

## 7 Advanced Learning Algorithms for Multilayer Perceptrons

The basic back-propagation learning law is a gradient-descent algorithm based on the estimation of the gradient of the instantaneous sum-squared error for each layer:

$$\Delta W(n) = -\eta \cdot \nabla_W E(n) = \eta \cdot \delta(n) \cdot \mathbf{x}^T(n) \quad (7.1)$$

Such an algorithm is slow for three basic reasons:

- It uses an instantaneous sum-squared error  $E(W, n)$  to minimise the mean squared error,  $J(W)$ , over the training epoch. The gradient of the instantaneous sum-squared error is not a good estimate of the gradient of the mean squared error. Therefore, satisfactory minimisation of this error typically requires many repetitions of the training epochs.
- It is a first-order minimisation algorithm which is based on the first-order derivatives (a gradient). Faster algorithms utilise also the second derivatives (the Hessian matrix)
- The error propagation, which is conceptually very interesting, serialises computations on the layer by layer basis.

The mean squared error,  $J(W)$ , is a relatively complex surface in the weight space, possibly with many local minima, flat sections, narrow irregular valleys, and saddle points.

Example of such a surface is given in Figure 7–1

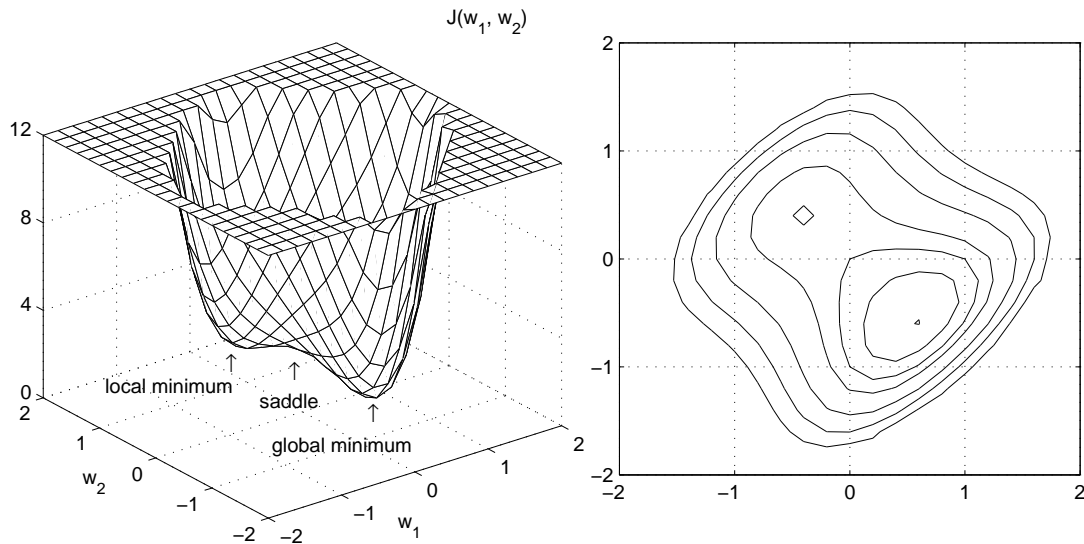


Figure 7–1: A 2-dimensional mean-squared error surface,  $J(W)$

Complexity of the error surface is the main reason that behaviour of a simple steepest descend minimisation algorithm can be very complex often with oscillations around a local minimum.

We first consider improvements to the basic back-propagation algorithms based on heuristic methods. These methods do not directly addresses the inherent weaknesses of the back-propagation algorithm, but aim at improvement of the behaviour of the algorithm by making modifications to its parameters, or to the form.

## 7.1 Heuristic Improvements to the Back-Propagation Algorithm

### The momentum term

One of the simple method to avoid an error trajectory in the weight space being oscillatory is to add to the weight update a momentum term. Such a term is proportional to the weight update at the previous step.

$$\Delta W(n) = \eta \cdot \delta(n) \cdot \mathbf{x}^T(n) + \alpha \cdot \Delta W(n-1), \quad 0 < \alpha < 1 \quad (7.2)$$

where  $\alpha$  is a momentum term parameter.

Such modification to the steepest descend learning law acts as a low-pass filter smoothing the error trajectory. As a result it is possible to apply higher learning rate,  $\eta$ .

## Adaptive learning rate

One of the ways of increasing the convergence speed, that is, to move faster downhill to the minimum of the mean-squared error,  $J(W)$ , is to vary adaptively the learning rate parameter,  $\eta$ .

A typical strategy is based on monitoring the rate of change of the mean-squared error and can be described as follows:

- If  $J$  is decreasing consistently, that is,  $\nabla J$  is negative for a prescribed number of steps, then the learning rate is increased linearly:

$$\eta(n+1) = \eta(n) + a, \quad a > 0 \quad (7.3)$$

- If the error has increased, ( $\nabla J > 0$ ), the learning rate is exponentially reduced:

$$\eta(n+1) = b \cdot \eta(n), \quad 0 < b < 1 \quad (7.4)$$

In general, increasing the value of the learning rate the learning tends to become unstable which is indicated by an increase in the value of the error function. Therefore it is important to quickly reduce  $\eta$ .

### Efficient initialization — nwini.m

Efficient initialization can speed up the convergence process significantly, even by the order of magnitude. A popular initialization algorithm developed by Nguyen and Widrow and used in the MATLAB Neural Network Toolbox is presented below.

