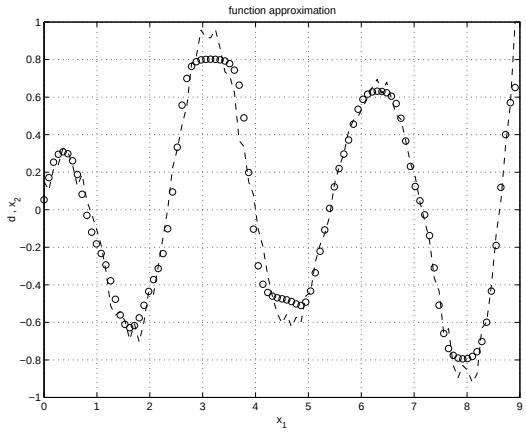
6 Self-Organizing Neural Networks

6.1 Supervised and Unsupervised Learning

- Learning algorithms which were considered for a single perceptron, linear adaline, and multilayer perceptron belong to the class of **supervised learning** algorithms.
- In this case the training data is divided into input signals, $\mathbf{x}(n)$, and target signals, $\mathbf{d}(n)$.
- A typical learning algorithm is driven by error signals $\boldsymbol{\varepsilon}(n)$ which are the differences between the actual network output, $\mathbf{y}(n)$, and the desire (or target) output for a given input.
- For a pattern learning, we can express the weight update in the following general form

$$\Delta\mathbf{w}(n) = \mathcal{L}(\mathbf{w}(n), \mathbf{x}(n), \boldsymbol{\varepsilon}(n))$$

where $\mathcal{L}$ represents a learning algorithm.



- If we say that a neural network can describe a model of data, then a multilayer perceptron describes the data in a form of a curve, or surface or hypersurface which approximates a functional relationship between $\mathbf{x}(n)$, and $\mathbf{d}(n)$.

Self-organizing neural networks — Unsupervised Learning

- Self-organising neural networks employ unsupervised learning laws and discover **characteristic features** in input data without using a target or desired output.
- Information about the characteristic features of input data is created during the learning process and stored in the synaptic weights.
- Output signals describe relationship between the current input signals and the weight vectors.
- Two basic groups of unsupervised learning algorithms and related self-organizing neural networks, namely:
  - (Generalised) Hebbian Learning
  - Competitive Learning

  can be distinguished by the type of characteristic features that they "discover" from the input data, namely, "shape" of data and constellation of clusters of points.

Generalised Hebbian Learning

- Generalised Hebbian Learning extracts from data a set of **principal directions** along which data is organised in a p-dimensional space.
- Each direction is represent by a relevant weight vector. The number of those principal directions is, at most, equal to the dimensionality of the input space p.
- In an illustrative example presented in Figure 6–1 the two-dimensional data is organised along two principal directions, $\mathbf{w}_1$ and $\mathbf{w}_2$.

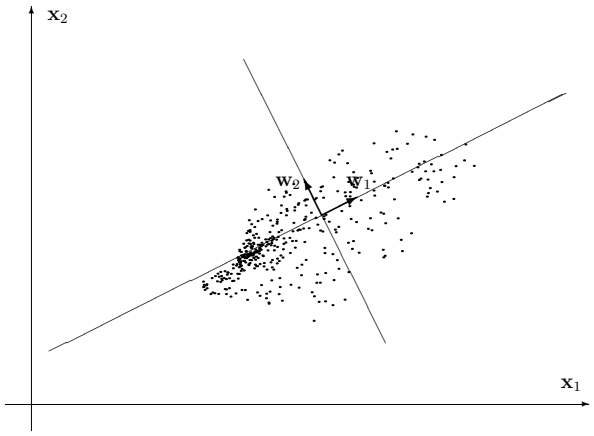


Figure 6–1: A 2-D pattern with principal directions

Competitive Learning

- Competitive Learning extracts from data a set of **centers of data clusters**.
- Each center point is stored as a weight vector. It is obvious that the number of clusters is independent of dimensionality of the input space.

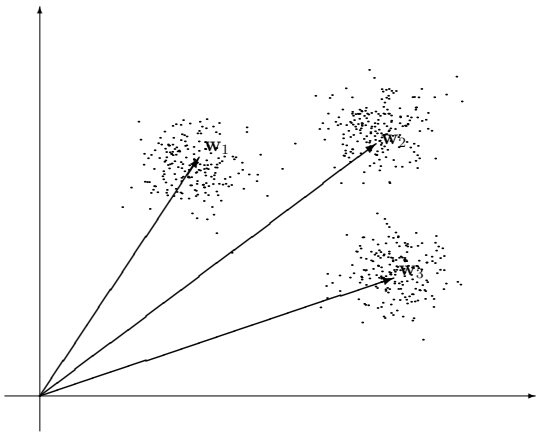


Figure 6–2: Example of two-dimensional data organised in three clusters. Claster centres are represented by three weight vectors.

- An important extension of a basic competitive learning is known as **feature maps**. A feature map is obtained by adding some form of topological organization to neurons.

6.2 Hebbian learning

6.2.1 Basic structure of Hebbian learning neural networks

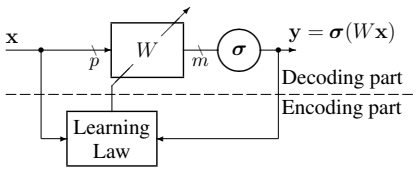


Figure 6–3: A block-diagram of a basic Hebbian learning neural network consisting of a single layer decoding part and a learning (encoding) part

- The decoding layer contains a single weight matrix W which is $m \times p$.
  Each row weight vector is associated with one neuron.
- The decoding layer is often linear, which further simplifies the network structure.
  The complexity of the network comes from the structure of the learning law employed.
- Note the absence of the target value in the encoding/learning part (un-supervised learning).
- The basic idea behind a Hebbian learning law is to make the update of a synaptic weight proportional to both input and output signals:

$$\begin{aligned} w_{ji}(n+1) &= f(w_{ji}(n), y_j(n), x_i(n)) \\ &= w_{ji}(n) + \eta y_j(n) x_i(n) \end{aligned} \qquad (6.1)$$

- These two signals, y_j and x_i are locally available at the ji synapse, therefore, this type of a learning law is termed as a **local learning law**.

The concept of the **local learning** law is illustrated in Figure 6–4.

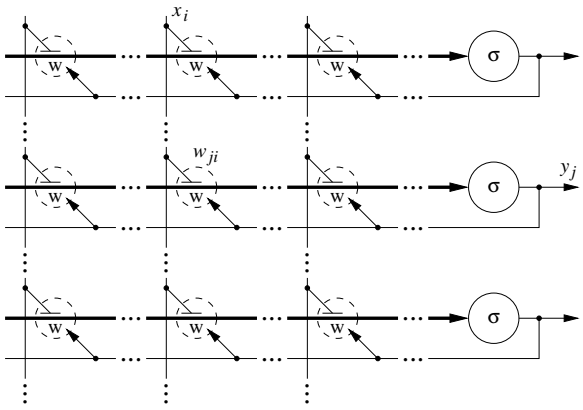


Figure 6–4: A neural network with a local learning law circuitry.

It may be observed that the neuron output signal, y_j, is available locally at the synapse through the additional feedback connection. This feedback is essential to the learning process.

