5 Feedforward Multilayer Neural Networks — part II

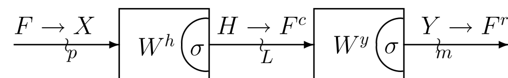
In this section we first consider selected applications of the multi-layer perceptrons.

5.1 Image Coding using Multi-layer Perceptrons

- In this example we study an application of a two-layer feed-forward neural network (perceptron) in image coding.

The general concept is as follows:

- A two-layer perceptron is trained using a representative set of images, F.

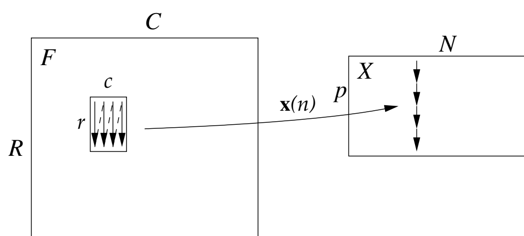


- Once the training is completed, appropriate hidden, W^h and output, W^y weights are available.
- An image, F, can now be encoded into an image F^c, represented by the hidden signals.
- If $L < p$ an image compression occurs.
- The encoded (compressed) image, F^c, represented by the hidden signals can now be reconstructed using the output layer of the perceptron as F^r.

Training procedure:

- Training is conducted for a representative class of images using the back-propagation algorithm.
- Assume that an image, F, used in training is of size $R \times C$ and consists of $r \times c$ blocks.
- Convert a block matrix F into a matrix X of size $p \times N$ containing training vectors, $\mathbf{x}(n)$, formed from image blocks. Note that

$$p = r \cdot c \,, \quad \text{and} \quad p \cdot N = R \cdot C$$



Use the following MATLAB function to perform this conversion: X = blkM2vc(F, [r c]);

```
function vc = blkM2vc(M, blkS)
[rr cc] = size(M) ;
r = blkS(1) ; c = blkS(2) ;
if  (rem(rr, r) ~= 0) | (rem(cc, c) ~= 0)
 error('blocks do not fit into matrix')
end
nr = rr/r ; nc = cc/c ; rc = r*c ;
vc = zeros(rc, nr*nc);
for ii = 0:nr-1
 vc(:,(1:nc)+ii*nc)=reshape(M((1:r)+ii*r,:),rc,nc)
end
```

- As a **target data** use the **input data**, that is:

$$D = X$$

- Train the network until the mean squared error, J, is sufficiently small. The matrices W^h and W^y will be subsequently used in the image encoding and decoding steps.

Image encoding:

- An image, F, is divided into $r \times c$ blocks of pixels. Each block is then scanned to form an input vector $\mathbf{x}(n)$ of size $p = r \times c$.

  The image to be encoded is now represented by an $N \times p$ matrix X, each column storing a block of pixels.

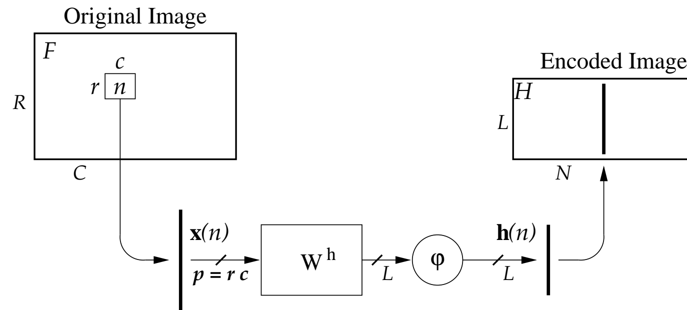


Figure 5–1: A hidden layer in image encoding

- Assume that the hidden layer of the neural network consists of L neurons each with p synapses, and that it is characterised by the appropriately selected weight matrix W^h.
- The encoding procedure can be described as follows:

$$F \longrightarrow X \ , \quad H = \sigma(W^h \cdot X) \longrightarrow F^c$$

  where F^c represented by H is an encoded image.

Image reconstruction:

- Assume that the output layer consists of $m = p = r \times c$ neurons, each with L synapses.

  Let W^y be an appropriately selected output weight matrix.
- The decoding procedure can be described as follows:

$$Y = \sigma(W^y \cdot H) \ , \quad Y \longrightarrow F^r$$

- Re-assemble the output signals into $p = r \times c$ image blocks to obtain a re-constructed image, F^r.

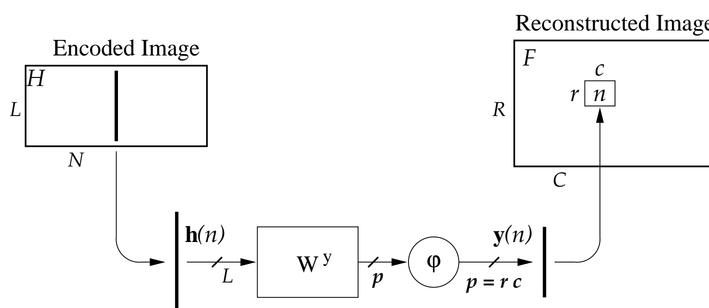


Figure 5–2: the outpur layer in image reconstruction

- The quality of image coding is typically assessed by the Signal-to-Noise Ratio (SNR) defined as

$$\mathrm{SNR} = 10 \log \frac{\Sigma_{i,j}(F_{i,j})^2}{\Sigma_{i,j}(F^r_{i,j} - F_{i,j})^2}$$

- The conversion of the vectors of the reconstructed image stored in the $p \times N$ matrix Y into blocks of the reconstructed image, F^r, can be performed using the following MATLAB function:

