3 ADALINE — The Adaptive Linear Element

- The Adaline can be thought of as the smallest, linear building block of the artificial neural networks.
- This element has been extensively used in science, statistics (in the linear **linear regression analysis**), engineering (the **adaptive signal processing**, **control systems**), and many other areas.

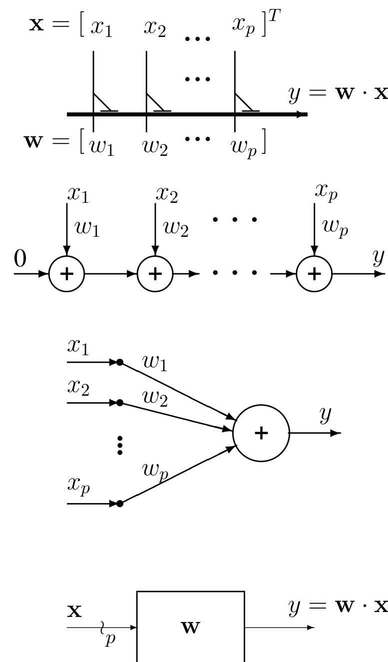
In general, the Adaline is used to perform

linear approximation of a "small" segment of a nonlinear hyper-surface, which is generated by a p–variable function, $y = f(\mathbf{x})$.
In this case, the bias is usually needed, hence, $w_p = 1$.

linear filtering and prediction of data (signals)

pattern association, that is, generation of m–element output vectors (using m Adalines) associated with respective p–element input vectors.

Various representations of the Adaline:



3.1 Linear approximation of a p-variable function

- A function of p variables, $y = f(\mathbf{x})$, can be interpreted as a hyper-surface in a $(p + 1)$–dimensional space, $[x_1, \dots x_p, y]$.
- In this section, we will discuss methods of approximating such a surface by a hyperplane using an Adaline. It is also known as a linear regression problem: Given

  $d = f(\mathbf{x})$, we find a hyperplane $y = \mathbf{w} \cdot \mathbf{x}$ such that the error $\varepsilon = |d - y|$ is small for all $\mathbf{x}$.

- We start with a bit more general problem, namely, approximation of m such functions using m p-input Adalines.
- Let the functions to be linearly approximated be known at N points, $\mathbf{x}(n)$, $\mathbf{d}(n)$ being a vector of values of functions.
- N points (training patterns) can be arranged, as previously, in the following two matrices:

$$\begin{aligned} X &= [\, \mathbf{x}(1) \ \dots \ \mathbf{x}(n) \ \dots \ \mathbf{x}(N) \,] \quad \text{is} \quad p \times N \quad \text{matrix,} \\ D &= [\, \mathbf{d}(1) \ \dots \ \mathbf{d}(n) \ \dots \ \mathbf{d}(N) \,] \quad \text{is} \quad m \times N \quad \text{matrix} \end{aligned}$$

- In order to approximate the above function let us consider a p-input m-output Adaline characterised by an $m \times p$ weight matrix, W, each row related to a single neuron.
- For each input vector, $\mathbf{x}(n)$, the Adaline calculates the actual output vector

$$\mathbf{y}(n) = W \cdot \mathbf{x}(n) \ . \qquad (3.1)$$

- All output vectors can also be arranged in an output matrix:

$$Y = [\, \mathbf{y}(1) \ \ldots \ \mathbf{y}(n) \ \ldots \ \mathbf{y}(N) \,] \quad \text{is} \quad m \times N \quad \text{matrix}$$

- The complete set of the output vectors can also be calculated as:

$$Y = W \cdot X \tag{3.2}$$

- Typically, the actual output vector, $\mathbf{y}(n)$ differs from the desired output vector, $\mathbf{d}(n)$, and the **pattern error**:

$$\boldsymbol{\varepsilon}(n) = \mathbf{d}(n) - \mathbf{y}(n) \quad \text{is a } \ m \times 1 \ \text{ vector,} \tag{3.3}$$

each component being equal to:

$$\varepsilon_j(n) = d_j(n) - y_j(n) \ . \tag{3.4}$$

- The problem of approximation of the surfaces specified by D by the hyper-planes specified by weight vectors stored in the weight matrix, W, is to select the weights so that the errors are as small as possible.
- The total measure of the goodness of approximation, or the **performance index**, can be specified by the **mean-squared error** over m neurons and N training vectors:

$$J(W) = \frac{1}{2mN} \sum_{n=1}^{N} \sum_{j=1}^{m} \varepsilon_j^2(n) \tag{3.5}$$

- Defining the total **instantaneous** error over m neurons as:

$$E(n) = \frac{1}{2} \sum_{j=1}^{m} \varepsilon_j^2(n) = \frac{1}{2} \boldsymbol{\varepsilon}^T(n) \cdot \boldsymbol{\varepsilon}(n) \tag{3.6}$$

the performance index can be expressed as

$$J(W) = \frac{1}{2mN} \sum_{n=1}^{N} E(n) = \frac{1}{2mN} \ \mathbf{e}^T \cdot \mathbf{e} \tag{3.7}$$

where $\mathbf{e}$ is a $mN \times 1$ vector consisting of all errors which can be calculated as:

$$\mathcal{E} = D - Y \ ; \ \ \mathbf{e} = \mathcal{E}(:)$$

where ‘:’ is the MATLAB column-wise scan operator.

- The performance index, $J(W)$, is a non-negative scalar function of $(m \cdot p)$ weights (a quadratic surface in the **weight space**).
- To solve the approximation problem, we will determine the weight matrix which minimises the performance index, that is, the **mean-squared error**, $J(W)$.
- For simplicity, solution to the approximation problem will be given for a **single-neuron** case (single output), when $m = 1$. Now, $\mathbf{e}^T = \mathcal{E} = D - Y$.
- The weight matrix, W, becomes the $1 \times p$ vector, $\mathbf{w}$ and the **mean-squared error**, $J(\mathbf{w})$, can now be calculated in the following way:

$$J(\mathbf{w}) = \frac{1}{2N}(D - Y)(D - Y)^T = \frac{1}{2N}(D \cdot D^T - D \cdot Y^T - Y \cdot D^T + Y \cdot Y^T)$$

where D and $Y = \mathbf{w} \cdot X$ are now $1 \times N$ row-matrices.