- The **basic Hebbian learning** law in the form as in eqn (6.1) cannot be used because it is fundamentally **unstable**, that is, weights reveal an unlimited grow during the learning process.
- It is relatively simple to show that if a stable solution to the learning law (6.1) exists it must be zero.
- Assuming for simplicity linear neurons and re-writing eqn (6.1) in a matrix form gives:

$$\mathbf{W}(n+1) = \mathbf{W}(n) + \eta \mathbf{y}(n)\mathbf{x}^T(n) = \mathbf{W}(n) + \eta \mathbf{W}(n)\mathbf{x}(n)\mathbf{x}^T(n)$$

- Application of the expectation operator, $E[\cdot]$, to both sides yields

$$E[\mathbf{W}] = E[\mathbf{W}] + \eta E[\mathbf{W}]E[\mathbf{x}\mathbf{x}^T]$$

- Hence, the steady-state value of the weight matrix, $\bar{\mathbf{W}} = E[\mathbf{W}]$, must satisfy the following equation

$$\bar{\mathbf{W}} R = 0$$

  where

$$R = E[\mathbf{x}\mathbf{x}^T] \approx \frac{1}{N} X X^T$$

  is the input **correlation matrix**.
- The input correlation matrix is non-singular, therefore, the only possible steady-state value of the weight matrix is $\bar{\mathbf{W}} = 0$.

6.2.2 Stable Hebbian learning

In order to stabilize a Hebbian learning law two basic steps are required

- Assuming that $\mathbf{x}$ is a p-dimensional random vector representing the input data, it is required that its **mean** to be zero:

$$E[\mathbf{x}] = 0$$

  In practical calculations with MATLAB, when input vectors are collected in the $p \times N$ input matrix X, the non-zero mean is removed from the input data in the following way:

```
mX=mean(X,2) ; X=X-mX(:,ones(1,N));
```

  Note that if the mean is zero **correlation** matrix is identical to the **covariance** matrix.
- The basic Hebbian learning law is to be modified in such a way that the magnitude (lenght) of weight vectors should tend to unity.

  Assuming for simplicity a single neuron network, it can be achieve by the following normalization:

$$\mathbf{w} \Leftarrow \mathbf{w} + \eta y \mathbf{x}^T \; ; \quad \mathbf{w} \Leftarrow \mathbf{w}/||\mathbf{w}|| \tag{6.2}$$

  where the vector magnitude is calculated in the usual way as:

$$||\mathbf{w}|| = \sqrt{\sum_{i=1}^{p} w_i^2}\,, \quad \text{also} \quad ||\mathbf{w}||^2 = \mathbf{w}\mathbf{w}^T$$

- Normalization as in eqn (6.2) is computationally relatively complex, therefore, we can use the following simplification based on the Taylor series expansion.

$$\frac{1}{||\mathbf{w}+\eta y\mathbf{x}^T||} = \frac{1}{\sqrt{(\mathbf{w}+\eta y\mathbf{x}^T)(\mathbf{w}+\eta y\mathbf{x}^T)^T}} = \frac{1}{\sqrt{\mathbf{w}\mathbf{w}^T + 2\eta y\mathbf{w}\mathbf{x} + \eta^2 y^2\mathbf{x}^T\mathbf{x}}}$$

- If we assume that the previous weight vector was normalised, that is,

$$\mathbf{w}\mathbf{w}^T = ||\mathbf{w}||^2 = 1$$

  and that $\eta \ll 1$ is small so that η^2 is negligible, then we can further write:

$$\begin{aligned}\frac{\mathbf{w}+\eta y\mathbf{x}^T}{||\mathbf{w}+\eta y\mathbf{x}^T||} &\approx \frac{\mathbf{w}+\eta y\mathbf{x}^T}{\sqrt{1+2\eta y\mathbf{w}\mathbf{x}}} \approx (\mathbf{w}+\eta y\mathbf{x}^T)(1-\eta y\mathbf{w}\mathbf{x}) \\ &\approx \mathbf{w}+\eta y\mathbf{x}^T - \eta y^2\mathbf{w} - \eta^2 y^2\mathbf{x}^T \approx \mathbf{w}+\eta y(\mathbf{x}^T - y\mathbf{w})\end{aligned}$$

- It can be proved that if we update weights according to the last equation, that is,

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta y(n)(\mathbf{x}^T(n) - y(n)\mathbf{w}(n)) \tag{6.3}$$

  then the magnitude of the weight vector will be close to unity, $||\mathbf{w}|| \to 1$.

- This is the form of the weight update used in the **Generalised Hebbian Learning** (GHL).

6.2.3 A single neuron case — the Oja's rule

The unsupervised learning algorithm described in eqn (6.3) for a single neuron case is known as the Oja's rule, and can be written in the following form:

$$\Delta\mathbf{w} = \eta y(\mathbf{x}^T - y\mathbf{w}) = \eta y\tilde{\mathbf{x}}^T\ , \quad y = \mathbf{w}\mathbf{x} = \mathbf{x}^T\mathbf{w}^T \tag{6.4}$$

where the augmented input vector is:

$$\tilde{\mathbf{x}} = \mathbf{x} - y\mathbf{w}^T \tag{6.5}$$

The negative term brings in the required stabilization of the learning law. To show this we calculate the projection of the update vector, $\Delta\mathbf{w}$ onto the current weight vector, $\mathbf{w}$:

$$\Delta\mathbf{w}\,\mathbf{w}^T = \mathbf{x}^T\mathbf{w}^T - y\mathbf{w}\mathbf{w}^T = y(1-||\mathbf{w}||^2)$$

Therefore, if the current weight vector is not on the unit circle, the update vector will bring the next weight vector closer to the the unit circle.

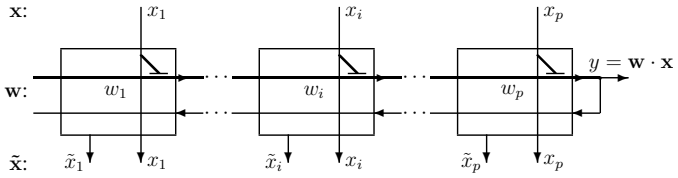


Figure 6–5: Internal structure of a neural network implementing the Oja's rule (6.4)

Each synapse aggregates the dendritic signal, and, during learning, generates the augmented input signal, and updates its weight.

Extraction of the first principal direction

- It is now possible to show that applying the learning law of eqn (6.4), the weight vector $\mathbf{w}$ converges to the **eigenvector** $\mathbf{q}_1$ of the input **correlation matrix** R associated with the largest **eigenvalue** λ_1 of R.
- The direction of the eigenvector $\mathbf{q}_1$ is referred to as the first **principal direction**.
- The sketch of the proof of the necessary convergence condition is as follows. Substitution of eqn (6.5) in eqn (6.4) yields:
$$\Delta\mathbf{w} = \eta(\mathbf{w}\mathbf{x}\mathbf{x}^T - \mathbf{w}\mathbf{x}\mathbf{x}^T\mathbf{w}^T\mathbf{w})$$
- Applying the statistical expectation operator to both sides gives:
$$E[\Delta\mathbf{w}] = \eta(\mathbf{w}E[\mathbf{x}\mathbf{x}^T] - \mathbf{w}E[\mathbf{x}\mathbf{x}^T]\mathbf{w}^T\mathbf{w}) \tag{6.6}$$
- The terms in eqn (6.6) can be estimated as follows. If we assume that the weight vector converges to a steady-state value, then the expectation of the weight update vectors is zero, that is, $E[\Delta\mathbf{w}] = 0$.
- The expectation of outer products of input vectors is the covariance (correlation) matrix:
$$R = E[\mathbf{x}\mathbf{x}^T] \approx \frac{1}{N}XX^T$$
- Now, eqn (6.6) can be written as
$$\mathbf{w}R = (\mathbf{w}R\mathbf{w}^T)\mathbf{w} \tag{6.7}$$
- Denote the scalar:
$$\lambda_1 = \mathbf{w}R\mathbf{w}^T \tag{6.8}$$