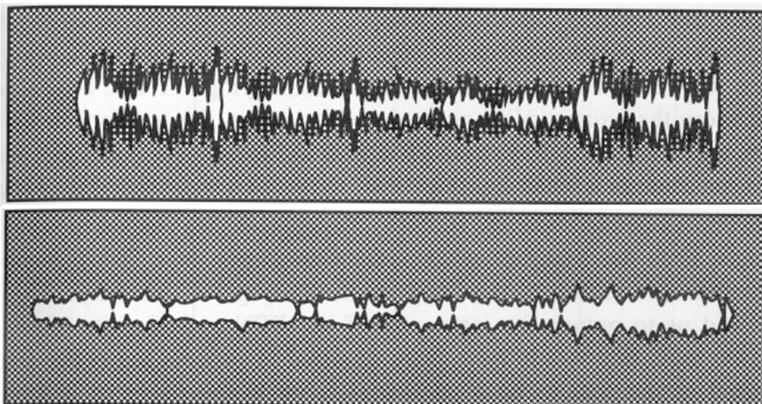
Fr = vc2blkM(Y, r, R);

```
function M = vc2blkM(vc, r, rM)
%vc2blkM  Reshaping a matrix vc of rc by 1 vectors into a
%         block-matrix   M  of  rM  by  cM  size
% Each rc-element column of vc is converted into a  r by c
% block of a matrix  M  and placed as a block-row element
[rc nb] = size(vc) ; pxls = rc*nb ;
if ( (rem(pxls, rM) ~= 0) | (rem(rM, r) ~= 0) )
  error('incorrect number of rows of the matrix')
end
cM = pxls/rM ;
if ( (rem(rc, r) ~= 0) | (rem(nb*r, rM) ~= 0) )
  error('incorrect block size')
end
c = rc/r ;
xM = zeros(r, nb*c);
xM(:) = vc ;
nrb = rM/r ;
M = zeros(rM, cM);
for ii = 0:nrb-1
  M((1:r)+ii*r, :) = xM(:, (1:cM)+ii*cM) ;
end
```


5.2 Paint-Quality Inspection

Adapted from (Freeman and Skapura, 1991)

- Visual inspection of painted surfaces, such as automobile body panels, is a very time-consuming and labor-intensive process.
- To reduce the amount of time required to perform this inspection, one of the major U.S. automobile manufacturers reflects a laser beam off the painted panel and on to a projection screen.
- Since the light source is a coherent beam, the amount of scatter observed in the reflected image of the laser provides an indication of the quality of the paint finish on the car.

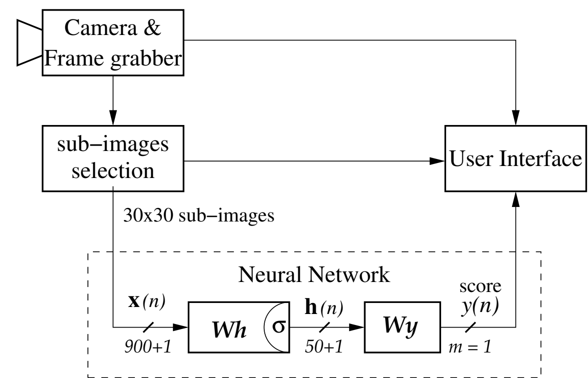


Reflection of a laser beam off painted sheet-metal surfaces:

top — a poor-quality paint finish: reflection is relatively difused.

bottom — a better-quality paint finish: reflection is very close to uniform throughout its image.

- A neural network, a two-layer perceptron in this case, is used to capture the expertise of the human inspectors scoring the paint quality from observation of the reflected laser images.
- The block-diagram of the Automatic Paint QA System:



- Images of the reflected laser beam are recorded by a camera and an associated frame grabber. Each image contains 400-by-75 8-bit pixels.
- To keep the size of the network needed to solve the problem manageable, we elected to take 10 sub-images from the snapshot, each sub-image consisting of a 30-by-30-pixel square centered on a region of the image with the brightest intensity.
- These 8-bit pixels are input to the neural network. In addition there is one biasing input, therefore, $p = 901$.
- The hidden layer consists of $L = 50$ neurons, hence the hidden-matrix, W^h is 900×50, and there are $900 \times 50 = 45000$ synapses in the hidden layer. A unipolar sigmoidal function is used.
- A single output signal from the network represents a **numerical score** in the range of 1 through 20 (a 1 represented the best possible paint finish; a 20 represented the worst).
- The output layer is **linear** and the output matrix W^y has a size 51×1 (there is a biasing input to the output layer).

- Once the network was constructed (and trained), 10 sub-images were taken from the snapshot using two different sampling techniques.
- In the first test, the samples were selected randomly from the image (in the sense that their position on the beam image was random).
- In the second test, 10 sequential samples were taken, so as to ensure that the entire beam was examined.
- In both cases, the input sample was propagated through the trained MLP, and the score produced as output by the network was averaged across the 10 trials.
- The average score, as well as the range of scores produced, were then provided to the user for comparison and interpretation.

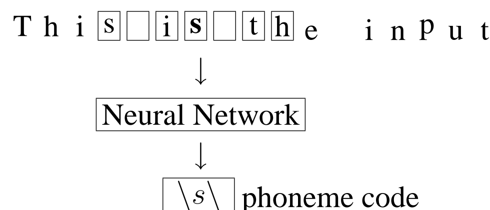
Training the Paint QA Network

- At the time of the development of this application, (1988) this network was significantly larger than any other network we had yet trained.
- The network is relatively big: 901 inputs, 51 hidden neurons, 1 output. Total number of trainable weights (synapses) is 45 101.
- The number of training patterns with which we had to work was a function of the number of control paint panels to which we had access (18), as well as of the number of sample images we needed from each panel to acquire a relatively complete training set (approximately 6600 images per panel).
- During training, the samples were presented to the network randomly to ensure that no single paint panel dominated the training.
- From these numbers, we can see that there was a great deal of computer time consumed during the training process.
- For example, one training epoch required the computer to perform approximately 13.5 million weight updates, which translates into roughly 360,000 floating-point operations (FLOPS) per pattern (2 FLOPS per connection during forward propagation, 6 FLOPS during error propagation), or 108 million FLOPS per epoch.
- However, once the network was trained, decoding is very fast and can be efficiently used in manufacturing.

