CSE5301 Neuro-Fuzzy Computing

Tutorial/Assignment 3: Unsupervised Learning

About this tutorial

The objective of this tutorial is to study unsupervised learning, in particular:

- (Generalized) Hebbian learning.
- Competitive learning.
- Self-Organized Maps.

3.1 Hebbian learning

Donald Hebb, after whom the learning is named, 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."

3.1.1 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 (output) 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.

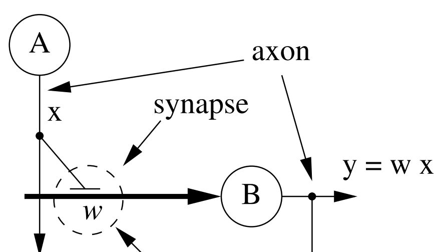


Figure 1: Illustration of the Hebbian learning

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 steps, $w(n+1), w(n)$ 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.

It is obvious that if A and B keep firing together in time during a lifetime, w would exponentially grow very large. Such learning law does not make any sense. Therefore, we must introduce a mechanism for decreasing weight w, because "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.

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 could save in a file `myprac3a.m`, or something similar:

```
% myprac3a.m
%       April 21, 2005
x = 0.5 ;
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.

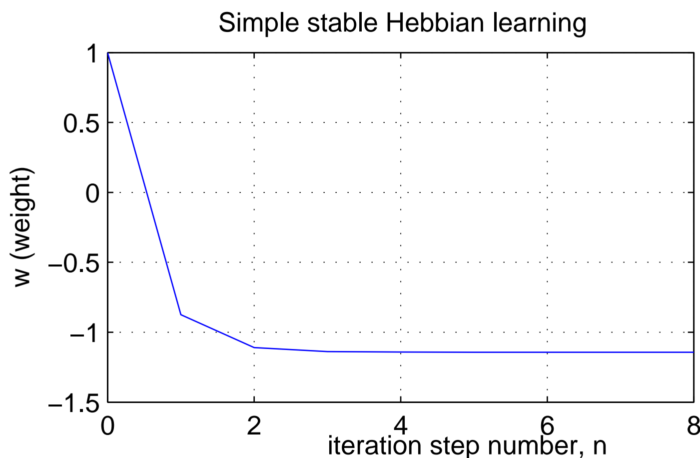


Figure 2: Evolution of weight values in a simple Hebbian learning

Exercise 3. 1
(Join with Exercise 3.2)
Observe the weight change for different values of α_1, α_2.

□

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:

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

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 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.

In Lecture Notes we discuss a **Generalized Hebbian Learning** which produces weight vectors of the unit length. Such weight vectors form a description of a shape in the form of its principal directions. We start with a single neuron version of the Generalized Hebbian Learning known as Oja's rule.

3.1.2 Generalized 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:

$$\Delta w_i = \alpha \cdot y \cdot \tilde{x}_i \,, \quad \text{where} \quad \tilde{x}_i = x_i - y \cdot w_i \tag{6}$$

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:

$$\Delta \mathbf{w} = \alpha \cdot y \cdot \tilde{\mathbf{x}}^T = \alpha \cdot y \cdot (\mathbf{x}^T - y \cdot \mathbf{w}) \,, \quad \text{where} \quad \tilde{\mathbf{x}}^T = \mathbf{x}^T - y \cdot \mathbf{w} \,, \quad \text{and} \quad y = \mathbf{w} \cdot \mathbf{x} \tag{7}$$

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 (7) is the fact that the **length (Euclidian norm) of the weight vector tends to unity**, that is,

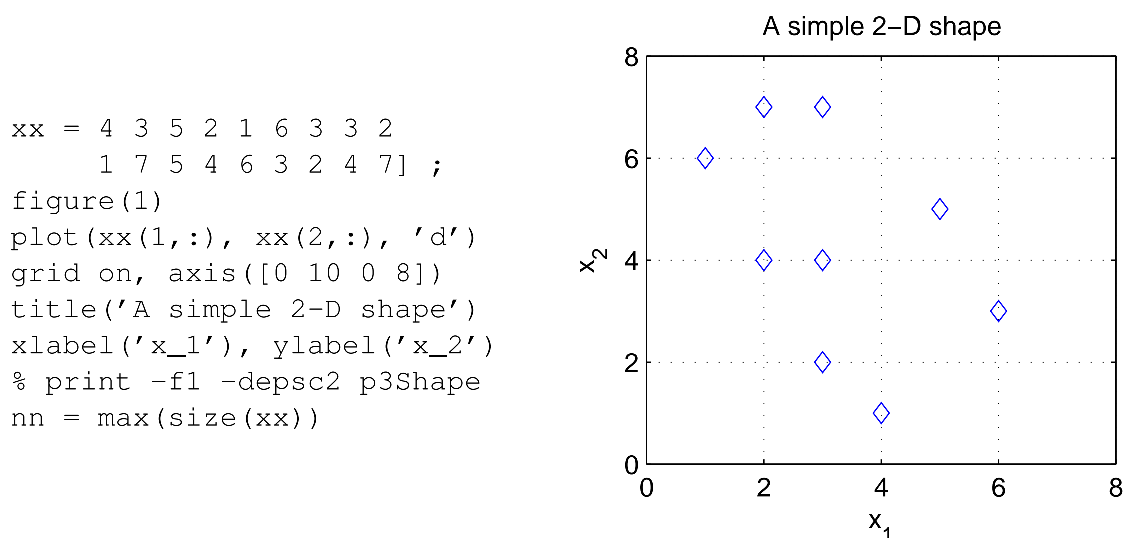
$$||\mathbf{w}|| \longrightarrow 1 \tag{8}$$

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

3.1.3 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 3. To create this shape execute the following MATLAB commands (prepare a relevant script) as in Figure 3.