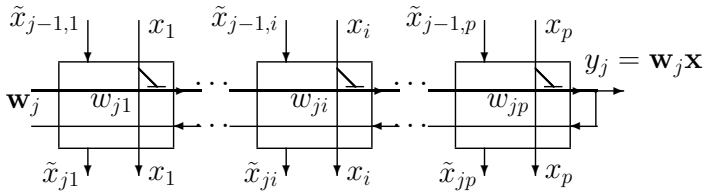
- Using definition (6.8) we can finally re-write eqn (6.7) in the following form
$$\mathbf{w}R = \lambda_1\mathbf{w} \tag{6.9}$$
- Alternatively, if we assume that
$$\mathbf{w}(n) \to \pm\mathbf{q}_1^T \;\; \text{as} \;\; n \to \infty \tag{6.10}$$
we have from eqn (6.9):
$$R\mathbf{q}_1 = \lambda_1\mathbf{q}_1 \quad \text{where} \quad \lambda_1 = \mathbf{q}_1^T R\mathbf{q}_1 \tag{6.11}$$
- Eqn (6.11) specifies a pair: an eigenvector $\mathbf{q}_1$ and related eigenvalue λ_1 which are characteristic values of a matrix (R in this case) such that, if R acts on $\mathbf{q}_1$ it modifies only the magnitude of the eigenvector.
- It can be shown that a "well behaving" $p \times p$ matrix has exactly p eigenvector-eigenvalue pairs.
- It can also be shown that λ_1 defined in eqn (6.8) or (6.11) and obtained using the Oja's rule is the largest eigenvalue of the input correlation matrix, R.
- As it is stated in eqn (6.10) the weight vector converges to the eigenvector $\mathbf{q}_1$, but the orientation of these two vectors does not have to be the same.
- Therefore, comparing the weight vector with the eigenvector when monitoring the convergence process, it is better to use the projection rather than the difference, that is:
$$|\mathbf{w} \cdot \mathbf{q}_1| \to 1 \quad \text{whereas} \quad ||\mathbf{w} - \mathbf{q}_1|| \to 0 \text{ or } +2 \tag{6.12}$$
- Condition (6.12) can be used to conveniently monitor the convergence process.

6.3 Self-Organizing Principal Component Analysis

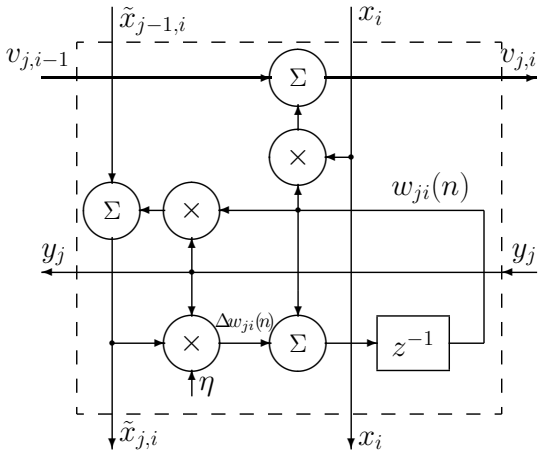
- The single neuron structure can be extended into a p-neuron network, the weight vector associated with subsequent neurons extracting the subsequent eigenvectors of the input correlation matrix.
- Such a network performs the **Principal Component Analysis** also known as the Karhunen-Loève transform.
- The objective of this analysis (transform) is to extract all principal directions characterising input data.
- The learning law involved is known as the Sanger's rule or Generalized Hebbian Algorithm (GHA).
- The idea behind the generalization of the Oja's rule neural network is to use in the learning part of neurons the **augmented input vectors** as specified by eqn (6.5).
- The weight update in the Sanger's rule, that is, the **Generalized Hebbian Algorithm** (GHA) can be described in the following way:

$$\begin{array}{lll} y_1 = \mathbf{w}_1\mathbf{x}\ , & \tilde{\mathbf{x}}_1 = \mathbf{x} - y_1\mathbf{w}_1^T\ , & \Delta\mathbf{w}_1 = \eta y_1\tilde{\mathbf{x}}_1^T \\ y_2 = \mathbf{w}_2\mathbf{x}\ , & \tilde{\mathbf{x}}_2 = \tilde{\mathbf{x}}_1 - y_2\mathbf{w}_2^T\ , & \Delta\mathbf{w}_2 = \eta y_2\tilde{\mathbf{x}}_2^T \\ & \cdots & \\ y_j = \mathbf{w}_j\mathbf{x}\ , & \tilde{\mathbf{x}}_j = \tilde{\mathbf{x}}_{j-1} - y_j\mathbf{w}_j^T\ , & \Delta\mathbf{w}_j = \eta y_j\tilde{\mathbf{x}}_j^T \\ & \cdots & \end{array} \tag{6.13}$$

- The jth neuron of the GHA has the following structure:



6.3.1 Structure of a single synapse implementing the Generalised Hebbian Learning



Identify:

- the dendritic activation signal

$$v_{j,i} = v_{j,i-1} + w_{ji}\cdot x_i\ , \quad (v_{j,0} = 0\ ,\ \ y_j = v_{j,p})$$

- the augmented input signal

$$\tilde{x}_{ji} = \tilde{x}_{j-1,i} - w_{ji}\cdot y_j$$

- the synaptic weight update (learning law)

$$\Delta w_{ji}(n) = \eta\cdot y_j(n)\cdot\tilde{x}_{ji}(n)$$

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n)$$

6.3.2 A matrix form of the Generalised Hebbian Learning

- In order to re-write the **Generalised Hebbian Learning** algorithm in a matrix for, let us first note that from eqn (6.13), we can write the augmented input signal vectors in the following form:

$$\tilde{\mathbf{x}}_j^T = \mathbf{x}^T - [y_1 \dots y_j] \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_j \end{bmatrix} = \mathbf{x}^T - [y_1 \dots y_j \, 0 \dots 0] \, W = \mathbf{x}^T - \tilde{\mathbf{y}}_j^T W \tag{6.14}$$

  where

$$\tilde{\mathbf{y}}_j^T = [y_1 \dots y_j \, 0 \dots 0]$$

  is the output vector $\mathbf{y}$ in which the last $m - j$ components are set to zero.

- Subsequently, the weight update for the j neuron specified in eqn (6.13) can be re-written in the following form

$$\Delta \mathbf{w}_j = \eta(y_j \mathbf{x}^T - y_j \tilde{\mathbf{y}}_j^T W)$$

- Finally, the **pattern update** of the whole weight matrix in the GHA algorithm can be expressed in the following compact form:

$$\mathbf{y} = W\mathbf{x} \; , \quad \Delta W = \eta(\mathbf{y}\mathbf{x}^T - \text{tril}(\mathbf{y}\mathbf{y}^T)W) \tag{6.15}$$

  where $\text{tril}(\cdot)$ denotes the lower-triangular matrix with elements above the main diagonal set to zero.