- If we take into account that the inner product of vectors is commutative, that is, $\mathbf{u}^T \cdot \mathbf{v} = \mathbf{v}^T \cdot \mathbf{u}$, then we have

$$\begin{aligned} J &= \frac{1}{2N}(\|D\|^2 - 2DY^T + YY^T) \\ &= \frac{1}{2N}(\|D\|^2 - 2DX^T\mathbf{w}^T + \mathbf{w}XX^T\mathbf{w}^T) \end{aligned}$$

- Denote by

$$\mathbf{q} = (D \cdot X^T)/N \quad \text{the } 1 \times p \textbf{ cross-correlation} \text{ vector,} \tag{3.8}$$

$$R = (X \cdot X^T)/N \quad \text{the } p \times p \textbf{ input correlation} \text{ matrix.} \tag{3.9}$$

- Then the mean-squared error finally becomes

$$J(\mathbf{w}) = \frac{1}{2}(\|D\|^2/N - 2\mathbf{q}\mathbf{w}^T + \mathbf{w}R\mathbf{w}^T) \tag{3.10}$$

Example

Consider an example of a 2-D performance index when $(p = 2)$ and $\mathbf{w} = [w_1\ w_2]$.
Eqn (3.10) is of the following matrix form:

$$J(\mathbf{w}) = \frac{1}{2}\mathbf{w}R\mathbf{w}^T - \mathbf{q}\mathbf{w}^T + c$$

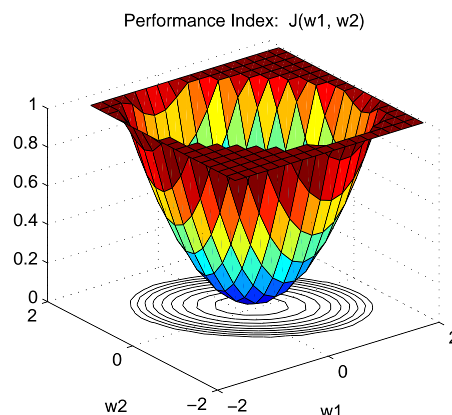
Let

$$J(w_1, w_2) = [w_1\ w_2] \begin{bmatrix} 9 & 4 \\ 4 & 10 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - [5\ \ 4] \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + 1 \tag{3.11}$$

We can re-write eqn (3.11) in the following "scalar" form:

$$J(w_1, w_2) = 9w_1^2 + 8w_1w_2 + 10w_2^2 - 5w_1 - 4w_2 + 1$$

The plot of the normalised performance index, $J(w_1, w_2)$ has been generated by a MATLAB script `cJ2x.m`:



In order to find the optimal weight vector which minimises the mean-squared error, $J(\mathbf{w})$, we calculate the gradient of J with respect to $\mathbf{w}$:

$$\nabla J(\mathbf{w}) = \frac{\partial J}{\partial \mathbf{w}} = \left[\frac{\partial J}{\partial w_1} \ \cdots \ \frac{\partial J}{\partial w_p}\right] = \frac{1}{2}\nabla(\|D\|^2/N - 2\mathbf{q}\mathbf{w}^T + \mathbf{w}R\mathbf{w}^T) = -\mathbf{q} + \mathbf{w}R^T$$

Taking into account that $R = R^T$ (a symmetric matrix), the gradient of the performance index finally becomes:

$$\nabla J(\mathbf{w}) = -\mathbf{q} + \mathbf{w}R \tag{3.12}$$

The gradient, $\nabla J(\mathbf{w})$, becomes zero for:

$$\mathbf{w}R = \mathbf{q} \tag{3.13}$$

This is a very important equation known as the **normal** or **Wiener–Hopf** equation.

This is a **set of** p **linear equations** for $\mathbf{w} = [w_1 \ \cdots \ w_p]$.

The solution, if exists, can be easily found, and is equal to:

$$\mathbf{w} = \mathbf{q} \cdot R^{-1} = \mathbf{q}/R = DX^T(XX^T)^{-1} \tag{3.14}$$

Using a concept of the **pseudo-inverse** of a matrix X defined as:

$$X^+ \stackrel{\mathrm{def}}{=} X^T(XX^T)^{-1} \tag{3.15}$$

the optimal in the least-mean-squared sense weight vector can be also calculated as

$$\mathbf{w} = D \cdot X^+ = D/X \tag{3.16}$$

Example: The performance index of eqn (3.11) attains minimum for

$$\mathbf{w} = [w_1 \ w_2] = \frac{1}{2}\,[5 \ \ 4]\begin{bmatrix} 9 & 4 \\ 4 & 10 \end{bmatrix}^{-1} = [0.23 \ \ 0.11]$$

□

In the **multi-neuron** case, when D is a $m \times N$ matrix, the optimal weight matrix W (which is $m \times p$) can be calculated in a similar way as:

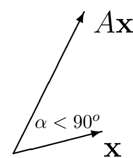
$$W = D \cdot X^+ = D/X \tag{3.17}$$

- In order to check that the above weight vector really minimises the performance index, we calculate the **second derivative** of J which is known as the **Hessian matrix**:

$$H(\mathbf{w}) = \frac{\partial^2 J}{\partial \mathbf{w}^2} = \frac{\partial}{\partial \mathbf{w}}(\nabla J(\mathbf{w})) = R \quad (3.18)$$

- The second derivative is independent of the weight vector and is equal to the input correlation matrix, R.
- R, as a product of X and X^T, can be proved to be a **positive-definite** matrix.
- Moreover, if the number of linearly independent input vectors is at least p, then the R matrix is of full rank.
- This means that the performance index attains minimum for the optimal weight vector, and that the minimum is unique.
- A matrix A is said to be positive-definite if and only if for all non-zero vectors $\mathbf{x}$

$$\mathbf{x}^T \cdot A \cdot \mathbf{x} > 0$$



- It can be shown that the eigenvalues of a positive-definite matrix are real and positive.

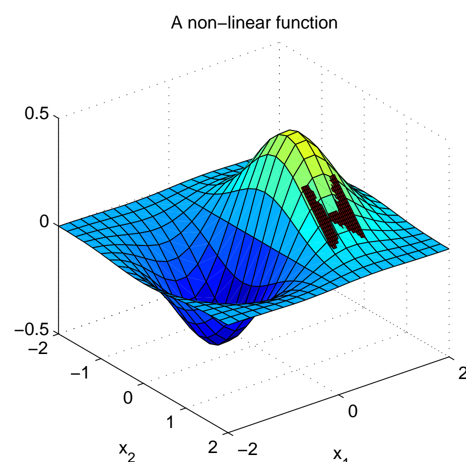
Approximation by a plane — MATLAB example (adln1.m)

In this example, we approximate a small section of a nonlinear 2-D surface with a plane which is specified by a weight vector of a linear neuron.

First, for the 2-D input domain $\mathbf{x} = [x_1, x_2] \in \{-2, 2\}$ we calculate 21 × 21 points of the Gaussian-like function:

$$y = f(\mathbf{x}) = x_1 e^{(-x_1^2 - x_2^2)}$$

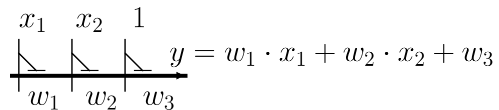
```
N = 20; NN = (0:N)/N ; M = (N+1)^2 ;
x1 = 4*NN - 2 ;
[X1 X2] = meshgrid(x1, x1);
y = X1 .* exp(-X1.^2 - X2.^2);
figure(1), clf reset
surf(x1, x1, y), axis('ij'), hold on
```



Next, we select a small segment of the surface, for $x_1 \in \{0.4 \ .. \ 1\}, x_2 \in \{0.6 \ .. \ 1.4\}$, and form the set of input points (training vectors), X, taken from the points of the 21 × 21 grid.