```
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 3: 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:

$$X = [\, \mathbf{x}(1) \;\; \mathbf{x}(2) \;\; \ldots \;\; \mathbf{x}(N) \,] \;, \quad \text{where, for } p = 2 \quad \mathbf{x}(n) = \begin{bmatrix} x_1(n) \\ x_2(n) \end{bmatrix} \tag{9}$$

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 3. Since our shape is two-dimensional, we need a neuron with just two synapses $(p = 2)$ as in Figure 4. Note that the

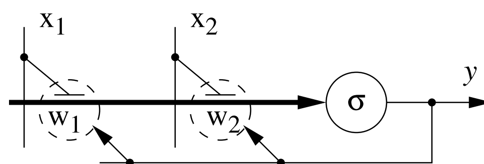


Figure 4: A neural network with a local learning law

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 (7). 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 5. A few additional comments and explanations regarding

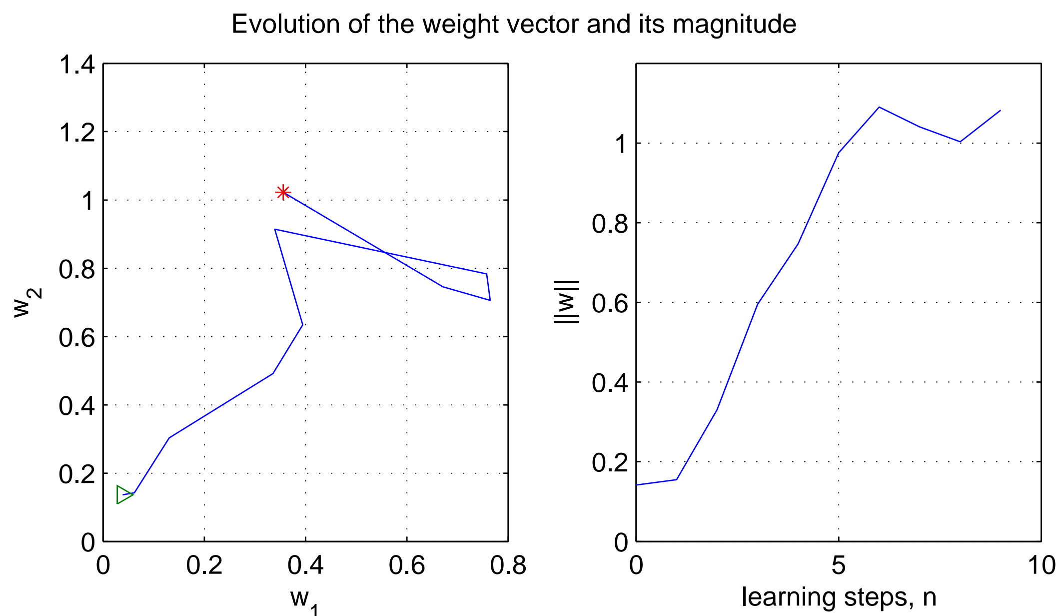


Figure 5: Evolution of the weight vector and its length during one presentation of the shape. The initial and final weights are marked with '▷' and '*', respectively

the MATLAB script.

- 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 5 that as predicted in eqn (8) 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 6. We can now see from Figure 6 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 6 that when the length is close to unity, the weight vector moves on a segment of a **unity circle**.