- The sketch of the prove that the network weight vectors converge to the eigenvalues of the input correlation matrix is as follows.
- Taking the statistical expectation operator on both sides of the weight update equation (6.15), and assuming that in the steady-state the updates are zeros, we have

$$0 = E[W\mathbf{x}\mathbf{x}^T] - E[\text{tril}(W\mathbf{x}\mathbf{x}^T W^T)W]$$

- This can be re-written as

$$WR = \text{tril}(WRW^T)W$$

  or

$$\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} R = \text{tril}(\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} R \begin{bmatrix} \mathbf{w}_1^T & \dots & \mathbf{w}_p^T \end{bmatrix}) \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix}$$

- The lower-triangular matrix in the right-hand side is of the form:

$$\Lambda = \text{tril}(WRW^T) = \begin{bmatrix} \lambda_1 & & \\ \{\lambda_{ji}\} & \ddots & \mathbf{0} \\ & & \lambda_p \end{bmatrix} , \quad \text{where} \quad \lambda_{ji} = \mathbf{w}_j R \mathbf{w}_i^T$$

- Therefore we finally have

$$\begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} R = \begin{bmatrix} \lambda_1 & & \\ \{\lambda_{ji}\} & \ddots & \mathbf{0} \\ & & \lambda_p \end{bmatrix} \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_p \end{bmatrix} \tag{6.16}$$

or

$$WR = \Lambda W \tag{6.17}$$

- The first row of eqn (6.16), namely,

$$\mathbf{w}_1 R = \lambda_1 \mathbf{w}_1 \,, \qquad \text{where} \qquad \lambda_1 = \mathbf{w}_1 R \mathbf{w}_1^T$$

  is exactly the same as eqn (6.11) from the Oja's network, the weight vector of the first neuron, $\mathbf{w}_1$, converging to the eigenvector of the input correlation matrix associated with the largest eigenvalue, λ_1.

- In order to show that the other weight vectors converge to subsequent eigenvectors it is enough to show that the off-diagonal coefficients $\lambda_{ji} = \mathbf{w}_j R \mathbf{w}_i^T$ converge to zero due to orthogonality of the eigenvectors.

6.4 Example of image compression using GHA

An image to be compressed is a `rr×cc` sub-image from `'gatlin'`:

```
load gatlin
rr = 120 ;  cc = 180 ;  % numbers of rows and columns of Img
Img = X((1:rr)+20, (1:cc)+20) ;
figure(1), image(Img), colormap(map)
```

The image is divided into $r \times c$ blocks, each nth block being converted into a $p = r \times c$ component vector $\mathbf{x}(n)$. These vectors are stored in a $p \times N$ matrix X:

```
r = 4 ; c = 4 ; p = r*c ;
X = blkM2vc(Img, [r c]) ;
[p N] = size(X) ;        % X  is  16 by 1350  = 120 by 180
```

The next step is to remove the mean from X, that is, to normalize it. The pattern matrix X is now ready to be used in the learning algorithm:

```
Xm = mean(X')' ;
X = X - Xm(:, ones(1, N)) ;
X = X/max(max(abs(X))) ;
```

The internal learning loop goes through all patterns from X (one epoch) updating weights in a 'pattern mode'. In order to monitor the convergence process the length of the weight vectors, `lw`, is calculated. It is expected that when the weights converge to respective eigenvectors their lengths will be unity. This condition is checked in the outer loop.

```
m = 4 ;                      %  number of neurons
W = 0.6*(rand(m, p)-0.5) ;  % weight initialisation
lw = sum((W.^2)') ;          % length of weight vectors
W2 = zeros(N, m) ;  % changes to the length of weight vectors
figure(2)
eta = 2e-2 ;         % learning gain
er = .05 ;           % the length convergence error
ep = 0 ;             % number of training epochs
while (sum( abs(1-lw) < er ) < m) & (ep < 16)
  [rs rn] = sort(rand(1, N)) ;
  for n = 1:N
    x  = X(:, rn(n)) ;   % randomised selection of patterns
    y  = W*x ;
    dW = eta*(y*x' - tril(y*y')*W) ;
    W  = W + dW ;
    lw = sum((W.^2)') ;    % length of weight vectors
    W2(n, :) = lw ;
  end
  plot(W2),
  ep = ep+1 ;
  if  ep == 1
    plot(W2),  axis([0 1400 0 1.1])
    title(['lengths of weight vectors during training'])
    xlabel('pattern number')
  end
  grid on, drawnow
end
plot(W2),  axis([0 1400 0.5 1.1])
title(['lengths of weight vectors after ',num2str(ep),'epochs'])
xlabel('pattern number (the last epoch)'),  grid on, drawnow
```


It the above example, there are $m = 4$ neurons, that is, the neural network is trained for only $m = 4$ out of $p = 16$ 'principal directions', specified by the eigenvectors of the input correlation matrix.

After one pass through the training data only the first weight vector representing the most significant eigenvector has converged to achieve the unity length as demonstrated in Figure 6–6.

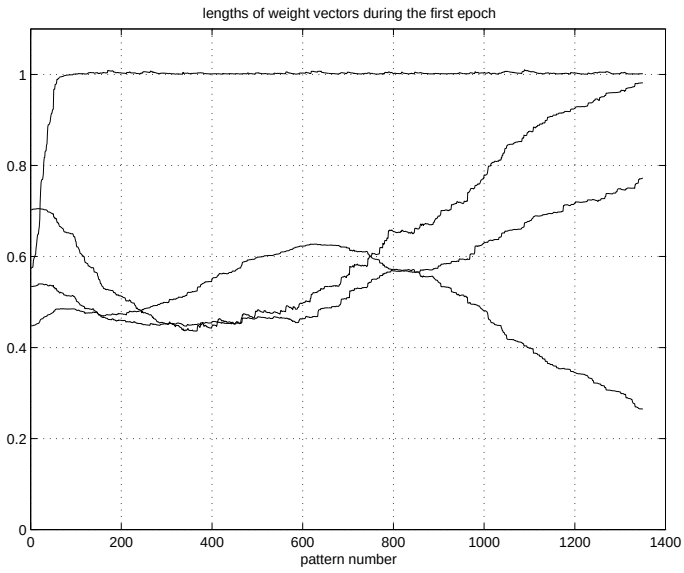


Figure 6–6: The length of the weight vectors representing the m most significant principal directions.

After `ep = 7` epochs, all $m = 4$ weight vectors have attained the unity length within the error specified

by `er` as presented in Figure 6–7.

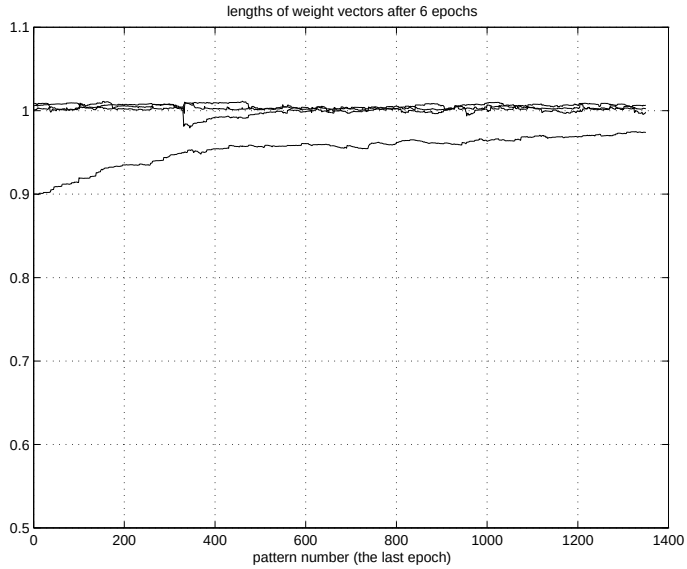


Figure 6–7: The length of the weight vectors representing the m most significant principal directions.

