CSE2330 Introduction to Computational Neuroscience

Tutorial/Assignment 3: Learning and Self-Organization

3.1 About this tutorial

The objective of this tutorial is to introduce concepts of learning and self-organization and practice the computational aspects of these concepts.

Contents

getting started ...

The amount of information and the number of technical terms, computational methods and concepts that you are expected to master increase quickly, therefore, you are strongly advised to come well prepared to the practical classes.

Read at least the previous prac manuals, but you cannot go wrong if you also read related lecture notes and book chapters.

3.2 Hebbian learning

From the lectures we remember the fundamental Hebb's law. Donald Hebb stated in 1949:
"When an axon of cell A is near enough to excite a cell B and repeatedly takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased."
When Hebb made his statement it was a hypothesis. It has since been verified in some parts of the brain so the term "law" is now legitimate.

3.2.1 Illustrative example

To incorporate Hebb's law in our computer simulations we must give it a mathematical formulation.

We begin our study with a simple network consisting of the two neurons A and B and their connecting synapse. The neuron on the presynaptic side, A, has the efferent signal x on its axon and the neuron on the postsynaptic side, B, has the efferent signal y on its axon. The synapse is assumed to have the strength w as illustrated in Figure 1.

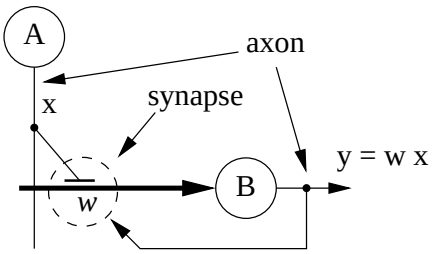


Figure 1: Illustration of the Hebbian learning

Formalizing the Hebb's law we say that **learning is based on modification of synaptic weights** depending both on the afferent and efferent signals and possibly on the current values of weights.

We typically start with an additive weight modification in the form:

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

where n denotes the processing step, $w(n+1), w(n)$ are two subsequent values of the weight, and $\Delta w(n)$ is the weight change, or update. This weight modification mechanism that takes place in the synapse is symbolised by the dashed circle in Figure 1.

The obvious first mathematical formulation of Hebb's law is in the form of the product of afferent and efferent signals as follows:

$$\Delta w = \alpha\, x\, y \tag{2}$$

where α is the "learning rate". When both x and y are large, i.e., both A and B are firing, w is increased. When one or both of x and y are small, w is changed very little. This may be seen as formalization of Long Term Potentiation (LTP).

It is obvious that if A and B keep firing together in time during a lifetime, w would grow very large. This could not be biologically sustained and it would not make any sense anyway.

One contribution of computational neuroscience is that it showed that Hebb's law is not sufficient for stable learning. We must introduce a mechanism for decreasing weight w, besause "forgetting" is an integral part of learning.

There are several ways of doing this, one choice is the following rule: "If A and B are not usually simultaneously active, then a synapse connecting them is not doing much good anyway, so let us decrease its w. So if A or B is active but not both, then we decrease w."
One way of formalizing such a statement is the following weight modification rule:

$$\Delta w = \alpha_1 xy - \alpha_2(x + y), \tag{3}$$

where α_1 and α_2 are appropriately chosen, as discussed below. Eqn (3) may be seen as an attempt of formalization of Long Term Depression (LTD).

We are now ready to test the first Hebbian learning law given in eqn (3) in MATLAB. Invoke MATLAB, open an editor window, and create the following m-file that you should save in a file `myprac3a.m`, or something similar:

```
% myprac3a.m
%            app
x = 0.5 ;   % mV are typical units
alf1 = 0.5 ; alf2 = 2;
nn = 8;   % number of iterations
w = zeros(1, nn+1) ; % to store all values of weights
w(1) = 1 ;  % initial value of the weight
for n = 1:nn
   y = w(n)*x ;
   w(n+1) = w(n) +  alf1*x*y - alf2*(x+y) ;
end
figure(1)  % visualisation of the results
plot(0:n, w), grid on
title('Simple stable Hebbian learning')
ylabel('w (weight)'), xlabel('n [iteration#]')
% print -f1 -depsc2 p3Hebb2
```

This script should produce the following plot showing how the weight values evolve. Note that for a given values of parameters the weight reaches very quickly its steady-state value. If a weight settles at a fixed value, we call it a **stable learning**. Since the steady-state value is negative, our synapse is **inhibitory**. Try to increase value of `x` by 4, and you will see that $-w$ will grow up indefinitely.

Exercise 3. 1
Observe the weight change for different values of α_1, α_2. Identify three pairs of values that characterize well the behaviour of the circuit (in your opinion)

□

With a little bit of basic mathematics we can find out why the weight evolves as it does. Combining eqns (1) and (3) and taking into account that $y = w \cdot x$, we can re-write the weight modification law as:

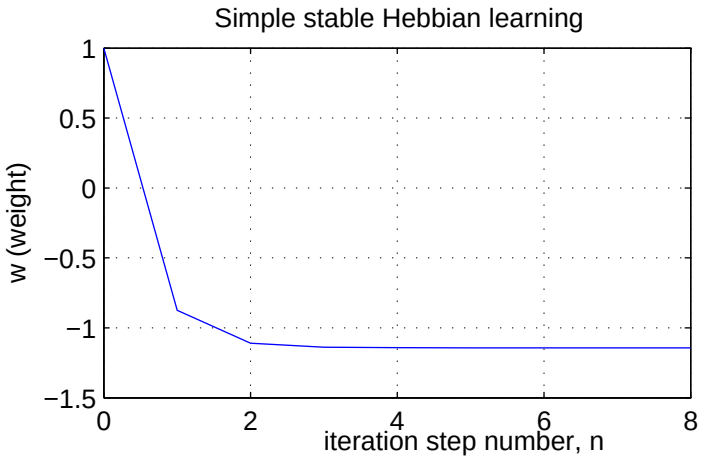


Figure 2: Evolution of weight values in a simple Hebbian learning

$$w(n+1) = w(n) + \alpha_1 x^2 w(n) - \alpha_2 x(1 + w(n)) \tag{4}$$

Re-arranging and grouping required terms together we can further write:

$$w(n+1) = (\alpha_1 x^2 - \alpha_2 x + 1)w(n) - \alpha_2 x$$

which can be simply written as:

$$w(n+1) = aw(n) - \alpha_2 x\ , \quad \text{where } a = \alpha_1 x^2 - \alpha_2 x + 1 \tag{5}$$

Eqn (5) is a familiar **geometric progression** with ratio a. Now we know that in order for such a progression not to grow indefinitely, its ratio must be less that one, or more precisely

$$|a| < 1$$

It is also easy to calculate the **steady state** value of the weight, when we assume that in a steady state we clearly have $w_s = w(n+1) = w(n)$. Substituting it into eqn (5) gives the steady state value of the weight:

$$w_s = \frac{-\alpha_2 x}{1 - a}$$

To check it, let us calculate in MATLAB

```
a = x*(alf1*x - alf2) + 1
ws = -alf2*x/(1-a)
```

I got the values: `a = 0.1250` and `ws = -1.1429`, which is consistent with the plot in Figure 2. Note that with a given ratio of the geometric progression, every step, the weight change will be reduced by the factor of $a = 0.125 = 1/8$.

Exercise 3. 2
Calculate the steady-state values of the weight for three sets of parameters as in Exercise 3.1.

□

This introductory example demonstrates a principle of **self-organization**:

- after the process of learning stabilizes, the synaptic weight has a value depending on the afferent signal(s).

We will demonstrate how various models of self-organization capture the essential aspects of data.

3.2.2 Basic structure of Hebbian learning neural networks

A more general block diagram illustrating a general concept of Hebbian learning is given in Figure 3. The upper section of the network, called sometimes the **decoding part**, is a single layer

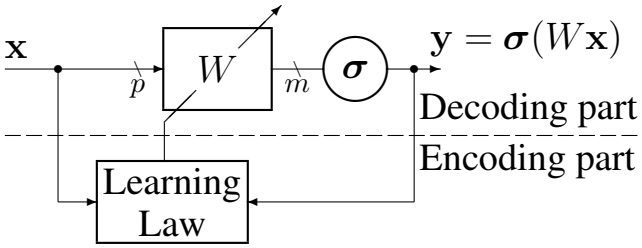


Figure 3: A block-diagram of a basic Hebbian learning neural network

network specified by an $m \times p$ weight matrix, W. As you remember each row of the matrix is associated with one neuron. The activation function σ keeps values of the signals between 0 and 1. For simplicity, we consider first a linear activation function.
The learning law is implemented by the **encoding part** of the network and the block-diagram illustrates that the **modification of weights during learning** is a function of afferent and efferent signals.
Generalizing the Hebbian learning law of eqn (4) we can write

$$w_{ji}(n+1) = f(w_{ji}(n), y_j(n), x_i(n)) \tag{6}$$

which means that, in general, the next value of weight is a function of the current value of weight and afferent and efferent signals.
Note that two signals, y_j and x_i and weight w_{ji} are locally available at the ji synapse, therefore, we often say that Hebbian learning law is an example of a **local learning law**.

The concept of the **local learning** law is further illustrated in Figure 4. Note that each synapse, represented by a dashed circle, receives locally available afferent and efferent signals, say x_i, y_j, so that its weight w_{ji} can be modified according to a specified learning law. The local feedback provides the efferent signal, y_j to all synapses along the neuron's dendrite. This feedback is essential for a learning process to occur.