Let us consider a single layer of  $m$  neurons with  $p$  synapses, each including the bias. Then for  $j$ th neuron we have

$$y_j = \varphi(v_j) , \quad \text{where} \quad v_j = \mathbf{w}_j \cdot \mathbf{x} , \quad x_p = 1$$

For an activation function  $\varphi(v)$  we can specify its active region  $\bar{v} = [v_{min} \ v_{max}]$  outside which the function is in saturation. For example, for the hyperbolic tangent we can assume

$$\bar{v} = [-2 \ +2] , \quad \text{then} \quad \tanh(v) \in [-0.96 \ 0.96]$$

In addition we need to specify the range of input signals,

$$\bar{\mathbf{x}}_i = [x_{i,min} \ x_{i,max}] \quad \text{for} \quad i = 1 \dots p - 1$$

**Unified range:** Assume first that the range of input signals and non-saturating activation potential is  $[-1 \ +1]$ . The initial weight vectors will now have evenly distributed magnitudes and random directions:

For  $p = 2$  the initialisation is as follows

- generate  $m$  random numbers  $a_j \in (-1, +1)$  for  $j = 1, \dots, m$
- Set up weights as follow

$$W(j, 1) = 0.7 \frac{a_j}{|a_j|} , \quad W(:, 2) = 0$$

For  $p > 2$  the initialisation is as follows

- Specify the magnitude of the weight vectors as

$$\bar{W} = 0.7 m^{\frac{1}{p-1}}$$

- generate  $m$  random unity vectors,  $\mathbf{a}_j$ , that is, generate an  $m \times (p-1)$  array  $A$  of random numbers,  $a_{ji} \in (-1, +1)$  and normalise it in rows:

$$\mathbf{a}_j = \frac{A(j, :)}{\|A(j, :)\|}$$

- Set up weights as follow

$$W(j, 1 : p-1) = \bar{W} \cdot \mathbf{a}_j \quad \text{for } j = 1, \dots, m$$

and the bias weights

$$W(j, p) = \text{sgn}(W(j, 1)) \cdot \bar{W} \cdot \beta_j \quad \text{for } \beta_j = -1 : \frac{2}{m-1} : 1$$

Finally, the weights are linearly rescaled to account for different range of activation potentials and input signals. Details can be found in the MATLAB script, `nwini.m`.

```

function w = nwini(xr, m, vr)
%
%nwini Calculates Nugyen-Widrow initial conditions.
% adapted from NNet toolbox      8 Axril 1999
% xr - p-1 by 2 matrix of [xmin xmax]
% assumes that the bias is added
% m - Number of neurons.
% vr - Active region of the transfer function
% vr = [Vmin Vmax].
% e.g. vr = [-2 -2] for tansig , [-4 4] for logsig
% w is m by p

r = size(xr,1); p = r+1 ;

% Null case
if (r == 0) | (m == 0)
    w = zeros(m,p) ;
    return
end

% Remove constant inputs that provide no useful info
R = r;
ind = find(xr(:,1) ~= xr(:,2));
r = length(ind);
xr = xr(ind,:);

% Nguyen-Widrow Method
% Assume inputs and activation potentials range in [-1 1].

% Weights
wMag = 0.7*m^(1/r); % weight vectors magnitude
% weight vectors directions: wDir are row unity vectors
a = 2*rand(m,r)-1 ;
if r == 1
    b = ones./abs(a);
else
    b=sqrt(ones./(sum((a.*a)'))');
end
wDir=b(:,ones(1,r)).*a;
w = wMag*wDir;

```

```
% Biases
if (m==1)
    wb = 0;
else
    wb = wMag*[2*(0:m-2)/(m-1)-1 1]'.*sign(w(:,1));
end

% Conversion of activation potentials of [-1 1] to [Nmin Nmax]
a1 = 0.5*(vr(2)-vr(1));
a2 = 0.5*(vr(2)+vr(1));
w = a1*w;
wb = a1*wb+a2;

% Conversion of inputs of xr to [-1 1]
a1 = 2./(xr(:,2)-xr(:,1));
a2 = 1-xr(:,2).*a1;

ap = a1';
wb = w*a2+wb;
w = w.*ap(ones(1,m),:);

% Replace constant inputs
ww = w;
w = zeros(m,R);
w(:,ind) = ww;
% combine with biasing weights
w = [w wb] ;
```

## 7.2 Some theoretical aspects of optimisation

The problem of minimisation of a function of many variables (multi-variable function),  $J(\mathbf{w})$ , has been researched since the XVII century and its principles were formulated by people as Kepler, Fermat, Newton, Leibnitz, Gauss.

**Taylor series expansion:** Consider values of the weight vector at two consecutive steps:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n) \quad (7.5)$$

which will be, for brevity, written as

$$\mathbf{w}_{n+1} = \mathbf{w} + \Delta\mathbf{w} \quad (\text{for consistence, these are row vectors}) \quad (7.6)$$

and associated values of the performance index,  $J(\mathbf{w}_{n+1})$  and  $J(\mathbf{w})$ . A good way to compare these two values is to apply the Taylor series expansion:

$$J(\mathbf{w}_{n+1}) = J(\mathbf{w}) + \Delta\mathbf{w} \cdot \nabla J(\mathbf{w}) + \frac{1}{2} \Delta\mathbf{w} \cdot \nabla^2 J(\mathbf{w}) \cdot \Delta\mathbf{w}^T + \dots \quad (7.7)$$

where

$\nabla J(\mathbf{w}) = \mathbf{g}$  is the gradient vector of the performance index,

$\nabla^2 J(\mathbf{w}) = H$  is the Hessian matrix (matrix of second derivatives).

