Chapter 3. Introduction to the Neural Networks

Luis Miguel Bergasa

Index

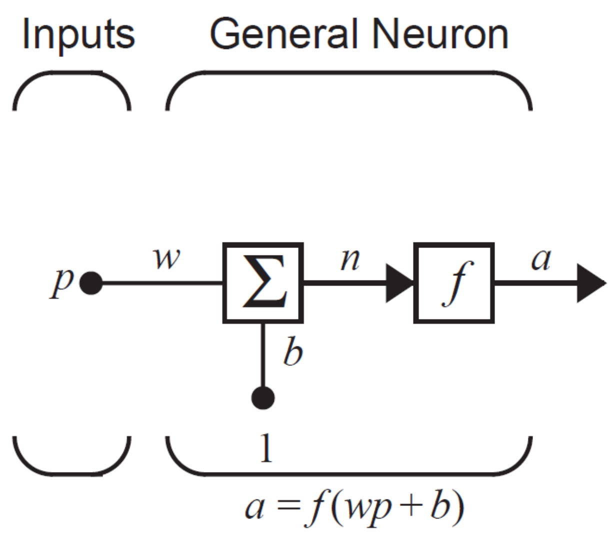
Neuron model and network architectures

Notation

- **Notation**
  - Scalars — small italic letters: a, b, c
  - Vectors — small **bold** nonitalic letters: **a,b,c**
  - Matrices — capital **BOLD** nonitalic letters: **A,B,C**

Neuron Model

➢ Single-Input Neuron




```
% Single-Input Neuron
net = linearlayer;
net = configure(net,[0],[0]);
view(net)
W = net.IW{1,1}
% W = 0
b = net.b{1}
%b = 0
% Assign arbitrary values to the
weights and bias
net.IW{1,1} = [3];
net.b{1} = -1.5;
% You can simulate the Net for a
particular input vector.
p = {2};
a = sim(net,p)
```

$$a = f(wp + b)$$

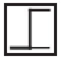
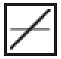
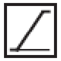
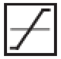
If, for instance, *w*=3, *p*=2 and *b*=-1.5 then:

$$a = f(3(2) - 1.5) = f(4.5)$$

Neural Networks

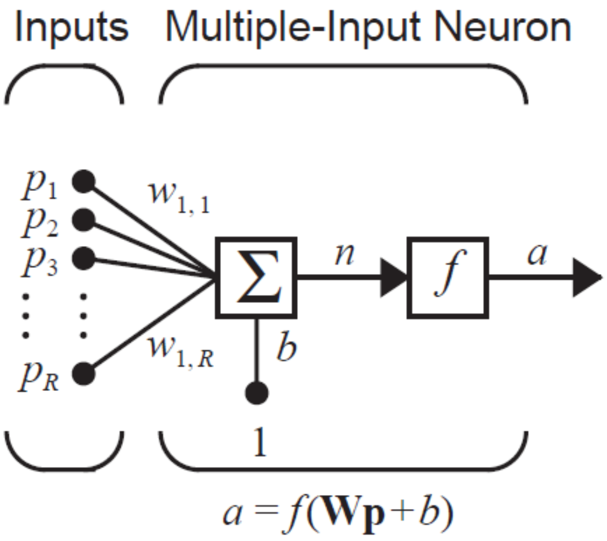
Neuron Model

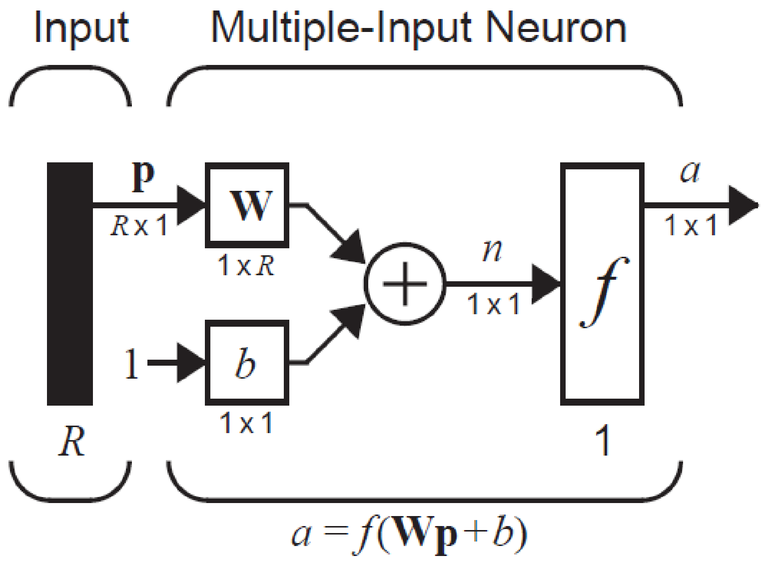
- **Transfer functions**

Name	Input/Output Relation	Icon	MATLAB Function
Hard Limit	$a = 0 \quad n < 0$ $a = 1 \quad n \geq 0$		hardlim
Symmetrical Hard Limit	$a = -1 \quad n < 0$ $a = +1 \quad n \geq 0$		hardlims
Linear	$a = n$		purelin
Saturating Linear	$a = 0 \quad n < 0$ $a = n \quad 0 \leq n \leq 1$ $a = 1 \quad n > 1$		satlin
Symmetric Saturating Linear	$a = -1 \quad n < -1$ $a = n \quad -1 \leq n \leq 1$ $a = 1 \quad n > 1$		satlins
Log-Sigmoid	$a = \frac{1}{1 + e^{-n}}$		logsig
Hyperbolic Tangent Sigmoid	$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$		tansig

MATLAB demo
nnd2n1

- **Multiple-input neuron**





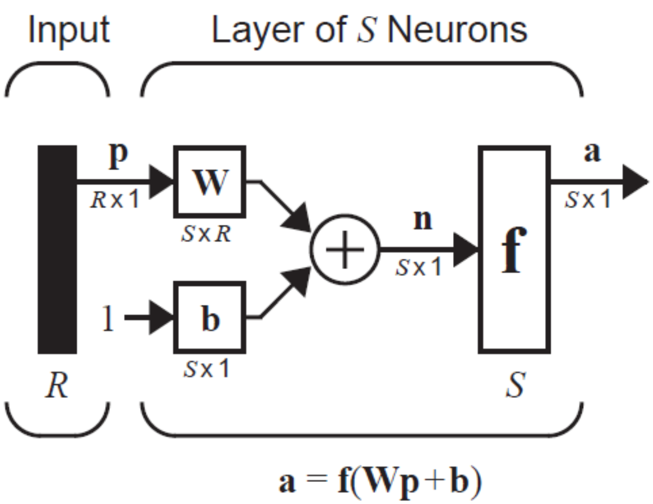
$$n = w_{1,1} p_1 + w_{1,2} p_2 + \ldots + w_{1,R} p_R + b$$

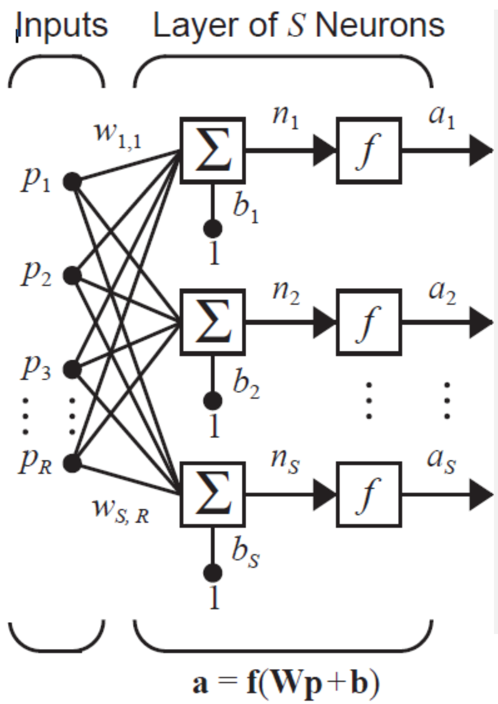
$$n = \mathbf{Wp} + b$$

$$a = f\left(\mathbf{Wp} + b\right)$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} \\ w_{1,2} \\ \vdots \\ w_{1,R} \end{bmatrix}^T \qquad \mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix}$$

- **Single-layer**



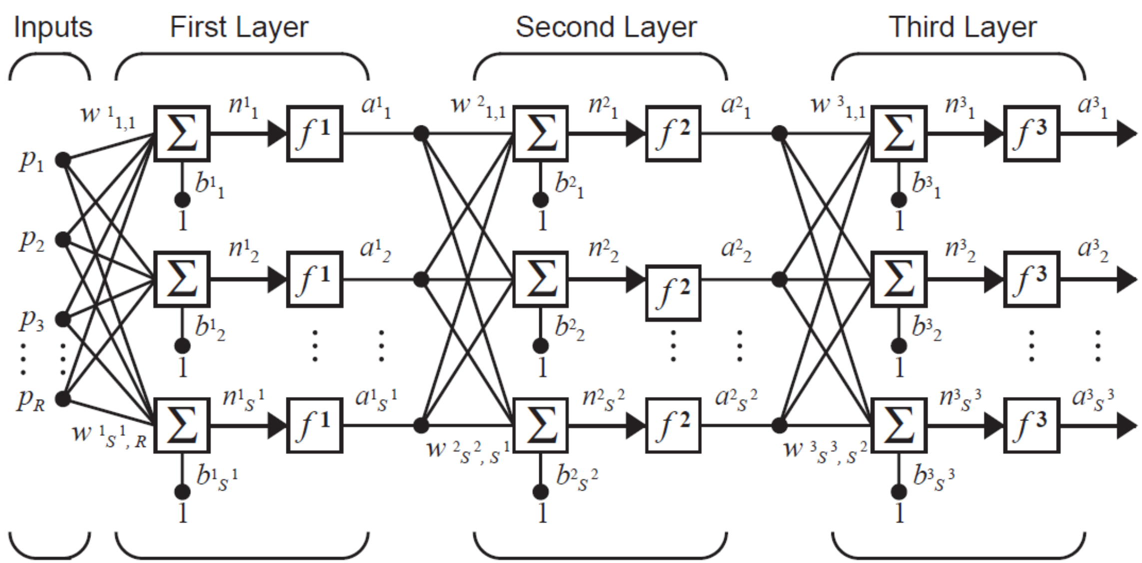



```
% Single-Layer Network
net = linearlayer;
net = configure(net,[0;0;0],[0;0]);
%p(3x1), a(2x1)
view(net)
% Assign arbitrary values to the
weights and bias
net.IW{1,1} = [1 2 3; 1 2 3];
%W(2x3)
net.b{1} = [1.5;-1.5]; %b(2x1)
% You can simulate the Net for a
particular input vector.
p = [2;1;-1];
a = sim(net,p)
%a =
%     2.5000
%    -0.5000
```

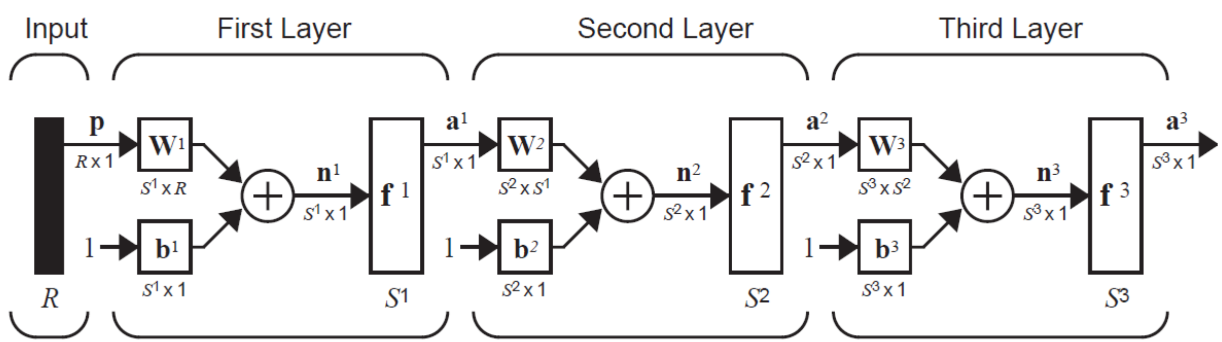
$$\mathbf{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,R} \\ w_{2,1} & w_{2,2} & \dots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \dots & w_{S,R} \end{bmatrix} \qquad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

Network Architectures

- **Multiple-layers**



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Neural Networks

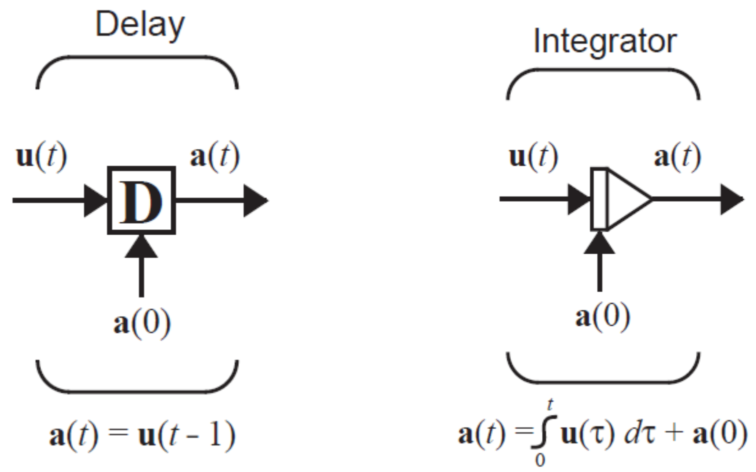
Network Architectures

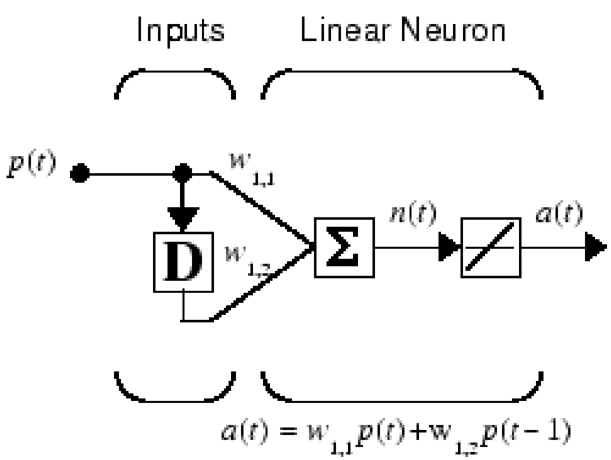
```
% Multi-Layer Network
% Create a two-layer feedforward network
net = feedforwardnet
view(net)
% Check the DIMMENSIONS: inputs, layers and outputs of the net
% Check the CONNECTIONS structure and change the bias
net.biasConnect=[1;0];
view(net)
% Check the SUBOBJECTS of the network
% View the layers subobject for the first layer
net.layers{1}
% Which is the number of neurons in the first layer?
% Change the number of neurons in the first layer to 3
net.layers{1}.dimensions=3;
% What is the transfer function used in the first layer?
% Change the transfer function to logsig
net.layers{1}.transferFcn = 'logsig';
% View the layerWeights subobject for the weight between layer 1 and layer 2
net.layerWeights{2,1}
% What is the size of this layer weights? Justify the answer
%Configure Neural Network Inputs and Outputs using sequential inputs
p = {[0;0] [0;1] [1;0] [1;1]}; % sequence of 4 inputs: p(2x1)
t = {[0;1] [1;0] [1;0] [0;1]}; % sequence of 4 targets: t(2x1)
net = configure(net,p,t);
view(net)
net.layerWeights{2,1}
% What is now the size of this layer weights?
```

Neural Networks

Network Architectures

- **Recurrent Networks**

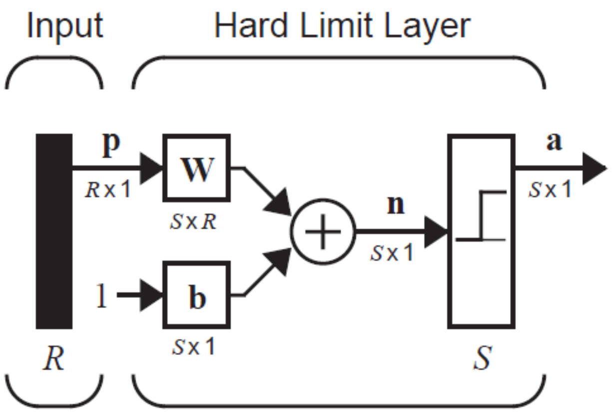





```
% Sequential Inputs in a Dynamic Network
net = linearlayer([0 1]); %[0 1] delay elements
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
view(net);
% Assign the weight matrix to be W = [1 2]
net.IW{1} = [1 2];
% Suppose that the input sequence is:
P = {1 2 3 4}; % Elements of a cell array
% You can now simulate the network:
A = net(P)
% A = [1] [4] [7] [10]
```

Perceptron

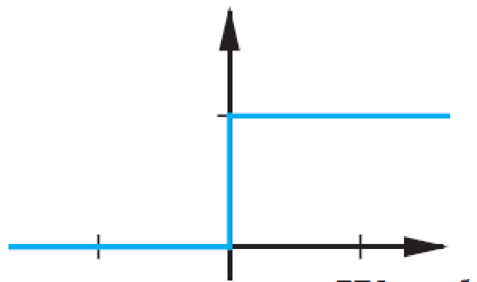
- **Architecture**



$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp}+\mathbf{b})$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix} \quad {}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,1} \\ \vdots \\ w_{i,R} \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}$$

$$a = hardlim\,(n)$$



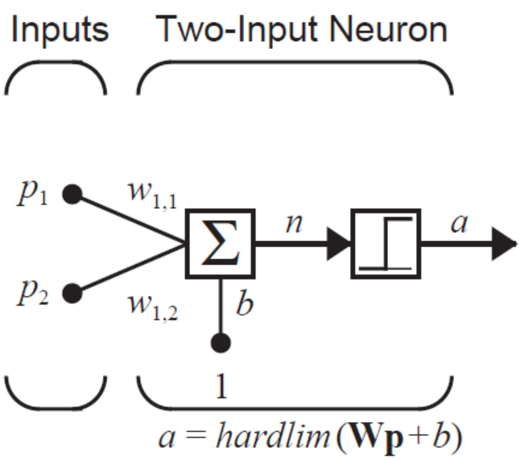
$$n = \mathbf{Wp} + b$$

$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i)$$

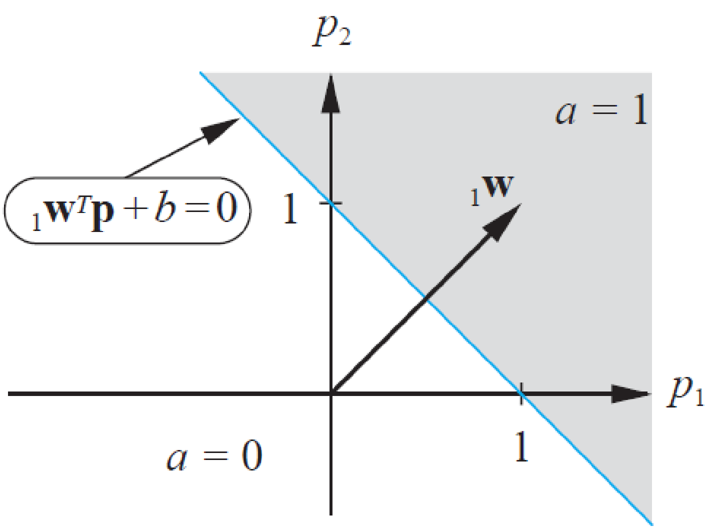
$$a = hardlim(n) = \begin{cases} 1 & if \quad n \geq 0 \\ 0 & otherwise \end{cases}$$