It is possible to observe that during training the weight vectors converge in a serial way, one by one, starting from the weight vector representing the most significant principal direction.

Next we will check that the weight vectors are really equal to the eigenvectors of the input correlation matrix, $R = X \cdot X'$. A number of aspects need to be taken into account in this comparison.

First, the `eig` function arranges the eigenvectors as the column vectors whereas the weights are row vectors.
Secondly, the eigenvalues must be sorted in the descending order and the respective eigenvectors must be re-arranged accordingly.
Thirdly, the orientation of the eigenvectors and the weight vectors might be opposite. This can be done as follows:

```
R = (X*X')/N ;  % the autocorrelation matrix
[V D] = eig(R) ;
dd = diag(D)
d = flipud(dd(p-m+1:p, :)) ;
VV = fliplr( V(:, p-m+1:p)) ;
WV = W*VV
```

Now, `d` contains m largest eigenvalues, and `VV` is a $p \times m$ matrix of respective eigenvectors of the input correlation matrix.
In order to verify that the weight matrix has converged to m principal directions we calculate all possible inner products between all m weight vectors and m eigenvectors. The $m \times m$ matrix `WV` stores these products. The diagonal terms of this matrix should ideally be equal to ± 1, whereas the off-diagonal terms should be zero. In our example, we have;

```
WV = W*VV = 1.0004   0.0308  -0.0029   0.0164
            0.0059   0.9972  -0.0465  -0.0316
            0.0039  -0.0199  -1.0029  -0.0001
           -0.0022   0.0353   0.0100  -0.9860
```

which looks as a sensible approximation.

The next test is to check that the variance of the output signals is equal to the eigenvalues of the input correlation matrix.

```
yy = W*X ;  % the output signals  m by N
vy = var(yy')' ;
  [   d        vy    ]
    2.2875    2.2913
    0.1575    0.1571
    0.1138    0.1147
    0.0424    0.0415
```

Indeed, the approximation seems to be satisfactory.

In order to obtain the compressed image, the matrix of output vectors Y= `yy` is transformed first into a reconstructed input matrix $\hat{X}$= `Xr`

$$\hat{X} = W' \cdot Y \tag{6.18}$$

The reconstructed input matrix can now be re-arranged into $r \times c$ image blocks. This can be done in the following way:

```
Xr = W'*yy ;
Imr = vc2blkM(Xr, r, rr) ;
Imr = round(mc*Imr/max(max(Imr))) ;
figure(3), image(Imr), colormap(gray(mc))
```

The original and compressed images are shown in Figure 6–8.

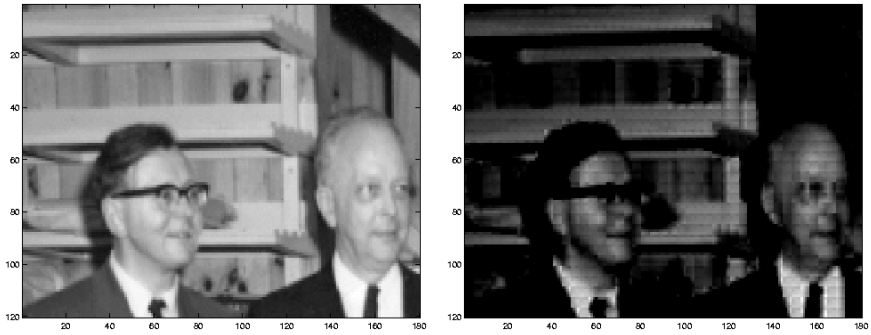


Figure 6–8: The original and compressed images.

7 Competitive Neural Networks

The basic competitive neural network consists of two layers of neurons:

- The similarity-measure layer,
- The competitive layer, also known as a "Winner-Takes-All" (WTA) layer

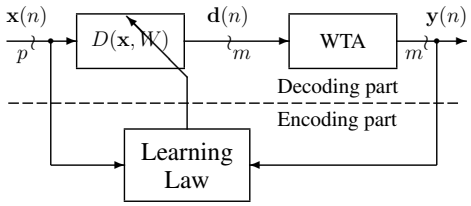


Figure 7–1: The structure of the competitive neural network

The **similarity-measure** layer contains an $m \times p$ weight matrix, W, each row associated with one neuron.

This layer generates signals $\mathbf{d}(n)$ which indicate the similarity between the current input vector $\mathbf{x}(n)$ and each synaptic vector $\mathbf{w}_j(n)$.

The **competitive layer** generates m binary signals y_j. This signal is asserted "1" for the neuron j-th winning the competition, which is the one for which the distance signal d_j attains minimum.

In other words, $y_j = 1$ indicates that the j-th weight vector, $\mathbf{w}_j(n)$, is most similar to the current input vector, $\mathbf{x}(n)$.

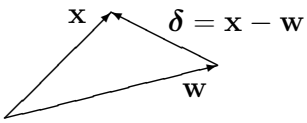
7.1 The similarity-Measure Layer

The detailed structure of the similarity-measure layer depends on the specific measure employed.
Let

$$d = D(\mathbf{x}, \mathbf{w})$$

denote a distance, or similarity measure between two vectors, $\mathbf{x}$ and $\mathbf{w}$. The following measures can be taken into considerations:

- The most obvious similarity measure is the **Euclidean norm**, that is, the magnitude of the difference vector, $\boldsymbol{\delta}$,



$$d = ||\mathbf{x} - \mathbf{w}|| = ||\boldsymbol{\delta}|| = \sqrt{\delta_1^2 + \ldots + \delta_p^2} = \sqrt{\boldsymbol{\delta}^T \cdot \boldsymbol{\delta}}$$

  Such a measure is relatively complex to calculate.

- The square of the Euclidean norm:

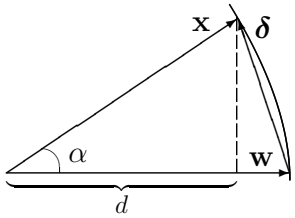
$$d = ||\mathbf{x} - \mathbf{w}||^2 = ||\boldsymbol{\delta}||^2 = \sum_{j=1}^{p} \delta_j^2 = \boldsymbol{\delta}^T \cdot \boldsymbol{\delta}$$

  The square root has been eliminated, hence calculations of the similarity measure have been simplified.

- The Manhattan distance, that is, the sum of absolute values of the coordinates of the difference vector

$$d = \sum_{j=1}^{p} |\delta_j| = \mathrm{sum}(\mathrm{abs}(\boldsymbol{\delta}))$$

- The **projection** of $\mathbf{x}$ on $\mathbf{w}$. This is the simplest measure of similarity of the **normalised** vectors:



$$d = \frac{\mathbf{w}^T}{||\mathbf{w}||} \cdot \mathbf{x} = ||\mathbf{x}|| \cdot \cos\alpha$$

For normalised vectors, when $||\mathbf{w}|| = ||\mathbf{x}|| = 1$

we have

$$d = \cos\alpha \in [-1, +1]$$

and also

if	$d = +1$	then	$\|\|\boldsymbol{\delta}\|\| = 0$	vectors are identical
if	$d = 0$	then	$\|\|\boldsymbol{\delta}\|\| = \sqrt{2}$	vectors are orthogonal
if	$d = -1$	then	$\|\|\boldsymbol{\delta}\|\| = 2$	vectors are opposite

In general, we have

$$||\boldsymbol{\delta}||^2 = (\mathbf{x} - \mathbf{w})^T(\mathbf{x} - \mathbf{w}) = \mathbf{x}^T\mathbf{x} - 2\mathbf{x}^T\mathbf{w} + \mathbf{w}^T\mathbf{w} = ||\mathbf{x}||^2 + ||\mathbf{w}||^2 - 2\mathbf{w}^T\mathbf{x}$$

Hence, for normalised vectors, the projection similarity measure, d, can be expressed as follows

$$d = \mathbf{w}^T\mathbf{x} = 1 - \frac{1}{2}||\boldsymbol{\delta}||^2$$

Normalisation of the input vectors can be achieved without any loss of information by adding another dimension to the input space during preprocessing of the input data.