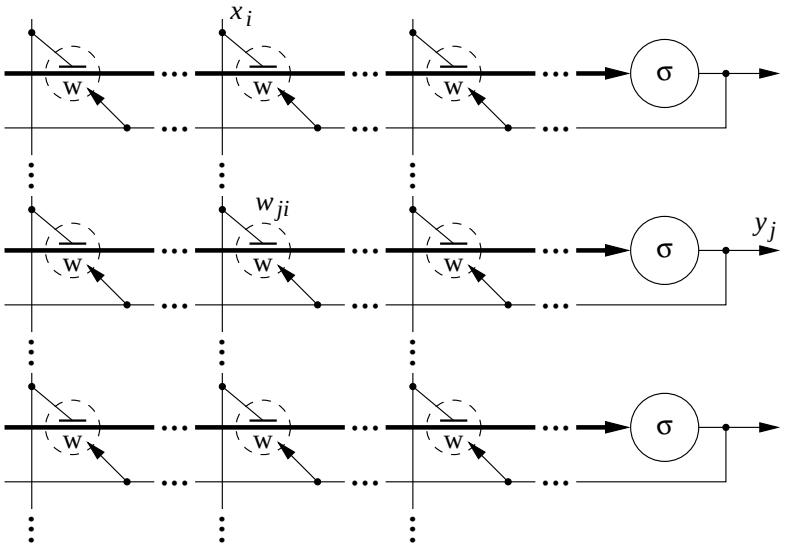


Figure 4: A neural network with a local learning law

3.2.3 Stable Hebbian learning. Oja's rule

One of the important forms of a stable Hebbian learning law, know as Oja's rule uses the **augmented afferent signals**, $\tilde{x}_i$ and can be written in the following form:

$$\Delta w_i = \alpha \cdot y \cdot \tilde{x}_i \,, \quad \text{where} \quad \tilde{x}_i = x_i - y \cdot w_i \tag{7}$$

Hence the input signal is augmented by a product of the output signal and the synaptic weight. The Oja's rule is primarily formulated for a single neuron with p synapses and can be written in a vectorised form in the following way:

$$\Delta \mathbf{w} = \alpha \cdot y \cdot \tilde{\mathbf{x}}^T = \alpha \cdot y \cdot (\mathbf{x}^T - y \cdot \mathbf{w}) \,, \quad \text{where} \quad \tilde{\mathbf{x}}^T = \mathbf{x}^T - y \cdot \mathbf{w} \,, \quad \text{and} \quad y = \mathbf{w} \cdot \mathbf{x} \tag{8}$$

where

$\Delta \mathbf{w} = \mathbf{w}(n+1) - \mathbf{w}(n)$	is a modification of the weight vector,
$\mathbf{w} = [w_1 \dots w_p]$	is a p-component weight row vector,
$\mathbf{x} = [x_1 \dots x_p]^T$	is a p-component column vector of afferent signals
$\tilde{\mathbf{x}} = [\tilde{x}_1 \dots \tilde{x}_p]^T$	is a p-component column vector of augmented afferent signals.
y	is the efferent signal created as an inner product of weights and afferent signals.

The important property of the learning law as in eqn (8) is the fact that the **length (Euclidian norm) of the weight vector tends to unity**, that is,

$$||\mathbf{w}|| \longrightarrow 1 \tag{9}$$

This important and useful condition means that Oja's rule is a stable learning law.

3.2.4 Self-organization I

Let us re-state that self-organization means that synaptic weights and possible efferent signal(s) capture some important properties of the afferent signals.

Assume that a collection of afferent signals can represent a **shape**. An example of such a simple shape is given in Figure 5. To create this shape execute the following MATLAB commands (prepare a relevant script) as in Figure 5.

```
xx = [4 3 5 2 1 6 3 3 2
      1 7 5 4 6 3 2 4 7] ;
figure(1)
plot(xx(1,:), xx(2,:), 'd')
grid on, axis([0 10 0 8])
title('A simple 2-D shape')
xlabel('x_1'), ylabel('x_2')
% print -f1 -depsc2 p3Shape
nn = max(size(xx))
```

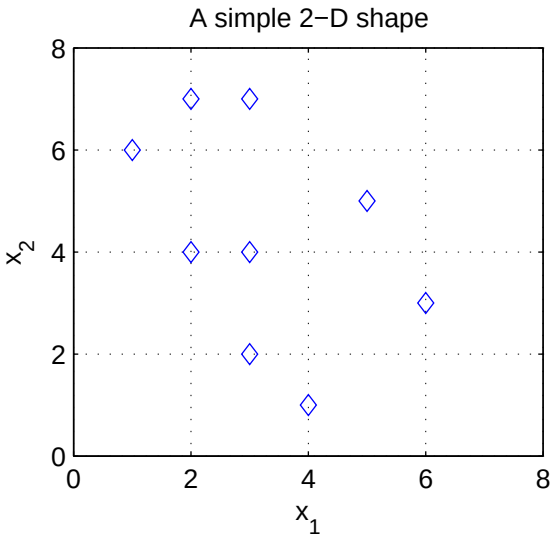


Figure 5: A simple shape in two-dimensional space formed from 9 points

We can say that a shape is a collection of N points $\mathbf{x}(n)$ which are arranged into a $p \times N$ matrix X (the matrix `xx` in MATLAB example) as follows:

$$X = [\,\mathbf{x}(1)\;\; \mathbf{x}(2)\;\; \ldots\;\; \mathbf{x}(N)\,]\;, \quad \text{where} \quad \mathbf{x}(n) = \begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix}\;, \quad (p = 2) \tag{10}$$

Now we will try to analyze the behaviour of a neuron implementing Oja's rule based learning when presented with afferent signals representing the shape as in Figure 5. Since our shape is two-dimensional, we need a neuron with just two synapses $(p = 2)$ as in Figure 6. Note that the

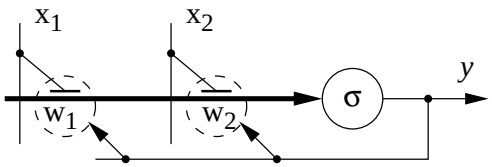


Figure 6: A neuron with two synapses and a local learning law

weight vector is also two-dimensional:

$$\mathbf{w} = [\; w_1\;,\;\; w_2\;]$$

Having all the background definitions we can now construct a simple MATLAB script which implements the Oja's rule as in eqn (8). We will apply points from the shape one-by-one and observe the development of the weight vector. The relevant script follows:

```
% Oja's rule -- single presentation (epoch)
 alf = 0.02 ;   % learning gain
 w = zeros(nn+1,2) ;
 w(1,:) = rand(1,2) ; % random initialization of the weight vector
 for n = 1:nn
   x = xx(:,n) ;   % current point (afferent signals)
   y = w(n,:)*x ;  % efferent signal
   w(n+1,:) = w(n,:) + alf*y*(x' - y*w(n,:)) ; % Oja's rule
 end
 ws = w(end,:)  % final weight matrix
 figure(1)      % presentation of the results
 subplot(1,2,1)
 plot(w(:,1),w(:,2),'-',w(1,1),w(1,2),'>',ws(1),ws(2),'*')
 grid on, xlabel('w_1'), ylabel('w_2')
 subplot(1,2,2)
 plot(0:nn, sqrt(sum(w'.^2)))
 grid on, axis([0 10 0 1.2])
 xlabel('learning steps, n'),
 ylabel('||w||')
 text(-10,1.3,'Evolution of the weight vector and its length')
```

Possible results are presented in Figure 7.

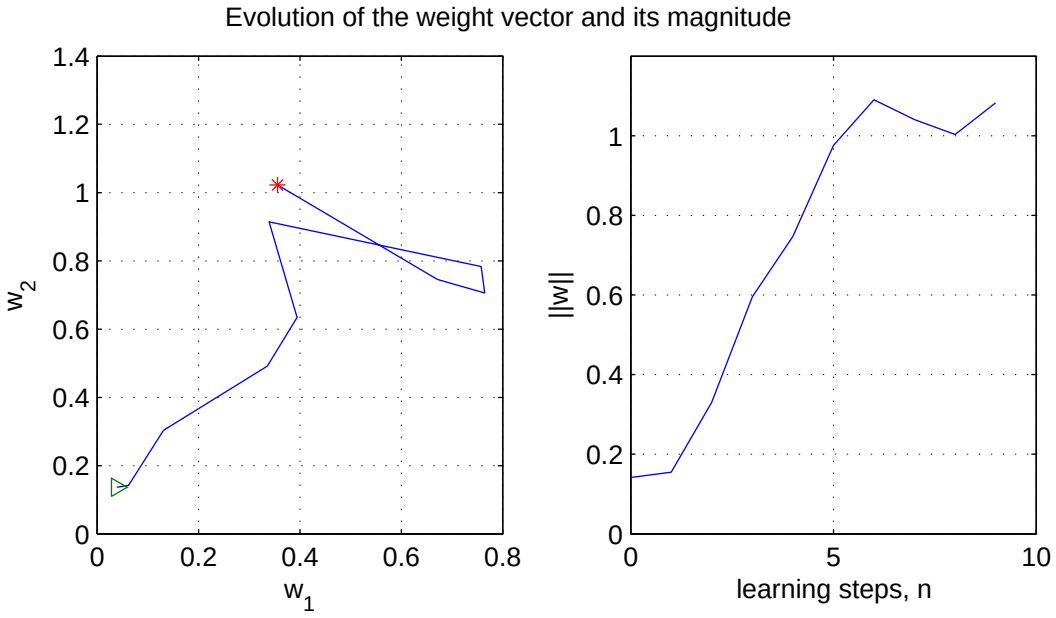


Figure 7: Evolution of the weight vector and its length during one presentation of the shape. The initial and final weights are marked with '▷' and '*', respectively

A few additional comments and explanations regarding the MATLAB script follows:

- Learning gain is often small. It influences the speed and convergence of the learning process.
- Initialization of weights: we start with a randomly selected weight vector. Often, initialization needs to be done carefully having the speed and convergence of the learning process in mind.
- Watch out for all those column and row vectors collected into relevant matrices. It is just confusing, but do not be.