Using this notation eqn (7.7) can be re-written as:

$$J(\mathbf{w}_{n+1}) = J(\mathbf{w}) + \Delta\mathbf{w} \cdot \mathbf{g}(\mathbf{w}) + \frac{1}{2} \Delta\mathbf{w} \cdot H(\mathbf{w}) \cdot \Delta\mathbf{w}^T + \dots \quad (7.8)$$

**Directional derivatives:** If  $\mathbf{p}$  is a unit vector, then derivatives in the direction of  $\mathbf{p}$  can be evaluated as:

$$\text{the first derivative:} \quad \nabla J(\mathbf{w}) \cdot \mathbf{p} = \mathbf{g} \cdot \mathbf{p}$$

$$\text{the second derivative:} \quad \mathbf{p}^T \cdot \nabla^2 J(\mathbf{w}) \cdot \mathbf{p} = \mathbf{p}^T \cdot H \cdot \mathbf{p}$$

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

$$J(\mathbf{w}(n+1)) < J(\mathbf{w}(n))$$

If we limit the Taylor expansion to the first-order term then we have

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

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

$$\Delta \mathbf{w} = -\eta \nabla J(\mathbf{w}) \tag{7.9}$$

which is the steepest descent condition that we have already employed.

**Optimal learning rate. Line minimisation** We will try to estimate an optimal value of the learning rate  $\eta$ .

If we combine eqns (7.5) and (7.9), we can write a steepest descent minimisation step in the form:

$$\mathbf{w}_{n+1} = \mathbf{w} - \eta \mathbf{g}, \quad \text{where } \mathbf{g} = \nabla J(\mathbf{w}) \quad (7.10)$$

is the gradient vector. Now, we can select the value of  $\eta$  which minimises the performance index,  $J(\mathbf{w}_{n+1})$ .

For varying  $\eta$ , eqn (7.10) describes a line in the weight space, therefore, we would minimise the performance index along this line:

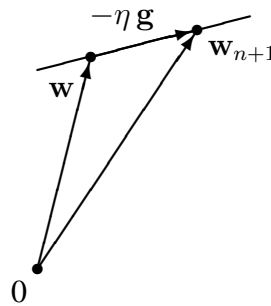


Figure 7-2: Minimisation along the line  $\mathbf{w}_{n+1} = \mathbf{w} - \eta \mathbf{g}$

The problem of line minimisation will be considered in the next section. Here, as an illustration, we will consider a problem of finding a maximum value of  $\eta$  for which the process of **learning is stable**, that is,  $J(\mathbf{w})$  converges to a minimum value.

### 7.3 Stable learning rates

Assume for simplicity that the performance index is a quadratic function of weights, that is:

$$J(\mathbf{w}) = \frac{1}{2} \mathbf{w} \cdot A \cdot \mathbf{w}^T + \mathbf{w} \cdot \mathbf{b} + c \quad (7.11)$$

The gradient of a quadratic function is:

$$\nabla J(\mathbf{w}) = \mathbf{w} \cdot A + \mathbf{b}^T \quad (7.12)$$

and its Hessian matrix:

$$\nabla^2 J(\mathbf{w}) = A \quad (7.13)$$

Eqn (7.7) for steepest descent can now be written as:

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta(\mathbf{w}(n) \cdot A + \mathbf{b}^T)$$

or

$$\mathbf{w}(n+1) = \mathbf{w}(n)(I - \eta A) - \eta \mathbf{b}^T \quad (7.14)$$

This recursive equation describes a linear dynamic system. It is known that such a system is stable, that is, the weight vector converges to a fix value, when all eigenvalues of the matrix  $(I - \eta A)$  are inside a unit circle, that is

$$|\text{eig}(I - \eta A)| < 1 \quad (7.15)$$

If  $(\lambda_i, \mathbf{v}_i)$  and  $(\bar{\lambda}_i, \bar{\mathbf{v}}_i)$  denote pairs of eigenvalues and eigenvectors of the matrices  $A$  and  $(I - \eta A)$ , respectively, then we can write:

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i, \text{ and } (I - \eta A)\bar{\mathbf{v}}_i = \bar{\lambda}_i \bar{\mathbf{v}}_i \quad (7.16)$$

The second equation can be re-written as

$$A\bar{\mathbf{v}}_i = \frac{1 - \bar{\lambda}_i}{\eta} \bar{\mathbf{v}}_i \quad (7.17)$$

Hence, due to uniqueness of the eigenvalue decomposition, we infer that  $\mathbf{v}_i = \bar{\mathbf{v}}_i$ , and

$$\lambda_i = \frac{1 - \bar{\lambda}_i}{\eta}, \text{ or } \bar{\lambda}_i = 1 - \eta \lambda_i \quad (7.18)$$

Condition (7.15) can be now written in terms of eigenvalues of the Hessian matrix  $A$  as

$$-1 < 1 - \eta \lambda_i < 1, \text{ for all } i = 1, \dots, p$$

where we assumed that the Hessian matrix  $A$  is positive definite, that is, its eigenvalues are real and positive, which is the case for the mean-squared error,  $J(\mathbf{w})$ .