5.3 NETtalk

Sejnowski and Rosenberg, 1987

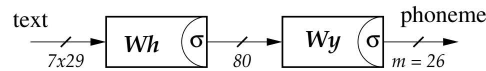
- The NETtalk project aimed at training a network to pronounce English text.
- The conceptual structure of the network is as follows:



- A character from a text and its three proceeding and three following characters are entered into a neural network which generates a phoneme code for the central character.
- The phoneme code can be sent to a speech generator giving the pronunciation of the central letter from the input window.

Network structure

- There are 29 English letters (including punctuation) and each letter is coded in a 1-of-29 code. Therefore, there are $p = 7 \times 29 = 203$ binary inputs to the network.
- Similarly, there are 26 different phonemes, hence, the network has 26 binary outputs.
- In addition, 80 hidden neurons are employed.



Training

- During training, the desired data were supplied by a commercially available DEC-talk, which is based on hand-coded linguistic rules.
- The network was trained on 1024 words, obtaining intelligible speech after 10 training epochs and 95% accuracy after 50 epochs.

5.4 Efficient initialization of the learning algorithms

- The simplest initialization of the weights is based on assigning them a "small" random values.
- This is not always a good solution because the activation potentials can be big enough to drive the activation functions into saturation.
- In saturation, the derivatives of the activation functions are zero, hence no weight update will take place.
- Efficient initialization can speed up the convergence process of the learning algorithms 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 for simplicity a **single layer** of m neurons with p synapses, each including the bias. Then for jth neuron we have

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

- For an activation function $\sigma(v)$ we can specify its **active region** $\bar{v} = [v_{min} \; v_{max}]$ outside which the function is consider to be in saturation.
- For example, for the hyperbolic tangent we can assume the active region as:

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

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

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

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 $p = 2$ (**single input plus bias**) the weights are initialised in the following way:

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

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

For $p > 2$ the weight 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

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

and the bias weights

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

- Finally, the weights are linearly rescaled to account for different range of activation potentials an 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(s,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] ;
```

5.5 Why backpropagation is slow

- The basic pattern-based 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:

$$\Delta W(n) = -\eta \cdot \nabla_W E(n) = \eta \cdot \boldsymbol{\delta}(n) \cdot \mathbf{x}^T(n) \tag{5.1}$$

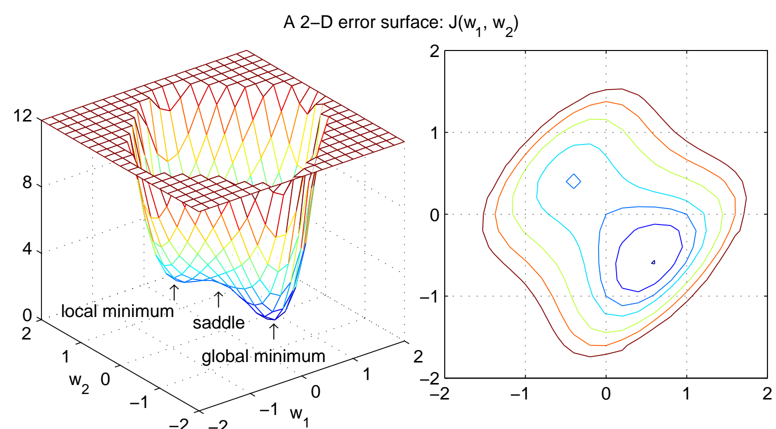
  Such an algorithm is slow for a few reasons:

- 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 back propagation, which is conceptually very interesting, serialises computations on the layer by layer basis.

A general problem is that 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, therefore, it is difficult to navigate directly to its minimum.

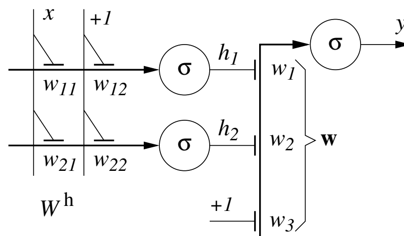
5.6 Examples of error surfaces

- Consider the following function of two weights, $J(w_1, w_2)$ representing a possible mean-squared error together with its contour map.
- Note the local and global minima, and a saddle point.
- Consider the importance of the proper initialisation to be able to reach the global minimum.



- 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.

- In order to gain more insight into the shape of the error surfaces let us consider a simple two-layer network approximating a single-variable function similar to that considered in sec. 4.3

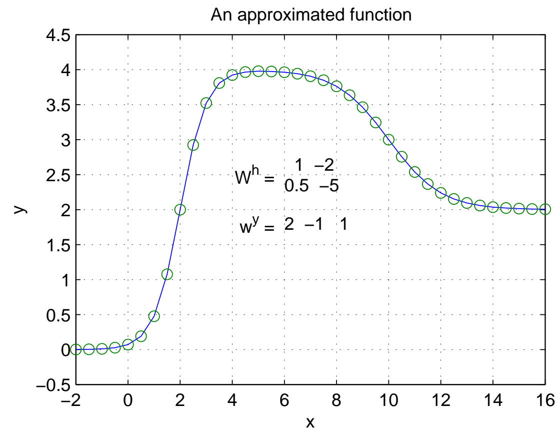


$$y(n) = \boldsymbol{\sigma}(\mathbf{w}^y \cdot \begin{bmatrix} \boldsymbol{\sigma}(W^h \cdot \begin{bmatrix} x(n) \\ 1 \end{bmatrix}) \\ 1 \end{bmatrix})$$

- For a specific set of parameters:

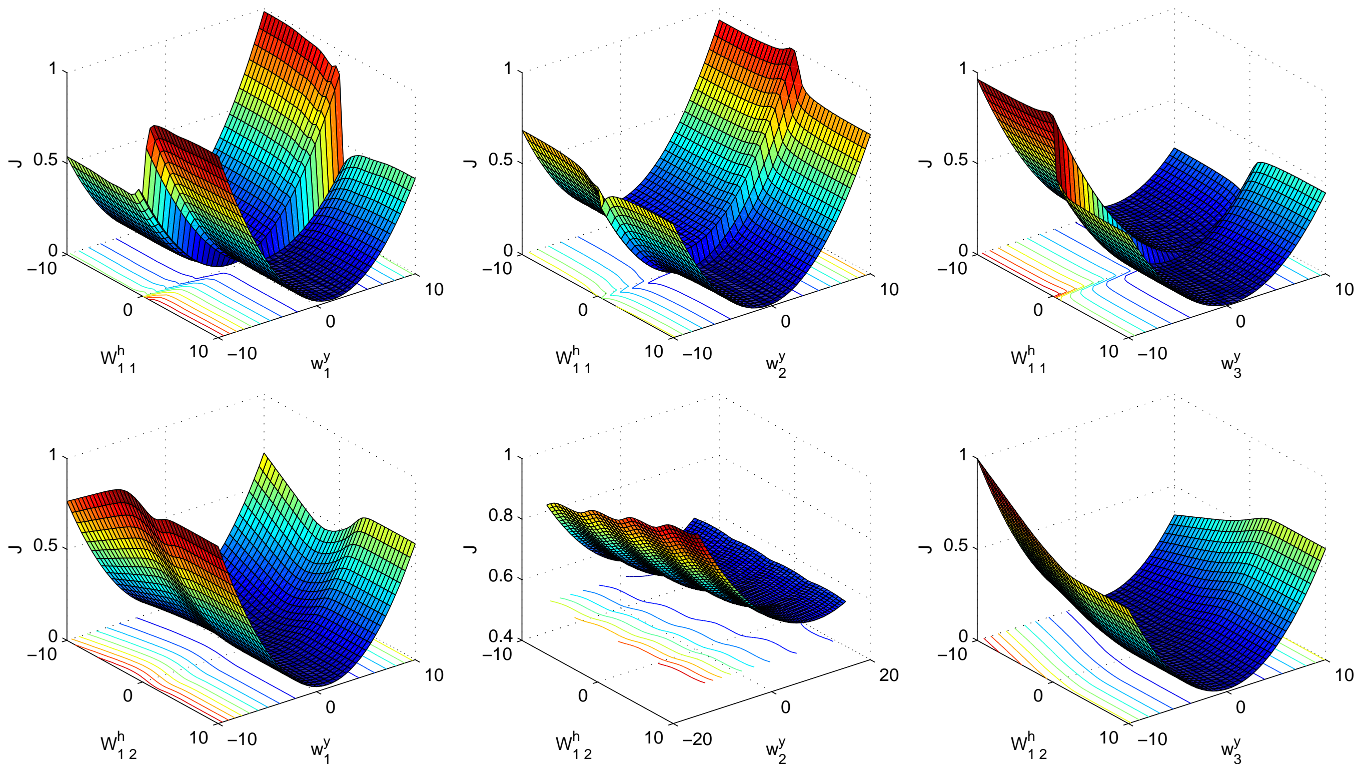
$$W^h = \begin{bmatrix} 1 & -2 \\ 0.5 & -5 \end{bmatrix} \quad \text{and} \quad \mathbf{w}^y = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix}$$

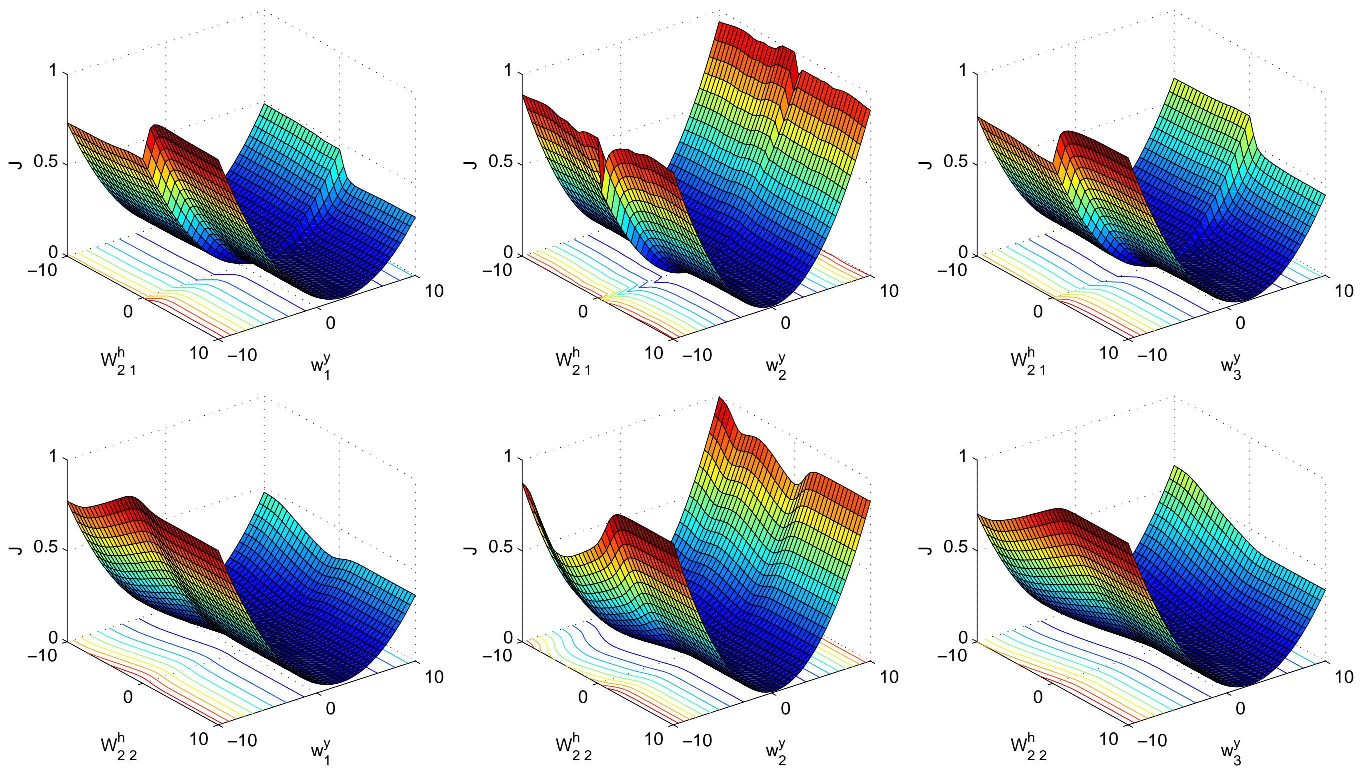
the network approximate the following function:



- In order to obtain the error surface, we will vary $\mathbf{w} = [W^h \ \ \mathbf{w}^y]$ and calculate $J(\mathbf{w})$ for the selected inputs X.
- The error function $J(\mathbf{w})$ is an 8-dimensional object, hence difficult to visualise. Therefore we will vary only a pair of selected weights at a time.

Neuro-Fuzzy Comp. — Ch. 5 March 24, 2005



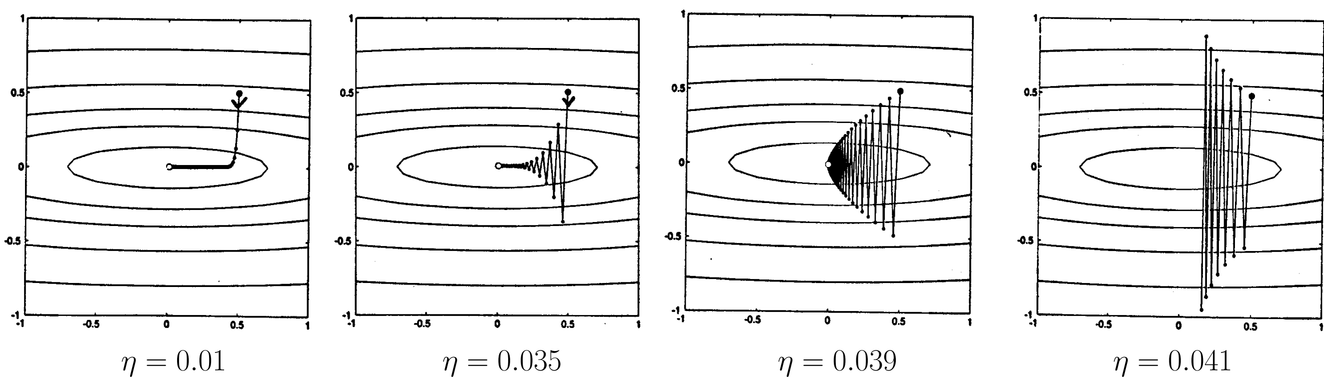


Note that the surfaces are very far away from and ideal second order paraboloidal shapes.
Finding the minimum is very sensitive to the initial position, learning gain and the direction of movement.

5.7 Illustration of sensitivity to a learning rate

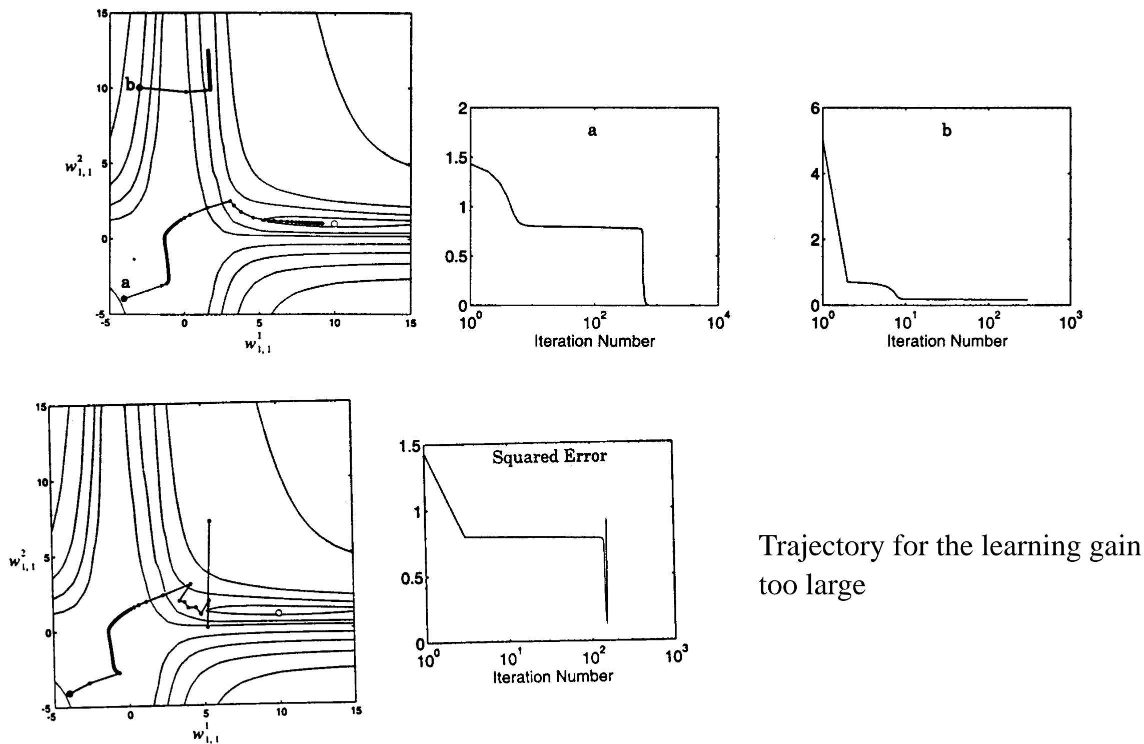
Figures 9.1, 9.2, 9.3, 12.6, 12.7 12.8 from: M.T. Hagan, H. Demuth, M. Beale, *Neural Network Design*, PWS Publishing, 1996

- For an Adaline, when the error surface is paraboloidal, the maximum stable learning gain can be evaluated from eqn (3.24) and is inversely proportional to the largest eigenvalue of the input correlation matrix, R.