Note from Figure 7 that as predicted in eqn (9) the length of the weight vector seems to be tending to unity. However, the weight vector itself that starts at the position marked with $\triangle$ does not seem to go to any specific location. A possible explanation is that presenting each point from the shape just once is not enough to pick up any pattern in the shape, that is, to achieve self-organization.

We can test this hypothesis by reapplying the points of shape a number of times. Each presentation is often referred to as an epoch. We just add the epoch loop to the previous script:

```
% Oja's rule -- multiply epochs
 kk = 3 ; % number of epochs
 alf = 5e-3 ; % learning gain
 w = zeros(kk*nn+1,2) ;
 w(1,:) = 0.4*rand(1,2) ; % random weight intialization
 k = 1 ;        % step counter
 for ep = 1:kk  % epoch loop
   for n = 1:nn   % point loop
     x = xx(:,n) ;
     y = w(k,:)*x ;
     w(k+1,:) = w(k,:) + alf*y*(x' - y*w(k,:)) ;
     k = k+1 ;
   end  % point loop
 end  % epoch loop
 ws = w(end,:)   % final value of the weight vector
 figure(1)       % visualization of the results
 subplot(1,2,1)
 plot(w(:,1),w(:,2),'-',w(1,1),w(1,2),'>',ws(1),ws(2),'*')
 grid on xlabel('w_1'), ylabel('w_2')
 subplot(1,2,2)
 plot(0:nn*kk, sqrt(sum(w'.^2)))
 grid on, axis([0 kk*nn 0 1.2])
 xlabel('learning steps, n'), ylabel('||w||')
 text(-25,1.25,'Evolution of the weight vector and its length')
% print -f1 -depsc2 p3LOjaEp
```

The results of self-organization are presented in Figure 8.
We can now see from Figure 8 that after presentation of the shape points three times (3 epochs) not only the length of the weight vector, $||\mathbf{w}||$ is very close to unity, but also the weight vector $\mathbf{w}$ itself oscillates around a steady-state value calculated in MATLAB as `ws`.
The value of length can be used to monitor the convergence of the learning process.
Note also from Figure 8 that when the length is close to unity, the weight vector moves on a segment of a **unity circle**.

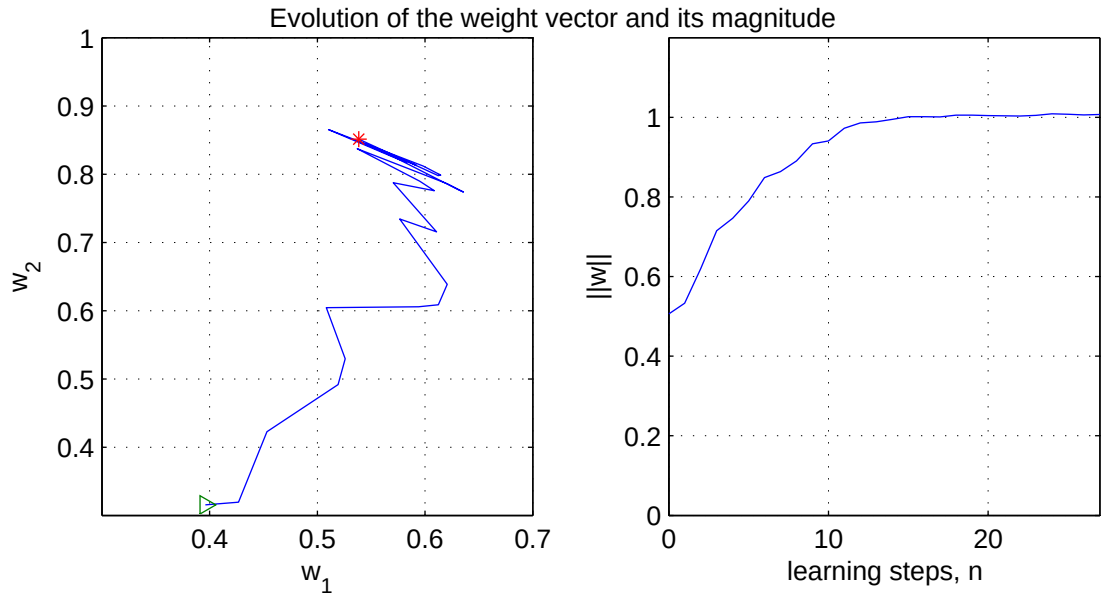


Figure 8: Self-organization in Hebbian learning

The next step is to interpret the weight vector in the input space. For that we plot the line along the final value of the weight vector and get the results presented in Figure 9.

```
figure(2)
plot(xx(1,:),xx(2,:),'d', ...
    [0 9*ws(1)],[0 9*ws(2)],'-', ...
     ws(1),ws(2),'*')
grid on, axis([0 8 0 8])
title('A simple 2-D shape')
text(0.9, 0.8, 'w')
xlabel('x_1'), ylabel('x_2')
```

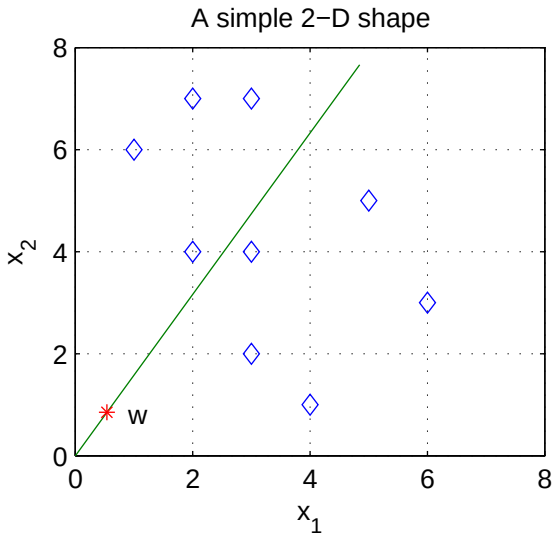


Figure 9: A simple shape with a line along the weight vector

The result does not look spectacular. The only thing we can say is that the weight line apparently bisects the afferent shape, which does not seem to mean too much in itself. The question is, can we do better, and the answer is affirmative:

> **In order to evaluate the shape itself independently of its position in space we have to perform Hebbian self-organization with respect to the centre of the shape.**

Computationally what we have to do is remove **mean** from the shape, that is, to move the coordinate system to the centre of the shape. Is such an operation biologically plausible? Consider that:

- when we look at the shape and try to establish what we are looking for its position is irrelevant.
- Removal of the mean is performed by means of the familiar lateral inhibition with a Mexican hat mask. We will not go into details of such an operation, but it can be done.

3.2.5 Extraction of the principal direction of a shape

Now we need just a small modification to our script to extract the principal direction of the shape, namely, removal of the mean. For convenience I give here a complete script for the extraction.

```
% myprac3b.m
%           app
clear
xS = [4 3 5 2 1 6 3 3 2
      1 7 5 4 6 3 2 4 7] ;
nn = max(size(xS)) ;
mnX = mean(xS,2) ;  % the mean of the shape
xx = xS - mnX(:,ones(1,nn)); % mean removal
         % Oja's rule -- multiply epochs
kk = 5 ;       % number of epochs
alf = 3e-2 ;  % learning gain
w = zeros(kk*nn+1,2) ;
w(1,:) = 0.4*rand(1,2) ;
k = 1 ;
for ep = 1:kk
  for n = 1:nn
    x = xx(:,n) ;
    y = w(k,:)*x ;
    w(k+1,:) = w(k,:) + alf*y*(x'-y*w(k,:)) ;
    k = k+1 ;
  end
end
ws = w(end,:)

figure(1)     % visualisation of the results
subplot(1,2,1)
plot(w(:,1), w(:,2), '-', w(1,1), w(1,2), '>', ws(1), ws(2), '*'),
grid on, axis([-0.6 0.6 0 1])
xlabel('w_1'), ylabel('w_2')

subplot(1,2,2)
plot(0:nn*kk, sqrt(sum(w'.^2))), grid on, axis([0 kk*nn 0 1.2])
xlabel('learning steps, n'), ylabel('||w||')
text(-30,1.25,'Evolution of the weight vector and its length')
```

```
figure(2)
plot(xx(1,:), xx(2,:), 'd', ...
   4*ws(1)*[-1 1], 4*ws(2)*[-1 1],'-', ...
   ws(1), ws(2),'*', 0, 0, '+k'),
grid on, axis([-4 4 -4 4])
title('The principal direction of a 2-D shape')
text(ws(1)+0.2*abs(ws(1)), ws(2), 'w_s')
xlabel('x_1'), ylabel('x_2')
```

Note that in the script:

- The mean has been removed using the command `mean` . It is computationally simpler than lateral inhibition.
- The learning gain and the number of epoch have been adjusted to get nice convergence.

The results of self-organization are presented in Figures 10 and 11.

