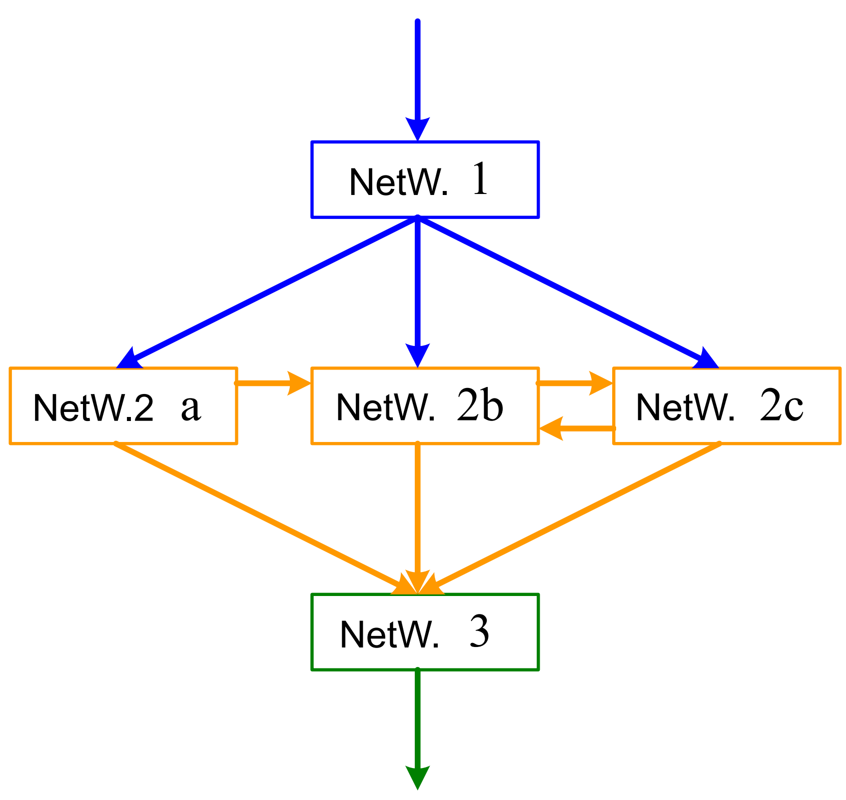


5 – Hybrid Network



Neural Macro-Structure

- # networks
- connection type
- size of networks
- degree of connectivity



Supervised Learning

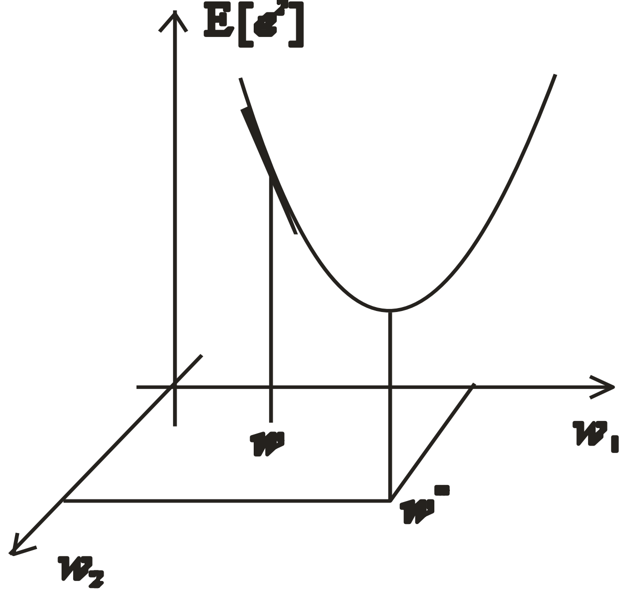
- Delta Rule → Perceptron

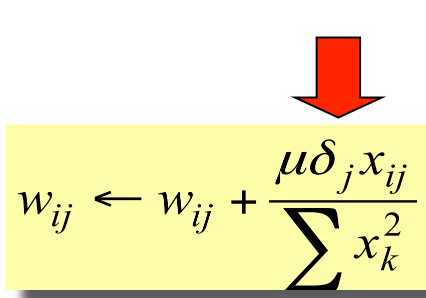
$$w \leftarrow w + \mu\delta x \qquad \delta \equiv d - y$$

μ – *learning rate*

x → y, d − y = δ

- Widrow-Hoff delta rule (LMS) → ADALINE, MADALINE
- Generalized Delta Rule





$$w_{ij} \leftarrow w_{ij} + \frac{\mu\delta_j x_{ij}}{\sum x_k^2}$$

Widrow-Hoff Delta Rule (LMS)

Delta rule → Perceptron

Perceptron – Rosenblatt, 1957

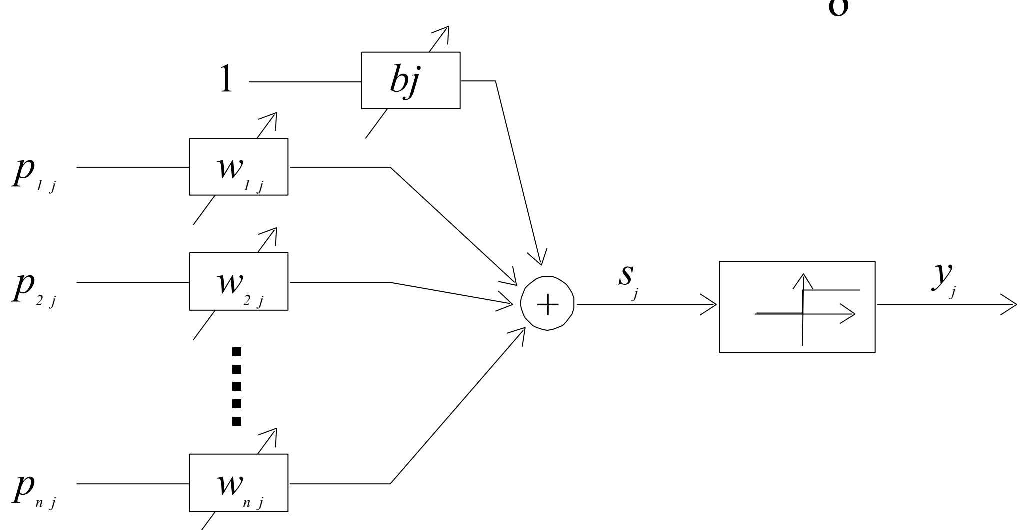
x → y

d
-
y
—
δ

Dynamics:

$$s_j = \sum_i w_{ij} p_{ij} + b_j$$

$$y_j = f(s_j) = \begin{cases} +1 & se\ s_j \geq 0 \\ 0 & se\ s_j < 0 \end{cases}$$



$$\delta_j = d_j - y_j$$

$$w_{ij} \leftarrow w_{ij} + \mu\, \delta_j x_{ij}$$ Delta Rule

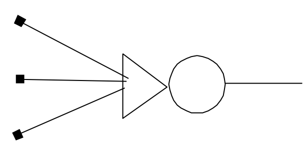
μ - learning rate

$\delta_j = 0 \rightarrow$ *the weight is not changed.*

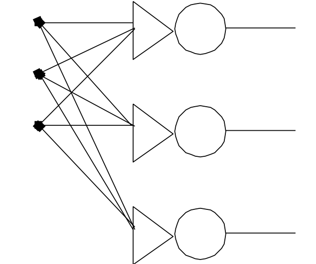
<u>Psychology Reasoning:</u>
- positive reinforcement
- negative reinforcement

ADALINE and MADALINE

Widrow & Hoff, 1960 – (Mult.) Adaptive Linear Element



$$y_j = \sum_i w_{ij} p_{ij} + b_j$$

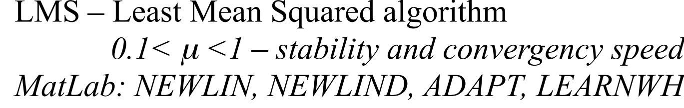


Training:

$$\varepsilon_j = d_j - s_j = d_j - \left(\sum w_{ij} p_{ij} + b_j \right)$$

$$w_{ij} \leftarrow w_{ij} + \frac{\mu \varepsilon_j x_{ij}}{\sum x_k^2}$$

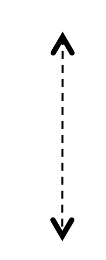
Widrow-Hoff delta rule

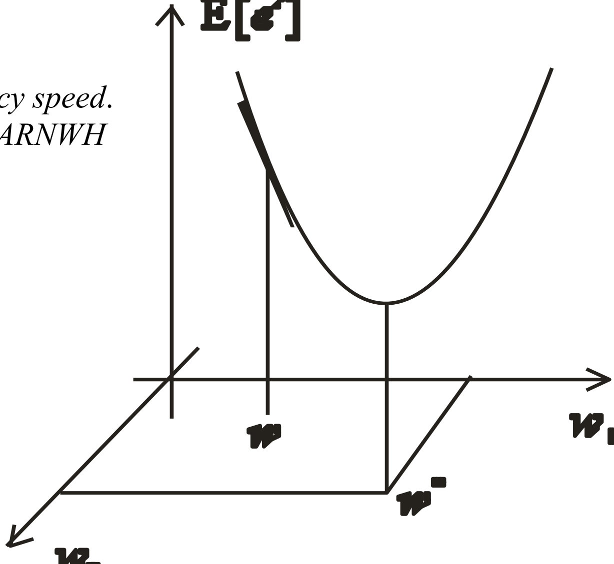


LMS – Least Mean Squared algorithm

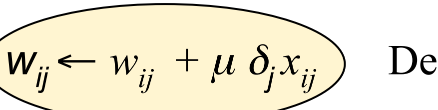
0.1< μ <1 – stability and convergency speed.

MatLab: NEWLIN, NEWLIND, ADAPT, LEARNWH





$$Obs: \ \varepsilon_j \equiv \delta_j$$



$$w_{ij} \leftarrow w_{ij} + \mu \, \delta_j x_{ij}$$

Delta Rule

LMS Algorithm

Objective: learn a function $f : \Re^n \rightarrow \Re$ from the samples ($\boldsymbol{x}_k$, d_k)