Therefore, finally, the condition for the **maximum stable learning rate** can be written as:

$$\eta < \frac{2}{\lambda_{\max}} \quad (7.19)$$

The largest eigenvalue,  $\lambda_{\max}$  represents the maximum curvature of the quadratic function, and the condition (7.19) states that the maximum stable learning rate is inversely proportional to this curvature.

## 7.4 Line search minimisation procedures

Let us consider a problem of minimisation of the performance index,  $J(\mathbf{w})$ , along the line similar to that in Figure 7–2:

$$\mathbf{w}_{n+1} = \mathbf{w} + \eta \mathbf{p} \quad (7.20)$$

where the next value of weight vector,  $\mathbf{w}_{n+1}$ , is obtained from the current value of weight vector,  $\mathbf{w}$ , by moving it in the direction of a vector,  $\mathbf{p}$ . This vector and an associated parameter  $\eta$  are to be selected so that the performance index

$$J(\mathbf{w}_{n+1}) = J(\mathbf{w} + \eta \mathbf{p}) \quad (7.21)$$

is minimised.

In order to find an optimal value of the parameter  $\eta$ , which, in particular, can be identified with the learning rate as in eqn (7.10), we calculate the following partial derivative:

$$\frac{\partial J(\mathbf{w}_{n+1})}{\partial \eta} = \frac{\partial J(\mathbf{w}_{n+1})}{\partial \mathbf{w}_{n+1}} \eta \mathbf{p}^T = \nabla J(\mathbf{w}_{n+1}) \eta \mathbf{p}^T \quad (7.22)$$

For the optimal value of  $\eta$ , this derivative is zero and we have the following relationship

$$\nabla J(\mathbf{w}_{n+1}) \cdot \mathbf{p}^T = \mathbf{g}_{n+1} \cdot \mathbf{p}^T = 0 \quad (7.23)$$

It states that the next estimate of the gradient,  $\mathbf{g}_{n+1}$ , is orthogonal to the current search direction,  $\mathbf{p}$ .

This means, in particular, that if we combine the **line minimisation** technique with the **steepest descent** algorithm when we move in the direction opposite to the gradient,  $\mathbf{p} = -\mathbf{g}$ , then we will be descending along the zig-zag line, each segment being orthogonal to the next one.

In order to smooth the descend direction, the steepest-descent technique is replaced with the conjugate gradient algorithm.

## Conjugate Gradient Algorithm

In this method, in order to avoid the zig-zag movement, the next search direction,  $\mathbf{p}_{n+1} = \mathbf{p}(n+1)$ , instead of being exactly orthogonal to the gradient, tries to maintain the current search direction,  $\mathbf{p}(n)$ , namely

$$\mathbf{p}(n+1) = -\mathbf{g}(n) + \beta(n)\mathbf{p}(n) , \text{ or } \mathbf{p}_{n+1} = -\mathbf{g} + \beta \mathbf{p} \quad (7.24)$$

where scalars  $\beta(n)$  are selected so that the directions  $\mathbf{p}(n+1)$  and  $\mathbf{p}(n)$  are conjugate with respect to the Hessian matrix,  $\nabla^2 J(\mathbf{w}) = H$ , that is,

$$\mathbf{p}(n+1) \cdot H \cdot \mathbf{p}^T(n) = 0 \quad (7.25)$$

In practice, the Hessian matrix is not being calculated and the following three choices of  $\beta(n)$  are the most commonly used

- Hestenes-Steifel formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{p}^T(n-1)} \quad (7.26)$$

- Fletcher-Reeves formula

$$\beta(n) = \frac{\mathbf{g}(n) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \quad (7.27)$$

- Polak-Ribière formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \quad (7.28)$$

In summary, the conjugate gradient involves: Initial search direction,  $\mathbf{p}(0) = -\mathbf{g}(0)$ , line minimisation with respect of  $\eta$ , calculation of the next search direction as in eqn (7.24), and  $\beta$  from one of the above formulae.

## 7.5 Newton's Methods

In Newton's methods minimisation is based on utilisation of the second derivatives of the performance index. Consider the Taylor series expansion of the performance index as in eqn (7.7) which can be re-written as

$$J(\mathbf{w}_{n+1}) = J(\mathbf{w}) + \Delta\mathbf{w} \cdot \nabla J + \frac{1}{2} \Delta\mathbf{w} \cdot \mathbf{H} \cdot \Delta\mathbf{w}^T + \dots$$

To minimise  $J(\mathbf{w}_{n+1})$  we calculate the gradient and equate it to zero

$$\nabla J(\mathbf{w}_{n+1}) = \nabla J + \Delta\mathbf{w} \cdot \mathbf{H} + \dots = 0$$

Neglecting the higher order expansion terms, we have

$$\Delta\mathbf{w} = -\nabla J \cdot \mathbf{H}^{-1} \quad (7.29)$$

This equation says that a better weight update is the direction opposite to the gradient modified (rotated) by the inverse of the Hessian matrix of the performance index  $J$ .

The Hessian matrix provides additional information about the shape of the performance index surface in the neighbourhood of  $\mathbf{w}(n)$ .

The Newton's method is typically faster than conjugate gradient algorithms. However, it requires computations of the inverse of the Hessian matrix which is relatively complex.

Many specific algorithms originate from the Newton's method, the fastest and most popular being the Levenberg-Marquardt algorithm, which originate from the Gauss-Newton method.

The Newton's methods use the batch training mode, rather than the pattern mode which is based on derivatives of instantaneous errors.