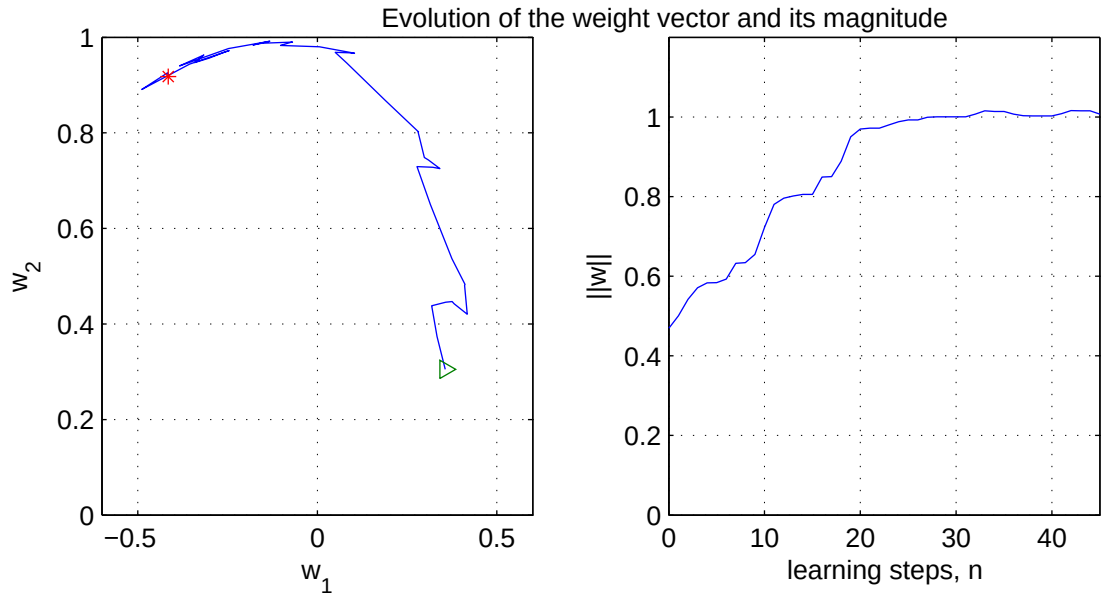


Figure 10: Evolution of the weight vector and its length during self-organization

Note that now the weight line bisects the shape in the direction where it is the most spread, that is, where there is most variation of data. This is called the **principal direction** of the shape.
In addition, the efferent signal y indicates the **variance of data** along the principal direction.

> **Hence, a neuron as in Figure 6, which implements Hebbian learning according to the Oja's rule, is a principal direction detector.**

It is also possible to extend the Oja's rule in such a way that the network is able to discover the next principal direction, orthogonal to the main principal direction.

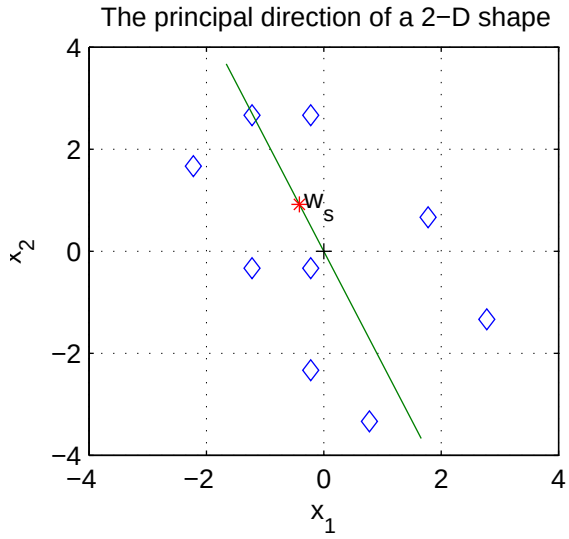


Figure 11: The shape with the final weight line inserted.

Exercise 3. 3
Define your own shape consisting of 12 points and find its principal direction.

□

3.2.6 Concluding remarks

Hebbian learning considered in this section has aimed at organising the synaptic weights in such a way that they approximated the shape of the data. The shape of data can be approximated by its principal direction (components), the first gives the direction of the highest variability of data, the second is orthogonal to the first direction and indicates the highest variability in that direction, and so on.
Mathematically the procedure is known as the principal component analysis.

3.3 Competitive learning

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

The block-structure of the competitive neural network is shown in Figure 12

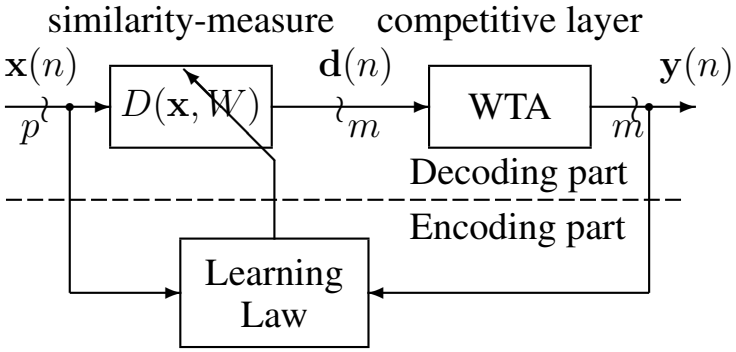


Figure 12: The block-structure of the competitive neural network

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 distances between the current input vector $\mathbf{x}(n)$ and each synaptic vector $\mathbf{w}_j(n)$.

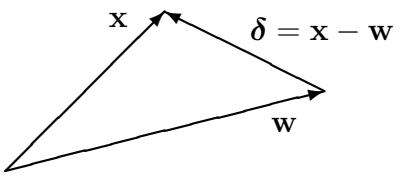
The competitive layer generates m binary signals y_j. This signal is asserted "1" for the neuron j-th winning the competition, which is the one for which the similarity signal d_j attains minimum.

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, $\mathbf{x}(n)$.

3.3.1 The Similarity-Measure Layer

The similarity-measure layer calculate the **similarity** between each stimulus and every weight vector. Such similarity can be measure in a number of possible ways.

- The most obvious way of measuring similarity or the distance between two vectors is the Euclidean norm (length) of the difference vector, $\boldsymbol{\delta}$,



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

The problem is that such a measure of similarity is relatively complex to calculate.

- A simpler way of measuring the similarity between two vectors is just the square of the length of the difference vector

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

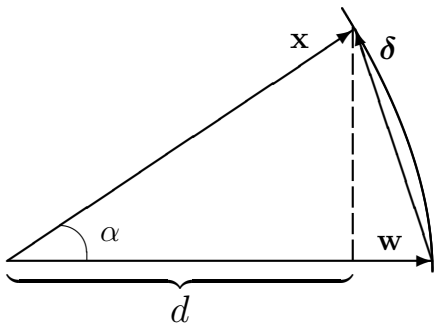
The square root has been eliminated, hence calculations of the similarity measure have been simplified.

- 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 that works particularly well for the **normalised** vectors. The projection is calculated using the inner product of two vectors, namely (assuming both are column 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 as the similarity measure, d, can be expressed in terms of the norm of the difference vector as follows

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

Note that the **greater** the signal d is, the **more similar** is the weight vector, $\mathbf{w}$ to the input signal $\mathbf{x}$.

Exercise 3. 4
Define a 3-component vector $\mathbf{x}$ and calculate all the above measures of similarity between x and the following weight matrix:

$$W = \begin{bmatrix} 2 & 1 & 3 \\ -1 & 2 & 1 \end{bmatrix}$$

□

In summary we say that for the **normalised vectors** the similarity-measure performs the "typical" operation of signal aggregation, that is:

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

The structure of the similarity-measure layer is illustrated in Figure 13 in the form of a typical dendritic diagram and related signal-flow block-diagram.

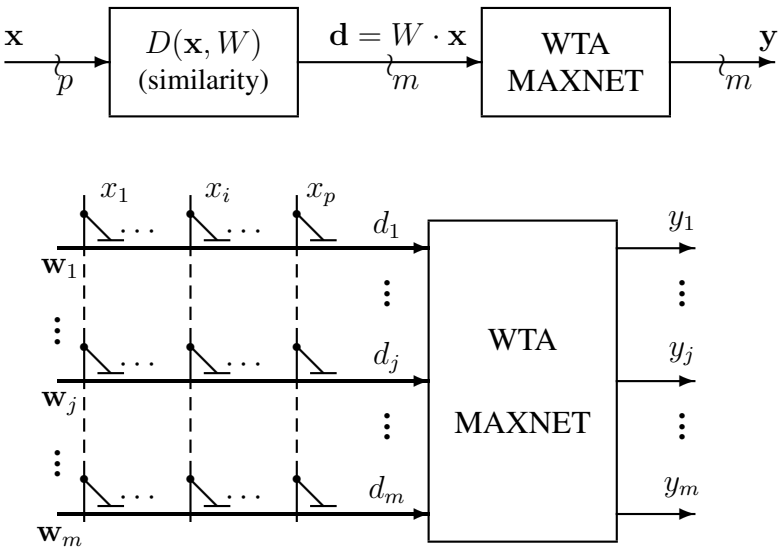


Figure 13: The structure of the similarity-measure layer for the normalised vectors

Exercise 3. 5
Consider a 2-D weight vector $\mathbf{w} = [2\ 1]$.
Normalise it so that its length is unity (refer to prac 1 for details).
Select three different unity afferent vectors $\mathbf{x}$ that form approximately $30^o, 100^o, 120^o$ angles with the weight vector $\mathbf{w}$.
Calculate the inner-product measure of similarity between $\mathbf{w}$ and $\mathbf{x}$s

□

3.3.2 The Competitive Layer

The competitive layer, also known as 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:

$$y_j = \begin{cases} 1 & \text{if } \ d_j \text{ is the largest of all } d_k \\ 0 & \text{otherwise} \end{cases}$$

that is, the jth weight vector $\mathbf{w}_{j:}$ is the **most similar** to the current afferent vector $\mathbf{x}$.