Perceptron

- **Single-Neuron Perceptron**



$$a = a_1 = hardlim(n_1) = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) =$$
$$= hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b)$$



- Decision boundary

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = 0$$

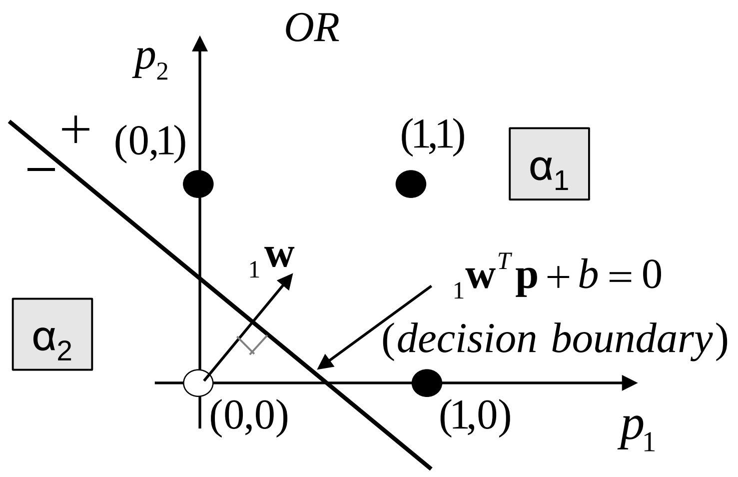
$$If\ w_{1,1} = 1, w_{1,2} = 1,\ b = -1 \ \rightarrow\ p_2 = -p_1 + 1$$

- The boundary is orthogonal to ${}_1\mathbf{w}$

Perceptron

Single-Neuron Perceptron

- How the $_1\mathbf{w}$ is calculated to implement an specific function?
- Supervised learning. Example: OR function



1. To select a decision boundary that separates the dark circles and the light circles
2. To choose a weight vector orthogonal to the decision boundary

$$\mathbf{w}=\begin{bmatrix}0.5\\0.5\end{bmatrix}$$

3. To find the bias (b) picking a point on the decision boundary that satisfies the equation $_1\mathbf{w}^T\mathbf{p}+b=0$

$$_1\mathbf{w}^T\mathbf{p}+b=\begin{bmatrix}0.5 & 0.5\end{bmatrix}\begin{bmatrix}0\\0.5\end{bmatrix}+b=0.25+b=0 \quad \Rightarrow b=-0.25$$

Perceptron learning rule

- Reward/punishment learning
- Supervised learning. Training set: $\{\mathbf{p}_1, t_1\}, \{\mathbf{p}_2, t_2\}, \ldots, \{\mathbf{p}_Q, t_Q\}$
- We define an error variable (e)
- We apply a recursive algorithm

$$\mathbf{w}(1) = random$$

$$e(k) = (t(k) - a(k))$$

$$If\ \ t(k) = 1\ and\ a(k) = 0 \rightarrow e(k) = 1 \rightarrow {}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + \mathbf{p}(k)$$

$$If\ \ t(k) = 1\ and\ a(k) = 1 \rightarrow e(k) = 0 \rightarrow {}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k)$$

$$If\ \ t(k) = 0\ and\ a(k) = 1 \rightarrow e(k) = -1 \rightarrow {}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) - \mathbf{p}(k)$$

$$If\ \ t(k) = 0\ and\ a(k) = 0 \rightarrow e(k) = 0 \rightarrow {}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k)$$

$$\Downarrow$$

$${}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + e(k)\mathbf{p}(k)$$

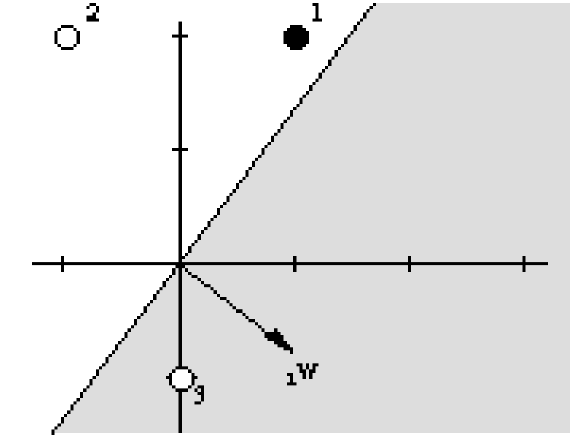
$$b(k+1) = b(k) + e(k)$$

Neural Networks

Perceptron: Example

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\} \quad \left\{\mathbf{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\} \quad \left\{\mathbf{p}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}$$

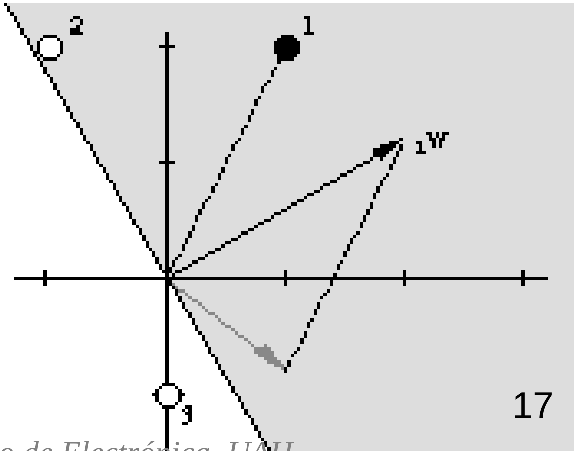
Random initial weight: $\quad {}_1\mathbf{w}(1) = \begin{bmatrix}1.0\\-0.8\end{bmatrix}$



Present $\mathbf{p}_1$ to the network:

a(1) $= hardlim({}_1\mathbf{w}^T\mathbf{p}_1) = hardlim\left(\begin{bmatrix}1.0 & -0.8\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix}\right)$

a(1) $= hardlim(-0.6) = 0$



$$e(1) = \big(t(1) - a(1)\big) = (1-0) = 1 \quad Incorrect\ classification$$

$${}_1\mathbf{w}(2) = {}_1\mathbf{w}(1) + e(1)\mathbf{p}(1) = \begin{bmatrix}1.0\\-0.8\end{bmatrix} + \begin{bmatrix}1\\2\end{bmatrix} = \begin{bmatrix}2.0\\1.2\end{bmatrix}$$

Perceptron: Example

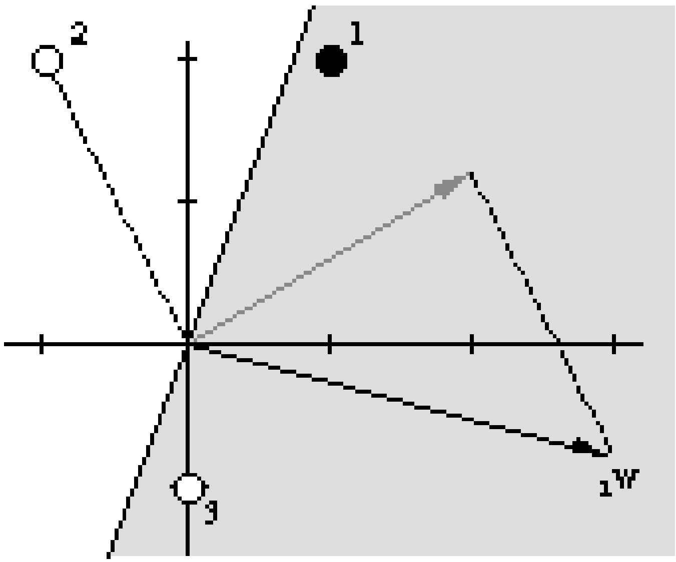
Present $\mathbf{p}_2$ to the network:

$$a(2) = hardlim({}_1\mathbf{w}^T\mathbf{p}_2) = hardlim\left(\begin{bmatrix}2.0 & 1.2\end{bmatrix}\begin{bmatrix}-1\\2\end{bmatrix}\right)$$

$$a(2) = hardlim(\ 0.4\ ) = 1$$

$$e(2) = \left(t(2) - a(2)\right) = \left(0 - 1\right) = -1 \quad \textit{Incorrect classification}$$

$${}_1\mathbf{w}(3) = {}_1\mathbf{w}(2) + e(2)\mathbf{p}(2) = \begin{bmatrix}2.0\\1.2\end{bmatrix} - \begin{bmatrix}-1\\2\end{bmatrix} = \begin{bmatrix}3.0\\-0.8\end{bmatrix}$$



Perceptron: Example

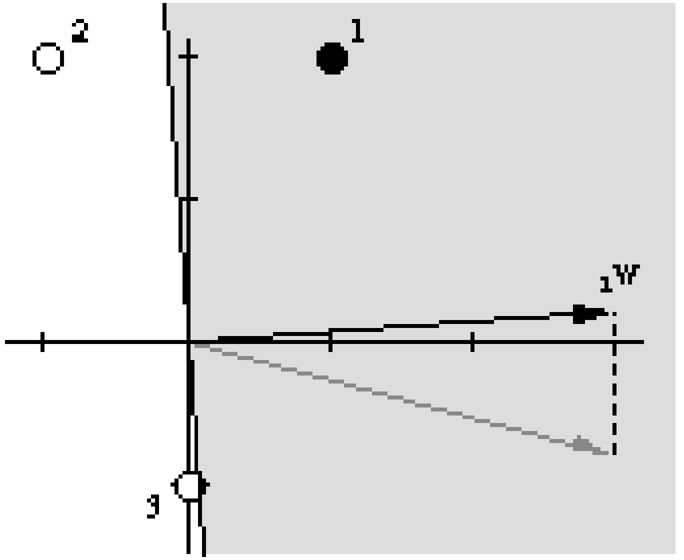
Present $\mathbf{p}_3$ to the network:

$$a(3) = hardlim({}_1\mathbf{w}^T \mathbf{p}_3) = hardlim\left(\begin{bmatrix} 3.0 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right)$$

$$a(3) = hardlim(0.8) = 1$$

$$e(3) = \big(t(3) - a(3)\big) = (0 - 1) = -1 \quad \textit{Incorrect classification}$$

$${}_1\mathbf{w}(4) = {}_1\mathbf{w}(3) + e(3)\mathbf{p}(3) = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}$$



Patterns are now correctly classified.

$$for\ k > 3 \rightarrow e(k) = 0 \ \rightarrow {}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k)$$

MATLAB demo
nnd4pr

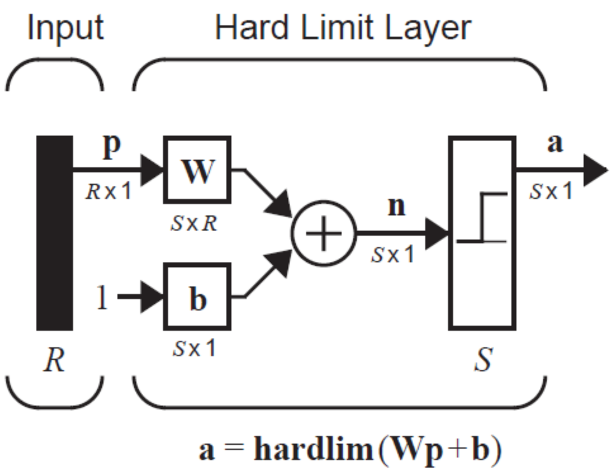
Perceptron: Example

```
% Perceptron example with sequential inputs
net = perceptron;
% The input vectors and targets are
p = {[1;2] [-1;2] [0;-1]}
t = {1 0 0}
net = configure(net,p,t);
% Remove bias
net.biasConnect = 0;
% Weights initialization
net.IW{1,1} = [1 -0.8];
view(net);
% Train the network
[net,a,e,pf] = adapt(net,p,t);
% Intermediate values
a % a = {[0]} {[1]} {[1]}
e % e = {[1]} {[-1]} {[-1]}
% The new weights after applying all input vectors are:
w = net.iw{1,1} % w = [3 0.2]
% Simulate the trained network for each of the inputs.
a = net(p) % a = 1 0 0
% Calculate the error
error = cell2mat(a)-cell2mat(t) % error = 0 0 0
%The outputs equal the targets, so you don't need to train the network
%for more than one pass. It this wasn't the case you should try more epochs.
%net.trainParam.epochs = 1;
```

Perceptron

➢ Multiple-Neuron Perceptron

- There will be one decision boundary for each neuron
- A single-neuron perceptron can classify input vectors into two categories
- A multiple-neuron perceptron can classify inputs into many categories (2^S)



$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \qquad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_S \end{bmatrix}$$

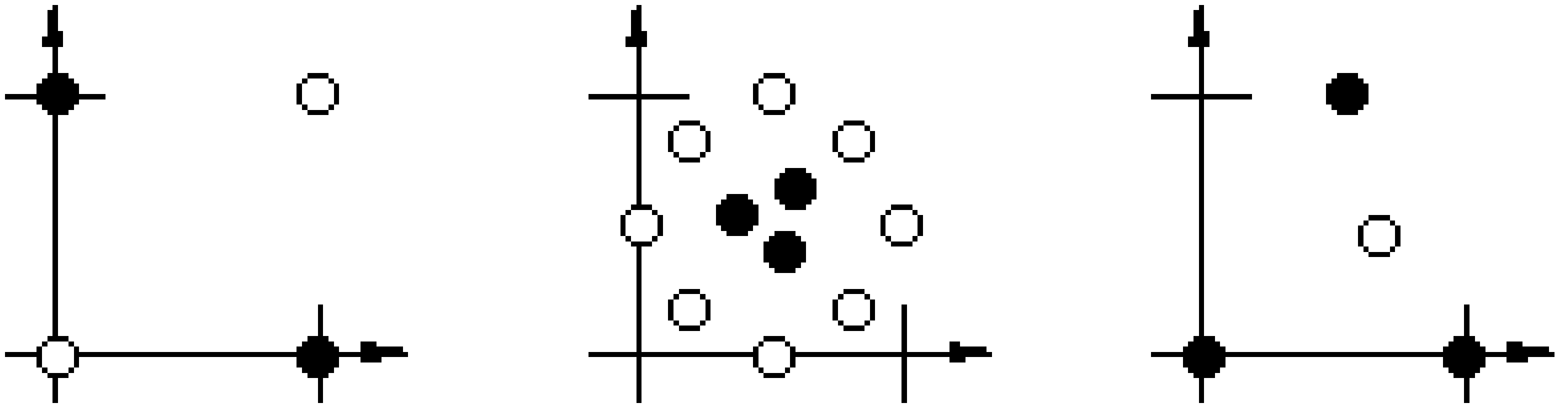
- We can generalize the perceptron rule for multiple-neuron perceptron

$$\mathbf{W}(k+1) = \mathbf{W}(k) + \mathbf{e}(k)\mathbf{p}^T(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + \mathbf{e}(k)$$

Perceptron. Limitations

- The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as **a solution exists**
- The perceptron can be used to classify input vectors that can be separated by a **linear boundary**
- Many **problems** are **not linearly separable**



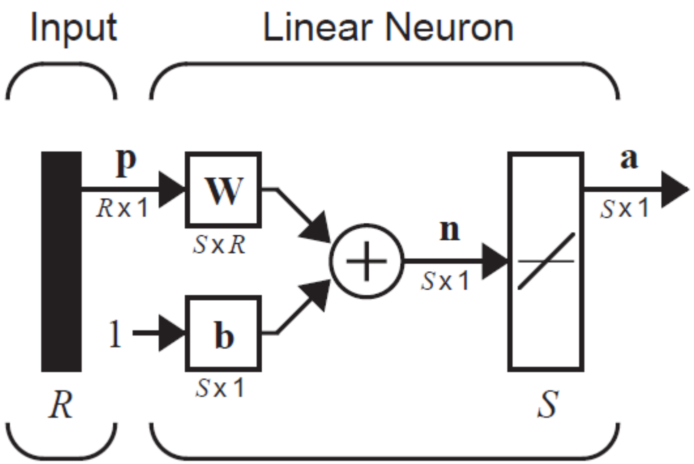
ADALINE Network and Widrow-Hoff Learning

ADALINE

- **ADALINE** (ADAptive LInear NEuron) network
- **Similar to the perceptron** except that its transfer function is linear
- They suffer from the **same limitations** that perceptron (linearly separable problems)
- Its **learning rule** called **LMS (Least Mean Square)** is more powerfull than the perceptron learning rule
- **Widrow-Hoff learning** is an approximate steepest descent algorithm, in which the performance index is mean square error (**LMS**)

ADALINE

- **Architecture:**



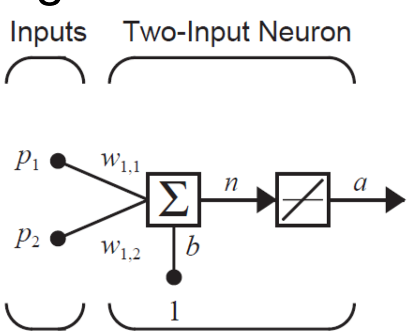
$\mathbf{a} = \mathbf{purelin}(\mathbf{Wp}+\mathbf{b})$

$$\mathbf{a} = purelin(\mathbf{Wp}+\mathbf{b}) = \mathbf{Wp}+\mathbf{b}$$

$$a_i = purelin(n_i) = purelin({}_i\mathbf{w}^T\mathbf{p}+b_i) = {}_i\mathbf{w}^T\mathbf{p}+b_i$$