The desired output values D are calculated from the equation of the function being approximated, $d = f(\mathbf{x})$

We need also the bias input $x_3 = 1$, so that the equation of the approximating plane and the related Adaline will be:




```
x1 = 0.6*NN+0.4;
x2 = 0.8*NN+0.6;
[X1 X2] = meshgrid(x1, x2);
d = X1 .* exp(-X1.^2 - X2.^2);
D = d(:)';
X = [X1(:)'; X2(:)'; ones(1,M)];
```

The three initial and four last training vectors $\mathbf{x}(k) = [x_1\ x_2\ 1]^T$, and $d(n)$ are of the following form:

```
X(:, [1:3 (M-3):M]) =
    0.40  0.40  0.40 ... 1.00  1.00  1.00  1.00
    0.60  0.64  0.68 ... 1.28  1.32  1.36  1.40
    1.00  1.00  1.00 ... 1.00  1.00  1.00  1.00

D(:, [1:3 (M-3):M]) =
   0.2378 0.2263 0.2147 ... 0.0715 0.0644 0.0579 0.0518
```

Then we calculate the cross-correlation vector $\mathbf{q}(n)$ and the input correlation matrix R.

```
q = (D*X')/M = 0.1093 0.1388 0.1555
R = (X*X')/M =  0.52  0.70  0.70
                0.70  1.06  1.00
                0.70  1.00  1.00
```


The eigenvalues of the input correlation matrix are real and positive which indicates that R is positive-definite and of full rank:

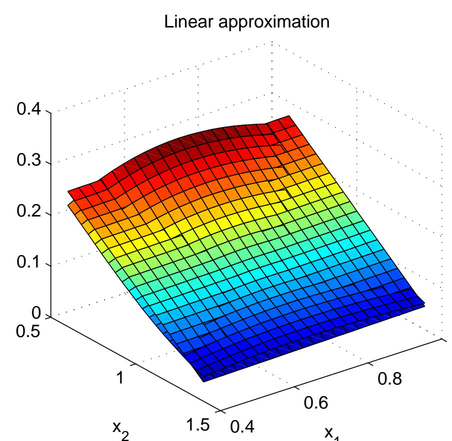
```
eig(R) = 0.0175 0.0438 2.5203

w =  q/R  = 0.0132  -0.2846  0.4309   % THE  SOLUTION!

Y = w*X;   Y(:, [1:3 438:441]) =
  0.2654  0.2540  0.2426 ... 0.0798  0.0684  0.0570  0.0456
```

```
err = sum(abs(D-Y)) =  1.54
YY = d ; YY(:) = Y;
surf(x1, x2, YY), hold off

figure(2)
surf(x1, x2, d),  axis('ij'), hold on
surf(x1, x2, YY), hold off
```



3.2 Method of steepest descent

- In order to calculate the optimal weights which minimise the approximation error, $J(\mathbf{w})$, we have to
  - calculate the correlation matrices, $\mathbf{q}$ and R, and
  - inverse the autocorrelation matrix, R, as in eqn (3.14).
- The above operations can be computationally intensive for a large number of training patterns, N.
- In addition we might prefer a procedure that finds an optimal weight vector iteratively, in a pattern-by-pattern fashion as for the perceptron.
- We assume that at each step the weight vector is updated by a small update vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n) \tag{3.19}$$

- We can now estimate how the above change in the weight vector is reflected in change of the value of the performance (error) function $J(\mathbf{w})$ defined in eqn (3.5). We have

$$\begin{aligned} J(\mathbf{w}+\Delta\mathbf{w}) &= \frac{1}{2}(\|D\|^2/N - 2\cdot\mathbf{q}\cdot(\mathbf{w}+\Delta\mathbf{w})^T + (\mathbf{w}+\Delta\mathbf{w})\cdot R\cdot(\mathbf{w}+\Delta\mathbf{w})^T) \\ &= J(\mathbf{w}) + \frac{1}{2}(-2\cdot\mathbf{q}\cdot\Delta\mathbf{w}^T + 2\cdot\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T + \Delta\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T) \\ &= J(\mathbf{w}) + (-q+\mathbf{w}\cdot R)\cdot\Delta\mathbf{w}^T + \frac{1}{2}\Delta\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T) \end{aligned}$$

Finally, we have

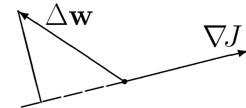
$$J(\mathbf{w}+\Delta\mathbf{w}) = J(\mathbf{w}) + \nabla J(\mathbf{w})\cdot\Delta\mathbf{w}^T + \frac{1}{2}\Delta\mathbf{w}\cdot R\cdot\Delta\mathbf{w}^T \tag{3.20}$$

- In an iterative search for the minimum of the performance index we would like its value to decrease at each step, that is

$$J(\mathbf{w}+\Delta\mathbf{w}) < J(\mathbf{w}) \;\text{ or }\; J(\mathbf{w}(n+1)) < J(\mathbf{w}(n))$$

- If in eqn (3.20) we neglect the second order term, then the above condition is equivalent to:

$$\Delta\mathbf{w}\cdot\nabla J(\mathbf{w}) < 0$$



- This condition means that in order to reduce the value of J we should move in the direction $\Delta\mathbf{w}$ such that its projection on $\nabla J(\mathbf{w})$ is negative.
- The most negative results is obtained when

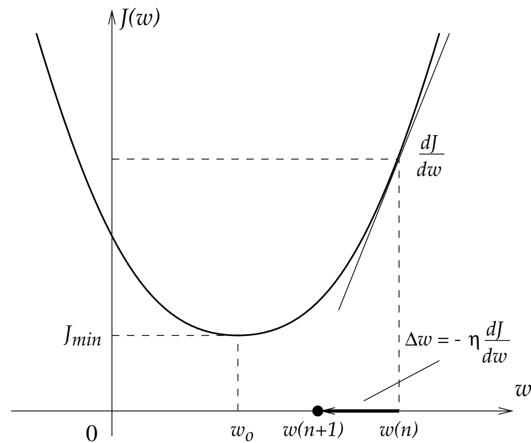
$$\Delta\mathbf{w} = -\eta\,\nabla J(\mathbf{w}) = \eta(q - \mathbf{w}\cdot R) \tag{3.21}$$

- This is the **steepest descent method** that states that in order to reduce J in successive steps, the weight vector should be modified in the direction **opposite to the gradient** of the error function J.
- η is an important parameter known as the **learning gain**, and $\mathbf{q}$ and R are cross- and input correlation matrices defined in eqns (3.8), (3.9).

- The steepest descent learning law of eqn (3.21) is a **batch method**, that is, for a complete set of data, N points being stored in X and D, the cross-correlation vector $\mathbf{q}$ and the input correlation matrix R is calculated and iterations are performed until $\Delta\mathbf{w}$ is close to zero.
- When $\Delta\mathbf{w} = 0$, then $\mathbf{w}R = q$, $\nabla J(\mathbf{w}) = 0 \implies$ $\mathbf{w}$ minimises J eqn (3.13)

Illustration of the steepest descent method:

- Moving in the direction opposite to the gradient of the performance index in the weight space takes us towards the minimum of error.
- When the weight vector attains the optimal value for which the gradient is zero (w_0 in the figure), the iterations are stopped.



Optimal learning gain

Optimal/maximum learning gain can be estimated in the following way.

- Using the steepest descent rule of eqn (3.21) the full expression for the weight update becomes:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(q - \mathbf{w}(n) \cdot R)$$

  or

$$\mathbf{w}(n+1) = \mathbf{w}(n)(I - \eta\, R) + \eta\, q \quad (3.22)$$

  where I is an identity matrix of the same size as R.