- As an illustration we consider the case when $\eta_{mx} = 0.04$ and observe the learning trajectory on the error surface for a linear case:



$\eta = 0.01$ $\eta = 0.035$ $\eta = 0.039$ $\eta = 0.041$

- Examples of learning trajectories for the steepest descent backpropagation algorithm in the batch mode. Plots on the right shows the error versus the iteration number.



Trajectory for the learning gain too large

5.8 Heuristic Improvements to the Back-Propagation Algorithm

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

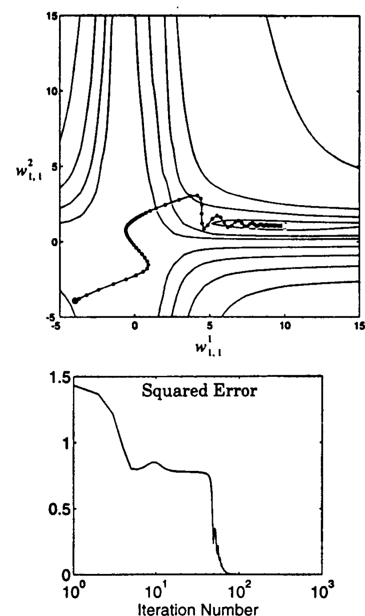
5.8.1 The momentum term

- 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.

$$\Delta W(n) = \eta \cdot \boldsymbol{\delta}(n) \cdot \mathbf{x}^T(n) + \alpha \cdot \Delta W(n-1) \ , \quad 0 < \alpha < 1 \quad (5.2)$$

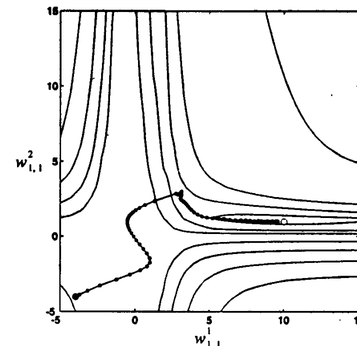
where α is a momentum term parameter.

- 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, η.



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

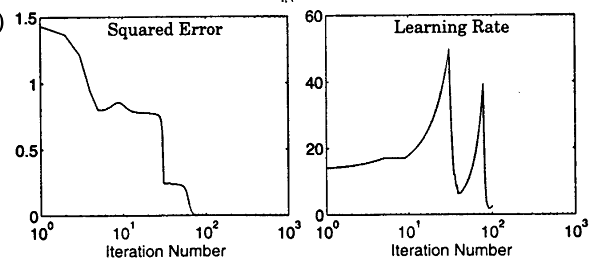


- If J is decreasing consistently, that is, ∇J is negative for a prescribed number of steps, then the learning rate is increased linearly:

$$\eta(n+1) = \eta(n) + a \ , \quad a > 0 \tag{5.3}$$

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

$$\eta(n+1) = b \cdot \eta(n) \ , \quad 0 < b < 1 \tag{5.4}$$



- In general, increasing the value of the learning rate the learning tends to become unstable which is indicated be an increase in the value of the error function.
- Therefore it is important to quickly reduce η.

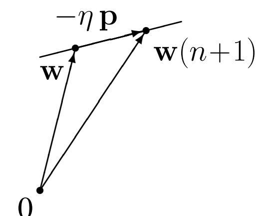
Advanced methods of optimisation

- Optimization or 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 such mathematicians as Kepler, Fermat, Newton, Leibnitz, Gauss.
- In general the problem is to find an optimal learning gain and the optimal search direction that takes into account the shape of the error function, that is its curvature.

5.9 Line search minimisation procedures

- Gradient descent minimization procedures are based on updating the weight vector

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta\, \mathbf{p}(n) \tag{5.5}$$



where η is the learning gain and the vector $\mathbf{p}(n)$ describes the direction of modification of the weight vector.

- The vector $\mathbf{p}$ is typically equal to the negative gradient of the error function

$$\mathbf{p}(n) = -\mathbf{g}(n) \ , \quad \text{where} \quad \mathbf{g}(n) = \nabla J(\mathbf{w}(n))$$

- Note that the next value of weight vector, $\mathbf{w}(n+1)$, is obtained from the current value of weight vector, $\mathbf{w}$, by moving it along the direction of a vector, $\mathbf{p}$.

- We can now find an optimal value of η for which the performance index

$$J(\mathbf{w}(n+1)) = J(\mathbf{w} + \eta\,\mathbf{p}) \tag{5.6}$$

is minimised.

- The optimal η is typically found through a search procedure along the direction $\mathbf{p}$ (line optimization)
- In order to find some properties of the line optimization we calculate the partial derivative of J with respect to η:

$$\frac{\partial J(\mathbf{w}(n+1))}{\partial \eta} = \frac{\partial J(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \frac{\partial \mathbf{w}(n+1)}{\partial \eta} = \frac{\partial J(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)}\,\eta\,\mathbf{p}^T = \eta\,\nabla J(\mathbf{w}(n+1)) \cdot \mathbf{p}^T \tag{5.7}$$