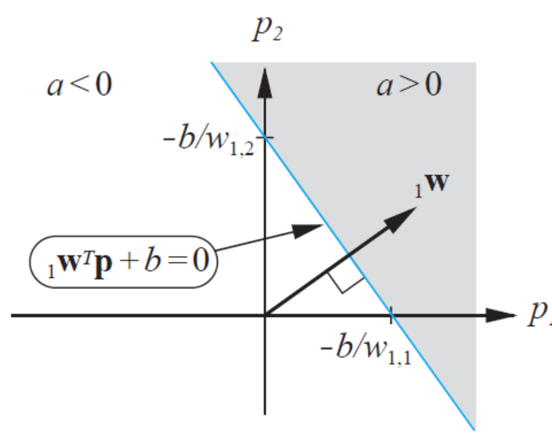
$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,1} \\ \vdots \\ w_{i,R} \end{bmatrix}$$

- Single ADALINE



$a = purelin(\mathbf{Wp}+b)$

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$



ADALINE

- **LMS algorithm. One step learning rule**
  - Supervised learning. Training set: $\{\mathbf{p}_1, t_1\}, \{\mathbf{p}_2, t_2\}, \ldots, \{\mathbf{p}_Q, t_Q\}$
  - To simply we will lump all of the parameters into one vector

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \qquad \longrightarrow \qquad a = {}_1\mathbf{w}^T \mathbf{p} + b \;\rightarrow\; a = \mathbf{x}^T \mathbf{z}$$

  - ADALINE network *mean square error*

$$\begin{aligned} F(\mathbf{x}) &= E\left[e^2\right] = E\left[(t-a)^2\right] = E\left[(t - \mathbf{x}^T\mathbf{z})^2\right] \\ &= E\left[t^2 - 2t\mathbf{x}^T\mathbf{z} + \mathbf{x}^T\mathbf{z}\mathbf{z}^T\mathbf{x}\right] \\ &= E\left[t^2\right] - 2\mathbf{x}^T E[t\mathbf{z}] + \mathbf{x}^T E\left(\mathbf{z}\mathbf{z}^T\right)\mathbf{x} \end{aligned}$$

  - This can be written in the following convenient form:

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}$$

$$c = E\left[t^2\right], \quad \mathbf{h} = E[t\mathbf{z}], \quad \mathbf{R} = E\left(\mathbf{z}\mathbf{z}^T\right)$$

R: input correlation matrix

h: target/input cross-correlation

ADALINE

- **LMS algorithm. One step learning rule**
  - The LMS is a quadratic function where:

$$F(\mathbf{x}) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} \qquad \mathbf{d} = -2\mathbf{h} \;\; and \;\; \mathbf{A} = 2\mathbf{R}$$

  - The stationary point of F(**x**) will be:

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{h} + 2\mathbf{R}\mathbf{x}$$

$$-2\mathbf{h} + 2\mathbf{R}\mathbf{x} = 0$$

$$\boxed{\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}}$$

```
%Linear System Design
P = [1 2 3];
T= [2.0 4.1 5.9];
net = newlind(P,T);
Y = net(P)
% Y = 2.0500 4.0000 5.9500
```

- If **R** is positive definite there will be a unique stationary point, which will be a strong minimum
- If **R** has some zero eigenvalues, the performance index will either have a weak minimum or no minimum, depending on the vector **d=-2h**

ADALINE

- **Incremental learning rule. LMS Widrow-Hoff algorithm**
  - It is not desirable or convenient in the practice to calculate **h** and $\mathbf{R}^{-1}$
  - We will use an approximate steepest descent algorithm (estimated gradient)
  - We aproximate the LMS by the squared error at iteration k to be used in the steepest descent algorithm with constant learning rate:

$$F(\mathbf{x}) \approx \hat{F}(\mathbf{x}) = \left(t(k) - a(k)\right)^2 = e^2(k)$$

$$\mathbf{x}(k+1) = \mathbf{x}(k) - \alpha \nabla \hat{F}(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}(k)} = \mathbf{x}(k) - \alpha \left.\frac{\partial \hat{F}(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}(k)}$$

  - Gradient estimation at each iteration (*stochastic gradient)*

$$\frac{\partial \hat{F}(\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial e^2(k)}{\partial \mathbf{x}} \rightarrow \begin{cases} \left[\dfrac{\partial e^2(k)}{\partial \boldsymbol{w}}\right]_j = \dfrac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\dfrac{\partial e(k)}{\partial w_{1,j}} \quad for \quad j = 1,2,\ldots,R \\ \left[\dfrac{\partial e^2(k)}{\partial b}\right]_{R+1} = \dfrac{\partial e^2(k)}{\partial b} = 2e(k)\dfrac{\partial e(k)}{\partial b} \end{cases}$$

ADALINE

- **Incremental learning rule. LMS Widrow-Hoff algorithm**

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial[t(k)-a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}\left[t(k)-\left({}_1\mathbf{w}^T\mathbf{p}(k)+b\right)\right] = \frac{\partial}{\partial w_{1,j}}\left[t(k)-\left(\sum_{i=1}^{R} w_{1,i}p_i(k)+b\right)\right]$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \qquad \frac{\partial e(k)}{\partial b} = -1$$

$$\mathbf{x}(k+1) = \mathbf{x}(k) - \alpha \left.\frac{\partial \hat{F}(\mathbf{x})}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}(k)} = \mathbf{x}(k) + 2\alpha e(k)\mathbf{z}(k) \qquad \mathbf{z} = \begin{bmatrix}\mathbf{p}\\ 1\end{bmatrix}$$

$$\boxed{\begin{aligned} {}_i\mathbf{w}(k+1) &= {}_i\mathbf{w}(k) + 2\alpha e_i(k)\mathbf{p}(k) \\ b_i(k+1) &= b_i(k) + 2\alpha e_i(k) \end{aligned}}$$

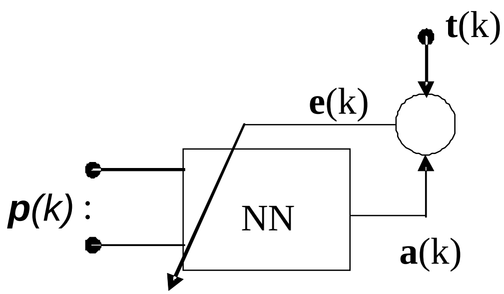
- In matrix notation:

$$\boxed{\begin{aligned} \mathbf{W}(k+1) &= \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k) \\ \mathbf{b}(k+1) &= \mathbf{b}(k) + 2\alpha\mathbf{e}(k) \end{aligned}}$$

ADALINE

- **Incremental vs batch learning rule. LMS Widrow-Hoff algorithm**
  - Incremental

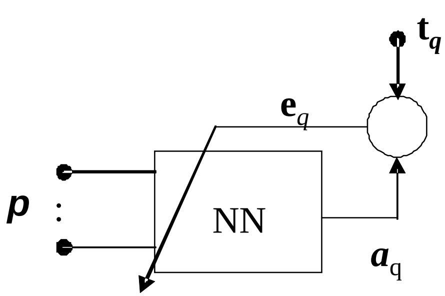
$$\mathbf{p} = \begin{bmatrix} p_1(1) & p_1(2) & \cdots & p_1(Q) \\ p_2(1) & p_2(2) & \cdots & p_2(Q) \\ \vdots & \vdots & \vdots & \vdots \\ p_R(1) & p_R(2) & \cdots & p_R(Q) \end{bmatrix} = \left[\ \mathbf{p}(1) \quad \mathbf{p}(2) \quad \cdots \quad \mathbf{p}(Q)\ \right]$$



$$F(\mathbf{x}) \approx \hat{F}(\mathbf{x}) = \left(\mathbf{t}(k) - \mathbf{a}(k)\right)^T \left(\mathbf{t}(k) - \mathbf{a}(k)\right) = \mathbf{e}^T(k)\mathbf{e}\ (k)$$

$$\boxed{\begin{aligned} \mathbf{W}(k+1) &= \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^T(k) \\ \mathbf{b}(k+1) &= \mathbf{b}(k) + 2\alpha\mathbf{e}(k) \end{aligned}}$$

  - Batch



$$F(\mathbf{x}) = E\left[\mathbf{e}^T\mathbf{e}\right] = E\left[(\mathbf{t} - \mathbf{a})^T(\mathbf{t} - \mathbf{a})\right] = \frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q) = \frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{e}_q)^T(\mathbf{e}_q)$$

$$\nabla F(\mathbf{x}) = \nabla\left\{\frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q)\right\} = \frac{1}{Q}\sum_{q=1}^{Q}\nabla\left\{(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q)\right\}$$

$$\boxed{\begin{aligned} \mathbf{W}(n+1) &= \mathbf{W}(n) + \frac{2\alpha}{Q}\sum_{q=1}^{Q}\mathbf{e}_q\mathbf{p}_q^T \\ \mathbf{b}(n+1) &= \mathbf{b}(n) + \frac{2\alpha}{Q}\sum_{q=1}^{Q}\mathbf{e}_q \end{aligned}}$$

ADALINE

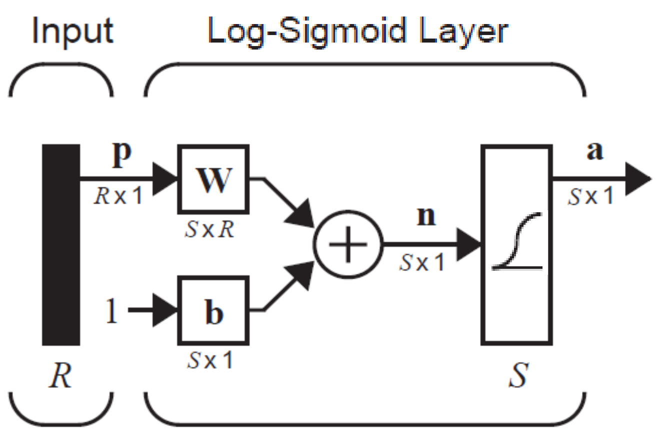
```
% Incremental Training of ADALINE (Static Network)
% Inputs and targets as sequences:
p = {[2;2] [1;-2] [-2;2] [-1;1]};
t = {4 5 6 7};
% First, set the delays and initial learning rate to zero. Also, set up the
% network with zero initial weights and biases to show the effect of incremental training.
net = linearlayer(0,0); % 1°: delays=0, 2°: lr=0
net = configure(net,p,t);
net.IW{1,1} = [0 0];
net.b{1} = 0;
view(net);
% train the network incrementally
[net,a,e,pf] = adapt(net,p,t);
% a = [0] [0] [0] [0]
% e = [4] [5] [6] [7]
% If you now set the learning rate to 0.1 you can see how the network is
% adjusted as each input is presented:
net.inputWeights{1,1}.learnParam.lr = 0.1;
net.biases{1,1}.learnParam.lr = 0.1;
net.trainParam.epochs = 1;
[net,a,e,pf] = adapt(net,p,t);
% a = [0] [-0.4] [-2.3] [3.47]
% e = [4] [5.4] [8.3] [3.53]
% To reach a better adjustment more epochs in the training are needed
net.trainParam.epochs = 100;
net.inputWeights{1,1}.learnParam.lr = 0.01;
net.biases{1,1}.learnParam.lr = 0.01;
net = adapt(net,p,t);
a = net(p)
e = cell2mat(a)-cell2mat(t)
% a = {[4.1716]}    {[-1.6165]}    {[6.6617]}    {[4.4365]}
% e = 0.1716   -6.6165    0.6617   -2.5635
```

ADALINE

```
% Batch Training of ADALINE (Static Network)
% Concurrent inputs and targets:
p = [[2;2] [1;-2] [-2;2] [-1;1]];
t = [4 5 6 7];
% Cells data can be used too (sequential data)
%p = {[2;2] [1;-2] [-2;2] [-1;1]};
%t = {4 5 6 7};
net = linearlayer(0,0.01);
net = configure(net,p,t);
net.IW{1,1} = [0 0];
net.b{1} = 0;
net.trainParam.epochs = 100;
% train the network in batch mode
net = train(net,p,t); %train is used for batch training
a = sim(net,p)
e = a-t
% a = 4.0502    4.8434    6.4645    5.9084
% e = 0.0502   -0.1566    0.4645   -1.0916
```

ADALINE Log-Sigmoid

- **Architecture**



a=logsig(Wp+b)

$$\mathbf{a} = \text{logsig}\left(\mathbf{W}\mathbf{p} + \mathbf{b}\right)$$

$$a_i = \text{logsig}\left(n_i\right) = \text{logsig}\left({}_i\mathbf{w}^T\mathbf{p} + b_i\right)$$

NN topology
$$\begin{cases} \hat{F} = \mathbf{e}^T(k)\mathbf{e}(k) = \sum_{i=1}^{S}\left(t_i - a_i\right)^2 = \sum_{i=1}^{S} e_i^2 \\ a_i = f\left(n_i\right) = \dfrac{1}{1+e^{-n_i}} \\ n_i = \sum_{j=1}^{R} w_{i,j}\, p_j + b_i \end{cases}$$

$$w_{i,j}(k+1) = w_{i,j}(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}} \qquad \frac{\partial \hat{F}}{\partial w_{i,j}} \underset{\substack{\uparrow \\ \textit{chain rule}}}{=} \frac{\partial \hat{F}}{\partial a_i}\frac{\partial a_i}{\partial n_i}\frac{\partial n_i}{\partial w_{i,j}} = \left(-2e_i\right)\left(a_i(1-a_i)\right)\left(p_j\right)$$

$$b_i(k+1) = b_i(k) - \alpha \frac{\partial \hat{F}}{\partial b_i} \qquad \frac{\partial \hat{F}}{\partial b_i} \underset{\substack{\uparrow \\ \textit{chain rule}}}{=} \frac{\partial \hat{F}}{\partial a_i}\frac{\partial a_i}{\partial n_i}\frac{\partial n_i}{\partial b_i} = \left(-2e_i\right)\left(a_i(1-a_i)\right)(1)$$

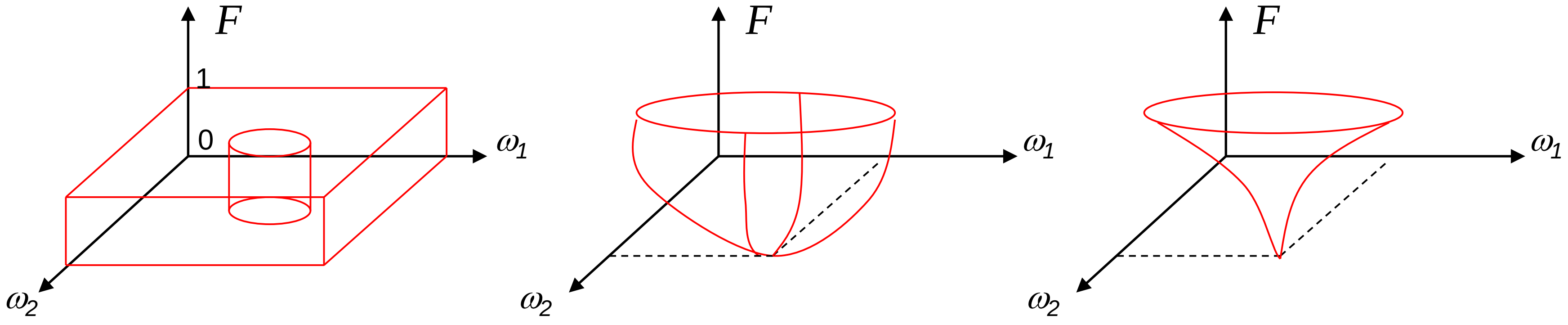
$$\boxed{\begin{aligned} w_{i,j}(k+1) &= w_{i,j}(k) + 2\alpha e_i(k)\left(a_i(k)(1-a_i(k))\right) p_j(k) \\ b_i(k+1) &= b_i(k) + 2\alpha e_i(k)\left(a_i(k)(1-a_i(k))\right) \end{aligned}} \qquad \boxed{\begin{aligned} \mathbf{w}\ (k+1) &= \mathbf{w}(k) + 2\alpha\mathbf{e}(k)\left(\mathbf{a}(k)(1-\mathbf{a}(k))\right)\mathbf{p}(k) \\ \mathbf{b}(k+1) &= \mathbf{b}(k) + 2\alpha\mathbf{e}(k)\left(\mathbf{a}(k)(1-\mathbf{a}(k))\right) \end{aligned}}$$

ADALINE Log-Sigmoid

```
% Batch Training of ADALINE Log-Sigmoid (Static Network)
% Concurrent inputs and targets:
p = [[2;2] [1;-2] [-2;2] [-1;1]];
t = [4 5 6 7];
net = linearlayer(0,0.01);
net = configure(net,p,t);
net.IW{1,1} = [0 0];
net.b{1} = 0;
net.layers{1}.transferFcn = 'logsig';
net.trainParam.epochs = 100;
% train the network in batch mode
t1=t/max(t); % Normalization process
net = train(net,p,t1); %train is used for batch training
a1 = sim(net,p);
a=a1*max(t) % Desnormalization process
e = a-t
% a = 4.0506    3.7104    5.8037    5.2982
% e = 0.0506   -1.2896   -0.1963   -1.7018
```

ADALINE

- **PERCEPTRON vs. ADALINE**
  - Perceptron presents a LMS error function (F) abrupt (discreet)
    - Ideal for convergence after some iterations
    - It is not possible to train it by using backpropagation because F is not derivable
  - Linear ADALINE presents a LMS error function (F) soft and derivable, with small distance between the minimum and the adjacent points
    - Steepest descent algorithm with slow convergence to the minimum
  - Logsig-ADALINE presents a LMS error function (F) abrupt and derivable. Similar to perceptron but we can use error backpropagation



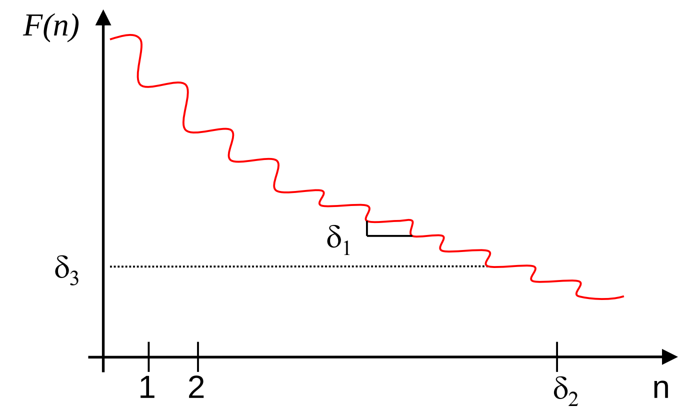
ADALINE

- **Stop in the learning process (incremental and batch)**

$$F(n) = \frac{1}{Q}\sum_{q=1}^{Q}\left(\mathbf{t}_q - \mathbf{a}_q(n)\right)^T\left(\mathbf{t}_q - \mathbf{a}_q(n)\right)$$
$$= \frac{1}{Q}\sum_{q=1}^{Q}\left(\mathbf{e}_q(n)\right)^T\left(\mathbf{e}_q(n)\right)$$