{ $\boldsymbol{x}_k$}, {d_k } and {e_k } → stationary stochastic processes

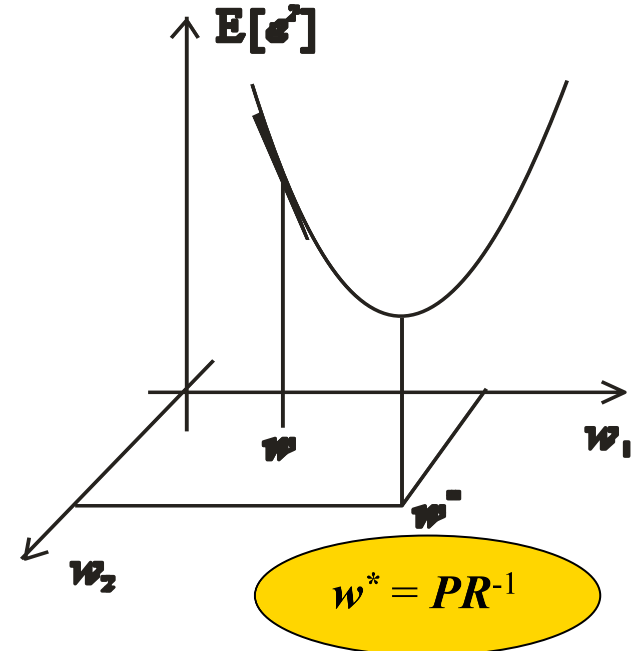
$e = d - y$ → actual stochastic error

→ Linear neuron

$$y = \sum_{i=1}^{n} x^i w^i = \mathbf{x}\mathbf{w}^t$$

Expected value

$$\begin{aligned} E[e^2] &= E[(d-y)^2] \\ &= E[(d-\boldsymbol{x}\boldsymbol{w}^t)^2] \\ &= E[d^2] - 2E[d\boldsymbol{x}]\boldsymbol{w}^t + \boldsymbol{w}E[\boldsymbol{x}^t\boldsymbol{x}]\boldsymbol{w}^t \end{aligned}$$



Optimal analytic solution of the optimization (solvelin.m)

Assuming $\boldsymbol{w}$ deterministic.
With
E [$\boldsymbol{x}^t\boldsymbol{x}$] ≡ $\boldsymbol{R}$ → autocorrelation input matrix
E [$d\boldsymbol{x}$] ≡ $\boldsymbol{P}$ → cross correlated vector

$$E[e^2] = E[d^2] - 2\boldsymbol{P}\boldsymbol{w}^t + \boldsymbol{w}\boldsymbol{R}\boldsymbol{w}^t$$

$$\mathbf{0} = 2\boldsymbol{w}^*\boldsymbol{R} - 2\boldsymbol{P}$$

(Partial derivatives equal 0 for optimal w*)

Iterative LMS Algorithm

Objective: adaptively learn a function $f : \Re^n \to \Re$ from the samples ($\boldsymbol{x}_k$, d_k)

Knowing $\boldsymbol{P}$ and $\boldsymbol{R}$, ∍ $\boldsymbol{R}^{-1}$, then for some $\boldsymbol{w}$:

$$\nabla_w \mathrm{E}[e^2] = 2\boldsymbol{wR} - 2\boldsymbol{P}$$

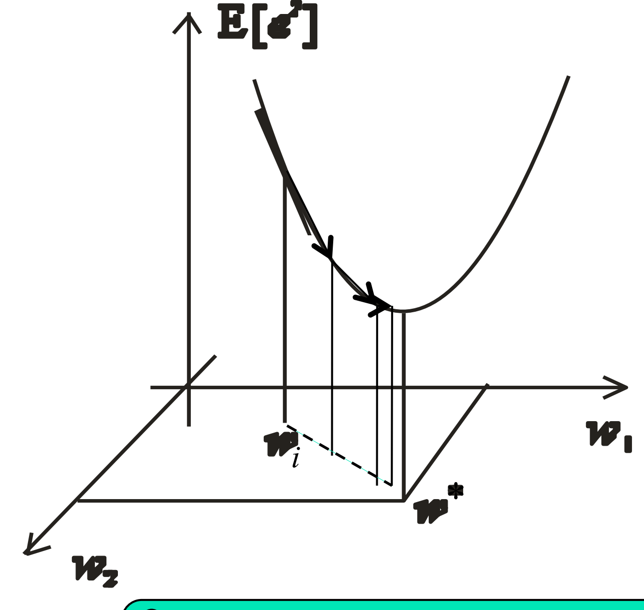
Post-multiplyting by ½ $\boldsymbol{R}^{-1}$

$$\tfrac{1}{2} \nabla_w \mathrm{E}[e^2]\, \boldsymbol{R}^{-1} = \boldsymbol{w} - \boldsymbol{P}\,\boldsymbol{R}^{-1} = \boldsymbol{w} - \boldsymbol{w}^*$$

$$\boldsymbol{w}^* = \boldsymbol{w} - \tfrac{1}{2} \nabla_w \mathrm{E}[e^2]\, \boldsymbol{R}^{-1}$$

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - c_k \nabla_w \mathrm{E}[e^2]\, \boldsymbol{R}^{-1}$$

($c_k = ½ \to$ Newton's method)



How to, ***cautiously*** find new (better) values for w_i , the free parameters ?

LMS Hypothesis:

$$\mathrm{E}[e^2_{k+1} | e^2_0, e^2_1, \ldots e^2_k] = e^2_k$$

Iterative LMS Algorithm...

assuming $\boldsymbol{R} = \boldsymbol{I}$ $\rightarrow$ estimated ***steppest decent algorithm***:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - c_k \nabla_w e^2_{\mathrm{k}}$$

Gradient of e^2_{k} with respect to $\boldsymbol{w}$

$$\nabla_w e^2_{\mathrm{k}} = \left[\frac{\partial e_k^2}{\partial w_1}, \quad \cdots \quad \frac{\partial e_k^2}{\partial w_n}\right]$$