Assume that $(p-1)$-dimensional input vectors, $\hat{\mathbf{x}}(n)$, have been already scaled into the $[-1, +1]$ range, that is,

$$||\hat{\mathbf{x}}(n)|| \leq 1 \quad \forall (n = 1, \ldots, N)$$

If we add the pth component, x_p to $\hat{\mathbf{x}}$, we obtain a p-dimensional vector, $\mathbf{x}$ and we can write the following relationship

$$||\mathbf{x}(n)||^2 = ||\hat{\mathbf{x}}(n)||^2 + x_p^2(n)$$

Now, in order to normalise all p-dimensional vectors, $\mathbf{x}$, the pth components must be calculated as follows

$$x_p^2(n) = 1 - ||\hat{\mathbf{x}}(n)||^2 \quad \forall (n = 1, \ldots, N)$$

This operation is equivalent to the projection of the input data from the $(p-1)$-dimensional hyper-plane up onto the p-dimensional unity hyper-sphere, as illustrated in Figures 7–2 and 7–3.

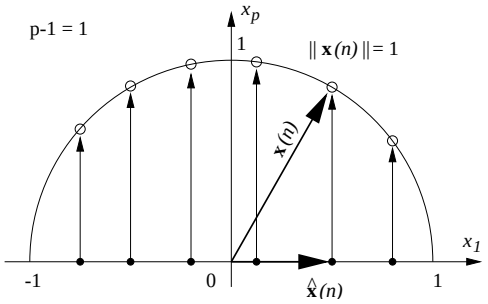


Figure 7–2: Normalisation of 1-D input data by projection onto a unity circle

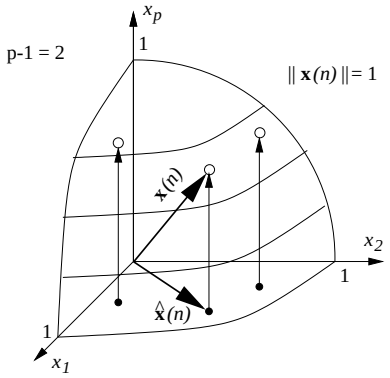


Figure 7–3: Normalisation of 2-D input data by projection onto a unity sphere

For the **normalised vectors** the similarity-measure layer is linear, that is,

$$\mathbf{d} = W \cdot \mathbf{x}$$

The structure of the competitive neural network which employs projections as the similarity measures is illustrated in Figure 7–4 in the form of a signal-flow block-diagram and the dendritic diagram.

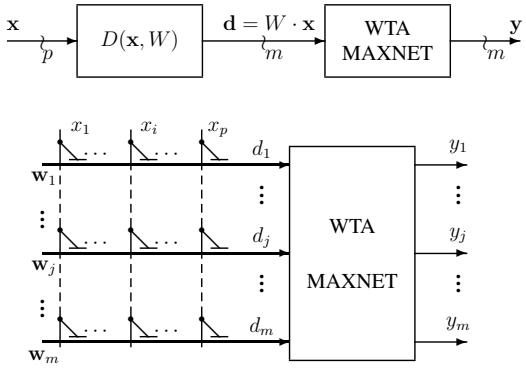


Figure 7–4: The structure of the similarity-measure layer for the normalised vectors

The greater the signal $d_j(n)$ is, the more similar is the jth weight vector, $\mathbf{w}_j$ to the current input signal $\mathbf{x}(n)$.

7.2 The Competitive Layer

The competitive layer, also known as the MinNet (MaxNet), or the "Winner-Takes-All" (WTA) network, generates binary output signals, y_j, which, if asserted, point to the winning neuron, that is, the one with the weight vector being closest to the current input vector:

$$y_j = \begin{cases} 1 & \text{if} \quad j = \arg\min\limits_k D(\mathbf{x}, \mathbf{w}_k) \\ 0 & \text{otherwise} \end{cases}$$

In other words, the MaxNet (MinNet) determines the largest (smallest) input signal, $d_j = D(\mathbf{x}, \mathbf{w}_j)$.

The competitive layer is, in itself, a **recurrent** neural network with the predetermined and fixed feedback connection matrix, M. The matrix M has the following structure:

$$M = \begin{bmatrix} 1 & & \\ & \ddots & -\alpha \\ & -\alpha & \ddots \\ & & & 1 \end{bmatrix}$$

where $\alpha < 1$ is a small positive constant. Such a matrix describes a network with a local unity feedback, and a feedback to other neurons with the connection strength $-\alpha$.

The MaxNet network

The MaxNet network is an implementation of the competitive layer described by the following block diagram

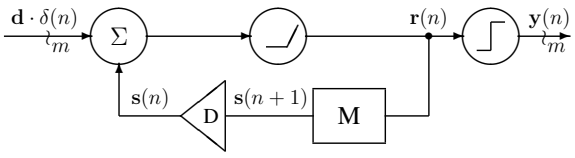


Figure 7–5: The block-diagram of the MaxNet network (competitive layer)

The input vector, $\mathbf{d}$, is active only at the initial time, $n = 0$, which is accomplished by means of the delta function

$$\delta(n) = \begin{cases} 1 & \text{for} \quad n = 0 \quad \text{(initial condition)} \\ 0 & \text{for} \quad n \neq 0 \end{cases}$$

The state equation, which describes how the state vector, $\mathbf{s}(n)$, evolves with time, can be written in the following form

$$\mathbf{s}(n+1) = M \cdot \mathbf{r}(n) + \mathbf{d} \cdot \delta(n)$$

where, the feedback signals $\mathbf{r}(n)$, are formed by clipping out the negative part of the state signals, that is,

$$r_j(n) = \max(0, s_j(n)) = \begin{cases} s_j(n) & \text{for} \quad s_j(n) \geq 0 \\ 0 & \text{for} \quad s_j(n) < 0 \end{cases} \quad \text{for} \quad j = 1, \ldots, m$$

Finally, the binary output signals $\mathbf{y}(n)$, are formed as follows

$$y_j(n) = \begin{cases} 1 & \text{for } r_j(n) > 0 \\ 0 & \text{for } r_j(n) = 0 \end{cases} \quad \text{for } j = 1, \ldots, m$$

Alternatively, we can evaluate the state signals as:

$$\begin{aligned} \mathbf{s}(1) &= \mathbf{d} \\ \mathbf{r}(n) &= \max(0, \mathbf{s}(n)) \\ s_j(n+1) &= r_j(n) - \alpha \sum_{k \neq j} r_k \text{ , for } n > 1 \text{ and } j = 1, \ldots, m \end{aligned}$$

At each step n, signals $s_j(n+1)$ consists of the self-excitatory contribution, $r_j(n)$, and a total lateral inhibitory contribution, $\alpha \sum_{k \neq j} r_k$.