NN IN **p**(k)	Targets **t**(k)	NN OUT **a**(k)	Error **e**(k)
p(1)	**t**(1)	**a**(1)	**e**(1)
p(2)	**t**(2)	**a**(2)	**e**(2)
:	:	:	:
p(Q)	**t**(Q)	**a**(Q)	**e**(Q)

n iterations



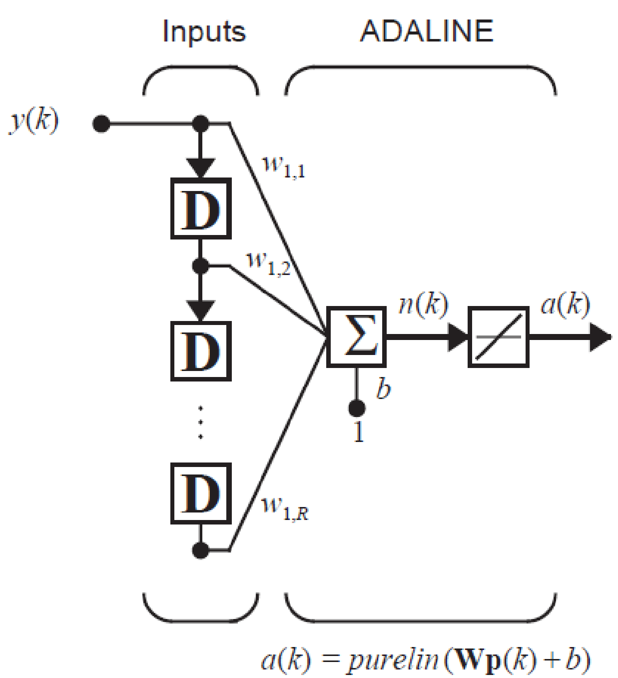
- 1) LMS error stops falling $\left|F(n+1) - F(n)\right| < \delta_1$
  - The system stops to learn
- 2) Maximum iteration times $n < \delta_2$
- 3) Maximum accepted LMS error $F(n) < \delta_3$

ADALINE

- **Stability**
  - Stability of the steepest descent algorithm:

$$0 < \alpha < 1/\lambda_{\max}$$

    - $\lambda_{\max}$ highest eigenvalue of the input correlation matrix **R (A=2R)**
- **Adaptive filtering**



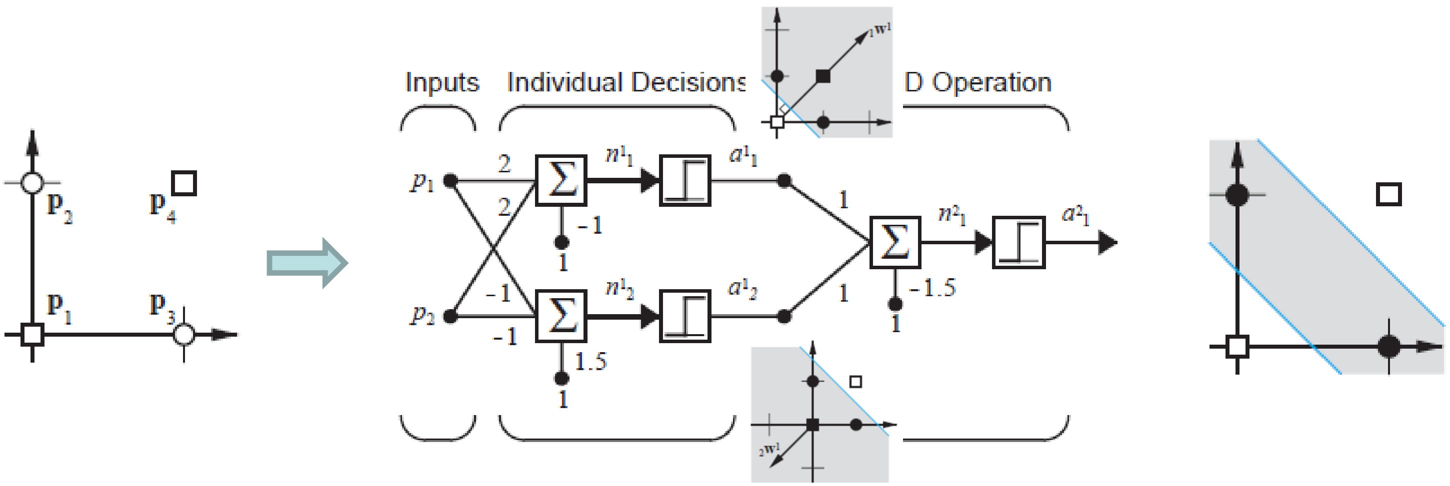
$$a(k) = purelin(\mathbf{Wp} + b)$$
$$= \sum_{i=1}^{R} w_{1,i}\, y(k - i + 1) + b$$

Incremental learning

MADALINE or FeedForward Network and Backpropagation Learning

MADALINE or FeedForward

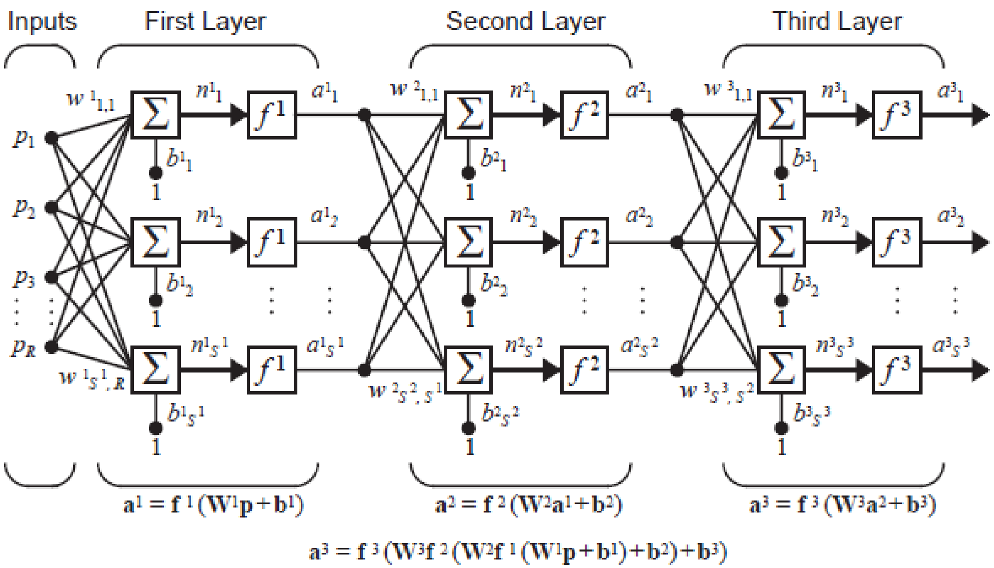
- **ADALINE and Perceptron:** single-layer NN
  - Separates the classification space in two classes through an hyperplane (a line in 2D)
- **XOR problem**
  - XOR function can’t be implemented by single-layer NN
  - Solution: to use multi-layers NNs → MADALINE
  - Problem: How to train a MADALINE NN? Perceptron and LMS Widrow-Hoff learning rule were designed to train single-layer NN



Neural Networks

MADALINE or FeedForward

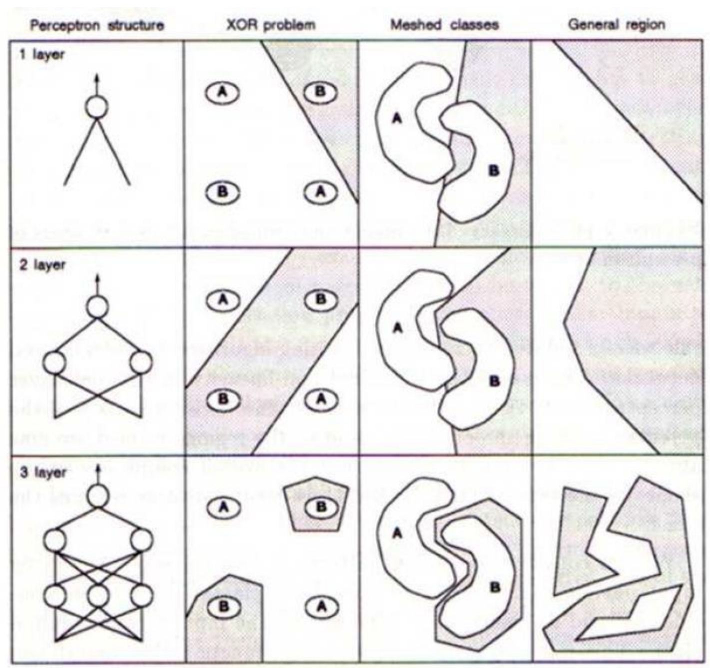
- **Architecture**
  - Neurons grouped by layers (input, hidden, output)
  - Total connectivity. Each neuron of the hidden layers are connected with all the neurons of the previous and posterior layers
  - The number of neurons in the input and output layers are predefined by the problem
  - The number of neurons of the hidden layers define the network complexity



Neural Networks

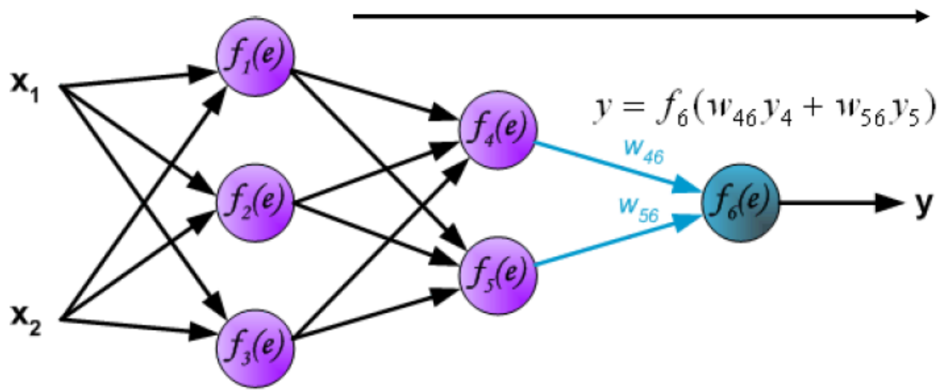
MADALINE or FeedForward

- The hidden-layers neurons increase the boundary decision complexity
- A three-layers NN is an universal classifier (input-hidden-output)
- No more layers are necessary to improve the classifier, increasing the number of neurons in the hidden-layer has the same effect
- No linear separation among classes require no linear activation functions (sigmoids)

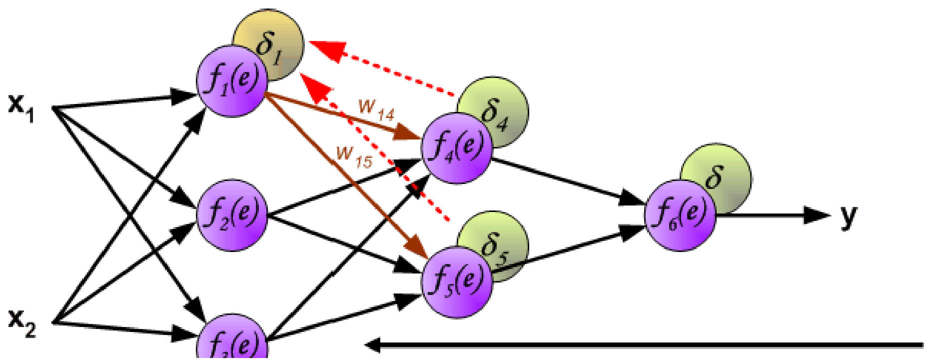


Backpropagation

- **Backpropagation: Learning algorithm for multilayer NNs**
  - Training:
    - It is a generalization of the LMS algorithm, in which the performance index is mean square error, for multilayer NNs
    - Iterative method
    - Supervised learning
    - High training time (slow convergence)
  - Process
    - 1) An initial weight is provided to the neurons
    - 2) From the input set we get an output set
      - NN Inputs: $\mathbf{x}_j$
      - NN Outputs: $\mathbf{y}_{\mathbf{Nj}}$

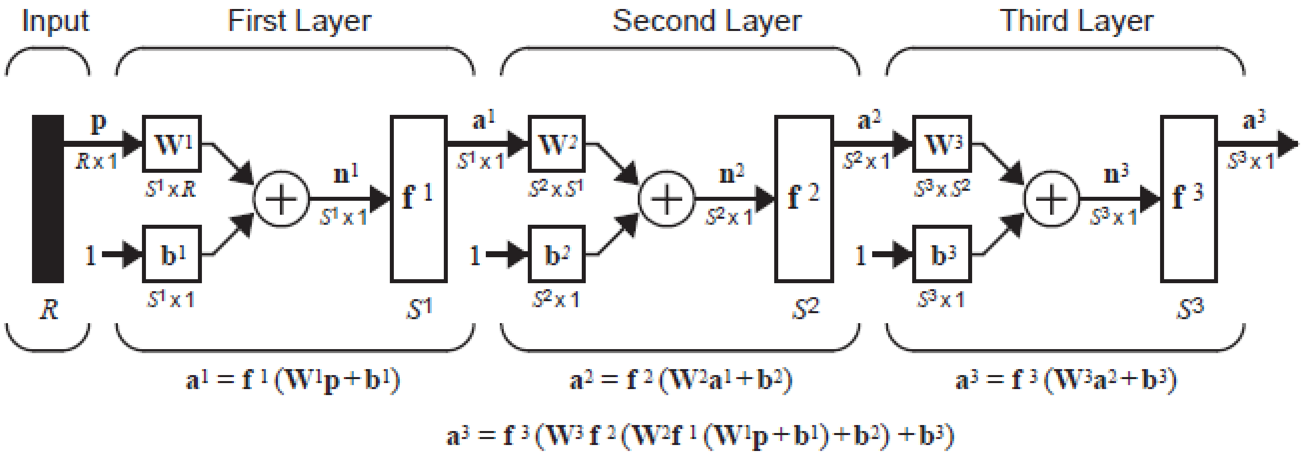


- **Backpropagation:**
  - Process
    - 3) Comparing the NN outputs with the targets some squared errors are obtained
      - Targets: $\mathbf{t}_j$
      - Errors: $(\mathbf{t}_j - \mathbf{y}_{\mathbf{N}j})^2$
    - 4) Errors are back-propagated from the output to the input and each neuron takes a part of the error proportional to its contributions to the global error



    - 5) The weight of each neuron is adapted as a function of its error

Incremental training algorithm



$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}\left(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}\right) \ \ for \ \ m = 0,1,\ldots,M-1$$

$$\mathbf{a}^0 = \mathbf{p} \qquad\qquad \mathbf{a} = \mathbf{a}^M$$

- Supervised learning. Training set: $\{\mathbf{p_1}, t_1\}, \{\mathbf{p}_2, t_2\}, \ldots, \{\mathbf{p}_Q, t_Q\}$
- We aproximate the LMS by the squared error at iteration k:

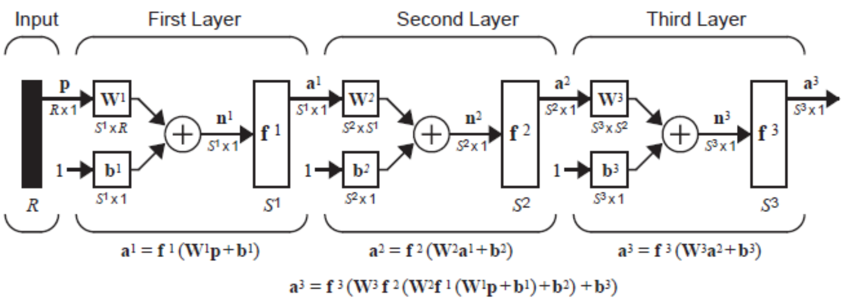
$$F(\mathbf{x}) = E\left[\mathbf{e}^T\mathbf{e}\right] = E\left[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})\right]$$

$$F(\mathbf{x}) \approx \hat{F}(\mathbf{x}) = \left(\mathbf{t}(k)-\mathbf{a}(k)\right)^T\left(\mathbf{t}(k)-\mathbf{a}(k)\right) = \mathbf{e}^T(k)\mathbf{e}(k)$$

- **Incremental training algorithm**
  - The steepest descent algorithm for the approximate mean square error is:

$$w^m_{i,j}(k+1) = w^m_{i,j}(k) - \alpha \frac{\partial \hat{F}}{w^m_{i,j}} \qquad b^m_i(k+1) = b^m_i(k) - \alpha \frac{\partial \hat{F}}{b^m_i}$$

  - Applying the chain rule $\frac{df(n(w))}{dw} = \frac{df(n)}{dn}\frac{dn(w)}{dw}$



$$\hat{F} = \mathbf{e}^T(k)\mathbf{e}(k) = \sum_{i=1}^{S^M}\left(t_i - a_i^M\right)^2$$

$$a_i^m = f\left(n_i^m\right)$$

$$n_i^m = \sum_{j=1}^{S^{m-1}} w^m_{i,j} a_j^{m-1} + b_i^m$$

$$\frac{\partial \hat{F}}{\partial w^m_{i,j}} = \frac{\partial \hat{F}}{\partial n_i^m}\frac{\partial n_i^m}{\partial w^m_{i,j}}$$

$$\frac{\partial \hat{F}}{\partial b^m_i} = \frac{\partial \hat{F}}{\partial n_i^m}\frac{\partial n_i^m}{\partial b^m_i}$$

⇨

sensitivity

$$s_i^m = \frac{\partial \hat{F}}{\partial n_i^m}$$

$$\frac{\partial n_i^m}{\partial w^m_{i,j}} = a_j^{m-1}\ , \quad \frac{\partial n_i^m}{\partial b^m_i} = 1$$

⇨

$$\boxed{\begin{aligned} w^m_{i,j}(k+1) &= w^m_{i,j}(k) - \alpha s_i^m a_j^{m-1} \\ b^m_i(k+1) &= b^m_i(k) - \alpha s_i^m \end{aligned}}$$

Backpropagation

- **Incremental training algorithm**
  - In matrix form:

$$\boxed{\begin{aligned}\mathbf{W}^m(k+1) &= \mathbf{W}^m(k) - \alpha \mathbf{s}^m \left(\mathbf{a}^{m-1}\right)^T \\ \mathbf{b}^m(k+1) &= \mathbf{b}^m(k) - \alpha \mathbf{s}^m\end{aligned}}$$

$$\mathbf{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \dfrac{\partial \hat{F}}{\partial n_1^m} & \dfrac{\partial \hat{F}}{\partial n_2^m} & \cdots & \dfrac{\partial \hat{F}}{\partial n_{S^m}^m} \end{bmatrix}^T$$