## Gauss-Newton method

Let us assume for that

- all weights have been arranged in one row vector

$$\mathbf{w} = [w_1 \dots w_j \dots w_K]$$

- all (instantaneous) errors form a column vector

$$\boldsymbol{\varepsilon}(\mathbf{w}, n) = \mathbf{d}(n) - \mathbf{y}(n) = [\varepsilon_1 \dots \varepsilon_k \dots \varepsilon_m]^T$$

- the instantaneous performance index  $E(\mathbf{w}, n)$  is a sum of squares of errors

$$E(\mathbf{w}, n) = \frac{1}{2} \sum_{k=1}^m \varepsilon_k^2(n) = \frac{1}{2} \boldsymbol{\varepsilon}(n) \cdot \boldsymbol{\varepsilon}^T(n)$$

(for brevity, arguments like  $\mathbf{w}$  and  $n$  are often omitted)

- The total performance index (mean squared error)

$$F(\mathbf{w}) = \frac{1}{M} \sum_{n=1}^N E(\mathbf{w}, n)$$

where  $M = m N$ . The symbol  $F$  is used in place of  $J$  to avoid confusion with the Jacobian matrix.

We consider first derivatives of instantaneous errors. The  $j$ th element of the instantaneous gradient vector can now be expressed as

$$\begin{aligned} [\nabla E(\mathbf{w}, n)]_j &= \frac{\partial E(\mathbf{w}, n)}{\partial w_j} = \sum_{k=1}^m \varepsilon_k(\mathbf{w}) \frac{\partial \varepsilon_k(\mathbf{w})}{\partial w_j} \\ &= \boldsymbol{\varepsilon}^T(\mathbf{w}) \left[ \frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_j} \dots \frac{\partial \varepsilon_M(\mathbf{w})}{\partial w_j} \right]^T \end{aligned}$$

This expression can be generalised into a matrix form for the gradient:

$$\nabla E(\mathbf{w}, n) = \boldsymbol{\varepsilon}^T(\mathbf{w}, n) \mathcal{J}(\mathbf{w}, n) \quad (7.30)$$

where

$$\mathcal{J}(\mathbf{w}, n) = \begin{bmatrix} \frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_1} & \cdots & \frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_K} \\ \vdots & & \vdots \\ \frac{\partial \varepsilon_m(\mathbf{w})}{\partial w_1} & \cdots & \frac{\partial \varepsilon_m(\mathbf{w})}{\partial w_K} \end{bmatrix} \quad (7.31)$$

is the  $m \times K$  matrix of first derivatives known as the **Jacobian** matrix.

In order to find the Hessian matrix of the instantaneous performance index we differentiate eqn (7.30):

$$\nabla^2 E(\mathbf{w}, n) = \frac{\partial(\boldsymbol{\varepsilon}^T(\mathbf{w}) \mathcal{J}(\mathbf{w}))}{\partial \mathbf{w}}$$

Let us calculate first, for simplicity, the  $k, j$  element of the Hessian matrix

$$\begin{aligned} [\nabla^2 E(\mathbf{w}, n)]_{k,j} &= \frac{\partial^2 E(\mathbf{w})}{\partial w_k \partial w_j} = \sum_{i=1}^M \left( \frac{\partial \varepsilon_i}{\partial w_k} \frac{\partial \varepsilon_i}{\partial w_j} + \varepsilon_i \frac{\partial^2 \varepsilon_i}{\partial w_k \partial w_j} \right) \\ &= \begin{bmatrix} \frac{\partial \varepsilon_1}{\partial w_k} & \cdots & \frac{\partial \varepsilon_M}{\partial w_k} \end{bmatrix} \begin{bmatrix} \frac{\partial \varepsilon_1}{\partial w_j} \\ \vdots \\ \frac{\partial \varepsilon_M}{\partial w_j} \end{bmatrix} + \boldsymbol{\varepsilon}^T \frac{\partial^2 \varepsilon_i}{\partial w_k \partial w_j} \end{aligned}$$

Generalising the above expression into a matrix form, we obtain:

$$\nabla^2 E(\mathbf{w}) = \mathcal{J}^T(\mathbf{w})\mathcal{J}(\mathbf{w}) + \boldsymbol{\varepsilon}^T(\mathbf{w})R(\mathbf{w}) \quad (7.32)$$

where

$$R(\mathbf{w}) = \left\{ \frac{\partial^2 \varepsilon_i}{\partial w_k \partial w_j} \right\}$$

it the matrix of all second derivatives of errors.

**Gauss-Newton method:** If we neglect the term

$$\boldsymbol{\varepsilon}^T(\mathbf{w})R(\mathbf{w})$$

due to the fact that errors are small, then we obtain the Gauss-Newton method.

In this method the Hessian matrix is approximated as:

$$H(\mathbf{w}, n) = \nabla^2 E(\mathbf{w}, n) \approx \mathcal{J}^T(\mathbf{w}, n)\mathcal{J}(\mathbf{w}, n)$$

and the weight update equation (7.29) becomes:

$$\Delta \mathbf{w}(n) = -\nabla E(n)H^{-1}(n) = -\boldsymbol{\varepsilon}^T(n)\mathcal{J}(n) \left( \mathcal{J}^T(n)\mathcal{J}(n) \right)^{-1} \quad (7.33)$$

For simplicity, we have considered the pattern update, however, in the following section we consider a modification of the Gauss-Newton algorithm in which the batch update of weights is employed.