- This recursive equation describe a linear dynamic system with weights being its state vector.
- It is known that for such a system to be stable, that is, weights to converge to a fixed value, the eigenvalues of the state matrix $(I - \eta\, R)$ must be located inside the unit circle, that is:

$$|\text{eig}(I - \eta\, R)| < 1 \quad (3.23)$$

- If $(\lambda_i, \mathbf{v}_i)$ are an eigenvalue–eigenvector pair, than we have:

$$(I - \eta\, R) \cdot \mathbf{v}_i = \lambda_i \cdot \mathbf{v}_i$$

- This can be re-written as:

$$R \cdot \mathbf{v}_i = \frac{1 - \lambda_i}{\eta}\mathbf{v}_i\,, \quad \text{or} \quad R \cdot \mathbf{v}_i = \lambda_{Ri}\, \mathbf{v}_i\,, \quad \text{where} \quad \lambda_{Ri} = \frac{1 - \lambda_i}{\eta}$$

  is an eigenvalue of the input correlation matrix R (due to the uniqueness of the eigen-decomposition).

- The condition (3.23) can now be written in terms of the eigenvalues of the correlation matrix as:

$$-1 < 1 - \eta\,\lambda_{Ri} < 1\ , \quad \text{for all} \quad i = 1, \ldots, p$$

  Note that eigenvalues of the correlation matrix are real and positive.

- Finally the condition for the **maximum stable learning gain** can be written in the following simple form:

$$\eta_{mx} = \frac{2}{\lambda_{mx}} \tag{3.24}$$

  where λ_{mx}, the largest eigenvalue of the input correlation matrix, represents the maximum curvature of the quadratic function J and

- the condition (3.24) states that the maximum stable learning gain is inversely proportional to this curvature.

Pattern learning

- Often we have a situation when the samples of data $(\mathbf{x}(n), d(n))$ arrive one at a time and we would like to get a weight update specifically for the new data sample.
- This is know as the pattern learning and two incremental learning law, LMS and RMS are discussed in subsequent sections.

3.3 The LMS (Widrow-Hoff) Learning Law

- The **Least-Mean-Square** (LMS) algorithm also known as the Widrow-Hoff Learning Law, or the **Delta Rule** is based on the instantaneous update of the correlation matrices, hence, on the **instantaneous update of the gradient** of the mean-squared error.
- To derive the instantaneous update of the gradient vector we will first express the current values of the correlation matrices in terms of their previous values (at the step $n-1$) and the updates at the step n.
- First observe that the current input vector $\mathbf{x}(n)$ and the desired output signal $d(n)$ are appended to the matrices $\mathbf{d}(n-1)$ and $X(n-1)$ as follows:

$$\mathbf{d}(n) = [\mathbf{d}(n-1)\ d(n)]\ , \quad \text{and} \quad X(n) = [X(n-1)\ \mathbf{x}(n)]$$

- Now using definitions of correlation matrices of eqns (3.8) and (3.9) we can write:

$$\begin{aligned}
\begin{bmatrix} \mathbf{q}(n) \\ R(n) \end{bmatrix} &= \left(\begin{bmatrix} \mathbf{d}(n-1) & d(n) \\ X(n-1) & \mathbf{x}(n) \end{bmatrix} \begin{bmatrix} X^T(n-1) \\ \mathbf{x}^T(n) \end{bmatrix} \right)/n = \left(\begin{bmatrix} \mathbf{d}(n-1)\,X^T(n-1) + d(n)\,\mathbf{x}^T(n) \\ X(n-1)\,X^T(n-1) + \mathbf{x}(n)\,\mathbf{x}^T(n) \end{bmatrix} \right)/n \\
&= \mu \begin{bmatrix} \mathbf{q}(n-1) \\ R(n-1) \end{bmatrix} + \begin{bmatrix} \Delta\mathbf{q}(n) \\ \Delta R(n) \end{bmatrix}
\end{aligned} \tag{3.25}$$

  where
$$\mu = \frac{n-1}{n} \approx 1 \ , \text{ and}$$

$$\Delta\mathbf{q}(n) = (d(n)\,\mathbf{x}^T(n))/n \quad \text{and} \quad \Delta R(n) = (\mathbf{x}(n)\,\mathbf{x}^T(n))/n \tag{3.26}$$

  are the instantaneous updates of the correlation matrices.

- The gradient of the mean-squared error at the step n can also be expanded into its previous values and the current update.
- From eqn (3.12), we have

$$\nabla J(n) = -\mu(\mathbf{q}(n-1) - \mathbf{w}(n)R(n-1)) - \Delta\mathbf{q}(n) + \mathbf{w}(n)\Delta R(n)$$

or

$$\nabla J(n) = \nabla\hat{J}(n-1) + \Delta\nabla J(n)\,, \quad \text{where} \quad \nabla\hat{J}(n-1) = \mu(-\mathbf{q}(n-1) + \mathbf{w}(n)R(n-1))$$

is the current (step n) estimate of the previous (step $n-1$) value of the gradient vector, and the gradient vector update is:

$$\begin{aligned}\Delta\nabla J(n) &= -\Delta\mathbf{q}(n) + \mathbf{w}(n)\Delta R(n) = -\frac{1}{n}(d(n)\,\mathbf{x}^T(n) - \mathbf{w}(n)\mathbf{x}(n)\,\mathbf{x}^T(n)) \\ &= -\frac{1}{n}(d(n) - \mathbf{w}(n)\mathbf{x}(n))\mathbf{x}^T(n) = -\frac{1}{n}(d(n) - y(n))\mathbf{x}^T(n) = -\frac{1}{n}\varepsilon(n)\mathbf{x}^T(n)\end{aligned}$$

- The **Least-Mean-Square** learning law replaces the gradient of the mean-squared error in eqn (3.21) with the **gradient update** and can be written in following form:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta_n \cdot \varepsilon(n) \cdot \mathbf{x}^T(n) \tag{3.27}$$

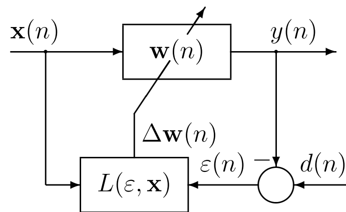
where the output error is

$$\varepsilon(n) = d(n) - y(n)$$

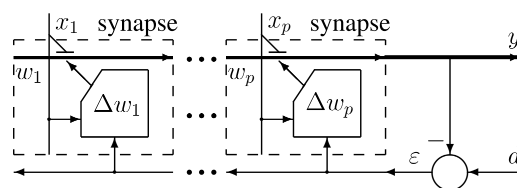
and the learning gain η_n can be either constant or reducible by the factor $1/n$.

Adaline with its error-correcting learning

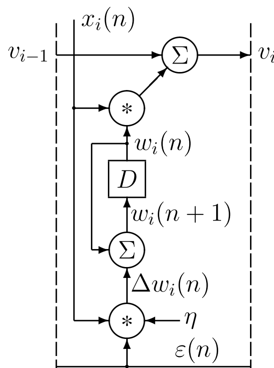
a. Block-diagram structure of an ADALINE



b. Dendritic structure of an ADALINE



c. Detailed structure of an ith synapse implementing the LMS learning



- The LMS weight update for the ADALINE is:

$$\Delta\mathbf{w}(n) = \eta \cdot \varepsilon(n) \cdot \mathbf{x}^T(n)$$

- The weight update for a single synapse is:

$$\Delta w_i(n) = \eta\,\varepsilon(n)\,x_i(n) \tag{3.28}$$

Some comments:

- The LMS learning law works because the instantaneous gradient update, $\Delta\nabla J(n)$ points, on average, in the direction of the gradient, $\nabla J(n)$, of the error function J.
- The LMS learning law of eqn (3.27) can be easily expanded into the case of the multi-neuron Adaline, describes a linear mapping from p-dimensional input space into an m-dimensional output space (m hyper planes).