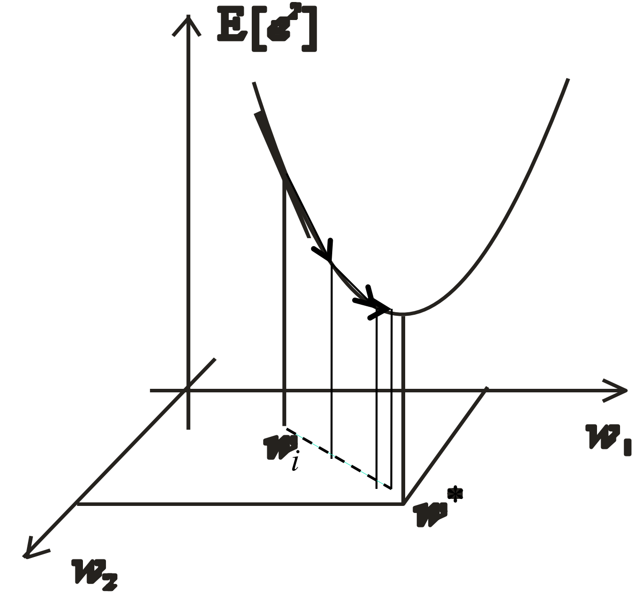
$$= \left[\frac{\partial (d_k - y_k)^2}{\partial w_1}, \quad \cdots \quad \frac{\partial (d_k - y_k)^2}{\partial w_n}\right]$$

$$= \left[-2(d_k - y_k)\frac{\partial y_k}{\partial w_1}, \quad \cdots \quad -2(d_k - y_k)\frac{\partial y_k}{\partial w_n}\right]$$

$$= -2e_k\left[\frac{\partial y_k}{\partial w_1}, \quad \cdots \quad \frac{\partial y_k}{\partial w_n}\right]$$

$$= -2e_k\left[x_k^1, \quad \cdots \quad x_k^n\right] = -2e_k\mathbf{x}_k \qquad (y_k = \mathbf{x}_k\mathbf{w}_k^t)$$

LMS algorithm reduces to $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + 2\mathrm{c}_k e_k \boldsymbol{x}_k$



$$w_{ij} \leftarrow w_{ij} + \frac{\mu\delta_j x_{ij}}{\sum x_k^2}$$

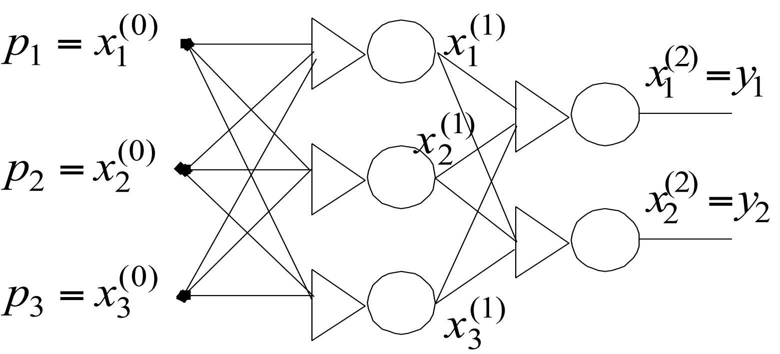
Normalization

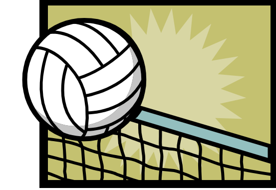
Iterative (adaptive) solution
(The optimal solution is never reached!)
MADALINE i-input, j-neuron

The Multilayer Perceptron

- The Generalized Delta Rule

Rumelhart, Hinton e Williams, PDP/MIT, 1986





Neuron Dynamics:

Processing Element (PE) j
in layer k
input i

$$s_j^{(k)} = w_{0j}^{(k)} + \sum_i w_{ij}^{(k)} x_i^{(k-1)}$$

with f (activation function) ***continuous differentiable***

$$x_j^{(k)} = f(s_j^{(k)})$$

Turning Point Question: How to find the error associated with an internal neuron??

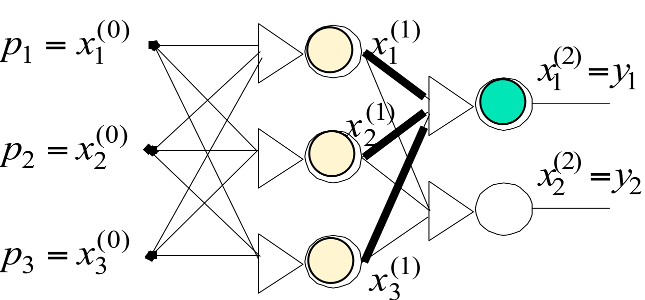
The generalized delta rule

Training

$$\varepsilon^2 = \sum_{j=1}^{m} (d_j - y_j)^2$$ - quadratic error

$$\mathbf{w}_j^{(k)} = (w_{oj}^{(k)}, w_{1j}^{(k)}, \dots, w_{mj}^{(k)})$$ - weigths of PE j

$$\mathbf{x}_j^{(k-1)} = (1, x_{1j}^{(k-1)}, \dots, x_{nj}^{(k-1)})$$ - input vector of PE j



With $s_j^{(k)} = \mathbf{w}_j^{(k)} \mathbf{x}_j^{(k-1)} \rightarrow \dfrac{\partial s_j^{(k)}}{\partial \mathbf{w}_j^{(k)}} = \mathbf{x}_j^{(k-1)}$

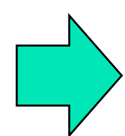
Instantaneous gradient:

$$\nabla_j^{(k)} = \frac{\partial \varepsilon^2}{\partial \mathbf{w}_j^{(k)}} = \left[\frac{\partial \varepsilon^2}{\partial w_{0j}^{(k)}}, \quad \frac{\partial \varepsilon^2}{\partial w_{1j}^{(k)}}, \quad \cdots \quad \frac{\partial \varepsilon^2}{\partial w_{mj}^{(k)}} \right]$$

$$\nabla_j^{(k)} = \frac{\partial \varepsilon^2}{\partial \mathbf{w}_j^{(k)}} = \frac{\partial \varepsilon^2}{\partial s_j^{(k)}} \frac{\partial s_j^{(k)}}{\partial \mathbf{w}_j^{(k)}}$$

so $\nabla_j^{(k)} = \dfrac{\partial \varepsilon^2}{\partial \mathbf{w}_j^{(k)}} = \dfrac{\partial \varepsilon^2}{\partial s_j^{(k)}} \mathbf{x}_j^{(k-1)}$