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 $-\alpha$.

For example, for $\alpha = 0.1$

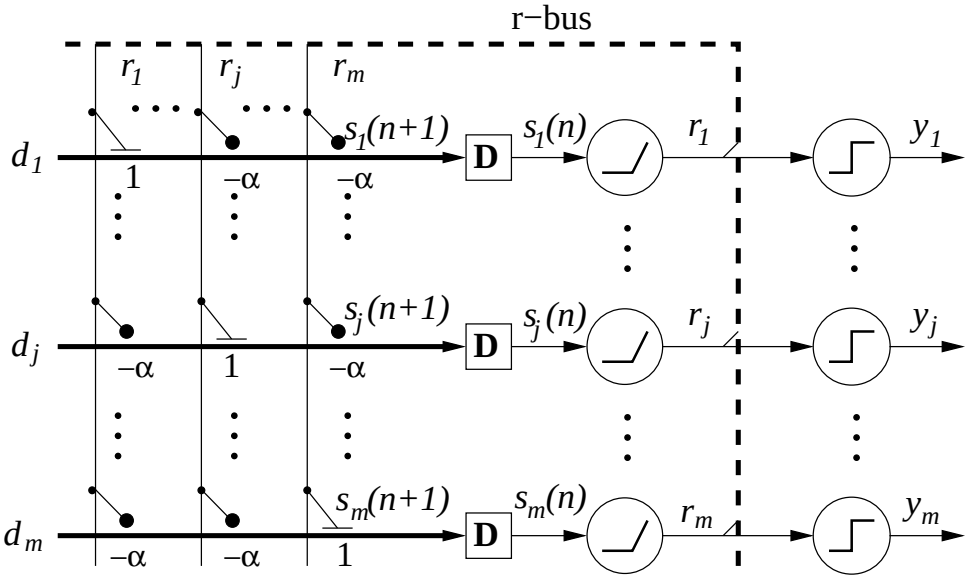
$$M = \begin{bmatrix} 1 & -0.1 & -0.1 & -0.1 \\ -0.1 & 1 & -0.1 & -0.1 \\ -0.1 & -0.1 & 1 & -0.1 \\ -0.1 & -0.1 & -0.1 & 1 \end{bmatrix}$$

The structure of the competitive layer is presented in Figure 14. In the dendritic view note that each signal r_j is fed back (through the "r-bus") and connected to an excitatory synapse belonging to the same neuron, and laterally to all inhibitive synapses from other neurons.

From the signal-flow view we can equivalently observe the unity self-excitatory connections, and the lateral inhibitory connections.

Signals $s_j(n+1)$ are delayed by one unit and form respective $s_j(n)$ signals. Signals r_j are from from s_j by removing their negative part before they are fed back to synapses. Finally signals r_j are converted into binary signals y_j.

Dendritic view:



Signal-flow view:

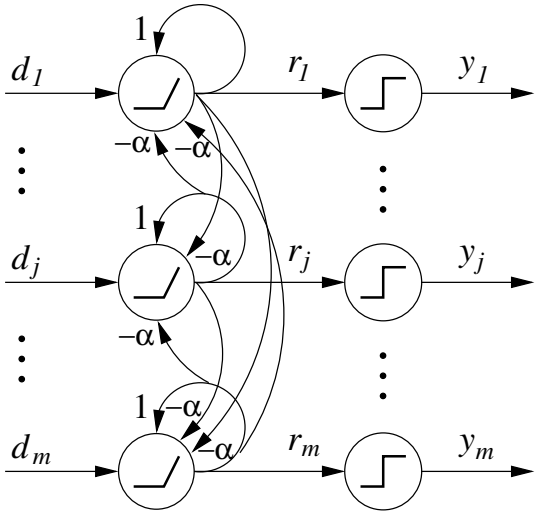


Figure 14: The dendritic and the signal-flow views of the competitive layer

Mathematically, we can evaluate the signals in the following way:

$$\begin{aligned} \mathbf{s}(0) &= \mathbf{d} \\ \mathbf{r}(n) &= \max(0, \mathbf{s}(n)) \\ \mathbf{s}(n+1) &= M \cdot \mathbf{r}(n) \\ \text{or} & \\ s_j(n+1) &= r_j(n) - \alpha \sum_{k \neq j} r_k \ , \text{ for } \ n > 0 \ \text{ and } \ j = 1, \ldots, m \end{aligned} \tag{11}$$

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

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

Let us implement in MATLAB the "Winner-Takes-All" procedure described in eqns (11)

```
% p3wta.m
s = [7.3 4.2 9.6 .7 5.5 2.9 8.6 3.4]' ; % given vector of signals
rHist = s ;      % variable to store successive columns of  r
r = max(0, s) ;
mm = size(s, 1) ;
M = -0.2*ones(mm,mm) + 1.2*eye(mm);  % formation of the matrix  M
while (sum(r~=0)) > 1   % loop until there is only one non-zero r
   s = M*r ;
   r = max(0, s) ;
   rHist = [rHist r] ;
end
n = size(rHist,2)  % number of iteration steps
kwin = find(r > 0)  % position of the winner
rHist  % successive values of r
```

The script should produce the following results of calculation of the winner:

```
         n =  6 ,  kwin = 3
rHist =
      7.3    0.3200         0         0         0         0
      4.2         0         0         0         0         0
      9.6    3.0800    2.6400    2.4000    2.2656    2.2272
      0.7         0         0         0         0         0
      5.5         0         0         0         0         0
      2.9         0         0         0         0         0
      8.6    1.8800    1.2000    0.6720    0.1920         0
      3.4         0         0         0         0         0
```

Exercise 3. 6

- Explain details of the WTA script
- Repeat the above WTA script for a different vector, s, and a different value of α

□

3.3.3 Simple Competitive Learning

With reference to Figure 12, you can note that we have already described the decoding part of the network, namely, the similarity-measure and competitive layers. Now, we will concentrate on the encoding or learning part which is respnsible for self-organization of data.

Hebbian learning considered in sec. 3.2 has organised the synaptic weights in such a way that they approximated the shape of the data by their principal directions (components).

The competitive learning aims at approximation of data organised in **clusters**. If we assume that the input data is organised in, possibly overlapping, **clusters**, then each synaptic vector, $\mathbf{w}_j$, should converge to a **centroid** of a cluster of the input data as illustrated for 2-D space in Figure 15. In other

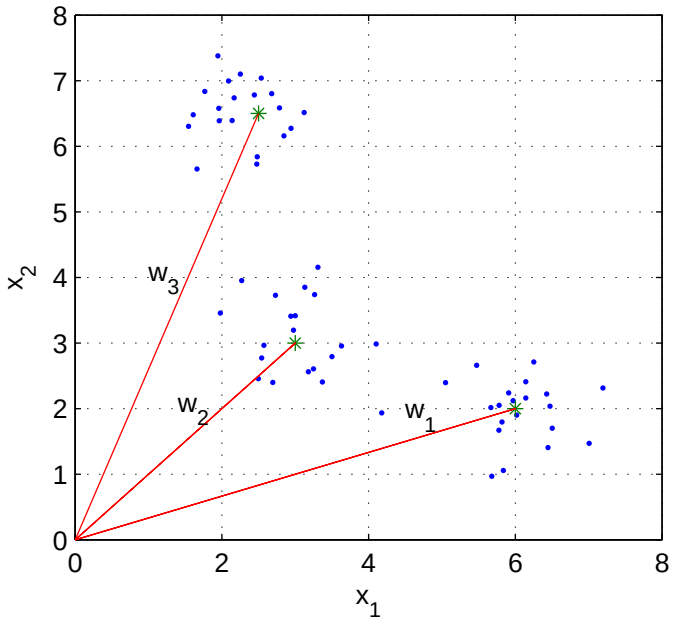


Figure 15: Approximation of 2-D clusters of data with weights

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

A simple competitive learning works in such a way that for each afferent signals (stimulus) presented to the network, its weight are pulled towards this stimulus. This should result in weight vectors being eventually located close to the centres of data (stimuli) clusters.