Stopping criteria of the learning process:

- If it is possible, the learning process goes through all training examples (an epoch) number of times, until a **stopping criterion** is reached.
- The convergence process can be monitored with the plot of the mean-squared error function $J(W(n))$.
- The popular stopping criteria are:
  - the mean-squared error is sufficiently small:

$$J(W(n)) < \epsilon$$

  - The rate of change of the mean-squared error is sufficiently small:

$$\frac{\partial J(W(n))}{\partial n} < \epsilon$$

3.4 A Sequential Regression algorithm

The **sequential regression** algorithm also known as the **recursive least-square** (RLS) algorithm is based on the sequential update of the inverse of the input correlation matrix, $R^{-1}(n)$ for each new input pattern, $\mathbf{x}(n)$.

- Re-call that according to the **Wiener–Hopf** equation (3.13), the optimal weight vector can be calculated as

$$\mathbf{w} = \mathbf{q} \cdot R^{-1} \tag{3.29}$$

- In the sequential regression algorithm, the inverse of the input correlation matrix, R, is calculated iteratively as

$$R^{-1}(n) = \left(X(n) \cdot X^T(n)\right)^{-1} = f\left(R^{-1}(n-1), \mathbf{x}(n)\right) \tag{3.30}$$

  that is, the current value of the inverse, $R^{-1}(n)$ is calculated from the previous value of the inverse, $R^{-1}(n-1)$, and the input vector, $\mathbf{x}(n)$.

- As a starting point let us rewrite eqn (3.25) and the Wiener–Hopf equation (3.13) in the following forms:

$$\mathbf{q}(n) = \mu\, \mathbf{q}(n-1) + (d(n) \cdot \mathbf{x}^T(n))/n \;, \quad \text{where} \quad \mu = \frac{n-1}{n} \approx 1 \tag{3.31}$$

$$\mathbf{q}(n) = \mathbf{w}(n) \cdot R(n) \;, \quad \mathbf{q}(n-1) = \mathbf{w}(n-1) \cdot R(n-1) \tag{3.32}$$

- Substituting eqns (3.32) into eqn (3.31) we have

$$\mathbf{w}(n) \cdot R(n) = \mu \mathbf{w}(n-1) \cdot R(n-1) + (d(n) \cdot \mathbf{x}^T(n))/n \quad (3.33)$$

- From eqn (3.25), the current value of the input correlation matrix is related to its next value in the following way:

$$\mu R(n-1) = R(n) - (\mathbf{x}(n) \cdot \mathbf{x}^T(n))/n \quad (3.34)$$

- Substitution of eqn (3.34) into eqn (3.32) yields:

$$\mathbf{w}(n) \cdot R(n) = \mathbf{w}(n-1) \cdot R(n) - (\mathbf{w}(n-1) \cdot \mathbf{x}(n) \cdot \mathbf{x}^T(n))/n + (d(n) \cdot \mathbf{x}^T(n))/n$$

- Let us denote the scaled **inverse of the input correlation matrix** as:

$$P(n) = \frac{1}{n} R^{-1}(n)$$

- Post-multiplying eqn (3.4) by the inverse, $P(n)$, gives:

$$\mathbf{w}(n) = \mathbf{w}(n-1) + (d(n) - \mathbf{w}(n-1) \cdot \mathbf{x}(n)) \cdot \mathbf{x}^T(n) \cdot P(n) \quad (3.35)$$

- Denote by

$$\tilde{y}(n) = \mathbf{w}(n-1) \cdot \mathbf{x}(n)$$

  the **estimated output signal** based on the previous weight vector, $\mathbf{w}(n)$, and by

$$\varepsilon(n) = d(n) - \tilde{y}(n) = d(n) - \mathbf{w}(n-1) \cdot \mathbf{x}(n) \quad (3.36)$$

the error between the desired and estimated output, and by

$$\mathbf{k}(n) = \mathbf{x}^T(n) \cdot P(n) \quad (3.37)$$

the **update vector** known as the **Kalman gain**.

- Then, from eqns (3.35), (3.36) and (3.37), the sequential **weight update** can be expressed as

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \varepsilon(n) \cdot \mathbf{k}(n) \quad (3.38)$$

- Eqn (3.38) describes the sequential regression algorithms in terms of the output error, $\varepsilon(n)$, and the update vector (Kalman gain), $\mathbf{k}(n)$, which involves calculation of the inverse of the input correlation matrix, $P(n)$.
- In order to derive an iterative expression for this inverse we will need the **matrix inversion lemma**.

  According to this lemma, it is possible to show that if R, A, B are appropriately dimensioned matrices (i.e., $p \times p,\ p \times m,\ m \times p$, respectively), then

$$(R + A \cdot B)^{-1} = R^{-1} - R^{-1} \cdot A \cdot (I_m + B \cdot R^{-1} \cdot A)^{-1} \cdot B \cdot R^{-1} \quad (3.39)$$

- Let us re-write eqn (3.34) as

$$(n-1)R(n-1) = n\,R(n) - \mathbf{x}(n) \cdot \mathbf{x}^T(n) \quad (3.40)$$

  and apply the matrix inversion lemma to it.

- The scaled inverse of the input correlation matrix can be now written in the following form:

$$P(n) = P(n-1) - \mathbf{r}(n)\left(1 + \mathbf{r}^T(n)\mathbf{x}(n)\right)^{-1}\mathbf{r}^T(n)$$

or

$$P(n) = P(n-1) - \frac{\mathbf{r}(n)\cdot\mathbf{r}^T(n)}{1 + \mathbf{r}^T(n)\cdot\mathbf{x}(n)} \tag{3.41}$$

where

$$\mathbf{r}(n) = P(n-1)\cdot\mathbf{x}(n) \tag{3.42}$$

is the **input gain vector** similar to the Kalman gain.

- The update vector (Kalman gain) specified in eqn (3.37) can now be expressed as

$$\mathbf{k}(n) = \mathbf{r}^T(n) - \frac{\mathbf{x}^T(n)\cdot\mathbf{r}(n)\cdot\mathbf{r}^{\mathbf{T}}(n)}{1 + \mathbf{r}^T(n)\cdot\mathbf{x}(n)}$$

- or, finally, in the form

$$\mathbf{k}(n) = \frac{\mathbf{r}^T(n)}{1 + \mathbf{r}^T(n)\cdot\mathbf{x}(n)} \tag{3.43}$$

- Substitution of eqn (3.43) into eqn (3.41) finally yields equation for the iteration step for the inverse of the input correlation matrix:

$$P(n) = P(n-1) - P(n-1)\cdot\mathbf{x}(n)\cdot\mathbf{k}(n) \tag{3.44}$$

- This equation can, alernatively, be written in the following form

$$P(n) = P(n-1)\left(I_p - \mathbf{x}(n)\cdot\mathbf{k}(n)\right) \tag{3.45}$$

The Sequential Regression (SR) or Recursive Least-Square (RLS) algorithm — Summary

- It can be shown that using the SR algorithm the final value of the estimated input correlation matrix is

$$\hat{R}(N) = R(N) + \frac{P^{-1}(0)}{N}$$

Therefore, the initial value of the inverse of the input correlation matrix should be large to minimise the final error.