### Levenberg-Marquardt algorithm

One problem with the Gauss-Newton method is that the approximated Hessian matrix may not be invertible. To overcome this problem in the **Levenberg-Marquardt** algorithm a small constant  $\mu$  is added such that

$$\mathbf{H}(\mathbf{w}) = \mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + \mu I$$

where  $\mathbf{H}(\mathbf{w})$  and  $\mathbf{J}(\mathbf{w})$  are the batch Hessian and Jacobian matrices, respectively,  $I$  is the identity matrix and  $\mu$  is a small constant.

The gradient of the batch performance index,  $F(\mathbf{w})$ , can be calculated as

$$\nabla F(\mathbf{w}) = \sum_{n=1}^N \nabla E(\mathbf{w}, n) = \sum_{n=1}^N \boldsymbol{\varepsilon}^T(\mathbf{w}, n) \cdot \mathcal{J}(\mathbf{w}, n) = \mathbf{e}^T(\mathbf{w})\mathbf{J}(\mathbf{w})$$

where

$$\mathbf{e}^T = \text{scan}(D - Y) = [\boldsymbol{\varepsilon}^T(1) \dots \boldsymbol{\varepsilon}^T(N)]$$

is the vector of all instantaneous errors. The batch Jacobian matrix,  $\mathbf{J}(\mathbf{w})$ , is a block-column matrix consisting of the instantaneous Jacobian matrices,  $\mathcal{J}(n)$

$$\mathbf{J}(\mathbf{w}) = \begin{bmatrix} \mathcal{J}(1) \\ \vdots \\ \mathcal{J}(N) \end{bmatrix}$$

As a result, the batch weight update is as in eqn (7.34):

$$\Delta \mathbf{w} = -\nabla F \cdot \mathbf{H}^{-1} = -\mathbf{e}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) (\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + \mu I)^{-1} \quad (7.34)$$

**Calculation of the Jacobian matrix** is similar to calculation of the gradient of the performance index. The main difference is that in the case of the gradient we differentiate the sum of squared errors, whereas in the case of the Jacobian we differentiate errors themselves, see eqn (7.31).

Following the derivation of the basic backpropagation algorithm we consider a **two-layer perceptron** with two weight matrices,  $W^h, W^y$ . The weight vector  $\mathbf{w}$  is formed by scanning these matrices in rows, so that we have:

$$\mathbf{w} = \text{scan}(W^h, W^y) = [w_{11}^h \dots w_{1p}^h \dots w_{Lp}^h | w_{11}^y \dots w_{1L}^y \dots w_{mL}^y]$$

The length of  $\mathbf{w}$  is  $K = L(p + m)$ .

The instantaneous Jacobian matrix,  $\mathcal{J}(n)$ , is  $m \times K$ , one column per weight, and can be partitioned into two blocks related to the hidden and output weights, respectively:

$$\mathcal{J}(n) = [\mathcal{J}^h(n) \quad \mathcal{J}^y(n)]$$

## Output layer

The output Jacobian matrix  $\mathcal{J}^y(n)$  is  $m \times mL$ , where  $m$  is the number of output neurons and  $L$  is the number of hidden signals,  $h_j$ , as in Figure 5–4. The elements of  $\mathcal{J}^y(n)$  are the first derivatives

$\frac{\partial \varepsilon_i}{\partial w_{kj}^y}$  of errors  $\varepsilon_i(n)$  with respect to weights  $w_{kj}^y$ . We have

$$\varepsilon_i(n) = d_i(n) - y_i(n) , \quad y_i(n) = \varphi(v_i(n)) , \quad v_i(n) = W_{i:}^y \cdot \mathbf{h}(n)$$

Now, an element of the output Jacobian matrix can be calculate in the following way

$$\frac{\partial \varepsilon_i}{\partial w_{kj}^y} = \begin{cases} 0 & \text{if } i \neq k \text{ (error is local to the } k\text{th neuron)} \\ -\frac{\partial y_k}{\partial w_{kj}^y} & \text{if } i = k \end{cases}$$

Hence, the output Jacobian matrix has a block-diagonal structure.

Subsequently, we have

$$\frac{\partial y_k}{\partial w_{kj}^y} = \frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{kj}^y} = \varphi'_k \cdot h_j , \quad \text{where } \varphi'_k = \frac{\partial y_k}{\partial v_k}$$

which can be generalised to the matrix of non-zero blocks of  $\mathcal{J}^y$  as

$$\frac{\partial \varepsilon_k}{\partial W_{k:}^y} = -\varphi'_k \cdot \mathbf{h}^T , \quad P = \frac{\partial \varepsilon_k}{\partial W^y} = -\boldsymbol{\varphi}' \cdot \mathbf{h}^T$$

Rows of the matrix  $P$  form the diagonal blocks of the Jacobian  $\mathcal{J}^y$ .

More formally, we can write

$$\mathcal{J}^y = -\text{diag}(\boldsymbol{\varphi}') \otimes \mathbf{h}^T \tag{7.35}$$

where  $\otimes$  denotes the Kronecker product.

## Hidden layer

The hidden Jacobian matrix  $\mathcal{J}^h(n)$  is  $m \times mp$ , where  $m$  is the number of output neurons and  $p$  is the number of input signals,  $x_i(n)$ .