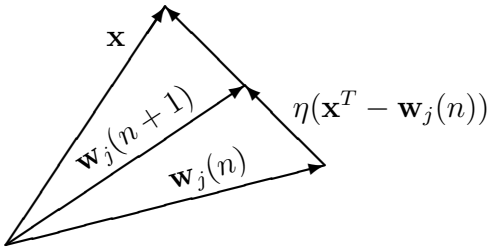
More formally, **simple competitive learning** can be describe as follow:

- Weight vectors are usually initialise with m randomly selected input vectors (stimuli):

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

- 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

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

The first MATLAB scripts generates 2-D input data matrix X consisting of m clusters of normally distributed points. It also initialises the weights with points from the data matrix X.

```
 %  CmpInit.m
 %            A.P. Paplinski   7 September 2005
 % Initialisation of competitive learning
% Generation of a 2-D pattern consisting of  m  clusters
%   of normally distributed points
 clear
 n = 2; m = 5 ;      % n inputs, m  outputs
 clst = randn(n, m); %  cluster centroids
 clst = [1 2 4 6 5
         5 1 4 2 6] ;
 Nk = 20;          % points per cluster
 K = m*Nk ;        % total number of points
 sprd = 0.5 ;      % a relative spread of the Gaussian "blob"
 X = zeros(n,K); % X is  n by m+N input data matrix
 wNk = ones(1, Nk);
 for k = 1:m
   xc = clst(:,k) ;
   X(:,(1+(k-1)*Nk):(k*Nk)) = sprd*randn(n,Nk)+xc(:,wNk) ;
 end
 [xc k] = sort(rand(1,K)); % random integers 1..K
 X = X(:, k) ;      % input data is shuffled randomly
 [xc k] = sort(rand(1,K));
```

```
Win = X(:,k(1:m))';   % Initial values of weights
figure(1), clf reset
plot(X(1, :), X(2, :), 'g.', Win(:, 1), Win(:, 2), 'b^', ...
  clst(1, :), clst(2, :), 'ro' ), grid on,
  title('Afferent data'), xlabel('x_1'), ylabel('x_2'
```

Note that for simplicity the number of data clusters is equal to number of neurons. Each neuron has two synapses, two being the dimensionality of afferent signals (stimuli). The stimuli (or data points) are marked with green dots. Centres of clusters are marked with read circles, and initial weights with the blue triangles.

In the following 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".

```
%   CmpLrn.m
%            A.P. Paplinski   7 September 2005
%   A simple competitive learning
figure(1), clf reset
CmpInit   % initialisation script
W = Win ;
Whist = zeros(K, m, n);  % to store all weights
Whist(1,:,:) = W ;
wnm = ones(m,1) ;
eta = 0.1 ;     % learning gain
deta = 1-1/K ;  % learning gain decaying factor
for  k = 1:K
  xk = X(:,k)' ;
% the current vector is compared with all weight vectors
  xmw = xk(wnm,:)-W ;
  [win kwin] = min(sum((xmw.^2),2)) ;
% the weights of the winning neurons are update
  W(kwin,:) = W(kwin,:) + eta*xmw(kwin,:) ;
  Whist(k,:,:) = W ;
% eta = eta*deta ;
plot(X(1,:),X(2,:),'g.',clst(1,:),clst(2,:),'ro', ...
    Win(:,1),Win(:,2),'b^' , ...
    Whist(1:k,:,1), Whist(1:k,:,2), 'b', W(:,1),W(:,2) , 'r*'),
  title(['Simple Competitive Learning    k = ', num2str(k)])
  xlabel('x_1'), ylabel('x_2'), grid on
  pause(1)
end
figure(1)
plot(X(1,:),X(2,:),'g.',clst(1,:),clst(2,:),'ro', ...
    Win(:,1),Win(:,2),'b^' , ...
    Whist(1:k,:,1), Whist(1:k,:,2), 'b', W(:,1),W(:,2) , 'r*'), grid
 title(['Simple Competitive Learning    k = ', num2str(k)])
 xlabel('x_1'), ylabel('x_2')
% print -f1 -depsc2 CmpLrn
```

Run the above script a number of times and observe how the weights evolve from their initial position to the final location that ideally should be at the centres of clusters.

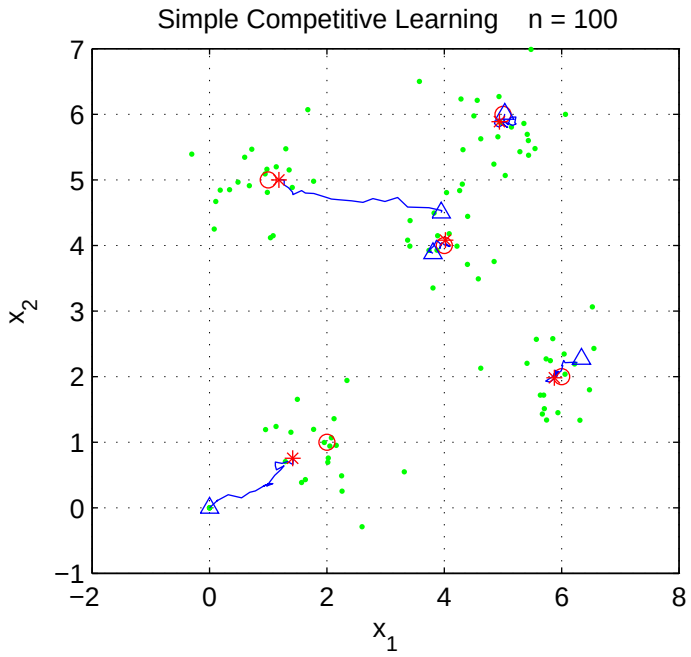


Figure 16: Simple competitive learning: 'o' – centroids of generated clusters, '△' – initial weights, '*' – Final weights

In Figure 16 there is an example of the simple competitive learning. Note that after one training epoch, the weights are relatively close to the centroids of the clusters. This is an indication that synaptical weights approximate the input date in a satisfactory way.

Exercise 3. 7
Repeat the above exercise with the following modifications:

number of clusters `nc = 6`
size of clusters `Nk = 16`
number of neurons `m = 12`

Comment how neurons are distributed around the input data at the conclusion of learning. Experiment with different values of the learning gain η.

□

3.4 Self-Organizing Feature Maps

3.4.1 Structure of Self-Organizing Feature maps

Self-Organizing Feature Maps (SOFMs, or SOMs) also known as Kohonen maps or topographic maps were first introduced by von der Malsburg (1973) and in its present form by Kohonen (1982).

Self-Organizing Feature Maps are **competitive neural networks** as those presented in Figure 12 in which, in addition neurons are organized in an l-dimensional **lattice** (grid) as illustrated in Figure 17. The network in Figure 17 consists of $m = 12$ neurons located on a 2-D ($l = 2$) $3 \times 4 = 12$ lattice.

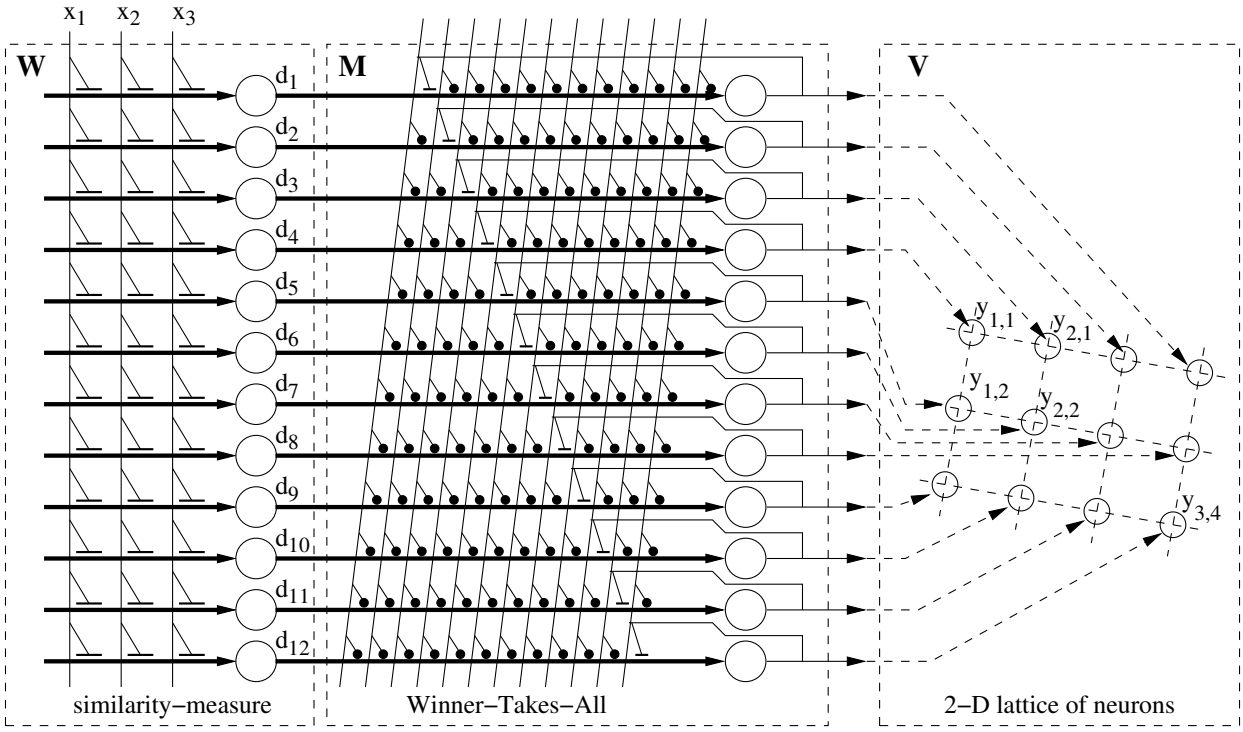


Figure 17: A structure of a 2×3 self-organizing feature map

Note the similarity measure layer and the "Winner-Take-All" or competitive layer. Note also the usual structure of the competitive layer based on lateral inhibition and self-excitatory connections. The neuronal lattice characterizes a relative position of neurons with regards to its neighbours.

It is often convenient to chracterize a position of each neuron by an $m \times l$ **position matrix**, V. For a network of Figure 17, the position matrix will store 12 positions from $(1, 1)$ to $3, 4$.