  - Backpropagating the sensitivities
    - Output layer:

$$m = M \rightarrow s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial (\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})}{\partial n_i^M} = \frac{\partial \sum_{l=1}^{S^M}(t_l - a_l)^2}{\partial n_i^M} = -2(t_i - a_i)\frac{\partial a_i}{\partial n_i^M}$$

$$\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M\left(n_i^M\right)}{\partial n_i^M} = \dot{f}^M\left(n_i^M\right)$$

$$\boxed{s_i^M = -2(t_i - a_i)\dot{f}^M\left(n_i^M\right)} \qquad \boxed{\mathbf{s}^M = -2\dot{\mathbf{F}}^M\left(\mathbf{n}^M\right)(\mathbf{t}-\mathbf{a})}$$

$$\dot{\mathbf{F}}^M\left(\mathbf{n}^M\right) = \begin{bmatrix} \dot{f}^m\left(n_1^m\right) & 0 & \cdots & 0 \\ 0 & \dot{f}^m\left(n_2^m\right) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \dot{f}^m\left(n_{S^m}^m\right) \end{bmatrix}$$

- **Incremental training algorithm**
  - Backpropagating the sensitivities
    - The sensitivities are propagated backward through the network from the last layer to the first layer

$$\mathbf{s}^M \rightarrow \mathbf{s}^{M-1} \rightarrow \ldots \mathbf{s}^2 \rightarrow \mathbf{s}^1$$

$$\mathbf{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}}$$

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix}$$

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial\left(\sum_{l=1}^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial a_j^m}{\partial n_j^m} = w_{i,j}^{m+1} \frac{\partial f^m\left(n_j^m\right)}{\partial n_j^m} = w_{i,j}^{m+1} \dot{f}^m\left(n_j^m\right)$$

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m(\mathbf{n}^m) \quad \Longrightarrow \quad \boxed{\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)\left(\mathbf{W}^{m+1}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)\left(\mathbf{W}^{m+1}\right)^T \mathbf{s}^{m+1}}$$

- **Incremental training algorithm summary**

1. To propagate the input forward through the network:

$$\mathbf{a}^0 = \mathbf{p}$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}\left(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}\right) \; for \;\; m = 0,1,\ldots,M-1$$

$$\mathbf{a} = \mathbf{a}^M$$

2. To propagate the sensitivities backward through the network:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M\left(\mathbf{n}^M\right)\left(\mathbf{t}-\mathbf{a}\right)$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)\left(\mathbf{W}^{m+1}\right)^T\mathbf{s}^{m+1} \qquad for \;\; m = M-1,\ldots,2,1$$

3. The weights and biases are updated using the approximate steepest descent rule:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m\left(\mathbf{a}^{m-1}\right)^T$$

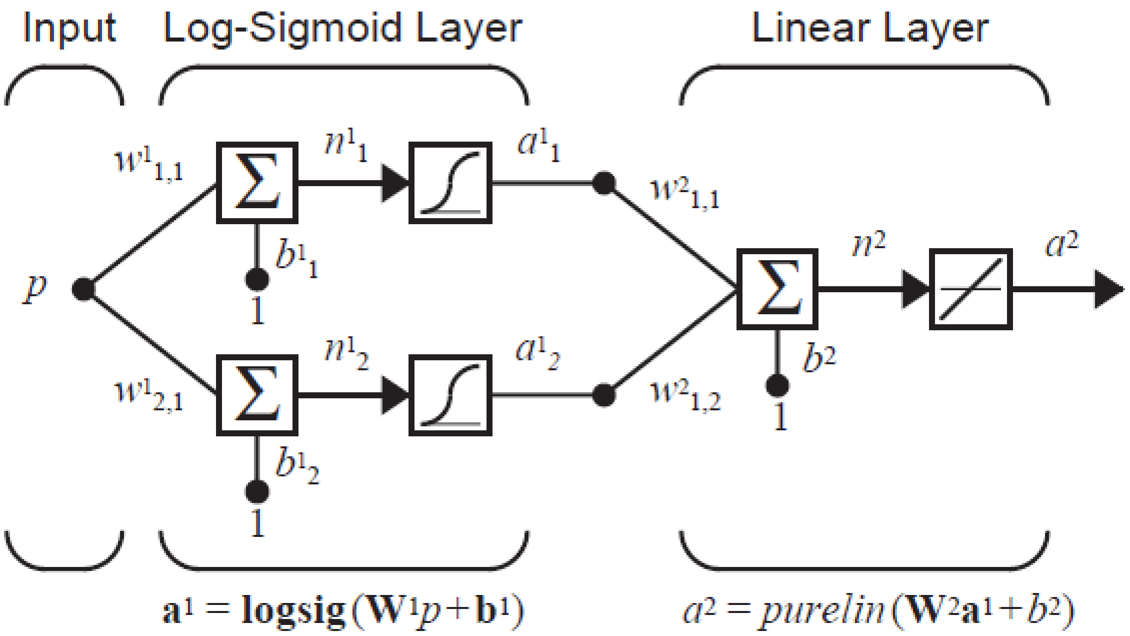
$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m$$

Backpropagation. Example

- Approximate the following function:

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2 .$$

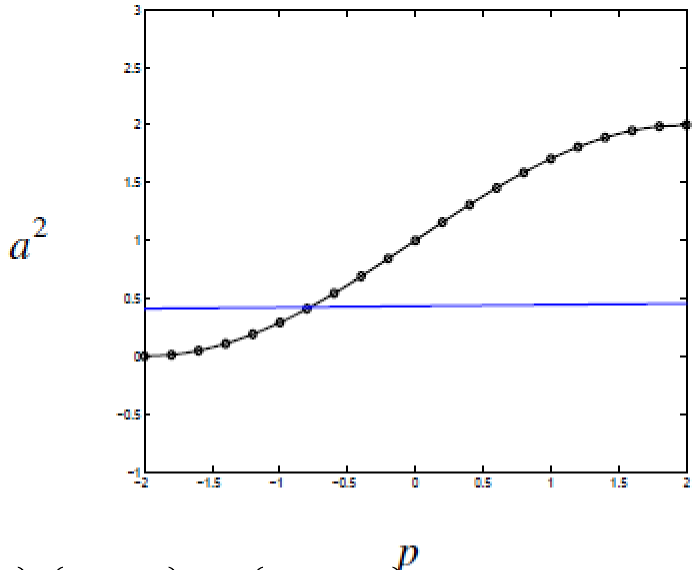
- Using the following NN topology (1-2-1)



Backpropagation. Example

- Random Initial conditions:

$$\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix},\ \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix},\ \mathbf{W}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix},\ \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}$$



- Training set: $\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_{21}, t_{21}\}$
- $a^0 = p_{16} = 1$

Backpropagation. Example

- Output of the first layer:

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{a}^0 + \mathbf{b}^1) = \mathbf{logsig}\left(\begin{bmatrix}-0.27\\-0.41\end{bmatrix}\begin{bmatrix}1\end{bmatrix} + \begin{bmatrix}-0.48\\-0.13\end{bmatrix}\right) = \mathbf{logsig}\left(\begin{bmatrix}-0.75\\-0.54\end{bmatrix}\right) = \begin{bmatrix}\dfrac{1}{1+e^{0.75}}\\ \dfrac{1}{1+e^{0.54}}\end{bmatrix} = \begin{bmatrix}0.321\\0.368\end{bmatrix}$$

- Output of the second layer:

$$a^2 = f^2(\mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2) = purelin\left(\begin{bmatrix}0.09 & -0.17\end{bmatrix}\begin{bmatrix}0.321\\0.368\end{bmatrix} + \begin{bmatrix}0.48\end{bmatrix}\right) = \begin{bmatrix}0.446\end{bmatrix}$$

- The error would be:

$$e = t - a = \left\{1 + \sin\left(\frac{\pi}{4}p\right)\right\} - a^2 = \left\{1 + \sin\left(\frac{\pi}{4}1\right)\right\} - 0.446 = 1.261$$

- To backpropagate the sensitivities:

$$\dot{f}^1(n) = \frac{d}{dn}\left(\frac{1}{1+e^{-n}}\right) = \frac{e^{-n}}{(1+e^{-n})^2} = \left(1 - \frac{1}{1+e^{-n}}\right)\left(\frac{1}{1+e^{-n}}\right) = (1-a^1)(a^1)$$

$$\dot{f}^2(n) = \frac{d}{dn}(n) = 1$$

Backpropagation. Example

- To backpropagate the sensitivities:

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2\left[\dot{f}^2(n^2)\right](1.261) = -2\left[1\right](1.261) = -2.522$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = \begin{bmatrix} (1-a_1^1)(a_1^1) & 0 \\ 0 & (1-a_2^1)(a_2^1) \end{bmatrix}\begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix}\begin{bmatrix} -2.522 \end{bmatrix}$$

$$= \begin{bmatrix} (1-0.321)(0.321) & 0 \\ 0 & (1-0.368)(0.368) \end{bmatrix}\begin{bmatrix} 0.09 \\ -0.17 \end{bmatrix}\begin{bmatrix} -2.522 \end{bmatrix}$$

$$= \begin{bmatrix} 0.218 & 0 \\ 0 & 0.233 \end{bmatrix}\begin{bmatrix} -0.227 \\ 0.429 \end{bmatrix} = \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix}.$$

Backpropagation. Example

➢ To update the weights:

$$\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha\mathbf{s}^2(\mathbf{a}^1)^T = \begin{bmatrix}0.09 & -0.17\end{bmatrix} - 0.1\begin{bmatrix}-2.522\end{bmatrix}\begin{bmatrix}0.321 & 0.368\end{bmatrix}$$

$$= \begin{bmatrix}0.171 & -0.0772\end{bmatrix},$$

$$\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha\mathbf{s}^2 = \begin{bmatrix}0.48\end{bmatrix} - 0.1\begin{bmatrix}-2.522\end{bmatrix} = \begin{bmatrix}0.732\end{bmatrix},$$

$$\mathbf{W}^1(1) = \mathbf{W}^1(0) - \alpha\mathbf{s}^1(\mathbf{a}^0)^T = \begin{bmatrix}-0.27\\-0.41\end{bmatrix} - 0.1\begin{bmatrix}-0.0495\\0.0997\end{bmatrix}\begin{bmatrix}1\end{bmatrix} = \begin{bmatrix}-0.265\\-0.420\end{bmatrix},$$

$$\mathbf{b}^1(1) = \mathbf{b}^1(0) - \alpha\mathbf{s}^1 = \begin{bmatrix}-0.48\\-0.13\end{bmatrix} - 0.1\begin{bmatrix}-0.0495\\0.0997\end{bmatrix} = \begin{bmatrix}-0.475\\-0.140\end{bmatrix}.$$

➢ Batch training algorithm

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})] = \frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q)$$

$$\nabla F(\mathbf{x}) = \nabla\left\{\frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q)\right\} = \frac{1}{Q}\sum_{q=1}^{Q}\nabla\{(\mathbf{t}_q-\mathbf{a}_q)^T(\mathbf{t}_q-\mathbf{a}_q)\}$$

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m(\mathbf{a}_q^{m-1})^T$$

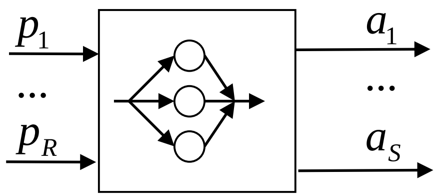
$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m .$$

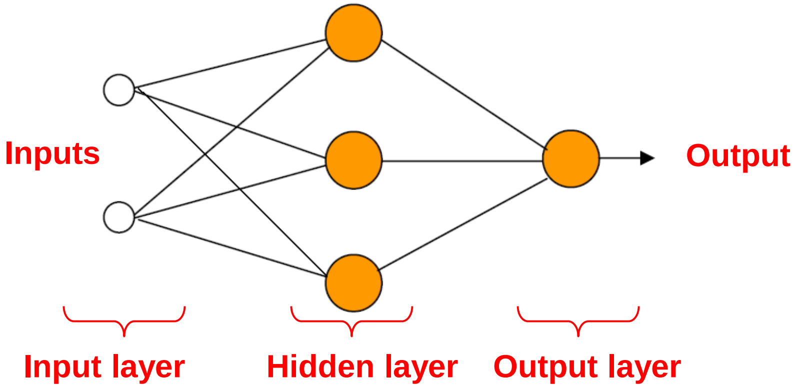
Neural Networks

MATLAB example

```
% Create a two-layer feedforward network
net = feedforwardnet
% Change the transfer function to logsig
net.layers{1}.transferFcn = 'logsig';
% Change the number of neurons in the firts layer to 3
net.layers{1}.dimensions=2;
%Configure Neural Network Inputs and Outputs
p = -2:.2:2;
t = 1+sin(pi/4*p);
net = configure(net,p,t);
view(net)
% Set initial values
net.IW{1} = [-0.27;-0.41]; %IW{1}(2x1)
net.b{1} = [-0.48;-0.13]; %b{1}(2x1)
net.LW{2} = [0.09 -0.17]; %IW{2}(1x2)
net.b{2} = [0.48]; %b{2}(1x1)
% Plot initial results
a = sim(net,p); % Simulate the net at the beginning
figure(1);
plot(t,'r-'); hold on; , plot(a,'b-'); hold off;
xlabel('p'); ylabel('target (red) - output (blue)'); title('Results before the training');
% Training the net
net = train(net,p,t); %train is used for batch training
% Simulate the net
a = sim(net,p);
% Mean Square Error calculation
e = mse(a,t) % e = 9.4024e-05
% Plot final results
figure(2)
plot(t,'r-'); hold on; , plot(a,'b-'); hold off;
xlabel('p'); ylabel('target (red) - output (blue)'); title('Results after the training');
```

- **Practical questions in the design of the MADALINE NN**
  - In control systems 3 layer NN are used (input, hidden, output)
  - Neurons of the input layer are fixed by the input vector
  - Neurons of the output layer are fixed by the output vector

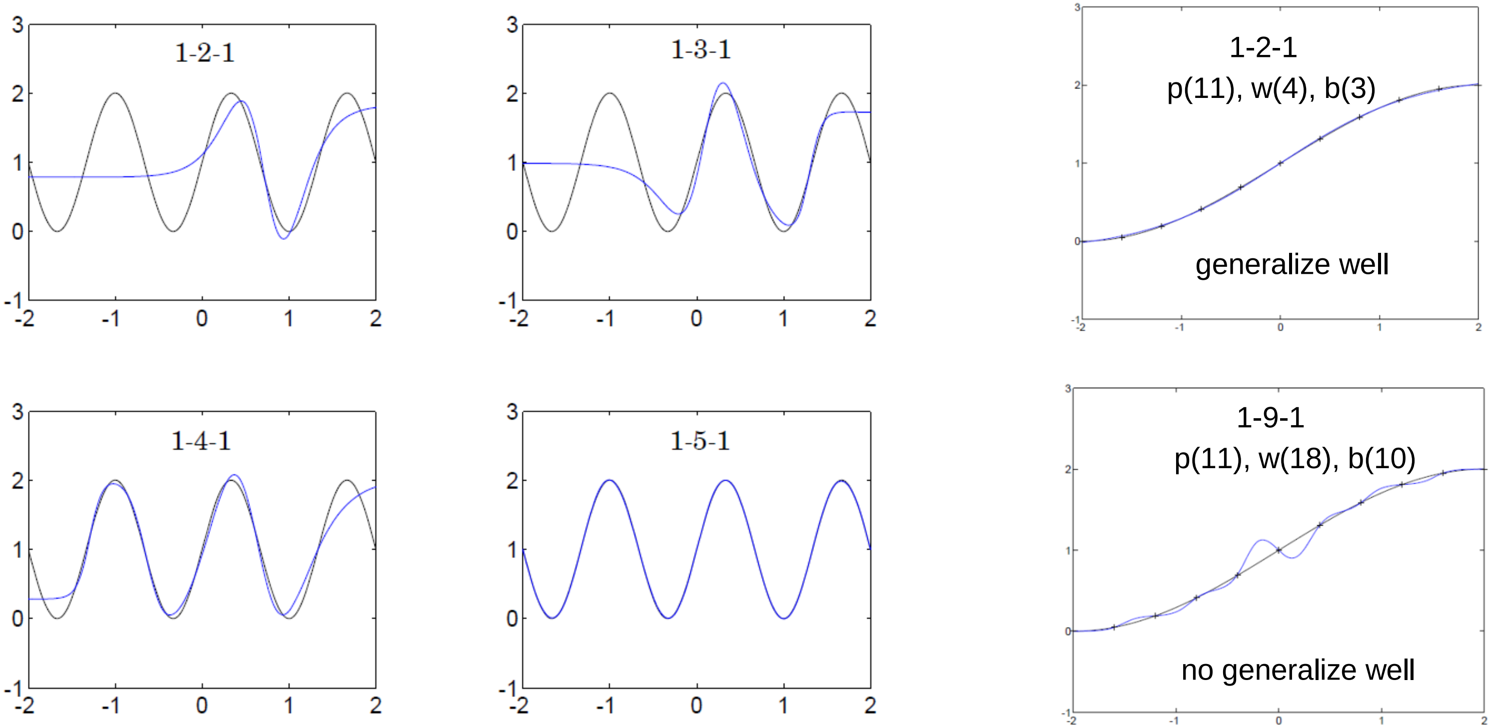




  - Neurons of the hidden layer must be lower of the 15% of the training vectors. It is calculated by rehearsal and error
  - It is convenience to use weight initialization methods to provide values closed to the final ones (stochastic methods)
    - The learning converges faster and avoids local minima

MADALINE

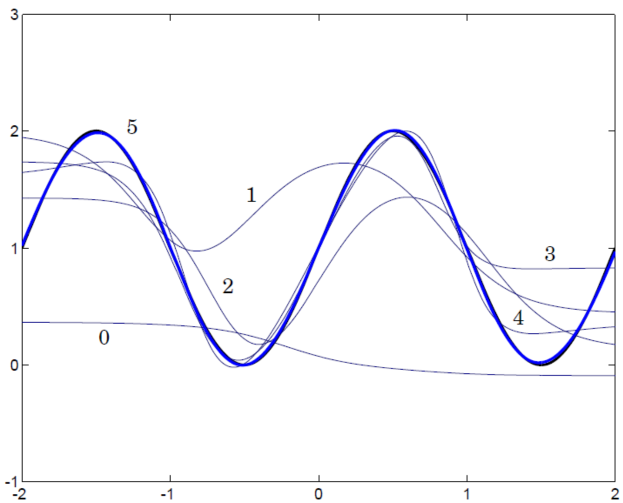
- **Practical questions in the design of the MADALINE NN**
  - **Network architecture:** new neurons are added to the hidden layer until the NN output fits the target (LMS error is low)