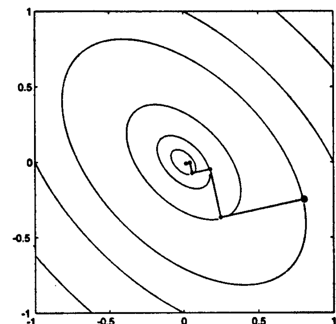
- For the optimal value of η, this derivative needs to be zero and we have the following relationship

$$\nabla J(\mathbf{w}(n+1)) \cdot \mathbf{p}^T = \mathbf{g}(n+1) \cdot \mathbf{p}^T = 0 \tag{5.8}$$

- It states that the next estimate of the gradient, $\mathbf{g}(n+1) = \nabla J(\mathbf{w}(n+1))$, is to be orthogonal to the current search direction, $\mathbf{p}$ for the optimal value of η

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.

5.10 Conjugate Gradient Algorithm

- The conjugate gradient algorithms also involved the line optimisation with respect to η, but
- in order to avoid the zig-zag movement through the error surface, the next search direction, $\mathbf{p}(n+1)$, instead of being exactly orthogonal to the gradient, tries to maintain the current search direction, $\mathbf{p}(n)$, namely

$$\mathbf{p}(n+1) = -\mathbf{g}(n) + \beta(n)\mathbf{p}(n) \tag{5.9}$$

where scalar $\beta(n)$ is selected in such a way that

- the directions $\mathbf{p}(n+1)$ and $\mathbf{p}(n)$ are **conjugate with respect to the Hessian matrix**, $\nabla^2 J(\mathbf{w}) = H$ (the matrix of all second derivatives of J), that is,

$$\mathbf{p}(n+1) \cdot H \cdot \mathbf{p}^T(n) = 0 \tag{5.10}$$

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

- Hestenes-Steifel formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{p}^T(n-1)} \tag{5.11}$$

- Fletcher-Reeves formula

$$\beta(n) = \frac{\mathbf{g}(n) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \tag{5.12}$$

- Polak-Ribiére formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \tag{5.13}$$

In summary, the conjugate gradient involves:

- initial search direction, $\mathbf{p}(0) = -\mathbf{g}(0)$,
- line minimisation with respect of η,
- calculation of the next search direction as in eqn (5.9), and
- β from one of the above formulae.

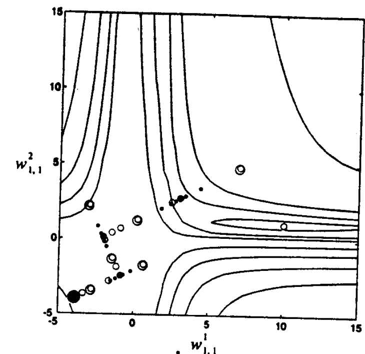


Figure 12.15 Intermediate Steps of CGBP

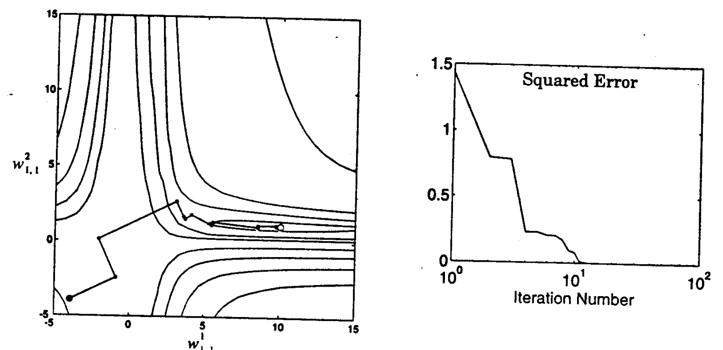


Figure 12.16 Conjugate Gradient Trajectory

5.11 Newton's Methods

- In Newton's methods minimisation is based on utilisation of not only the **first derivatives** (the gradient) of the error function, but also its **second derivatives** (Hessian matrix).
- Consider the Taylor series expansion of the performance index as in eqn (4.5) which can be re-written as

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

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

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

- Neglecting the higher order expansion terms, we have the following fundamental for Newton's methods equation:

$$\Delta\mathbf{w} = -\nabla J \cdot \mathbf{H}^{-1} \tag{5.14}$$

- This equation says that a more accurate weight update is the direction opposite to the gradient vector 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 methods are typically faster than conjugate gradient algorithms.
- However, they require computations of the inverse of the Hessian matrix which are 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 then the pattern mode which is based on derivatives of instantaneous errors.

5.12 Gauss-Newton method

In the Gauss-Newton method the **Hessian matrix** is approximated by a product of the **Jacobian matrix**.

In order to explain details of the method let us repeat the standard assumption:

- 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 jth element of the instantaneous gradient vector can now be expressed as

$$[\nabla E(\mathbf{w}(n))]_j = \frac{\partial E(\mathbf{w}(n))}{\partial w_j} = \sum_{i=1}^{m} \varepsilon_k(\mathbf{w}) \frac{\partial \varepsilon_i(\mathbf{w})}{\partial w_j} = \boldsymbol{\varepsilon}^T(\mathbf{w}) \left[\frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_j} \dots \frac{\partial \varepsilon_m(\mathbf{w})}{\partial w_j} \right]^T$$

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

$$\nabla E(\mathbf{w}(n)) = \boldsymbol{\varepsilon}^T(\mathbf{w}(n)) \mathcal{J}(\mathbf{w}(n)) \tag{5.15}$$

  where

$$\mathcal{J}(\mathbf{w}(n)) = \begin{bmatrix} \frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_1} & \dots & \frac{\partial \varepsilon_1(\mathbf{w})}{\partial w_K} \\ \vdots & & \vdots \\ \frac{\partial \varepsilon_m(\mathbf{w})}{\partial w_1} & \dots & \frac{\partial \varepsilon_m(\mathbf{w})}{\partial w_K} \end{bmatrix} \tag{5.16}$$

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

- In order to find the Hessian matrix of the instantenous performance index we differentiate eqn (5.15):

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

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

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

- Generalizing the above expression into a matrix form, we obtain:

$$
\nabla^2 E(\mathbf{w}) = \mathcal{J}^T(\mathbf{w})\mathcal{J}(\mathbf{w}) + \boldsymbol{\varepsilon}^T(\mathbf{w})R(\mathbf{w}) \tag{5.17}
$$

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.

- 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 (5.14) becomes:

$$
\Delta \mathbf{w}(n) = -\nabla E(n) H^{-1}(n) = -\boldsymbol{\varepsilon}^T(n)\mathcal{J}(n) \left(\mathcal{J}^T(n)\mathcal{J}(n) \right)^{-1} \tag{5.18}
$$

- 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.

5.13 Levenberg-Marquardt algorithm

5.13.1 The 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 μ is added such that

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

where $\mathbf{H}(\mathbf{w})$ and $\mathbf{J}(\mathbf{w})$ are the **batch Hessian and Jacobian matrices**, respectively, I is the identity matrix and μ 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

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

is the vector of all instantaneous errors.

Neuro-Fuzzy Comp. — Ch. 5 March 24, 2005

- 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 (5.19):

$$\Delta\mathbf{w} = -\nabla F \cdot \mathbf{H}^{-1} = -\mathbf{e}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) \left(\mathbf{J}^T(\mathbf{w})\mathbf{J}(\mathbf{w}) + \mu I\right)^{-1} \tag{5.19}$$

5.13.2 Calculation of the Jacobian matrix

- 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 (5.16).
- 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:

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

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:

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

5.13.3 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 4–3.
- The elements of $\mathcal{J}^y(n)$ are the first derivatives $\frac{\partial \varepsilon_i}{\partial w^y_{kj}}$ of errors $\varepsilon_i(n)$ with respect to weights w^y_{kj}. We have

$$\varepsilon_i(n) = d_i(n) - y_i(n) \; , \quad y_i(n) = \sigma(v_i(n)) \; , \quad v_i(n) = W^y_{i:} \cdot \mathbf{h}(n)$$