An element of the hidden Jacobian matrix can be calculate in the following way

$$\frac{\partial \varepsilon_k}{\partial w_{ji}^h} = -\frac{\partial y_k}{\partial w_{ji}^y} = -\frac{\partial y_k}{\partial v_k} \frac{\partial v_k}{\partial w_{ji}^h} = -\varphi'_k \frac{\partial v_k}{\partial w_{ji}^h}$$

If we take into account that

$$v_k = W_{k:}^y \cdot \mathbf{h} = \dots + w_{kj}^y \cdot h_j + \dots$$

and

$$h_j = \psi(W_{j:}^h \cdot \mathbf{x}) = \psi(\dots + w_{ji}^h \cdot x_i + \dots)$$

then we can arrive at the final form for a single element of the hidden Jacobian matrix:

$$[\mathcal{J}^h]_{k,ji} = \frac{\partial \varepsilon_k}{\partial w_{ji}^h} = -\varphi'_k \cdot w_{kj}^y \cdot \frac{\partial h_j}{\partial w_{ji}^h} = -\varphi'_k \cdot w_{kj}^y \cdot \psi'_j \cdot x_i \quad (7.36)$$

A  $1 \times p$  block of the hidden Jacobian matrix can be expressed as follows

$$[\mathcal{J}^h]_{k,j:} = -s_{kj} \cdot \mathbf{x}^T, \quad \text{where } s_{kj} = \varphi'_k \cdot w_{kj}^y \cdot \psi'_j$$

and finally, we have

$$\mathcal{J}^h = -S^h \otimes \mathbf{x}^T, \quad \text{where } S^h = \text{diag}(\boldsymbol{\varphi}') \cdot W^y \cdot \text{diag}(\boldsymbol{\psi}') \quad (7.37)$$

### The complete algorithm — general description

The complete Levenberg-Marquardt algorithm can be described as follows:

1. For the input matrix,  $X$ , calculate the matrices of:
  - hidden signals,  $H$ ,
  - output signals,  $Y$ ,
  - related derivatives,  $\Psi'$ , and  $\Phi'$ ,
  - errors,  $D - Y$ , and  $e$ .
2. Calculate instantaneous Jacobian matrices,  $\mathcal{J}^h$  and  $\mathcal{J}^y$  as in eqns (7.37) and (7.35), and arrange them in the batch Jacobian  $\mathbf{J}$ .
3. Calculate the weight update,  $\Delta \mathbf{w}$ , according to eqn (7.37) for a selected value of  $\mu$ .
4. Calculate the batch performance index,  $F(\mathbf{w} + \Delta \mathbf{w})$ , and compare it with the previous value,  $F(\mathbf{w})$ .  
 If  $F(\mathbf{w} + \Delta \mathbf{w}) > F(\mathbf{w})$ , reduce the value of the parameter  $\mu$  and recalculate  $\Delta \mathbf{w}$  (step 3), until  $F(\mathbf{w} + \Delta \mathbf{w}) > F(\mathbf{w})$  is reduced.
5. Repeat calculations from step 1 for the updated weights,  $\mathbf{w} = \mathbf{w} + \Delta \mathbf{w}$ .

### Some computational details

$\mathbf{J}^T \mathbf{J}$ :

The size of  $\mathbf{J}$  is  $M \times K = mN \times L(p + m)$ , that is, (size of the data set)  $\times$  (set of the weight set) which might be prohibitively large for a big data set. In order to reduce the size of the intermediate data, we can proceed in the following way:

$$\mathbf{J}^T \mathbf{J} = [\mathcal{J}^T(1) \dots \mathcal{J}^T(N)] \begin{bmatrix} \mathcal{J}(1) \\ \vdots \\ \mathcal{J}(N) \end{bmatrix} = \sum_{n=1}^N \mathcal{J}^T(n) \mathcal{J}(n)$$

The size of the matrix to be inverted,  $(\mathbf{J}^T \mathbf{J} + \mu I)$  is  $K \times K$ .

$\mathbf{e}^T \mathbf{J}$ :

Similarly, we can calculate the above gradient as

$$\nabla F = \mathbf{e}^T \mathbf{J} = \sum_{n=1}^N \boldsymbol{\varepsilon}^T(n) \mathcal{J}(n)$$

In this way, we need not store the complete batch Jacobian,  $\mathbf{J}$ .

## 7.6 Neural Network Toolbox – Multilayer perceptron

This section is an extract from *Neural Network Toolbox* that is available in the pdf format in

`$MATLAB/help/pdf_doc/nnet/nnet.pdf`

### Creating a Network (newff)

The first step in training a feedforward network is to create the network object. The function **newff** creates a trainable feedforward network. It requires four inputs and returns the network object:

1. The first input is an  $(p - 1) \times 2$  matrix  $X_R$  of minimum and maximum values for each of the  $(p - 1)$  elements of the input vector.
2. The second input is an array containing the sizes of each layer.
3. The third input is a cell array containing the names of the transfer functions to be used in each layer.
4. The final input contains the name of the training function to be used.

For example, the following command will create a two-layer network. There will be one input vector with two elements ( $p - 1 = 2$ ), three neurons in the first layer ( $L - 1 = 3$ ) and one neuron in the second (output) layer ( $m = 1$ ).