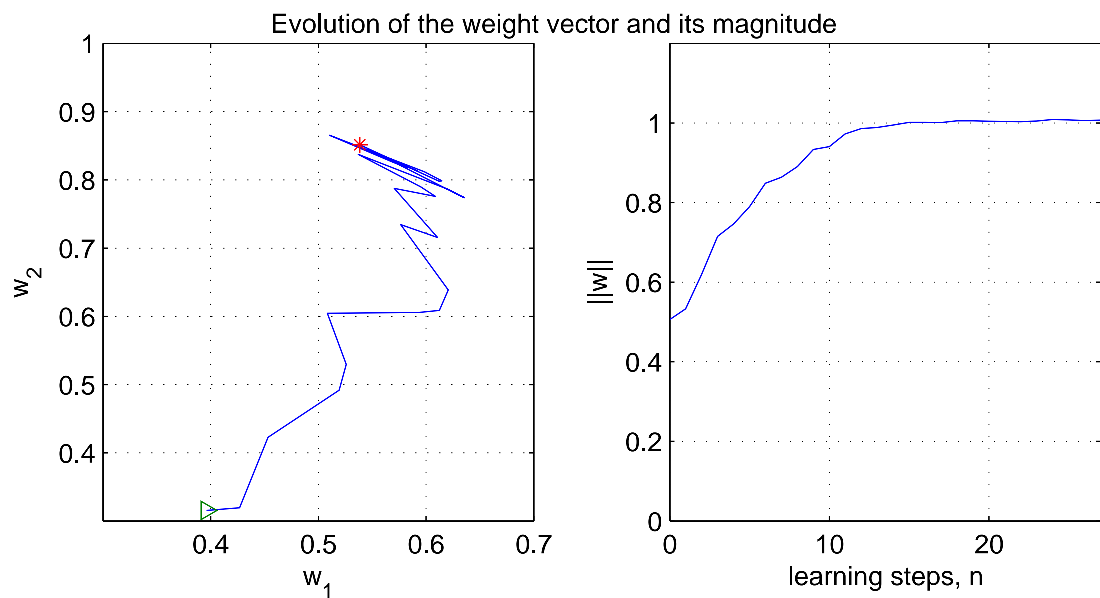


Figure 6: Self-organization in Hebbian learning

```
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')
```

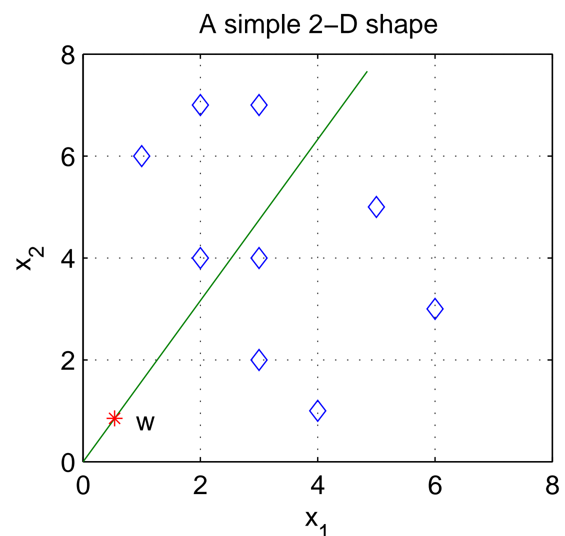


Figure 7: A simple shape with a line along the weight vector

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 7.
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.