The complete network of neurons is characterized by the following parameters:

p	—	dimensionality of the **input space**
l	—	dimensionality of the **feature space**
m	—	the total number of neurons
W	—	$m \times p$ matrix of synaptic weights
V	—	$m \times l$ matrix of topological positions of neurons

In subsequent considerations neurons will be identified either by their index $k = 1, \ldots, m$, or by their position vector $V(k, :)$ in the neuronal grid.

3.4.2 Feature Maps

If the dimensionality of the input (stimulus) space is low, typically $p = 1, 2, 3$, a **Feature Map** is a plot of synaptic **weights** in the **input space** in which weights of the neighbouring neurons are joined by lines or plane segments (patches).

For a typical case when both input and feature spaces are two-dimensional the feature map can be plotted in the way illustrated by the following script:

```
% p3SOM22.m
 %
 % Plotting a 2-D Feature map in a 2-D input space

 clear, close all
 m = [3 4]; mm = prod(m) ; % p = 2 ;
 %   formation of the neuronal position matrix
 [V2, V1] = meshgrid(1:m(2), 1:m(1)) ;
 V = [V1(:), V2(:)] ;
 % Example of a weight matrix
 % W = V-1.4*rand(mm, 2) ;
 W =[0.83    0.91
     0.72    2.01
     0.18    2.39
     2.37    0.06
     1.38    2.18
     1.41    2.82
     2.38    1.27
     2.06    1.77
     2.51    2.61
     3.36    0.85
     3.92    2.05
     3.16    2.90  ] ;
 [W V]
 figure(1)
 % Plotting a feature map
 FM1 = full(sparse(V(:,1), V(:,2), W(:,1))) ;
 FM2 = full(sparse(V(:,1), V(:,2), W(:,2))) ;
 FM = FM1+j*FM2;
 plot(FM), hold on, plot(FM.'), plot(FM, '*'), hold off
 grid on
 % the following section marks coordinates of each neuron
 tt = 'w_{11}' ; tt = tt(ones(mm,1),:) ;
 tt(:,4:5) = [num2str(V(:,1),1) num2str(V(:,2),1)] ;
 text(W(:,1)+0.05, W(:,2)+0.05, tt) ;
 axs = axis ;
 text(0.95*axs(2), 0.04*axs(2), 'x_1');
 text(0.04*axs(4), 0.95*axs(4), 'x_2') ;
 % print -f1 -depsc2 p3SOM22
```

The resulting weight and position matrices and are as follows

```
W = 0.83   0.91     V = 1  1
    0.72   2.01         2  1
    0.18   2.39         3  1
    2.37   0.06         1  2
    1.38   2.18         2  2
    1.41   2.82         3  2
    2.38   1.27         1  3
    2.06   1.77         2  3
    2.51   2.61         3  3
    3.36   0.85         1  4
    3.92   2.05         2  4
    3.16   2.90         3  4
```

The feature map for a given weight and position matrices is given in Figure 18.

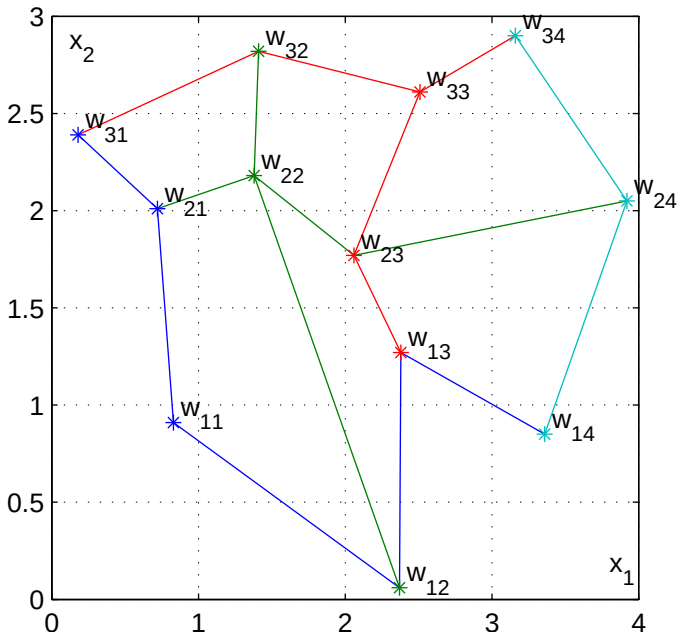


Figure 18: A Feature map for 2-D input and feature map

If the weight matrix has captured an essential features of input stimuli, then the feature map gives an additional information regarding the stimuli.

Exercise 3. 8
Plot a feature map for `m = [3 5]` and the weight matrix of your choice.

□

3.4.3 Learning Algorithm for Self-Organizing Feature Maps

The objective of the learning algorithm for a SOFM neural network is formation of a feature map which captures the essential characteristics of the p-dimensional input data and maps them on an l-D feature space. The learning algorithm consists of two essential aspects of the map formation, namely, **competition** and **cooperation** between neurons of the output lattice.

Competition is implemented as in the competitive learning: each input vector $\mathbf{x}(n)$ is compared with each weight vector from the weight matrix W and the position $V(k(n),:)$ of the winning neuron $k(n)$ is established. For the winning neuron the distance

$$d_k = |\mathbf{x}^T(n) - W(k(n),:)|$$

attains a minimum (as in MATLAB used as a simulation tool ':)' denotes all column elements of a matrix).

Cooperation All neurons located in a topological neighbourhood of the winning neurons $k(n)$ will have their weights updated usually with a strength $\Lambda(j)$ related to their distance $\rho(j)$ from the winning neuron,

$$\rho(j) = |V(j,:) - V(k(n),:)| \quad \text{for} \quad j = 1, \ldots, m.$$

The **neighbourhood function**, $\Lambda(j)$, is usually an l-dimensional Gausssian function:

$$\Lambda(j) = \exp(-\frac{\rho^2(j)}{2\sigma^2})$$

where σ^2 is the variance parameter specifying the spread of the Gaussian function.

The following script illustrate the concept of the **shrinking neighbourhood**. The weight modification will be proportional to the value of the neighbourhood function Λ for each nuron from the proximity of the winner.

```
% p3nbrhood.m
% Illustration of a shrinking 1-D neighbourhood
k =  -10:10 ;
kk = -10:0.1:10 ;
sig = 4;   %  sigma parameter
figure(2)
for n = 1:15
  sig2 = 2*((0.8^n)*sig)^2 ; % reducible variance
  Lmbdakk = exp(-kk.^2/sig2) ;
  Lmbdak  = exp(-k.^2/sig2) ;
  plot(kk, Lmbdakk), grid on, hold on
  stem(k, Lmbdak, '*'), hold off
  title(['Shrinking neighbourhood,    n = ', num2str(n)])
  pause(1)
end
```

Feature map formation is critically dependent on the learning parameters, namely, the learning gain, η, and the spread of the neighbourhood function specified for the Gaussian case by the variance, σ^2. In general, both parameters should be time-varying, but their values are selected experimentally.

Usually, the **learning gain**, η, should stay close to unity during the **ordering phase** of the algorithm which can last for, say, 1000 iteration (epochs). After that, during the **convergence phase**, should be reduced to reach the value of, say, 0.1.

The **spread**, σ^2, of the neighbourhood function should initially include all neurons for any winning neuron and during the ordering phase should be slowly reduced to eventually include only a few neurons in the winner's neighbourhood. During the convergence phase, the neighbourhood function should include only the winning neuron.

Details of the SOFM learning algorithm

The complete algorithm can be described as consisting of the following steps

1. Initialise:
   (a) the weight matrix W with a random sample of m input vectors.
   (b) the learning gain and the spread of the neighbourhood function.
2. for every input vector, $\mathbf{x}(n), \quad n = 1, \ldots, N$:
   (a) Determine the winning neuron, $k(n)$, and its position $V(k,:)$ as
   $$k(n) = \arg\min_j |\mathbf{x}^T(n) - W(j,:)|$$
   (b) Calculate the neighbourhood function
   $$\Lambda(n,j) = \exp(-\frac{\rho^2(j)}{2\sigma^2})$$
   where
   $$\rho(j) = |V(j,:) - V(k(n),:)| \quad \text{for} \quad j = 1, \ldots, m.$$
   (c) Update the weight matrix as
   $$\Delta W = \eta(n) \cdot \Lambda(n) \cdot (\mathbf{x}^T(n) - W(j,:))$$
   All neurons (unlike in the simple competitive learning) have their weights modified with a strength proportional to the neighbourhood function and to the distance of their weight vector from the current input vector (as in competitive learning).
   The step (2) is repeated E times, where E is the number of epochs.
3. (a) During the ordering phase, shrink the neighbourhood until it includes only one neuron:
   $$\sigma^2(e) = \frac{\sigma_0^2}{e}$$
   where e is the epoch number and σ_0^2 is the initial value of the spread (variance).

(b) During the convergence phase, "cool down" the learning process by reducing the learning gain. We use the following formula:

$$\eta(e) = \frac{\eta_0}{1 + \eta_p e}$$

where η_0 is the initial value of the learning gain, and η_p is selected so that the final value of the learning gain, reaches the prescribed value, $\eta(E) = \eta_f$.