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

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

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

Neuro-Fuzzy Comp. — Ch. 5 March 24, 2005

- Subsequently, we have

$$\frac{\partial y_k}{\partial w^y_{kj}} = \frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w^y_{kj}} = \sigma'_k \cdot h_j \; , \quad \text{where} \quad \sigma'_k = \frac{\partial y_k}{\partial v_k}$$

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

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

- Rows of the matrix P form the diagonal blocks of the Jacobian $\mathcal{J}^y$.
- More formally, we can write

$$\mathcal{J}^y = -\text{diag}(\boldsymbol{\sigma}') \otimes \mathbf{h}^T \tag{5.20}$$

where $\otimes$ denotes the Kronecker product.

5.13.4 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

$$\frac{\partial \varepsilon_k}{\partial w^h_{ji}} = -\frac{\partial y_k}{\partial w^y_{ji}} = -\frac{\partial y_k}{\partial v_k}\frac{\partial v_k}{\partial w^h_{ji}} = -\sigma'_k\frac{\partial v_k}{\partial w^h_{ji}}$$

- If we take into account that

$$v_k = W^y_{k:} \cdot \mathbf{h} = \cdots + w^y_{kj} \cdot h_j + \cdots$$

and

$$h_j = \psi(W^h_{j:} \cdot \mathbf{x}) = \psi(\cdots + w^h_{ji} \cdot x_i + \cdots)$$

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

$$[\mathcal{J}^h]_{k,ji} = \frac{\partial \varepsilon_k}{\partial w^h_{ji}} = -\sigma'_k \cdot w^y_{kj} \cdot \frac{\partial h_j}{\partial w^h_{ji}} = -\sigma'_k \cdot w^y_{kj} \cdot \psi'_j \cdot x_i \tag{5.21}$$

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

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

and finally, we have

$$\mathcal{J}^h = -S^h \otimes \mathbf{x}^T \quad , \quad \text{where} \;\; S^h = \text{diag}(\boldsymbol{\sigma}') \cdot W^y \cdot \text{diag}(\boldsymbol{\psi}') \tag{5.22}$$

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, Ψ', and Φ',
   - errors, $D - Y$, and $\mathbf{e}$.
2. Calculate instantaneous Jacobian matrices, $\mathcal{J}^h$ and $\mathcal{J}^y$ as in eqns (5.22) and (5.20), and arrange them in the batch Jacobian $\mathbf{J}$.
3. Calculate the weight update, $\Delta\mathbf{w}$, according to eqn (5.22) for a selected value of μ.
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 μ 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, $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$.

5.13.5 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)×(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:

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

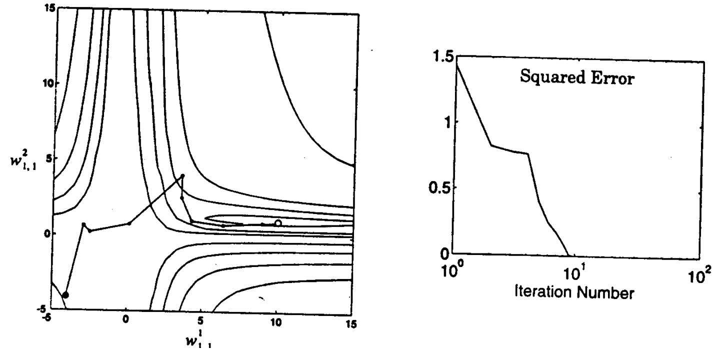
- The size of the matrix to be inverted, $(\mathbf{J}^T\mathbf{J} + \mu I)$ is $K \times K$.

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

- 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, $\mathbf{J}$.



LMbp trajectory

5.14 Speed comparison

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

Function	Algorithm	Relative time
LM trainlm	Levenberg-Marquardt	1.00
BFG trainbfg	BFGS Quasi-Newton	4.58
RP trainrp	Resilient Backpropagation	4.97
SCG trainscg	Scaled Conjugate Gradient	5.34
CGB traincgb	Conjugate Gradient with Powell/Beale Restarts	5.80
CGF traincgf	Fletcher-Powell Conjugate Gradient	6.89
CGP traincgp	Polak-Ribiére Conjugate Gradient	7.23
OSS trainoss	One-Step Secant	8.46
GDX traingdx	Variable Learning Rate Backpropagation	24.29