- Another problem to consider is that in practical applications we would like the algorithm to work continuously, that is, for large N, but with only the most recent input samples to contribute to the estimate of the correlation matrix.
- This is achieved by introduction of the "forgetting factor", λ in estimation of the correlation matrix.

The practical version of the RLS can be summarised as follows.

Initialisation:

$$\begin{aligned} P(1) &= R^{-1}(1) \quad \text{to be LARGE, e.g.} \quad P(1) = 10^6 I_p \\ \mathbf{w}(1) &= \text{small, random} \end{aligned}$$

an nth iteration step:

- Calculate the input gain vector $(p \times 1)$ as

$$\mathbf{r}(n) = \lambda^{-1} P(n-1) \cdot \mathbf{x}(n) \quad (3.46)$$

  where $0 < \lambda < 1$ is the forgetting factor.

- Calculate the Kalman gain vector $(1 \times p)$

$$\mathbf{k}(n) = \frac{\mathbf{r}^T(n)}{1 + \mathbf{r}^T(n) \cdot \mathbf{x}(n)} \quad (3.47)$$

  (A single neuron case $m = 1$ is assumed)

- Calculate the error signal

$$\varepsilon(n) = d(n) - \mathbf{w}(n) \cdot \mathbf{x}(n) \quad (3.48)$$

- Update the weight vector

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \varepsilon(n) \cdot \mathbf{k}(n) \quad (3.49)$$

- Calculate the next estimate of the **inverse** of the input correlation matrix

$$P(n+1) = \lambda^{-1} P(n) - \mathbf{r}(n) \cdot \mathbf{k}(n) \quad (3.50)$$

  where $\mathbf{r}(n)\mathbf{k}(n)$ is the outer product ($p \times p$ matrix) of the corresponding vectors.

  The forgetting factor, λ, de-emphasises contribution to the estimate of the inverse of the input correlation (covariance) matrix from older input data.

  A similar effect could be achieved by a periodic re-initialisation of the P matrix.

Another interpretation

- The RLS algorithm can be also seen as a way of optimal filtering the true signal, $d(n)$, from the output signal $y(n) = \mathbf{w}(n) \cdot \mathbf{x}(n)$.
- The error equation (3.48) can be re-written as a measurement equation:

$$d(n) = \mathbf{w}(n) \cdot \mathbf{x}(n) + \varepsilon(n)$$

  where $\varepsilon(n)$ is now the observation noise.

- The filtering procedure described above is known as the **Kalman filter**, and eqn (3.50) is known as the **Riccatti difference equation**.

3.5 ADALINE as an adaptive linear filter

- Traditional and very important applications of Linear Neural Networks are in the area one-dimensional adaptive signal processing, digital filtering and **time-series** prediction.
- A **digital filter**, such as an Adaline, is an algorithm executed either on a general purpose computer or specialised Digital Signal Processors (DSPs).
- In real-time signal processing we typically deal with an analog 1-D signal, $x(t)$, generated by some physical devices, for example, a microphone.
- The analog signal is passed through an Analog-to-Digital converter which does two operations: samples the analog signal with a given frequency, $f_s = 1/t_s$, and converts the samples into b-bit numbers, $x(n)$

$$x(t) \overset{t=n\cdot t_s}{\Longrightarrow} x(n)$$

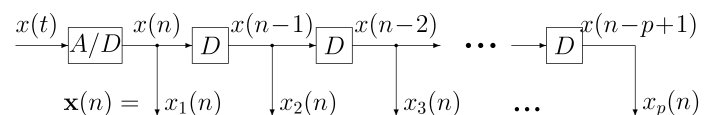
- Typically, more than one, say p, samples of the signal at each processing step are required.
- These samples are arranged into a p-element vector of input signals, $\mathbf{x}(n)$, supplied to a neural network:

$$\mathbf{x}(n) = [x(n)\ x(n-1) \ldots x(n-p+1)]^T \tag{3.51}$$

This vector of the **current and past samples** of the 1-D signal is created by a **tapped delay line** as illustrated below:

Conversion of a 1-D analog signal into digital samples supplied to a neural network using an Analog/Digital converter and a tapped delay line.

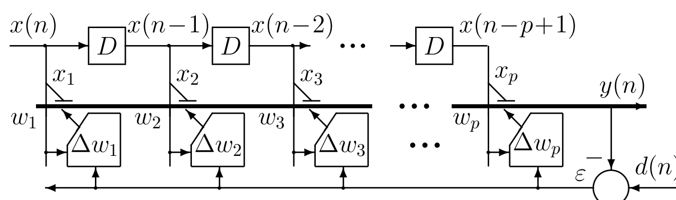
Note the difference between: $x(t), x(n)$, and $\mathbf{x}(n)$



Adaline as an adaptive FIR filter:

- If we connect outputs from the delay elements to the synapses of an Adaline as in the figure below, it will result in a signal processing structure known as an FIR (**Finite-Impulse-Response**) pth-order digital linear filter.
- In time-series processing such a system is called an MA (**Moving-Average**) model.

a. Detailed dendritic/synaptic structure:



b. Block-diagram:

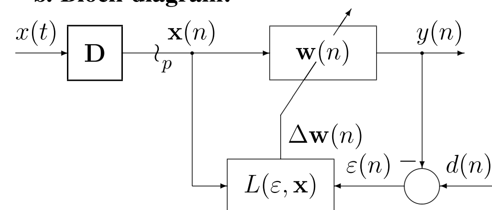


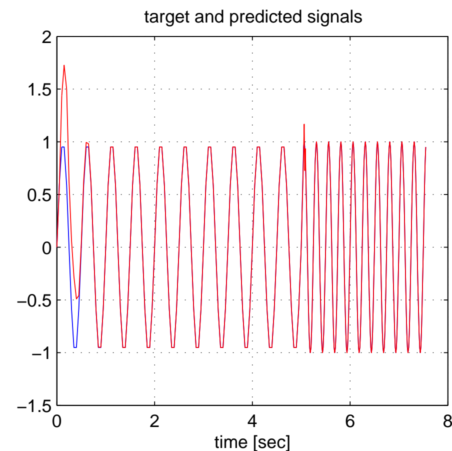
Figure 3–1: Adaline as an adaptive FIR filter

- If, in addition, the desired output signal, $d(n)$, is given, the filter's weights can be adapted using any of the previously discussed learning algorithms, so that the filter's output, $y(n)$ will track the desired output, $d(n)$.