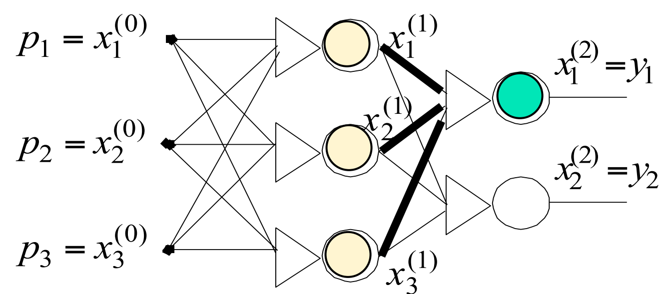
Defining the ***quadratic derivative error*** as $\delta_j^{(k)} = -\dfrac{1}{2} \dfrac{\partial \varepsilon^2}{\partial s_j^{(k)}}$



$$\nabla_j^{(k)} = -2 \delta_j^{(k)} \mathbf{x}_j^{(k-1)}$$

Gradient of the error with respect to the weights as function of the ***former layer*** signals!!

The generalized delta rule...



For the ***output layer***, the quadratic derivative error is:

$$\delta_j^{(k)} = -\frac{1}{2}\frac{\partial \sum_{i=1}^{N_k}(d_i - y_i)^2}{\partial s_j^{(k)}} = -\frac{1}{2}\frac{\partial \sum_{i=1}^{N_k}(d_i - f(s_i^{(k)}))^2}{\partial s_j^{(k)}}$$

The partial derivatives are 0 for $i \neq j$

$$\delta_j^{(k)} = -\frac{1}{2}\frac{\partial (d_j - f(s_j^{(k)}))^2}{\partial s_j^{(k)}} = -(d_j - f(s_j^{(k)}))\frac{\partial (d_j - f(s_j^{(k)}))}{\partial s_j^{(k)}} = (d_j - x_j^{(k)})f'(s_j^{(k)})$$

The output error associated with PE_j, in the last layer:

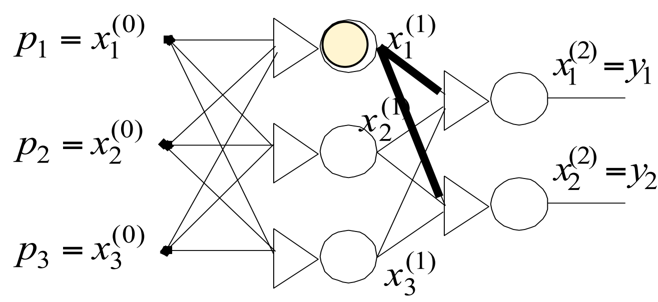
$$\varepsilon_j^{(k)} = d_j - x_j^{(k)} = d_j - y_j$$

Giving:

$$\delta_j^{(k)} = \varepsilon_j^{(k)} . f'(s_j^{(k)})$$

Remember,
"activation function, *f, **continuous differentiable***"

The generalized delta rule...



For a ***hidden layer*** *k*, the quadratic derivative error can be calculated using the linear outputs of layer *k+1*:

$$\delta_j^{(k)} = -\frac{1}{2}\frac{\partial \varepsilon^2}{\partial s_j^{(k)}} = -\frac{1}{2}\sum_{i=1}^{N_{k+1}}\left(\frac{\partial \varepsilon^2}{\partial s_i^{(k+1)}}\frac{\partial s_i^{(k+1)}}{\partial s_j^{(k)}}\right) \qquad \textit{(Chain Rule)}$$

$$= \sum_{i=1}^{N_{k+1}}\left(\left(-\frac{1}{2}\frac{\partial \varepsilon^2}{\partial s_i^{(k+1)}}\right)\frac{\partial s_i^{(k+1)}}{\partial s_j^{(k)}}\right) = \sum_{i=1}^{N_{k+1}}\left(\delta_i^{(k+1)}\frac{\partial s_i^{(k+1)}}{\partial s_j^{(k)}}\right)$$

Taking into account that $s_j^{(k)} = w_{0j}^{(k)} + \sum_{i=1}^{N_k} w_{ij}^{(k)} x_i^{(k-1)}$

$$\delta_j^{(k)} = \sum_{i=1}^{N_{k+1}}\left(\delta_i^{(k+1)}\frac{\partial}{\partial s_j^{(k)}}\left(w_{0i}^{(k+1)} + \sum_{l=1}^{N_k} w_{li}^{(k+1)} f\left(s_l^{(k)}\right)\right)\right)$$

$$\delta_j^{(k)} = \sum_{i=1}^{N_{k+1}}\left(\delta_i^{(k+1)}\sum_{l=1}^{N_k} w_{li}^{(k+1)}\frac{\partial}{\partial s_j^{(k)}} f\left(s_l^{(k)}\right)\right)$$

considering

$$\frac{\partial}{\partial s_j^{(k)}} f\left(s_l^{(k)}\right) = 0 \text{ if } l \neq j \text{ and that } \frac{\partial}{\partial s_j^{(k)}} f\left(s_j^{(k)}\right) = f'\left(s_j^{(k)}\right)$$

We have: $$\delta_j^{(k)} = \underbrace{\left(\sum_{i=1}^{N_{k+1}}\left(\delta_i^{(k+1)} w_{ji}^{(k+1)}\right)\right)}_{\equiv \varepsilon_j^{(k)}} . f'\left(s_j^{(k)}\right)$$