The transfer function in the first layer will be tan-sigmoid, and the output layer transfer function will be linear. The values for the first element of the input vector will range between -1 and 2, the values of the second element of the input vector will range between 0 and 5, that is, the matrix  $X_R$  is of the following form:

$$X_R = [-1 \ 2; \ 0 \ 5];$$

and the training function will be **traingd** (which will be described in a later section).

```
net=newff(XR,[L-1,m], 'tansig', 'purelin', 'traingd');
```

This command **creates** the network object and also **initializes** the weights and biases of the network, using the default Nguyen-Widrow algorithm (**initnw**). Therefore the network is ready for training.

### Initializing Weights (**init**, **initnw**, **rands**)

If you need re-initialisation, or change to the default initialisation algorithm you use the command **init**:

```
net=init(net);
```

This function takes a network object as input and returns a network object with all weights and biases initialized. This function is invoked by the **newff** command and uses the Nguyen-Widrow algorithm.

If, for example, we would like to re-initialise the weights and biases in the first layer to random values using the **rands** function, we would issue the following commands:

```
net.layers{1}.initFcn = 'initwb';  
net.inputWeights{1,1}.initFcn = 'rands';  
net.biases{1,1}.initFcn = 'rands';  
net.biases{2,1}.initFcn = 'rands';  
net = init(net);
```

## Simulation (**sim**)

The function **sim** simulates a network. The function **sim** takes the network input  $X$ , and the network object `net`, and returns the network outputs  $Y$ .

Here is how **simuff** can be used to simulate the network we created above for a single input vector:

```
x = [1;2];  
a = sim(net,x)  
a =  
    -0.1011
```

(If you try these commands, your output may be different, depending on the state of your random number generator when the network was initialized.)

Below, **sim** is called to calculate the outputs for a concurrent set of three input vectors.

```
X = [1 3 2;2 4 1];  
Y = sim(net,X)  
Y =  
    -0.1011    -0.2308     0.4955
```

## Incremental Training (**adapt**)

The function **adapt** is used to train networks in the incremental (pattern) mode. This function takes the network object and the inputs and the targets from the training set, and returns the trained network object and the outputs and errors of the network for the final weights and biases. There are two basic functions available for the incremental training: **leardgd** and **leardgdm**. Refer to the manual for details.

## Batch Training (**train**)

The alternative to incremental training is batch training, which is invoked using the function **train**. In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network.

Some of the functions available for the batch training are listed in the following table together with a relative time to reach convergence.

Function	Technique	Time	Epochs	Mflops
traingdx	Variable Learning Rate	57.71	980	2.50
trainrp	Rprop	12.95	185	0.56
trainscg	Scaled Conj. Grad.	16.06	106	0.70
traingcf	Fletcher-Powell CG	16.40	81	0.99
traingcp	Polak-Ribire CG	19.16	89	0.75
traingcb	Powell-Beale CG	15.03	74	0.59
trainoss	One-Step-Secant	18.46	101	0.75
trainbfg	BFGS quasi-Newton	10.86	44	1.02
trainlm	Levenberg-Marquardt	1.87	6	0.46

## 7.7 Function approximation — Example

In this example we use the Levenberg-Marquardt algorithm to approximate two functions of two variables.

```
% Generation of training points
na = 16 ; N = na^2; nn = 0:na-1;
X1 = nn*4/na - 2;
[X1 X2] = meshgrid(X1);
R = -(X1.^2 + X2.^2 + 0.00001);
D1 = X1 .* exp(R); D = (D1(:))';
D2 = 0.25*sin(2*R)./R ; D = [D ; (D2(:))']';
Y = zeros(size(D)) ;
X = [ X1(:)'; X2(:)']';
figure(1), clf reset
surfc([X1-2 X1+2], [X2 X2], [D1 D2]),
title('Two 2-D target functions'), grid, drawnow

% network specification
p = 2 ; % Number of inputs
L = 12; % Number of hidden signals
m = 2 ; % Number of outputs
xmnmx = [-2 2; -2 2] ;
net = newff(xmnmx, [L, m]) ;
figure(2)
net = train(net, X, D) ; % Training
Y = sim(net, X) ; % Verification

D1(:)=Y(1,:)' ; D2(:)=Y(2,:)' ;
% Two 2-D approximated functions are plotted side by side
figure(3)
surfc([X1-2 X1+2], [X2 X2], [D1 D2]), grid, ...
title('function approximation'), drawnow
```

## References

- [DB98] H. Demuth and M. Beale. *Neural Network TOOLBOX User's Guide. For use with MATLAB*. The MathWorks Inc., 1998.
- [FS91] J.A. Freeman and D.M. Skapura. *Neural Networks. Algorithms, Applications, and Programming Techniques*. Addison-Wesley, 1991. ISBN 0-201-51376-5.
- [Has95] Mohamad H. Hassoun. *Fundamentals of Artificial Neural Networks*. The MIT Press, 1995. ISBN 0-262-08239-X.
- [Hay99] Simon Haykin. *Neural Networks – a Comprehensive Foundation*. Prentice Hall, New Jersey, 2nd edition, 1999. ISBN 0-13-273350-1.
- [HDB96] Martin T. Hagan, H Demuth, and M. Beale. *Neural Network Design*. PWS Publishing, 1996.
- [HKP91] Hertz, Krogh, and Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991. ISBN 0-201-51560-1.
- [Kos92] Bart Kosko. *Neural Networks for Signal Processing*. Prentice-Hall, 1992. ISBN 0-13-617390-X.
- [Sar98] W.S. Sarle, editor. *Neural Network FAQ*. Newsgroup: comp.ai.neural-nets, 1998. URL: <ftp://ftp.sas.com/pub/neural/FAQ.html>.