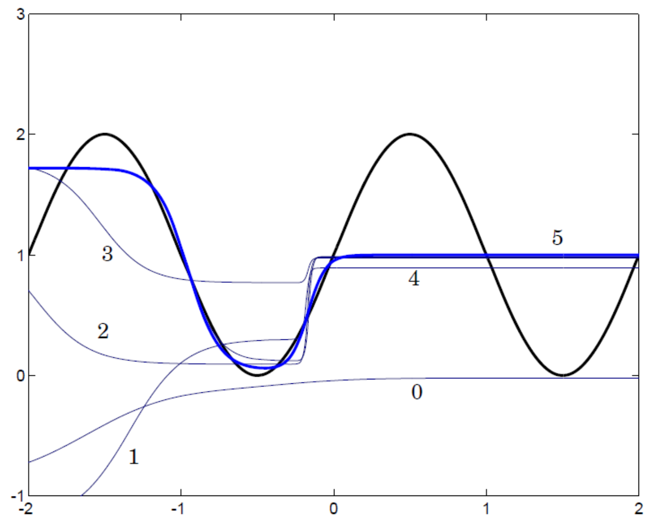
  - If too many neurons are added, the NN memorizes instead to **generalize**

MADALINE

- **Practical questions in the design of the MADALINE NN**
  - **Convergence**: a NN can no give an accurate approximation to the target even though the parameters minimize mean square error



Convergence to a global minimum



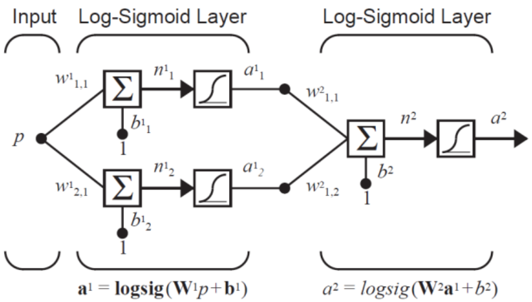
Convergence to a local minimum

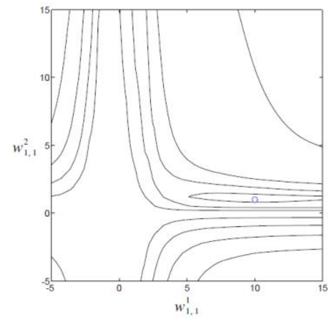
  - The higher the NN complexity is the higher the local minimum is too

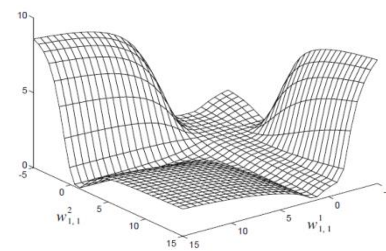
Variations on Backpropagation

Variations on Backpropagation

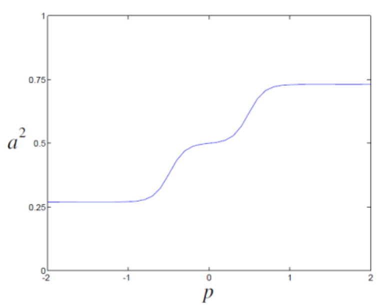
- **Drawbacks of Backpropagation**
  - BP is a LMS algorithm for multi-layer NN
  - Performance surface of a multilayer network
    - Many local minimum points
    - It is not a quadratic function
    - The curvature can vary widely in different regions

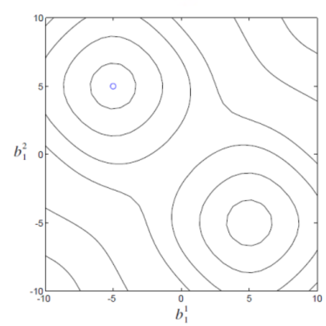


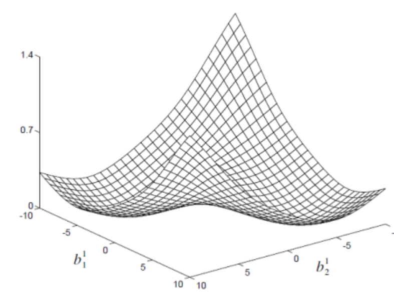




$$w^1_{1,1} = 10,\ w^1_{2,1} = 10,\ b^1_1 = -5,\ b^1_2 = 5,$$
$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = -1.$$

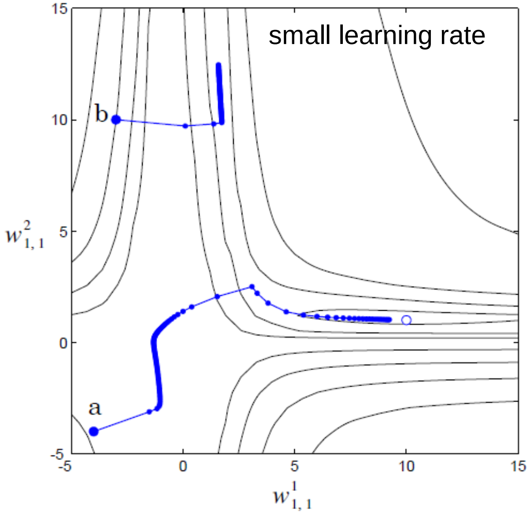


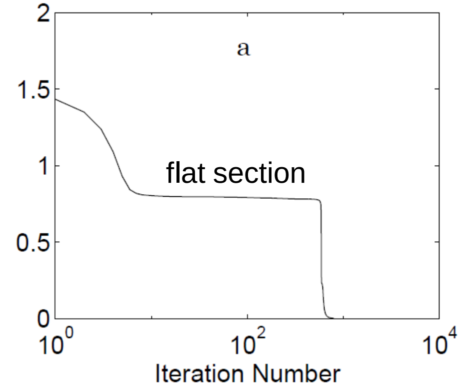


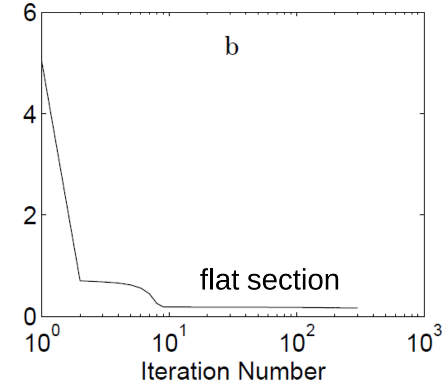


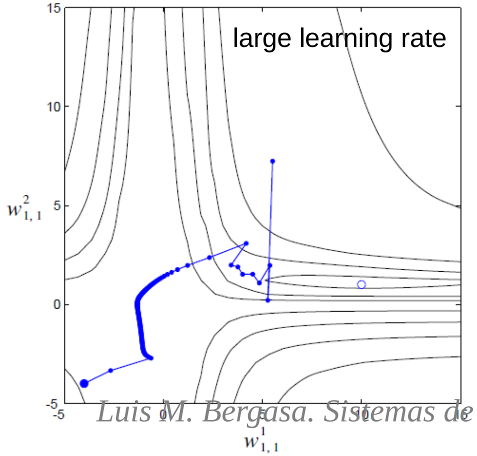
Variations on Backpropagation

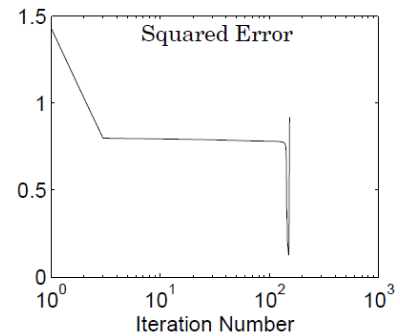
- **Drawbacks of Backpropagation**
  - Convergence is slow for a fix learning rate









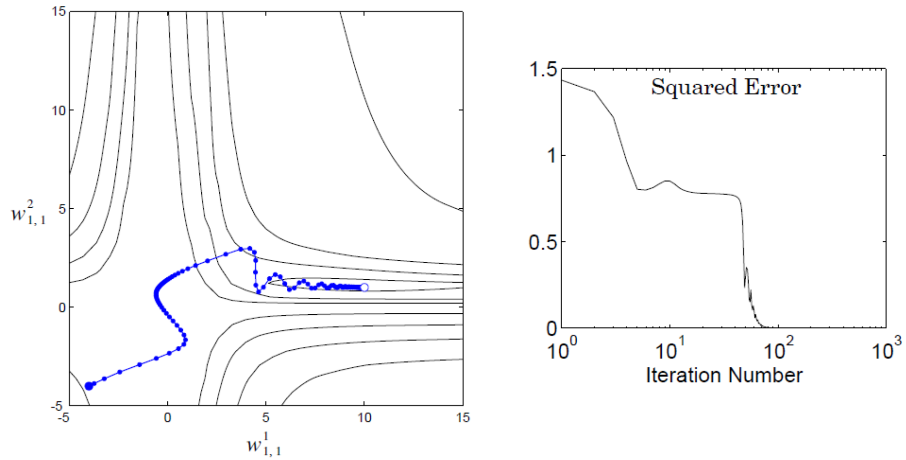


Variations on Backpropagation

- **Heuristic Modifications of Backpropagation**
  - **Momentum**
    - Adds an inertial momentum to the learning rate. It tends to make the trajectory continue in the same direction
    - A larger learning rate can be used while maintaining the stability of the algorithm
    - It tends to accelerate convergence when the trajectory is moving in a consistent direction

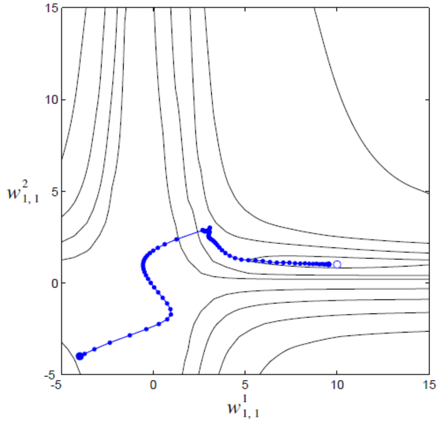
$$\Delta \mathbf{W}^m(k) = \gamma \Delta \mathbf{W}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T,$$

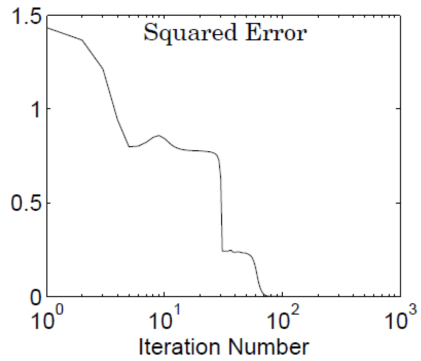
$$\Delta \mathbf{b}^m(k) = \gamma \Delta \mathbf{b}^m(k-1) - (1-\gamma)\alpha \mathbf{s}^m.$$

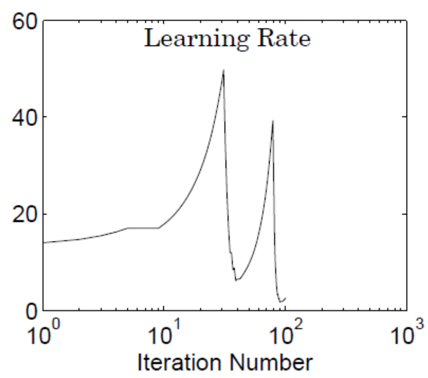


Variations on Backpropagation

- **Heuristic Modifications of Backpropagation**
  - **Variable Learning Rate**
    - Convergence can be speed up by adjusting the learning rate during the course of training







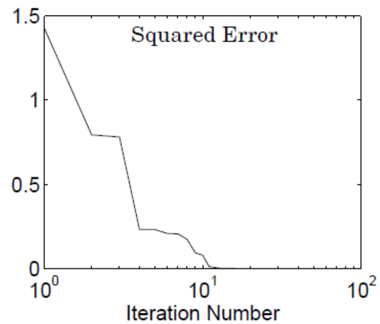
1. If the squared error increases by more than some set percentage ξ (typically 1% to 5%) after a weight update, then the weight update is discarded, the learning rate is multiplied by some factor $0 < \rho < 1$, and the momentum coefficient γ (if it is used) is set to zero.

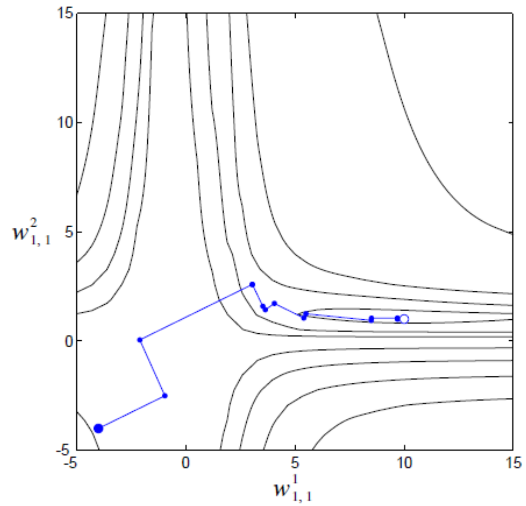
2. If the squared error decreases after a weight update, then the weight update is accepted and the learning rate is multiplied by some factor $\eta > 1$. If γ has been previously set to zero, it is reset to its original value.

3. If the squared error increases by less than ξ, then the weight update is accepted but the learning rate is unchanged. If γ has been previously set to zero, it is reset to its original value.

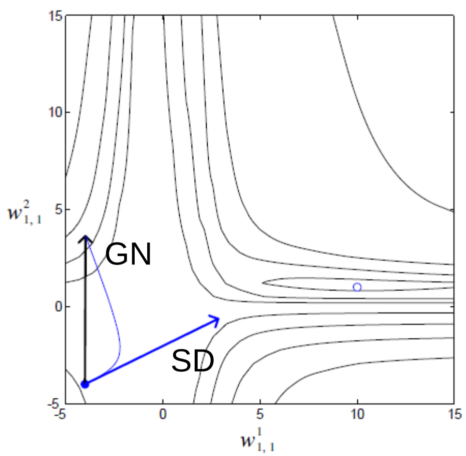
Variations on Backpropagation

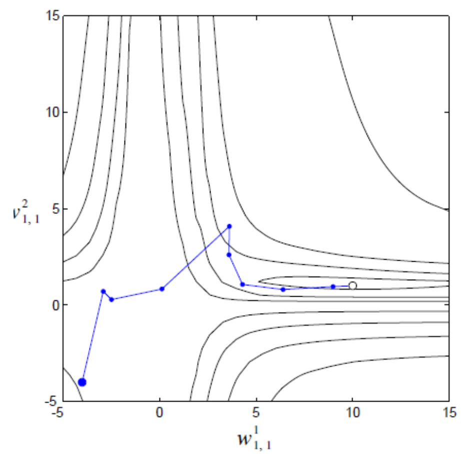
- **Numerical Optimization Techniques**
  - **Conjugate Gradient**



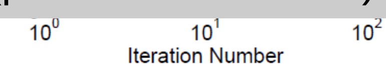


  - **Levenberg-Marquardt Algorithm**





Pros: fastest neural network training algorithm for moderate numbers of network parameters
Cons: storage requirement (parameters < 3000)



Variations on Backpropagation

- **MATLAB options**

Function	Algorithm
trainlm	Levenberg-Marquardt
trainbr	Bayesian Regularization
trainbfg	BFGS Quasi-Newton
trainrp	Resilient Backpropagation
trainscg	Scaled Conjugate Gradient
traincgb	Conjugate Gradient with Powell/Beale Restarts
traincgf	Fletcher-Powell Conjugate Gradient
traincgp	Polak-Ribiére Conjugate Gradient
trainoss	One Step Secant
traingdx	Variable Learning Rate Gradient Descent
traingdm	Gradient Descent with Momentum
traingd	Gradient Descent

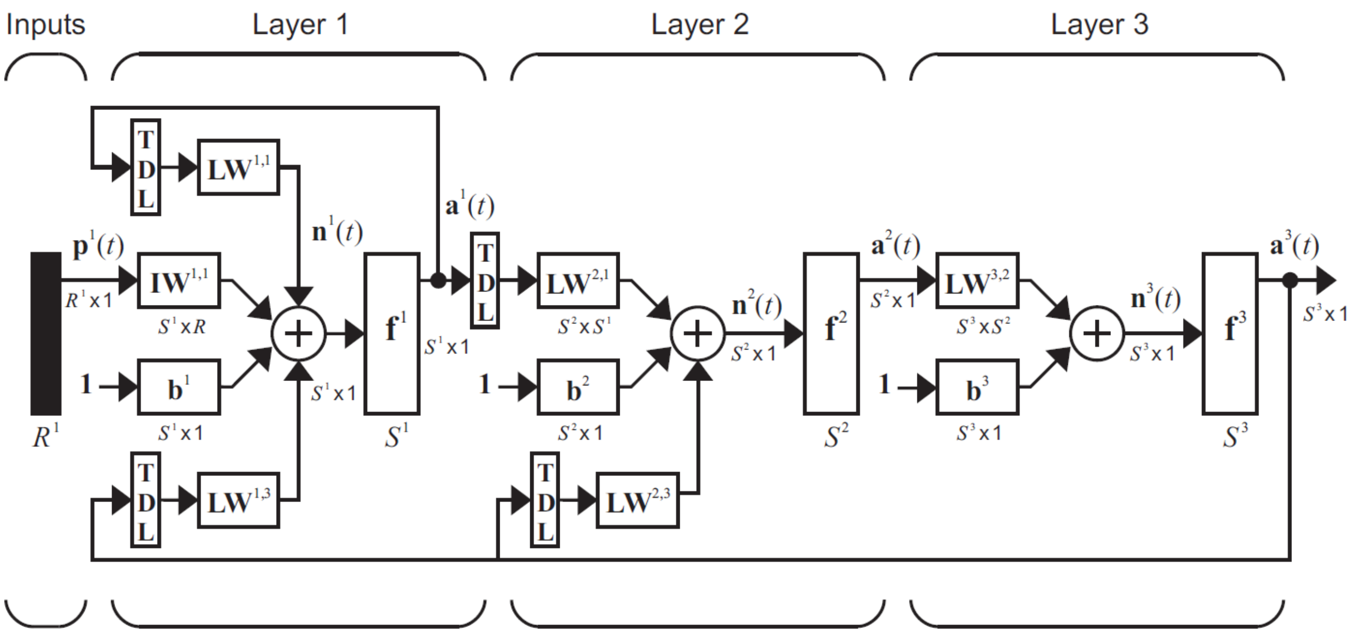
Dynamic Networks

Dynamic Networks

- Networks that contain **delays** and that operate on a sequence of inputs (they have **memory**)
- DNs **learn dynamic systems**
- DNs can be trained using the **standard optimization** methods **modifying** the **Jacobians** calculation
- Training approaches:
  - **Backpropagation-through time (BPTT)**
    - It works backward in time from the last time step
    - Efficient
    - Off-line
    - Used in Recurrent-dynamic network (NARX)
  - **Real-time recurrent learning (RTRL)**
    - It works forward through time
    - More calculations than BPTT
    - On-line
    - Used in Feedforward-dynamic networks