Finally, the ***quadratic derivative errror*** for a hidden layer:

$$\delta_j^{(k)} = \varepsilon_j^{(k)} . f'(s_j^{(k)})$$

The “Error Backpropagation” algorithm

1. $w_{ij}^{(k)} \leftarrow random$, initialize the network weigths

2. for (**x**,**d**), training pair, obtain y. Feedforward propagation: $\varepsilon^2 = \sum_{j=1}^{m}(d_j - y_j)^2$

3. k $\leftarrow last\ \ layer$

4. for each element j in the layer k do:

   Compute $\varepsilon_j^{(k)}$ using $\varepsilon_j^{(k)} = d_j - x_j^{(k)} = d_j - y_j$ if k is the last layer,

   $$\varepsilon_j^{(k)} = \sum_{i=1}^{N_{k+1}} \delta_i^{(k+1)} w_{ji}^{(k+1)}$$ if it is a hidden layer;

   Compute $\delta_j^{(k)} = \varepsilon_j^{(k)}.f'(s_j^{(k)})$

5. $k \leftarrow k-1$ if k > 0 go to step 4, else continue.

6. $\mathbf{w}_j^{(k)}(n+1) = \mathbf{w}_j^{(k)}(n) + 2\mu\delta_i^{(k)}\mathbf{x}_i^{(k)}$

7. For the next training pair go to step 2.

The Backpropagation Algorithm ***in practice***

1 – In the standard form BP is ***very slow***.

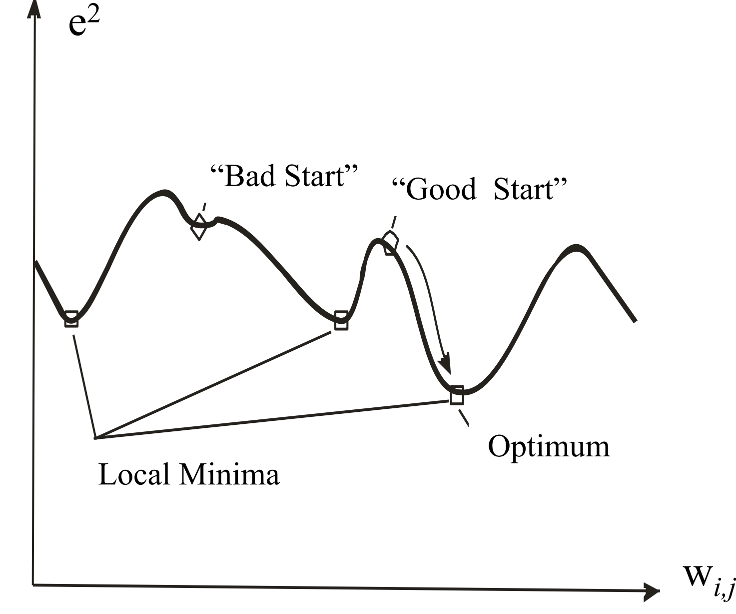
2 – BP Pathologies: ***paralysis*** in regions of small gradient.

3 – ***Initial conditions*** can lead to local minima.

4 – Stop conditions – number of epochs, $\Delta w_{ij} < \epsilon$

5 – BP variants
- trainbpm (with momentum)
- trainbpx (adaptive learning rate)
- ....
- trainlm (Levenberg-Marquard – *J*, Jacobian)

$$\Delta \mathbf{W}_j^{(k)} = (J^T J + \mu J)^{-1} J^T e$$



$e^2(w_{i,j})$ - Illustrative quadratic error as function of the weights

Obs: the error surface is, normally, unknown.

Steepest descent → go in the opposite direction of the ***local*** gradient ("downhill").

Computational Tools

- SNNS
- MatLab
  - Neural Network Toolbox
- NeuralWorks
- Java
- C++

- Hardware Implementations of RNAs

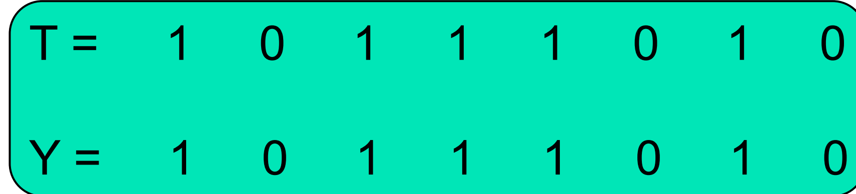
SNNS - Stuttgarter Neural Network Simulator