3.5.1 Adaptive Prediction with Adaline — Example (adlpr.m)

- In this example an Adaline configured as in Figure 3–1 is used to predict a 1-D signal (time series).
- To predict the next value of the input signal, p samples of it are sent to the Adaline.
- The input signal is also used as the target/desired signal.
- The LMS learning law as in eqn (3.27) is used to adapt the weight vector at each step.
- We start with specification of a sinusoidal signal of frequency 2kHz sampled every 50μsec.
- After 5sec the frequency of the signal quadruples with the sampling time being also reduced to 12.5μsec.

```
f1 = 2 ; % kHz
ts = 1/(40*f1); % 12.5usec - sampling time,
                %  fs = 80kHz
N = 100 ;       % number of time steps
t1 = (0:N)*4*ts ;            % 1st time segment
t2 = (0:2*N)*ts + 4*(N+1)*ts; % 2nd time segment
t = [t1 t2] ;     % 0 to 7.5 sec total time
N = size(t, 2) ;   % N = 302
xt = [sin(2*pi*f1*t1) sin(2*pi*2*f1*t2)];
plot(t, xt), grid on,
title('Signal to be predicted')
```



- The 1-D signal (time series) must be converted into a collection of input vectors, $\mathbf{x}(n)$, as in eqn (3.51) and stored in a $p \times N$ matrix X:

$$X = \begin{bmatrix} x(0) & x(1) & x(2) & x(3) & x(4) & \dots \\ 0 & x(0) & x(1) & x(2) & x(3) & \dots \\ 0 & 0 & x(0) & x(1) & x(2) & \dots \\ 0 & 0 & 0 & x(0) & x(1) & \dots \end{bmatrix}$$

- It can be observed that the matrix X is a **convolution matrix** (a Sylvester's resultant matrix) associated with a time series, $x(t)$. The relevant MATLAB function is called convmtx.
- Try `convmtx(1:8, 5)` to clarify the operation.

```
p = 4 ;         % Number of synapses
X = convmtx(xt, p) ; X = X(:, 1:N) ;
d = xt ;  % The target signal is equal to the input signal
y  = zeros(size(d)) ;    % memory allocation for y
eps = zeros(size(d)) ;   % memory allocation for eps
eta = 0.4   ;            % learning rate/gain
w = rand(1, p) ; % Initialisation of the weight vector
for n = 1:N                 % LMS learning loop
  y(n) = w*X(:,n) ;         % predicted output signal
  eps(n) = d(n) - y(n) ;    % error signal
  w = w + eta*eps(n)*X(:,n)'; % weight update
end
```

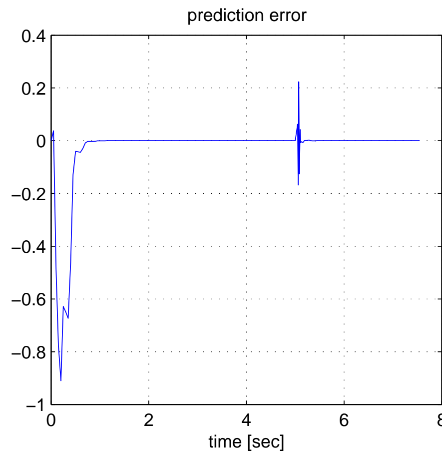


Figure 3–2: The Prediction error

- The resulting weight vectors can be as follows:

  Initial weight vector: `w = 0.9218 0.7382 0.1763 0.4057`

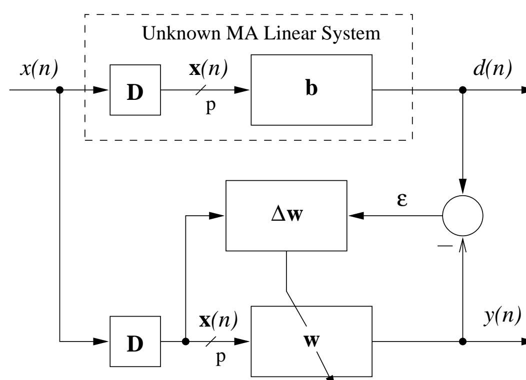
  Weight vectors at 5msec: `w = 0.7682 0.3797 -0.2394 0.0047`

  Final weight vector: `w = 0.7992 0.3808 -0.1987 -0.0011`
- Results vary from run to run not only because of a random initialisation, but because such a simple signal requires less than four parameter to be correctly predicted.

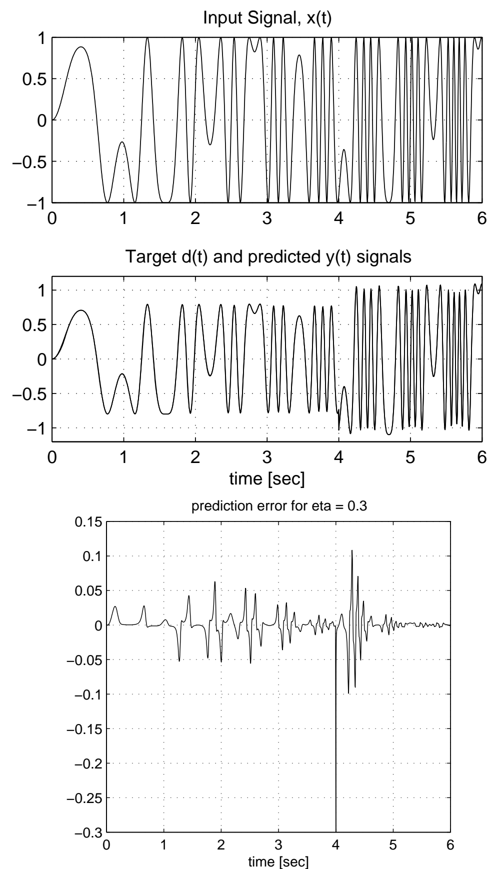
3.5.2 Adaptive System Identification

- Consider a discrete-time signal (time series), $x(n)$, which is processed by an unknown Moving-Average system.
- Such a system is an Adaline with parameters (weights) being a p-element vector b.
- It is assumed that the parameter vector is unknown.
- It is now possible to use another Adaline to observe inputs and outputs from the system and to adapt its weights using previously discussed learning algorithms so that the weight vector, **w**, approximates the unknown parameter vector, **b**: $\mathbf{w}(n) \longrightarrow \mathbf{b}$

Block diagram of the adaptive system identification:



Example – adsid.m




```
% Input signal x(t)
f = 0.8 ;      %  Hz
ts = 0.005; % 5 msec - sampling time
N1 = 800 ; N2 = 400 ; N = N1 + N2 ;
t1 = (0:N1-1)*ts ;  % 0 to 4 sec
t2 = (N1:N-1)*ts ;  % 4 to 6 sec
t = [t1 t2] ;       % 0 to 6 sec
xt = sin(3*t.*sin(2*pi*f*t)) ;

p = 3 ; % Dimensionality of the system
b1 = [ 1  -0.6 0.4]; % unknown system parameters during t1
b2 = [0.9 -0.5 0.7]; % unknown system parameters during t2
% formation of the input matrix X of size  p by N
 X = convmtx(xt, p) ; X = X(:, 1:N) ;
% The target signal
 d = [b1*X(:,1:N1) b2*X(:,N1+1:N)] ;
 y  = zeros(size(d)) ;   % memory allocation for y
eps = zeros(size(d)) ;   % memory allocation for eps
eta = 0.2  ; % learning rate/gain
w = 2*(rand(1,p)-0.5); % Initialisation of the weight vector
for n = 1:N  %  learning loop
   y(n) = w*X(:,n) ;      % predicted output signal
   eps(n) = d(n) - y(n) ; % error signal
   w = w + eta*eps(n)*X(:,n)' ;
   if n == N1-1, w1 = w ; end
end
```