Dynamic Networks

- **Layered Digital Dynamic Networks**

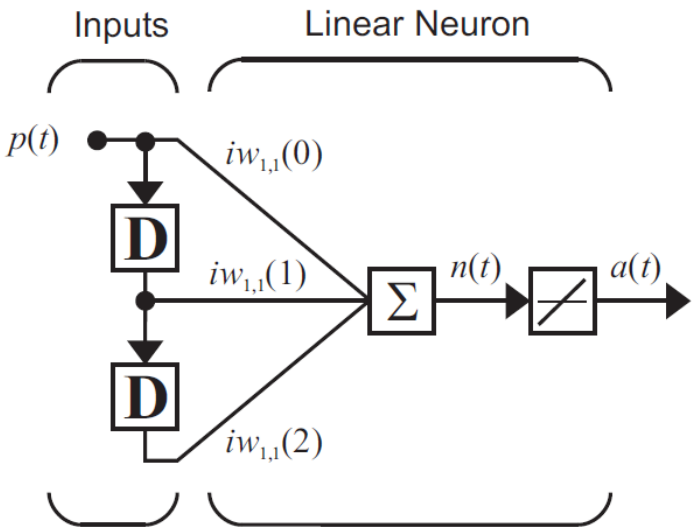


$$\mathbf{n}^m(t) = \sum_{l \in L_m^f} \sum_{d \in DL_{m,l}} \mathbf{LW}^{m,l}(d)\mathbf{a}^l(t-d) + \sum_{l \in I_m} \sum_{d \in DI_{m,l}} \mathbf{IW}^{m,l}(d)\mathbf{p}^l(t-d) + \mathbf{b}^m$$

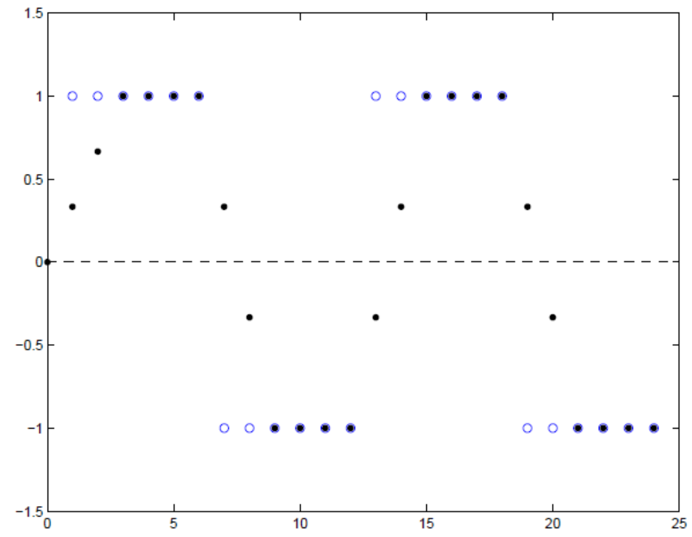
$$\mathbf{a}^m(t) = \mathbf{f}^m(\mathbf{n}^m(t))$$

Dynamic Networks

- **Example. Finite Impulse Response (FIR) filter**



$a(t) = iw_{1,1}(0)p(t)+iw_{1,1}(1)p(t\text{-}1)+iw_{1,1}(2)p(t\text{-}2)$



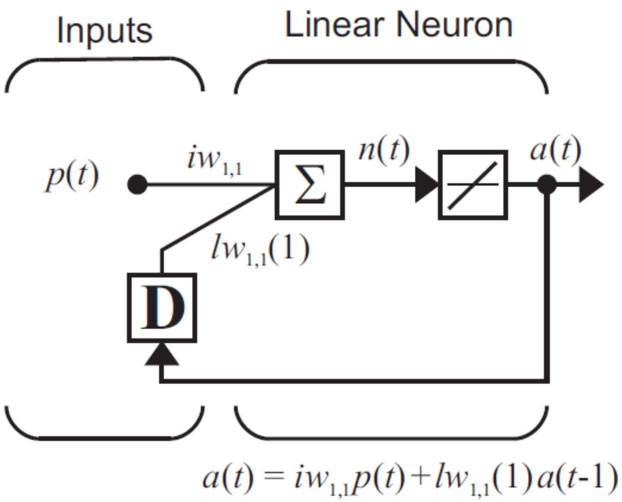
$$iw_{1,1}(0) = \frac{1}{3},\ iw_{1,1}(1) = \frac{1}{3},\ iw_{1,1}(2) = \frac{1}{3}$$

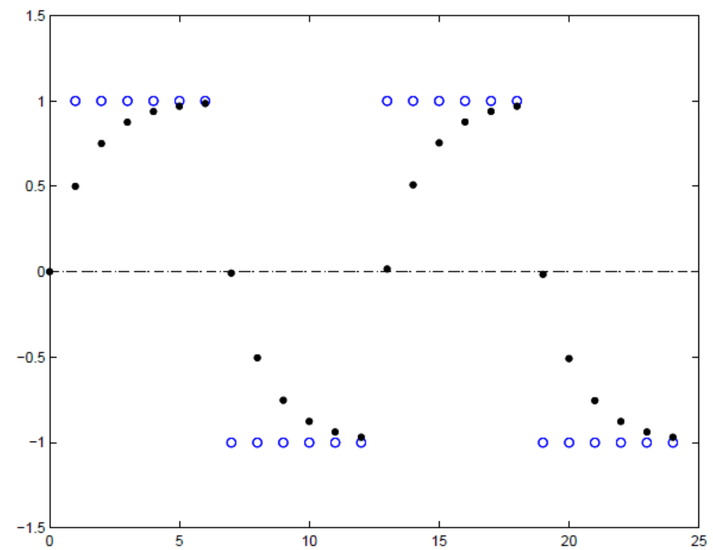
$$\mathbf{a}(t) = \mathbf{n}(t) = \sum_{d=0}^{2} \mathbf{IW}(d)\mathbf{p}(t-d)$$

$$= n_1(t) = iw_{1,1}(0)p(t) + iw_{1,1}(1)p(t-1) + iw_{1,1}(2)p(t-2)$$

Dynamic Networks

➢ Example. Infinite Impulse Response (IIR) filter





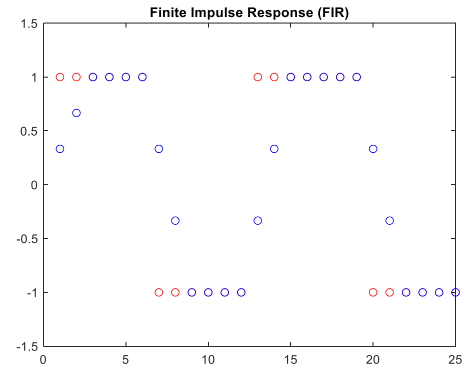
$$lw_{1,1}(1) = \frac{1}{2} \text{ and } iw_{1,1} = \frac{1}{2}$$

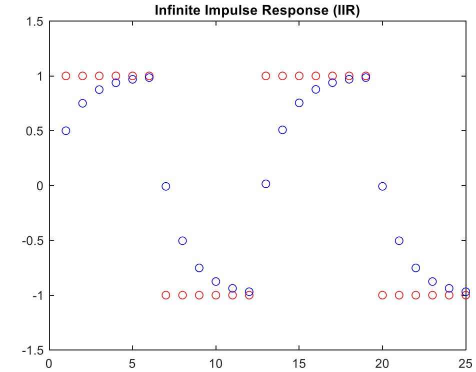
$$\begin{aligned}\mathbf{a}^1(t) = \mathbf{n}^1(t) &= \mathbf{LW}^{1,1}(1)\mathbf{a}^1(t-1) + \mathbf{IW}^{1,1}(0)\mathbf{p}^1(t)\\ &= lw_{1,1}(1)a(t-1) + iw_{1,1}p(t)\end{aligned}$$

Neural Networks

Dynamic Networks

```
% Input signal
p = {1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1};
% Feedforward-dynamic network (FIR)
net = linearlayer([0 1 2]);
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.IW{1,1} = [1/3 1/3 1/3];
view(net)
% Net simulation
a = net(p);
% Plot results
plot(cell2mat(p),'ro'); hold on;
plot(cell2mat(a),'bo'); hold off;
axis([0 25 -1.5 1.5]); title('Finite Impulse Response (FIR)');
% Recurrent-dynamic networks (IIR)
net = narxnet(0,1,[],'closed');
net.inputs{1}.size = 1;
net.layers{1}.dimensions = 1;
net.biasConnect = 0;
net.LW{1} = 0.5;
net.IW{1} = 0.5;
view(net)
% Net simulation
a = net(p);
% Plot results
plot(cell2mat(p),'ro'); hold on;
plot(cell2mat(a),'bo'); hold off;
axis([0 25 -1.5 1.5]); title('Infinite Impulse Response (IIR)');
```

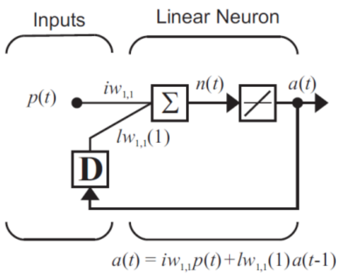




Neural Networks

Dynamic Networks

- **Principles of Dynamic Learning**



$$F(\mathbf{x}) = \sum_{t=1}^{Q} e^2(t) = \sum_{t=1}^{Q} (t(t) - a(t))^2$$

$$\frac{\partial F(\mathbf{x})}{\partial lw_{1,1}(1)} = \sum_{t=1}^{Q} \frac{\partial e^2(t)}{\partial lw_{1,1}(1)} = -2\sum_{t=1}^{Q} e(t)\frac{\partial a(t)}{\partial lw_{1,1}(1)}$$

$$\frac{\partial F(\mathbf{x})}{\partial iw_{1,1}} = \sum_{t=1}^{Q} \frac{\partial e^2(t)}{\partial iw_{1,1}} = -2\sum_{t=1}^{Q} e(t)\frac{\partial a(t)}{\partial iw_{1,1}}$$

$$a(t) = lw_{1,1}(1)a(t-1) + iw_{1,1}p(t)$$

$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} = a(t-1) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial lw_{1,1}(1)},$$

$$\frac{\partial a(t)}{\partial iw_{1,1}} = p(t) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial iw_{1,1}}.$$

➢ Principles of Dynamic Learning

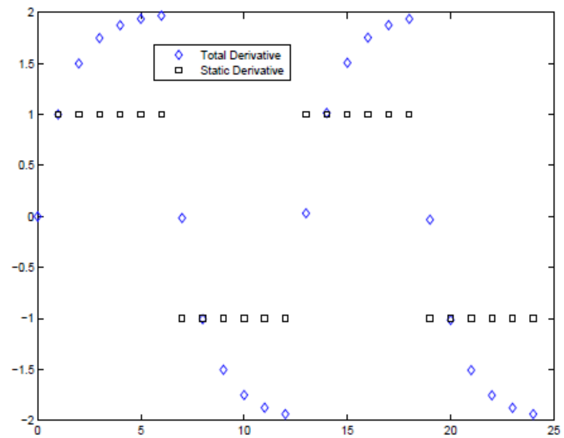
$$\frac{\partial a(t)}{\partial iw_{1,1}} = p(t) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial iw_{1,1}}$$

Total derivative *Static derivative* *Dynamic derivative*

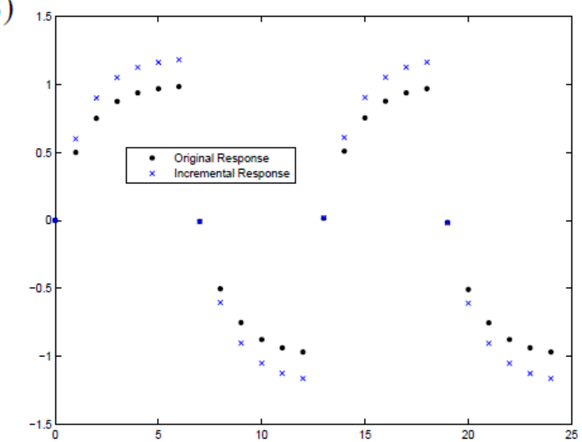
$$a(t) = lw_{1,1}(1)a(t-1) + iw_{1,1}p(t)$$

$$iw_{1,1} = 0.5 \rightarrow iw_{1,1} = 0.6$$

a)



b)



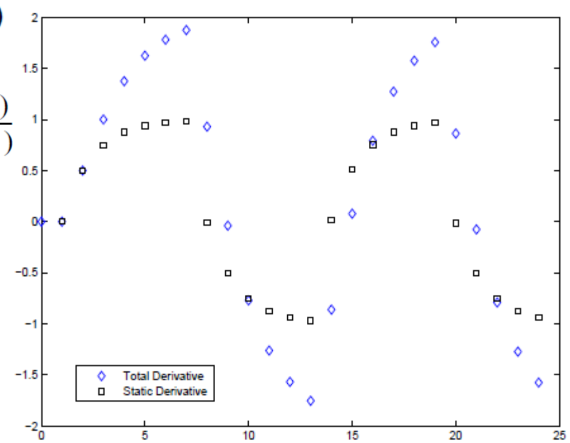
$$\frac{\partial a(t)}{\partial lw_{1,1}(1)} = a(t-1) + lw_{1,1}(1)\frac{\partial a(t-1)}{\partial lw_{1,1}(1)}$$

Total derivative *Static derivative* *Dynamic derivative*

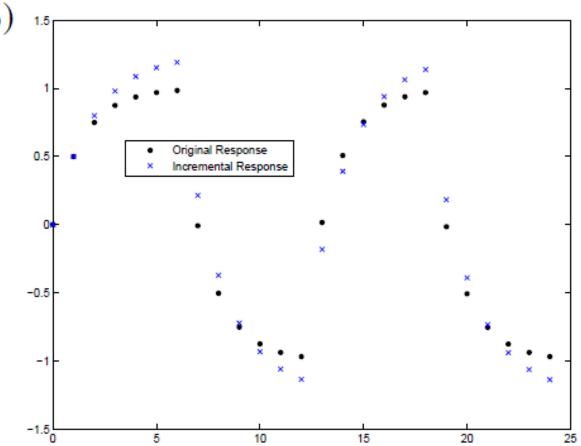
$$a(t) = lw_{1,1}(1)a(t-1) + iw_{1,1}p(t)$$

$$lw_{1,1} = 0.5 \rightarrow lw_{1,1} = 0.6$$

a)

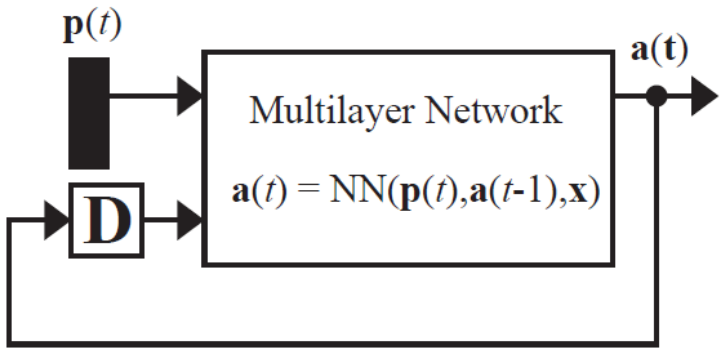


b)



Dynamic Networks

➢ Principles of Dynamic Learning



$$F(\mathbf{x}) = \sum_{t=1}^{Q} e^2(t) = \sum_{t=1}^{Q} (t(t) - a(t))^2$$

RTRL ⟶ $$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)} \qquad \frac{\partial \mathbf{a}(t)}{\partial \mathbf{x}^T} = \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T} + \frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{a}^T(t-1)} \times \frac{\partial \mathbf{a}(t-1)}{\partial \mathbf{x}^T}$$

BPTT ⟶ $$\frac{\partial F}{\partial \mathbf{x}} = \sum_{t=1}^{Q} \left[\frac{\partial^e \mathbf{a}(t)}{\partial \mathbf{x}^T}\right]^T \times \frac{\partial F}{\partial \mathbf{a}(t)} \qquad \frac{\partial F}{\partial \mathbf{a}(t)} = \frac{\partial^e F}{\partial \mathbf{a}(t)} + \frac{\partial^e \mathbf{a}(t+1)}{\partial \mathbf{a}^T(t)} \times \frac{\partial F}{\partial \mathbf{a}(t+1)}$$

- **Real Time Recurrent Learning. Example FIR**

$$a(t) = n(t) = iw_{1,1}(0)p(t) + iw_{1,1}(1)p(t-1) + iw_{1,1}(2)p(t-2)$$

$$F = \sum_{t=1}^{Q} (t(t) - a(t))^2 = \sum_{t=1}^{3} e^2(t) = e^2(1) + e^2(2) + e^2(3)$$

$$\{p(1), t(1)\}, \{p(2), t(2)\}, \{p(3), t(3)\}$$

$$a(1) = n(1) = iw_{1,1}(0)p(1) + iw_{1,1}(1)p(0) + iw_{1,1}(2)p(-1)$$

$$iw_{1,1}(d)_{k+1} = iw_{1,1}(d)_k + \Delta iw_{1,1}(d)_k = iw_{1,1}(d)_k - \left.\frac{\partial F}{iw_{1,1}(d)}\right|_k \quad \forall d \in [0,2]$$

$$\frac{\partial F}{iw_{1,1}(d)} = \sum_{t=1}^{3}\left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{iw_{1,1}}}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)} = \sum_{d=0}^{2} \frac{\partial a(1)}{\partial iw_{1,1}(d)} \times \frac{\partial^e F}{\partial a(1)} + \frac{\partial a(2)}{\partial iw_{1,1}(d)} \times \frac{\partial^e F}{\partial a(2)} + \frac{\partial a(3)}{\partial iw_{1,1}(d)} \times \frac{\partial^e F}{\partial a(3)}$$
$$= p(1-d)\big(-2e(1)\big) + p(2-d)\big(-2e(2)\big) + p(3-d)\big(-2e(3)\big)$$