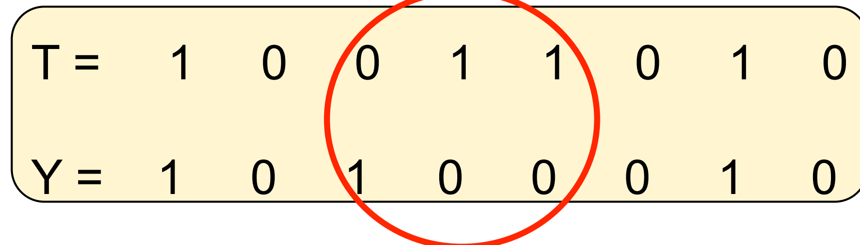
MatLab

- complete environment
  - -System Simulation
  - -Training
  - -Control

Demonstration - Perceptron

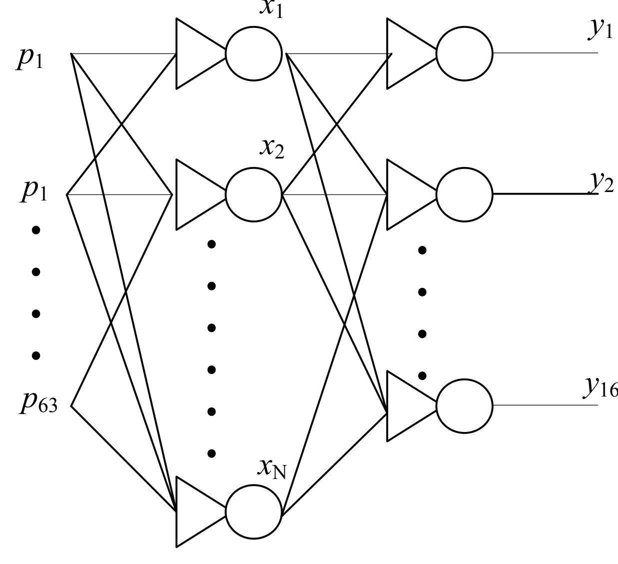
```
% Perceptron
% Training an ANN to learn to classify a non-linear problem
% Input Pattern
P=[ 0   0   0   0   1   1   1   1
    0   0   1   1   0   0   1   1
    0   1   0   1   0   1   0   1]
% Target
%T=[1 0 1 1 1 0 1 0]    % Linear separable
  T=[1 0 0 1 1 0 1 0]    %    non separable

% Try with Rosenblat's Perceptron
net=newp(P,T,'hardlim')

% train the network
net=train(net,P,T)

Y=sim(net,P)
```



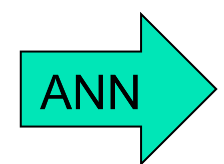


Demonstration - OCR

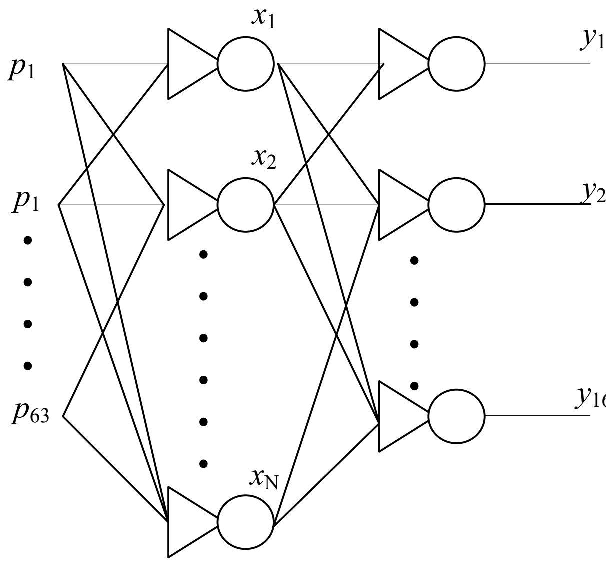
Training Vector



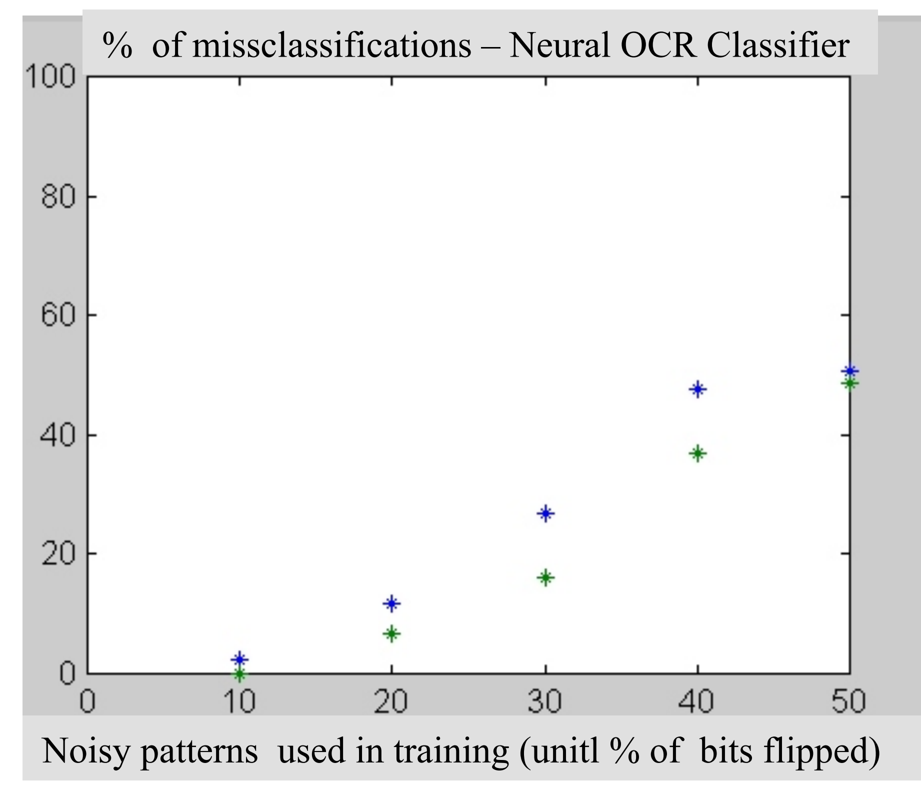
20 % Noise



Demonstration – OCR...