Estimated system parameters:

```
b1 = 1.0000 -0.6000  0.4000    b2 = 0.9000 -0.5000  0.7000
w1 = 0.9463 -0.5375  0.3908    w2 = 0.8690 -0.4369  0.6677
```


3.5.3 Adaptive Noise Cancelation

Consider a system as in Figure 3–3:

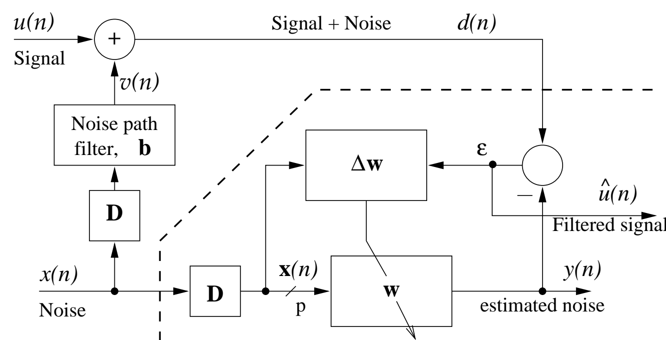


Figure 3–3: Adaptive Noise Cancelation

- A useful signal, $u(n)$, for example, voice of a pilot of an aircraft, is disturbed by a noise, $x(n)$, originated for example from an engine.
- The noise is coloured by an unknown FIR filter specified by an unknown vector $\mathbf{b}$ before it mixes with the signal. As a result, the observed signal is equal to:

$$d(n) = u(n) + v(n)$$

and the problem is to filter out the noise in order to obtain an estimate $\hat{u}(n)$ of the original signal $u(n)$.

Example – adlnc.m

With reference to Figure 3–3 we specified first the useful input signal, $u(n)$ and the noise, $x(t)$.

- The input signal is a sinusoidal signal modulated in frequency and amplitude:

$$u(t) = (1 + 0.2\sin(\omega_a)t) \cdot \sin(\omega \cdot (1 + 0.2\cos(\omega_m t) \cdot t)$$

where $\omega = 2\pi f$ is the fundamental signal frequency,
$\omega_m = 2\pi f_m$ is the frequency of the frequency modulation,
$\omega_a = 2\pi f_a$ is the frequency of the amplitude modulation, and
$t = nt_s,\ t_s$ being the sampling time.

```
f   = 4e3 ;          % 4kHz signal frequency
fm = 300 ;           % 300Hz frequency modulation
fa = 200 ;           % 200Hz amplitude modulation
ts = 2e-5 ;          % 0.2 msec sampling time
N  =  400 ;          % number of sampling points
t = (0:N-1)*ts ; %  discrete time from 0 to 8 msec
ut=(1+.2*sin(2*pi*fa*t)).*sin(2*pi*f*(1+.2*cos(2*pi*fm*t)).*t);
```

- The noise $x(n)$ is a triangular signal of frequency $f_n = 1$kHz.
- This noise is coloured by a linear FIR filter (an Adaline with fixed weights specified by a vector b).
- The resulting signal, $v(n)$, is added to the input noise signal, $x(t)$ to form the corrupted signal, $d(n)$ — see Figure 3–4


```
fn = 1e3 ;
xt = sawtooth(2*pi*1e3*t, 0.7); % the noise
b = [1  -0.6 -0.3]  ;  % noise path filter
vt = filter(b, 1, xt); % coloured noise
dt = ut + vt ;      % corrupted input signal
```

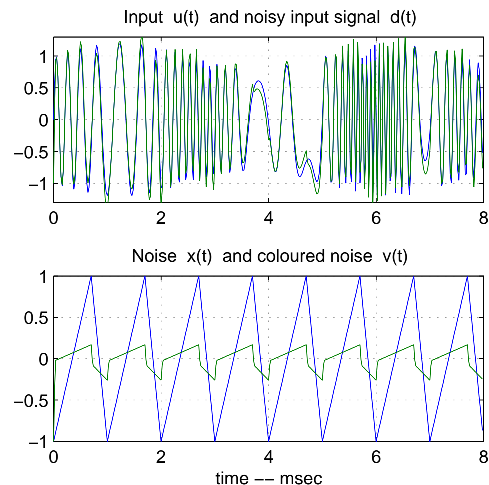


Figure 3–4: Input signal, $u(t)$, the corrupted-by-noise input signal, $d(t)$, noise, $x(t)$, and the coloured noise, $v(t)$

- It is assumed that the parameters of the noise colouring filter, b are unknown.
- The idea of noise cancellation is to estimate parameters of this noise colouring filter, thus to estimate the noise which corrupts the the input signal.
- This noise estimate, $y(n) \approx v(n)$, is available at the output of the Adaline.

- The difference between the corrupted signal, $d(n)$, and the noise estimate, $y(n)$ is the estimate of the original clean input signal:

$$\hat{u}(n) = \varepsilon(n) = d(n) - y(n) \approx u(n)$$

```
p = 4 ; % dimensionality of the input space
% formation of the input matrix X of size  p by N
X = convmtx(xt, p) ; X = X(:, 1:N) ;
y   = zeros(1,N) ;     % memory allocation for y
eps = zeros(1,N) ;     % memory allocation for uh = eps
eta = 0.03  ;          % learning rate/gain
w = 2*(rand(1,p)-0.5); % weight vector initialisation
```

- Note that the number of synapses in the adaptive Adaline, $p = 4$, is different that the order of the noise colouring filter, which is assumed to be unknown.
- Selection of the learning rate, η is very critical to good convergence of the LMS algorithm.
- In order to improve results, the learning loop which goes through all signal samples is repeated four time with diminishing values of the learning gain. Such a repetition is, of course, not possible for the real-time learning.

```
for c = 1:4
  for n = 1:N  %  learning loop
    y(n) = w*X(:,n) ;      % predicted output signal
    eps(n) = dt(n) - y(n) ; % error signal
    w = w + eta*eps(n)*X(:,n)' ;
```



```
  end
  eta = 0.8*eta ;
end
```

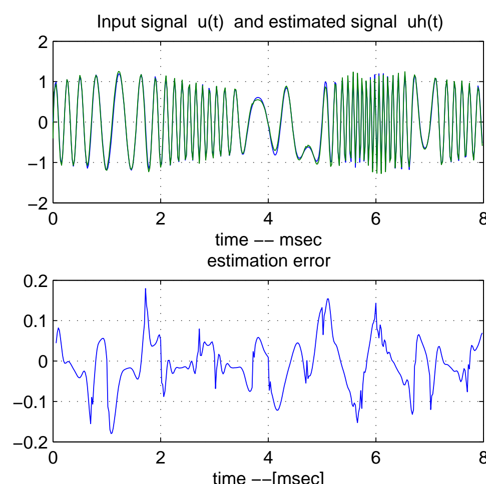


Figure 3–5: Input signal, $u(t)$, the estimated input signal, $\hat{u}(t)$, and the estimation error, $u(t) - \hat{u}(t)$,

- It can be noticed that the estimation error at each step is small, less that 4% of the signal amplitude. The resulting weight vector

```
w = 0.7933 -0.0646 -0.7231 0.0714
```

is similar to the parameters of the noise colouring filter, b.