- **Real Time Recurrent Learning. Example IIR**

$$a(t) = lw_{1,1}(1)a(t-1) + iw_{1,1}p(t)$$

$$F = \sum_{t=1}^{Q}(t(t)-a(t))^2 = \sum_{t=1}^{3}e^2(t) = e^2(1)+e^2(2)+e^2(3)$$

$$\{p(1), t(1)\}, \{p(2), t(2)\}, \{p(3), t(3)\}$$

$$a(1) = lw_{1,1}(1)a(0) + iw_{1,1}p(1)$$

$$iw_{1,1_{k+1}} = iw_{1,1_k} + \Delta iw_{1,1_k} = iw_{1,1_k} - \left.\frac{\partial F}{iw_{1,1}}\right|_k \qquad lw_{1,1}(1)_{k+1} = lw_{1,1}(1)_k + \Delta lw_{1,1}(1)_k = lw_{1,1}(1)_k - \left.\frac{\partial F}{lw_{1,1}(1)}\right|_k$$

$$\frac{\partial F}{iw_{1,1}} = \sum_{t=1}^{3}\left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{iw_{1,1}}}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)} \qquad \frac{\partial F}{lw_{1,1}(1)} = \sum_{t=1}^{3}\left[\frac{\partial \mathbf{a}(t)}{\partial \mathbf{lw_{1,1}}(1)}\right]^T \times \frac{\partial^e F}{\partial \mathbf{a}(t)}$$

- **Real Time Recurrent Learning. Example IIR**

$$a(1) = lw_{1,1}(1)a(0) + iw_{1,1}(1)p(1) \rightarrow \begin{cases} \dfrac{\partial a(1)}{\partial iw_{1,1}} = p(1) + lw_{1,1}(1)\dfrac{\partial a(0)}{\partial iw_{1,1}} = p(1) \\ \dfrac{\partial a(1)}{\partial lw_{1,1}(1)} = a(0) + lw_{1,1}(1)\dfrac{\partial a(0)}{\partial lw_{1,1}(1)} = a(0) \end{cases}$$

$$a(2) = lw_{1,1}(1)a(1) + iw_{1,1}(1)p(2) \rightarrow \begin{cases} \dfrac{\partial a(2)}{\partial iw_{1,1}} = p(2) + lw_{1,1}(1)\dfrac{\partial a(1)}{\partial iw_{1,1}} = p(2) + lw_{1,1}(1)p(1) \\ \dfrac{\partial a(2)}{\partial lw_{1,1}(1)} = a(1) + lw_{1,1}(1)\dfrac{\partial a(1)}{\partial lw_{1,1}(1)} = a(1) + lw_{1,1}(1)a(0) \end{cases}$$

$$a(3) = lw_{1,1}(1)a(2) + iw_{1,1}(1)p(3) \rightarrow \begin{cases} \dfrac{\partial a(3)}{\partial iw_{1,1}} = p(3) + lw_{1,1}(1)\dfrac{\partial a(2)}{\partial iw_{1,1}} = p(3) + lw_{1,1}(1)p(2) + \left(lw_{1,1}(1)\right)^2 p(1) \\ \dfrac{\partial a(2)}{\partial lw_{1,1}(1)} = a(2) + lw_{1,1}(1)\dfrac{\partial a(2)}{\partial lw_{1,1}(1)} = a(2) + lw_{1,1}(1)a(1) + \left(lw_{1,1}(1)\right)^2 a(0) \end{cases}$$

Dynamic Networks

- **Real Time Recurrent Learning. Example IIR**

$$\frac{\partial F}{\partial iw_{1,1}} = \frac{\partial a(1)}{\partial iw_{1,1}}\big(-2e(1)\big) + \frac{\partial a(2)}{\partial iw_{1,1}}\big(-2e(2)\big) + \frac{\partial a(3)}{\partial iw_{1,1}}\big(-2e(3)\big)$$

$$= -2e(1)\big[p(1)\big] - 2e(2)\big[p(2) + lw_{1,1}(1)\,p(1)\big] - 2e(3)\big[p(3) + lw_{1,1}(1)\,p(2) + \big(lw_{1,1}(1)\big)^2\,p(1)\big]$$

$$\frac{\partial F}{\partial lw_{1,1}(1)} = \frac{\partial a(1)}{\partial lw_{1,1}(1)}\big(-2e(1)\big) + \frac{\partial a(2)}{\partial lw_{1,1}(1)}\big(-2e(2)\big) + \frac{\partial a(3)}{\partial lw_{1,1}(1)}\big(-2e(3)\big)$$

$$= -2e(1)\big[a(0)\big] - 2e(2)\big[a(1) + lw_{1,1}(1)a(0)\big] - 2e(3)\big[a(2) + lw_{1,1}(1)a(1) + \big(lw_{1,1}(1)\big)^2\,a(0)\big]$$

Neural Networks

Dynamic Networks

```
% Incremental Training with Dynamic Networks
Pi = {1}; % Initial input
P = {2 3 4};
T = {3 5 7};
%Network architecture
net = linearlayer([0 1],0.1);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.biasConnect = 0;
net.trainParam.epochs = 1;
[net,a,e,pf] = adapt(net,P,T,Pi);
net.IW{1,1} % ans = [0.9880 0.5260]
% a = [0] [2.4] [7.98]
% e = [3] [2.6] [-0.98]

% Batch Training with Dynamic Networks
net = linearlayer([0 1],0.02);
net = configure(net,P,T);
net.IW{1,1} = [0 0];
net.biasConnect = 0;
net.trainParam.epochs = 1;
net = train(net,P,T,Pi);
net.IW{1,1} % ans = [0.9800 0.6800]
```

Neural Networks

Dynamic Networks

```
% Load the data
y = laser_dataset;
y = y(1:600);
% Arrange inputs and targets for
training a one-step-ahead predictor
p = y(9:end);
t = y(9:end);
Pi=y(1:8);

% Linear Dynamic Neural Network (LDNN)
lin_net = linearlayer([1:8],10);
lin_net.trainFcn='trainlm';
%lin_net.trainParam.epochs = 1000;
%lin_net.divideFcn = '';
[lin_net,tr] = train(lin_net,p,t,Pi);
% Simulate the network
lin_yp = lin_net(p,Pi);
% Calculate the prediction error
lin_e = gsubtract(lin_yp,t);
lin_rmse = sqrt(mse(lin_e)) % lin_rmse
= 21.1386
figure(1); plotresponse(t,lin_yp);
title('LDNN');
```

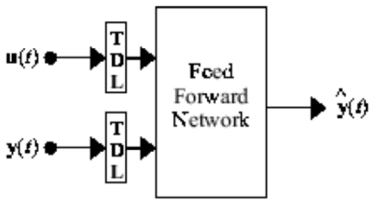
```
% Focused Time-Delay Neural Network (FTDNN)
ftdnn_net = timedelaynet([1:8],10);
ftdnn_net.trainParam.epochs = 1000;
ftdnn_net.divideFcn = '';
% Simulate the network
ftdnn_net = train(ftdnn_net,p,t,Pi);
% Calculate the prediction error
ftdnn_yp = ftdnn_net(p,Pi);
e = gsubtract(ftdnn_yp,t);
ftdnn_rmse = sqrt(mse(e)) % ftdnn_rmse = 1.1736
figure(2); plotresponse(t,ftdnn_yp); title('FTDNN');

% Time Series Distributed Delay Neural Networks (TDNN)
d1 = 0:4;
d2 = 0:3;
% The difference with timedelaynet is the first input
argument is a cell array with the tapped delays
dtdnn_net = distdelaynet({d1,d2},5);
dtdnn_net.trainFcn = 'trainbr';
dtdnn_net.divideFcn = '';
dtdnn_net.trainParam.epochs = 200;
dtdnn_net = train(dtdnn_net,p,t);
dtdnn_yp = sim(dtdnn_net,p);
e = gsubtract(dtdnn_yp,t);
dtdnn_rmse = sqrt(mse(e)) % dtdnn_rmse = 0.0010
figure(3); plotresponse(t,dtdnn_yp); title('TDNN');
```

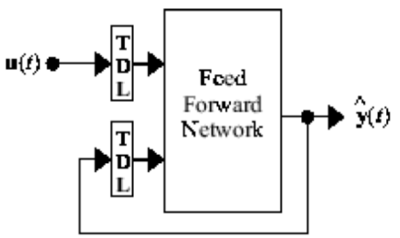
Neural Networks

Dynamic Networks

```
% Recurrent dynamic network (NARX)
% Data to model a magnetic levitation system
load magdata
u = con2seq(u); %inputs
y = con2seq(y); %outputs
% Create a series-parallel NARX network
d1 = [1:2];
d2 = [1:2];
narx_net = narxnet(d1,d2,10);
narx_net.divideFcn = '';
narx_net.trainParam.min_grad = 1e-10;
[p,Pi,Ai,t] = preparets(narx_net,u,{},y); %prepares data for the training
narx_net = train(narx_net,p,t,Pi);
% Simulate the network
yp = sim(narx_net,p,Pi);
% Calculate the resulting errors for the series-parallel implementation
e = cell2mat(yp)-cell2mat(t);
figure(1); plot(e);
xlabel('samples'); ylabel('error'); title('series-parallel error');
% Converting NARX from the series-parallel configuration (open loop) to the
parallel
% configuration (closed loop)
narx_net_closed = closeloop(narx_net);
view(narx_net); view(narx_net_closed);
% Use the closed-loop (parallel) configuration to perform an iterated prediction
y1 = y(1700:2600);
u1 = u(1700:2600);
[p1,Pi1,Ai1,t1] = preparets(narx_net_closed,u1,{},y1);
yp1 = narx_net_closed(p1,Pi1,Ai1);
figure(2); TS = size(t1,2); plot(1:TS,cell2mat(t1),'b',1:TS,cell2mat(yp1),'r')
xlabel('inputs'); ylabel('outputs'); title('Prediction with closed-loop NARX');
% You can also create a parallel (closed loop) NARX network, using the
% narxnet command. The training takes longer and the resulting performance is worse
% net = narxnet(d1,d2,10,'closed');
```



Series-Parallel Architecture



Parallel Architecture

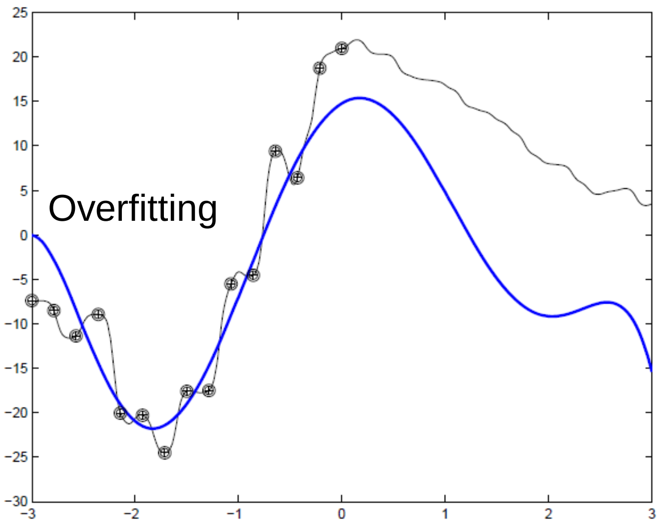
Generalization

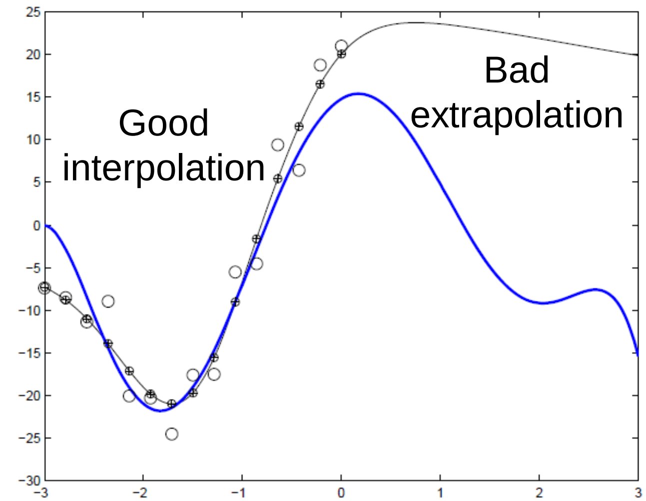
Neural Networks

Generalization

- A network trained to generalize will **perform as well in new situations** as it does on the data on which it was trained
- To find a network that generalizes well, we need to find the **simplest network** that fits the data
- **Methods:** growing, pruning, global searches, regularization, early stopping
- **Problem statement:** $\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$ $\quad \mathbf{t}_q = \mathbf{g}(\mathbf{p}_q) + \varepsilon_q$

$$F(\mathbf{x}) = E_D = \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q)$$





Generalization

- **Data-set in the training process**
  - **Training** set (70%).
    - For the training proccess (weight and bias update)
  - **Validation** set (15%).
    - For stopping training
  - **Test** set (15%).
    - For error calculation. It is a measure of the generalization NN capability

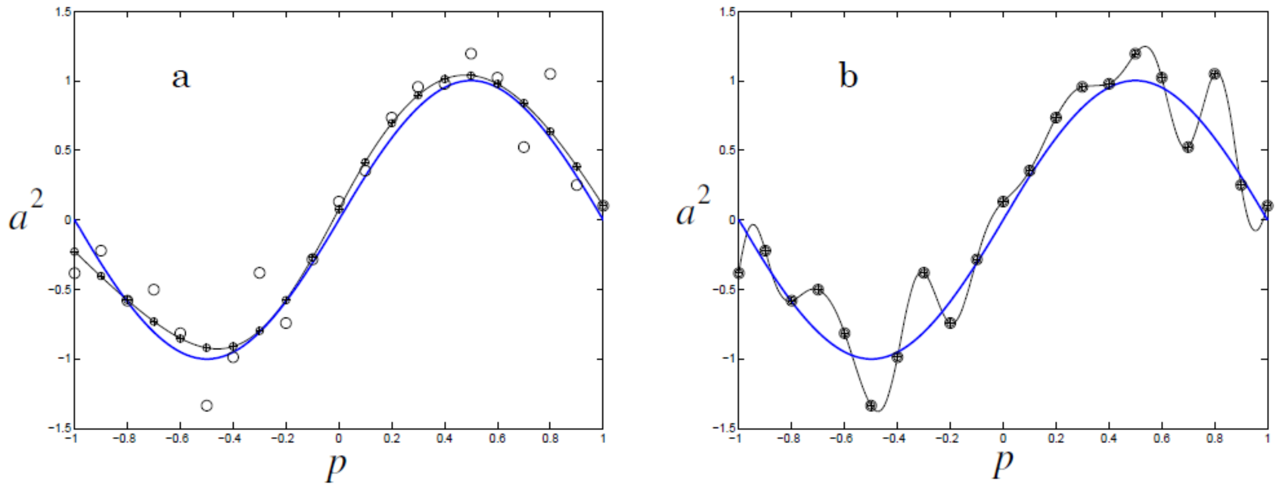
  - All datasets must be representative of all NN situations

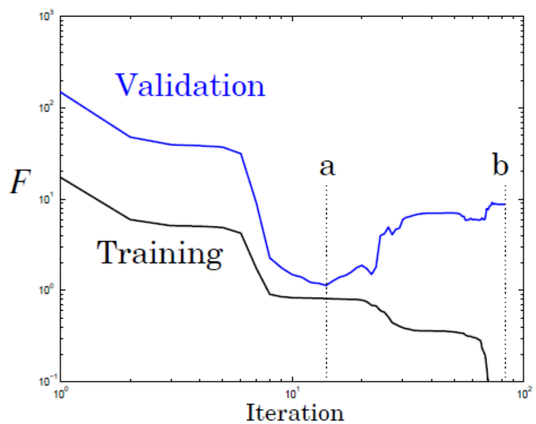
```
%Change the division function
net.divideFcn = '';
```

Function	Algorithm
dividerand	Divide the data randomly (default)
divideblock	Divide the data into contiguous blocks
divideint	Divide the data using an interleaved selection
divideind	Divide the data by index

Generalization

- **Early Stopping**
  - Cross-validation method. It uses a validation set to decide when to stop



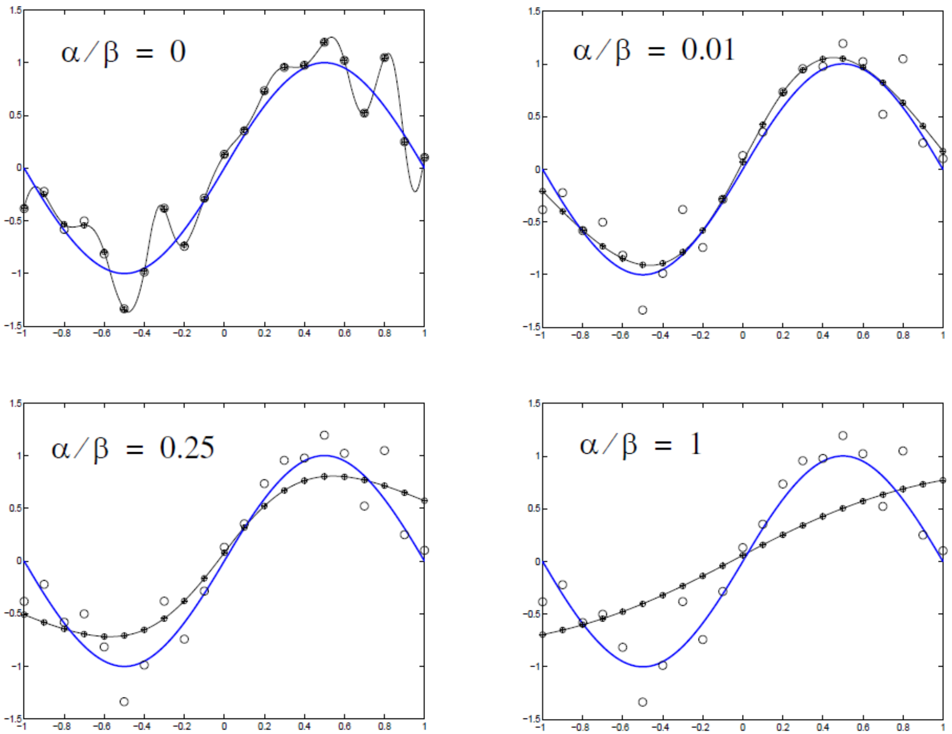



```
% When the validation error
increases for a specified
number of iterations
net.trainParam.max_fail=3;
```

Generalization

- **Regularization**
  - Sum squared error performance index is modified to include a term that penalizes network complexity

$$F(\mathbf{x}) = \beta E_D + \alpha E_W = \beta \sum^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q) + \alpha \sum^{n} x_i^2$$



Generalization

```
%Regularization
[x,t] = simplefit_dataset;
x1=x(1:60); x2=x(61:94); % Data to check overfitting
t1=t(1:60); t2=t(61:94);
net = feedforwardnet(10,'trainbfg');
net.divideFcn = '';
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
net.performParam.regularization = 0.1;
net = train(net,x,t);
y=sim(net,x);
e = gsubtract(y,t);
rmse = sqrt(mse(e))
% The problem with regularization is that it is difficult to determine the
% optimum value for the performance ratio parameter
```