Training with 10 x (0,10,20,30,40,50) % noise



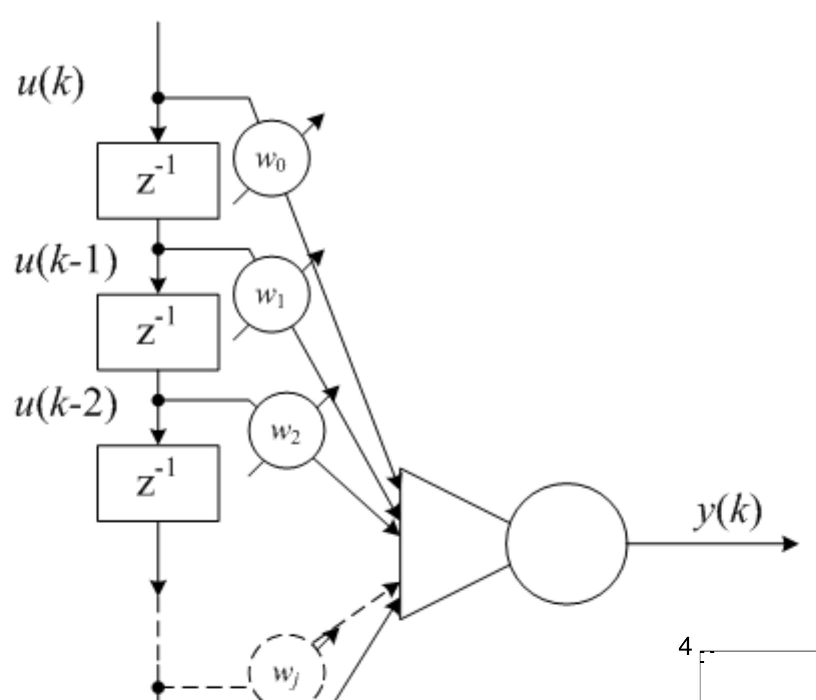
* - error without noisy training patterns
* - error using noisy training patterns

With Some Noisy Training Pattern
→ Learns how to treat "any" noise

Demonstration – LMS, ADALINE, FIR

$$y(k) = w_0 u(k) + w_1 u(k-1) + w_2 u(k-2) + \cdots w_n u(k-n)$$

$$\frac{Y(z)}{U(z)} = w_0 + w_1 z^{-1} + w_2 z^{-2} + \cdots w_n z^{-n}$$



FIR Model (always stable, only Zeros)

Obs: IIR Model is more compact, but can be unstable!

$$g_1(0-79.9\sec) = \frac{1}{s^2 + 0.2s + 1} \qquad g_2(80-150\sec) = \frac{3}{s^2 + 2s + 1}$$

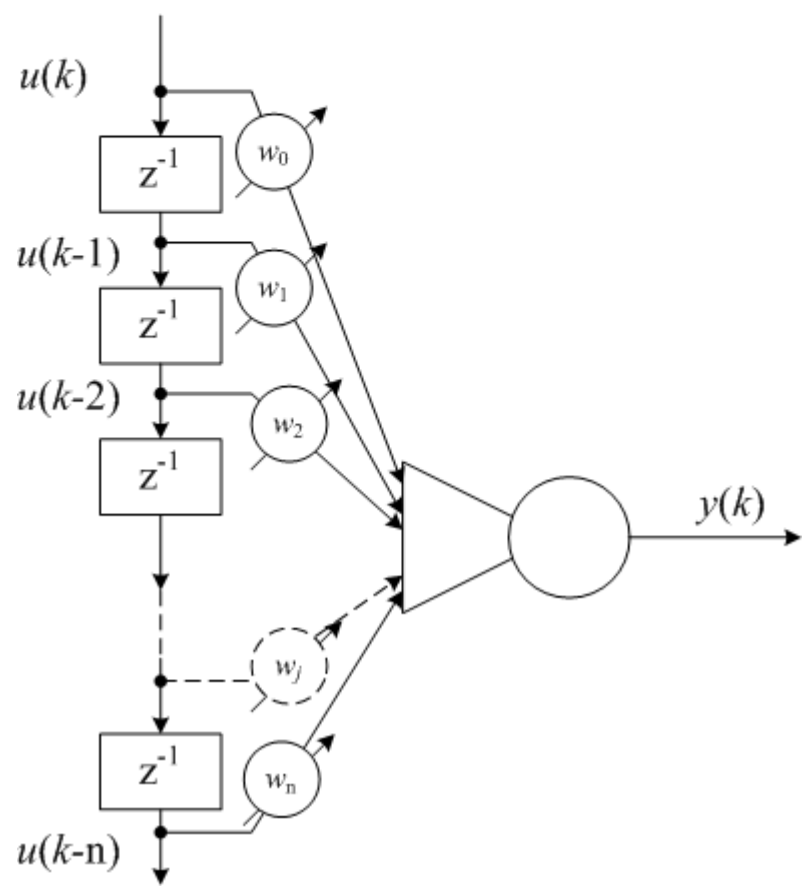
System changes at 80 sec

Sampling Time, $T_s = 0.1\sec$

4
2
0
-2
-4
0
50
100
150
sec

(TDL – Time Delay Line)

Demo – LMS, ADALINE, FIR...




```
% ADALINE - Adaptive dynamic system identification
% First sampled system - until 80 sec
g1=tf(1,[1 .2 1]), gd1=c2d(g1,.1)
 % Sytem changes dramatically - after 80 sec
g2=tf(3,[1 2 1]),gd2=c2d(g2,.1)

% Pseudo Random Binary Signal - good for identification
u=idinput(120*10,'PRBS',[0 0.01],[-1 1]);

% time vector
...
[y1,t1,x1]=lsim(gd1,u1,t1);
[y2,t2,x2]=lsim(gd2,u2,t2,x1);

% Creates new adaline nework with delayed inputs (FIR)
% Learning Rate = 0.09

net=newlin(t,y,[1 2 3 4 5 6 7 8 9 10],0.09)
[net,Y,E]=adapt(net,t,y)

% design an average transfer function
netd=newlind(t,y)
```

Demo – LMS, ADALINE, FIR...