After a certain number of iterations all r_k signals but the one associated with the largest input signal, d_j, are zeros.

Example

Let $\alpha = 0.2$ and $m = 8$. The feedback signals $\mathbf{r}(n)$ can be calculated as follows:

$\mathbf{r}(1)$	$-$	$\mathbf{r}(2)$	$-$	$\mathbf{r}(3)$	$-$	$\mathbf{r}(4)$	$-$	$\mathbf{r}(5)$	$-$	$\mathbf{r}(6)$
7.3	6.98	.32	.99	0	$*$	0	$*$	0	$*$	0
4.2	7.60	0	$*$	0	$*$	0	$*$	0	$*$	0
9.6	6.52	3.08	.44	2.64	.24	2.4	.13	2.27	.04	2.23
.7	8.30	0	$*$	0	$*$	0	$*$	0	$*$	0
5.5	7.34	0	$*$	0	$*$	0	$*$	0	$*$	0
2.9	7.86	0	$*$	0	$*$	0	$*$	0	$*$	0
8.6	6.72	1.88	.68	1.20	.53	.67	.48	.19	.45	0
3.4	7.76	0	$*$	0	$*$	0	$*$	0	$*$	0

The columns marked '$-$' represent the total inhibitory contribution.

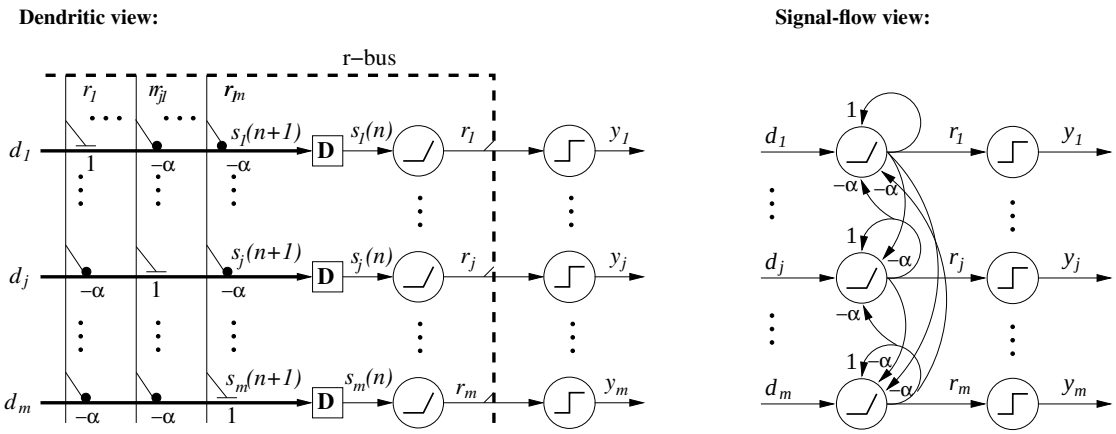


Figure 7–6: The dendritic and the signal-flow views of the competitive layer

Note the unity self-excitatory connections, and the lateral inhibitory connections.

7.3 Unsupervised Competitive Learning

The objective of the competitive learning is to adaptively quantize the input space, that is, to perform **vector quantization** of the input space

It is assumed that the input data is organised in, possibly overlapping, **clusters**. Each synaptic vector, $\mathbf{w}_j$, should converge to a **centroid** of a cluster of the input data.

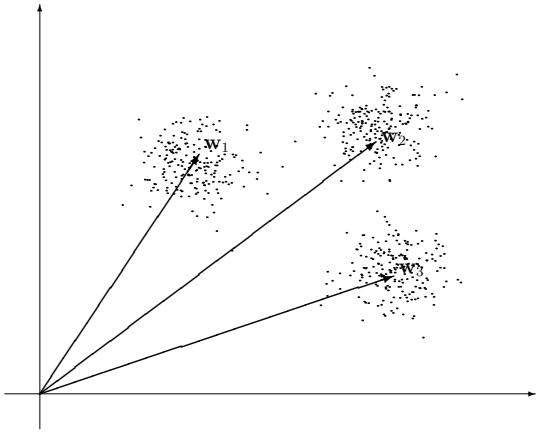


Figure 7–7: An example of a 2-D pattern with three clusters of data

In other words, the input vectors are categorized into m classes (clusters), each weight vector representing the center of a cluster.

It is said that such a set of weight vectors describes **vector quantization** also known as **Voronoi** (or **Dirichlet**) **tessellation** of the input space.

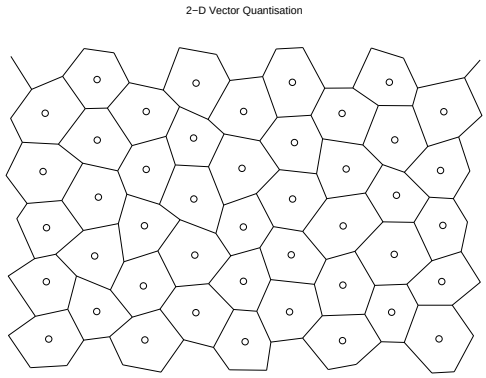


Figure 7–8: Example of Voronoi tessellation (vector quantization) of a 2-D space.

The space is partitioned into polyhedral regions with centres represented by weight vectors (dots in Figure 7–8).
The boundaries of the regions are planes perpendicularly bisecting lines joining pairs of centres (prototype vectors) of the neighbouring regions.

A very important application of the vector quantization is in data coding/compression. In this context the set of weights (prototype vectors) is referred to as a **codebook**.

We can find a set of prototype vectors with competitive learning algorithms.

A simple competitive learning

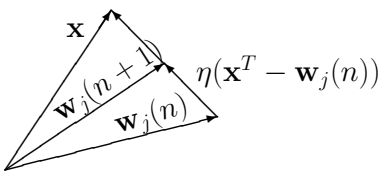
A simple competitive learning can be describe as follow:

- Weight vectors are usually initialise with n randomly selected input vectors:

$$\mathbf{w}_j(0) = \mathbf{x}^T(\mathrm{rand}(j))$$

- For each input vector, $\mathbf{x}(n)$, determine the winning neuron, j for which its weight vector, $\mathbf{w}_j(n)$, is closest to the input vector. For this neuron, $y_j(n) = 1$.
- Adjust the weight vector of the winning neuron, $\mathbf{w}_j(n)$, in the direction of the input vector, $\mathbf{x}(n)$; do not modify weight vectors of the loosing neurons, that is,

$$\Delta\mathbf{w}_j(n) = \eta(n)y_j(n)(\mathbf{x}^T(n) - \mathbf{w}_j(n))$$



- In order to arrive at a static solution, the learning rate is gradually linearly reduced, for example

$$\eta(n) = 0.1(1 - \frac{n}{N})$$

Example — scripts Cmpti.m, Cmpts.m

In this example we consider vector quantization of the 2-D input space into $m = 5$ regions specified by m weight vectors arranged in a 5×2 weight matrix W.