3.4.4 Example of a Self-Organizing Feature Map formation

In this example we consider two-dimensional stimuli provided by the sources are represented in the figures (e.g. see Figure 1) by o and + respectively. We use three groups of data each containing 10 stimuli (i.e. points in a two-dimensional space) in each source, sixty points all together.
The sources can be thought of as producing, e.g., two dialects of a very limited protolanguage, each with three protophonemes.
We can imagine that a source is a representation of a parent of a child pronouncing three phonemes in ten slightly different ways. The parallel is far from perfect but might be helpful for a conceptual understanding of the map formation and interpretation. Real sensory stimuli, like the phonemes of speech are of course larger in number and dimension.

```
% p3sofm.m
%             28 August 2004
% Self-Organizing map formation
clear
%------------------------
% generation of 2-D stimuli (data) organised in
%     ns   sources each with  nc  classes
nc = 3;   ns = 2;     % number of classses (nc) and sources (ns)
ncs = nc*ns ;         % total number of data clusters
Nps = 10 ;            % number of points per a claster
Np = Nps*nc ;         % number of points per source
N = Np*ns;            % A total number of datapoints.
wnc = ones(1,nc);     % a vector of  nc  ones
X = ones(2,N);    % memory allocation for the stimuli matrix
% centres of classes:  source A   sourceB
xmean = [0.9 1.8 0.7   1.1 2.1  1
         2  2.1  1     1.9  1.8 0.9 ];
% The spread of data
xvari = 0.05*ones(2, ncs) ;

% generation of normally distributed points around the means
for k = 1:ncs
  kN = k*ones(1,Nps) ;  kns = (1:Nps)+(k-1)*Nps ;
  X(1:2, kns) = xmean(:, kN) + xvari(:, kN).*randn(2,Nps) ;
end
figure(1)
plot(X(1,(1:Np)), X(2,1:Np),'o',X(1,(1:Np)+Np), X(2,(1:Np)+Np),'+')
```

```
axis([0.5 2.5 0.5 2.5]), grid on
title('Stimuli: 2 sources, 3 classes, 60 points'),
xlabel('x_1'), ylabel('x_2')
% print -f1 -depsc2 p3fmStim
```

Training stimuli generated by the above script should be similar to those in Figure 19

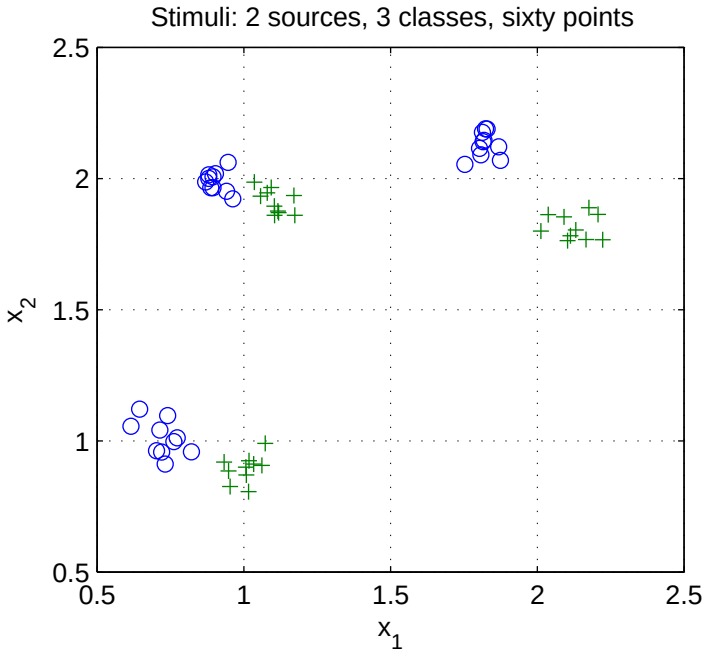


Figure 19: Training stimuli organized in 2 sources and 3 classes

The next step is to define details of the neural network and initialise the weights. This can be done in the following way:

```
% Specification of the neuronal lattice
p = 2 ;       % dimensionality of afferents
m = [3 3] ;   % neuronal lattice
     fprintf('neuronal lattice  m = [ %d , %d ]\n' , m)
mm = prod(m);      % total number of neurons
onesm = ones(1,mm) ;    % vector of mm  ones

%  formation of the neuronal position matrix
[V2, V1] = meshgrid(1:m(2), 1:m(1)) ;
V = [V1(:), V2(:)] ;

% Weight initialization with random values around  the mean of stimuli
Xmean = mean(X,2) ;    % the mean of all stimuli
W = 0.05*randn(mm,p)+Xmean(:,onesm)' ;
```

Befor we start the process of self-organization, that is, mapping neurons to stimuli, we set first the relevant parameters:

```
% ----------------------------------------
% Learning (map formation) parameters
% The initial spread of the neighbourhood function
 sig2 = 50 ;   %  2 sigma^2
    fprintf('The initial 2sigma.Sq of the NB function:  %4d\n', sig2)
 % The initial learning gain
 eta0 = 0.1 ;
     fprintf('The initial learning gain eta0: %1.3f\n', eta0)
 % learning gain reducing factor
  etap = 0.02 ;
 nepochs = 100;  % the number of epochs in each simulations
    fprintf('number of training epochs: %d\n' , nepochs)
 tic    % to start the stopwatch
 tim = clock;
 disp([date sprintf('%4d:%02d', tim(4:5))])
 fnm = [sprintf('p3d%02d',tim(3)),'h',sprintf('%02d',tim(4:5))];
 figure(2)
```

Finally, the script for the learning loop follows. We will plot the feature map at the end of each epoch.

```
%  Map formation
 for epch = 1:nepochs    %  the epoch loop
   % The learning gain  eta  decreases from  eta0  hyperbolically
   eta = eta0/(1+etap*epch) ;  % gain is reduced for every epoch

   for n=1:N     %  stimulus loop
     % training data is applied in a random order
     xn = X(:, round((N-1)*rand)+1) ; % a new stimulus (datapoint)

     % WTA: distance between the datapoint and weight vectors
     xW = xn(1:2,onesm) - W' ;
     % grid coordinates of the winning neuron V(kn, :)
     % and the difference between the datapoint and
     % the weight vector of the winner
     [TtlDif kn] = min(sum(xW.^2)) ;  vkn = V(kn, :) ;

     % squared distance from the winning neuron
     rho2 = sum(((vkn(onesm, :) - V).^2), 2) ;
     % Gaussian neighbourhood function, Lambda, is
     %        centered around winning neuron
     Lambda = exp(-rho2*epch/sig2) ;

     % Kohonen learning law
     W = W + eta*Lambda(:,ones(p,1)).*xW';
  end  % stimulus loop

  %  plot data points
```

```
  plot(X(1,(1:Np)), X(2,1:Np),'o',X(1,(1:Np)+Np), X(2,(1:Np)+Np),'+')
  axis([0.5 2.5 0.5 2.5]), grid on , hold on
  xlabel('x_1'), ylabel('x_2')

  %  plot a Feature MAp  for  W and positin matrix V
  FM1 = full(sparse(V(:,1),V(:,2),W(:,1)));
  FM2 = full(sparse(V(:,1),V(:,2),W(:,2)));
  FM = FM1+j*FM2;   % 2-D map in a complex form
  plot(FM,'*'), plot(FM), plot(FM.')
  title([sprintf('%d x %d map, ', m) sprintf('%d epochs', epch)]) ;
  hold off
  text(2.2, 2.6, fnm)
  pause(0.5)
end   %  the epoch loop
toc    % to stop the stopwatch
% print( '-f2',  '-deps2', fnm);
```

This concludes a rather complicated map formation script `p3sofm.m`. The final results might be similar the the one in Figure 20.

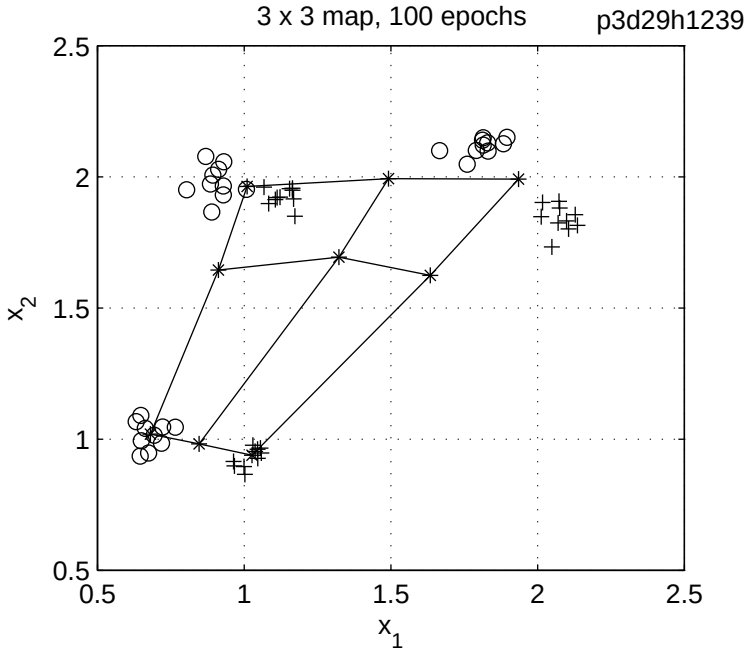


Figure 20: Self-organizing map of 3×3 lattice of neurons.

Note the way the nine neurons map the 60 stimuli organized in 3 pairs of clusters.

Exercise 3. 9
Repeat the map formation script with the following modifications

- Organize data in four pairs of clusters
- assume two different sizes of the neuronal maps: 2×3 and 3×4

If the time permits experiment with three map formation parameters, namely `sig2`, `eta0`, `etap` and draw conclusions regarding the map formation process.

□

Written Submission

Your written submission should include results of all exercises you have performed (MATLAB scripts, figures, numerical results, etc) with brief comments and explanations.
It should be in a form ready for electronic submission when requested.