We start with generation of a 2-D pattern consisting of m clusters of normally distributed points. Weights are initialised with points from the data matrix X.

```
%  Cmpti.m
% Initialisation of competitive learning
p = 2; m = 5 ;      % p inputs, m  outputs
clst = randn(p, m); %  cluster centroids
Nk = 100;       % points per cluster
N = m*Nk ;      % total number of points
sprd = 0.2 ;    % a relative spread of the Gaussian "blobs"
X = zeros(p,N+m); % X is  p by m+N input data matrix
wNk = ones(1, Nk);
for k = 1:m  % generation of m Gaussian "blobs"
  xc = clst(:,k) ;
  X(:,(1+(k-1)*Nk):(k*Nk))=sprd*randn(p,Nk)+xc(:,wNk) ;
end
[xc k] = sort(rand(1,N+m));
X = X(:, k) ;      % input data is shuffled randomly
winit = X(:,1:m)'; % Initial values of weights
X = X(:,m+1:N+m);  % Data matrix  is  p by N
```

In the script implementing a simple competitive learning algorithm we pass once over the data matrix, aiming at weights to converge to the centres of Gaussian "blobs".


```
%   Cmpts.m
W = winit ;
V = zeros(N, m, p);  % to store all weights
V(1,:,:) = W ;
wnm = ones(m,1) ;
eta = 0.08 ;    % learning gain
deta = 1-1/N ;  % learning gain decaying factor
for  k = 1:N    % main training loop
  xn = X(:,k)' ;
% the current vector is compared to all weight vectors
  xmw = xn(wnm,:)-W ;
  [win jwin] = min(sum((xmw.^2),2));
% the weights of the winning neurons are update
  W(jwin,:) = W(jwin,:) + eta*xmw(jwin,:);
  V(k,:,:) = W;
% eta = eta*deta ;
end

plot(X(1,:),X(2,:),'g.',clst(1,:),clst(2,:),'ro', ...
     winit(:,1),winit(:,2),'bx' , ...
     V(:,:,1), V(:,:,2), 'b', W(:,1),W(:,2) , 'r*'), grid
```

In Figure 7–9 there are four examples of data organised in five overlapping clusters. After one training epoch, the weights converged to centroids of the clusters with varying degree of success.

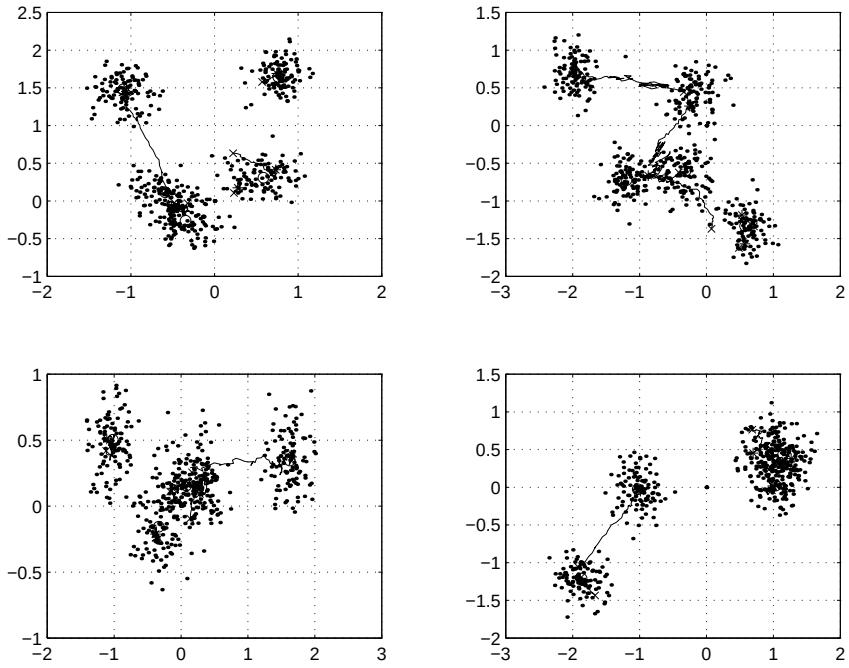


Figure 7–9: Simple competitive learning: ‘o’ – centroids of generated clusters, ‘$\times$’ – initial weights, ‘*’ – Final weights

7.4 Competitive Learning and Vector Quantization

- Competitive learning is used to create a **codebook**, that is a weight matrix, W, which stores the centers of data clusters.

  For a p-dimensional input space and m-neurons (size of the codebook) the structure of the codebook (weight matrix, W) is as follows

cluster #	W	1	$\cdots$	p
1	$\mathbf{w}_1$	w_{11}	$\ldots$	w_{1p}
2	$\mathbf{w}_2$	w_{21}	$\ldots$	w_{2p}
$\vdots$	$\vdots$	$\vdots$	W	$\vdots$
m	$\mathbf{w}_m$	w_{m1}	$\ldots$	w_{mp}

- The codebook (weight matrix) describes the tessellation of the input space known as **vector quantization** (VQ). For a 2-D input space the above concepts can be illustrated as in Figure 7–10.
- The codebook is created from a representative set of data using a competitive learning.

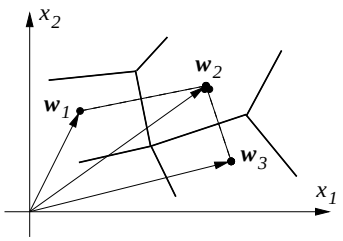


Figure 7–10: Voronoi tessellation (vector quantization) of a 2-D space.

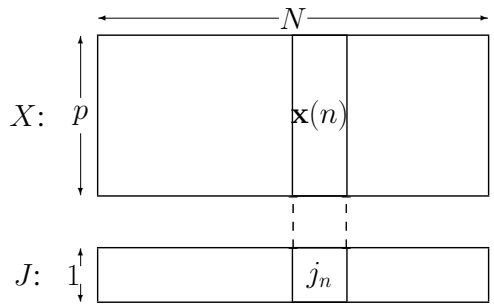
Vector quantization is used in **data compression**.

Given a codebook, $W = [\mathbf{w}_1; \ldots; \mathbf{w}_1]$ the process of data compression can be described as follows:

1. For every input vector, $\mathbf{x}(n)$ find the closest codebook entry, that is the weight vector $\mathbf{w}_{j_n}$, for which the distance

   $$|\mathbf{w}_{j_n} - \mathbf{x}(n)|$$

   attains minimum. Then we identify that the input vector, $\mathbf{x}(n)$ belongs to the j-th cluster.
2. Replace all input data $X = \{\mathbf{x}(n)\}_{1:N}$ by their corresponding indices $J = \{j_n\}_{1:N}$.



3. Transmit J instead of X. The codebook, W must be made known/transmitted to the receiver.
4. At the receiver, using a codebook, replace every j_n with the corresponding $\mathbf{w}_{j_n}$ which will now represent $\mathbf{x}(n)$. The representation error is:

   $$\varepsilon_n = |\mathbf{w}_{j_n} - \mathbf{x}(n)|$$

   The total squared error is

   $$F = \sum_{n=1}^{N} \varepsilon_n^2$$

In order to calculate the **compression ratio**, C, let us assume that

- Weights and input data are represented by B-bit numbers,
- The length of the codebook

$$m \leq 2^b$$

that is, all cluster indices are b-bit numbers

Then,

- the total number of bits to represent the input data, that is, the size of X is

$$B \times p \times N$$

- The size of the compressed data, J, is

$$b \times N$$

- The size of the codebook, W, is

$$B \times p \times m$$

- The compression ratio is:

$$C = \frac{BpN}{bN + Bpm} \approx \frac{Bp}{b}$$

For example, for $B = 16$, $p = 12$, $b = 8$

$$C \approx 24$$