3.1.4 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 attach here a complete script for the extraction.

```
% OjaPCA.m
%          5 May 2005
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 8 and 9.

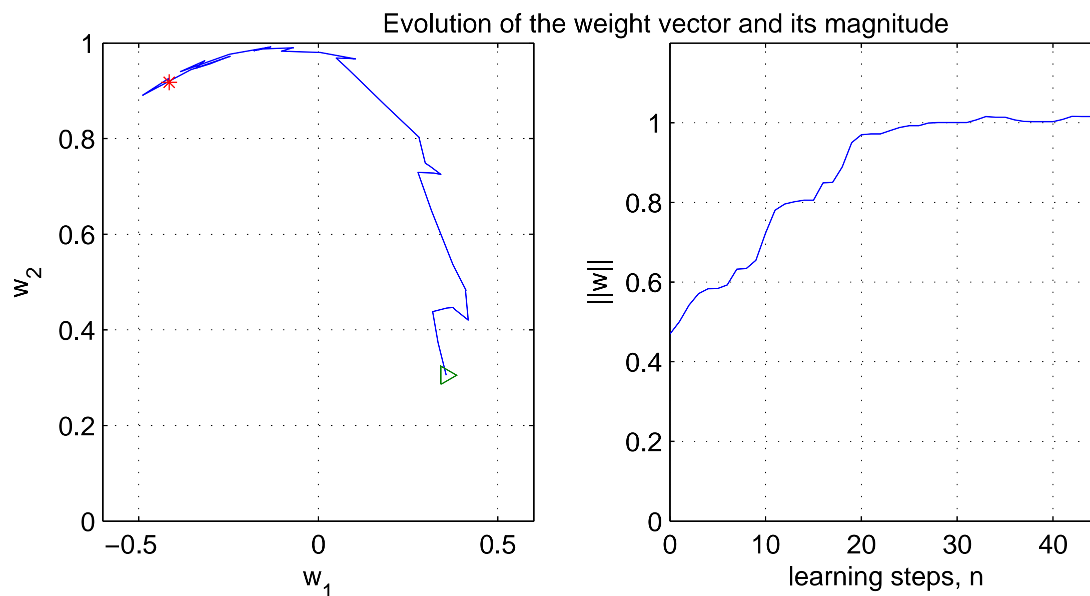


Figure 8: Evolution of the weight vector and its length during self-organization

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 4, 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.

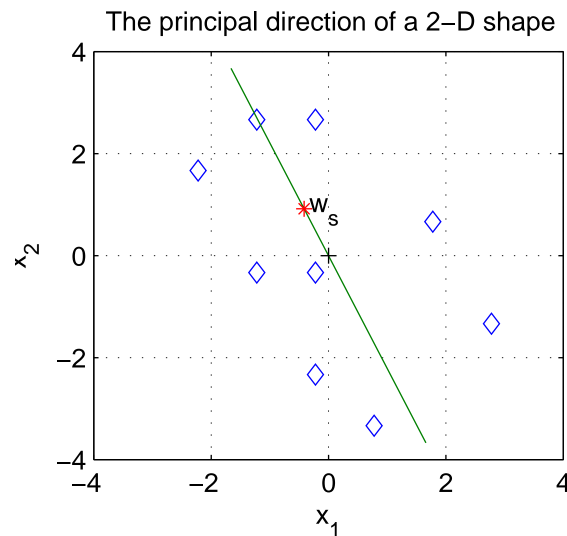


Figure 9: The shape with the final weight line inserted.

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

Show plots with evolution of weight vectors, their magnitudes and the shape with **two** principal directions inserted.

□

3.1.5 Image coding algorithm using the generalised Hebbian algorithm

Exercise 3. 4
Implement in MATLAB an image compression system using the generalised Hebbian learning algorithm (GHA) as described in lecture notes.

1. Determine the speed of training and the SNR ratio. Compare it with the equivalent results for the multilayer perceptron.
2. Demonstrate that the weight matrix W which is obtained by the GH learning algorithm is an approximation of the matrix of eigenvectors of the input correlation matrix, that is, $\hat{W} = \texttt{eig}(R)$.
3. Demonstrate that the variance of the output signals is equal to the eigenvalues of the input correlation matrix.
4. Compare briefly this image compression system to that based on the two-layer perceptron.

□

3.2 Competitive Learning

Exercise 3. 5
The script `$CSE5301/Mtlb/CmpLrn.m` demonstrates a simple competitive learning and produces the plot as in Figure 10

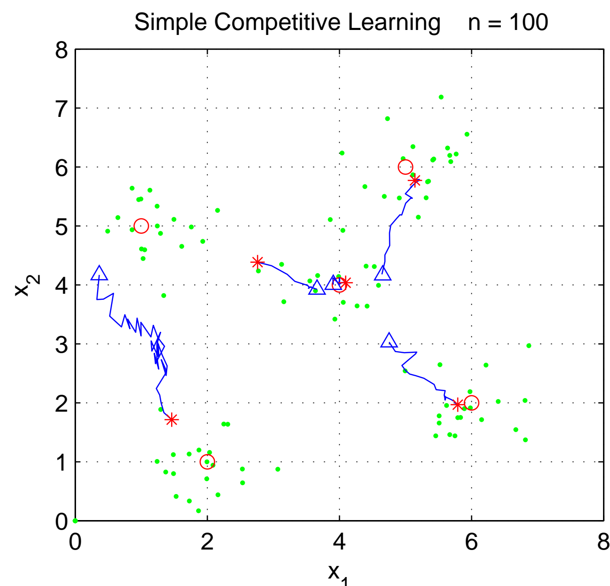


Figure 10: A simple competitive learning: 'o' – centroids of generated clusters, '△' – initial weights, '*' – Final weights

Repeat the above demo 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.

□

Exercise 3. 6
Modify the above script(s) to increase dimensionality of the input space from two to three, and to implement the Frequency-Sensitive Competitive Learning (FSCL):

1. In the FSCL, for each neuron we create the count, c_i, of the weight updates performed for this neuron, that is, the number of times this neuron won. The weight update is inversely proportional to the count, c_i, that is, the learning gain is proportional to $1/c_i$.

2. Initialize the weights with a sample of m input vectors (as oppose to the random initialization of weights).

□

Exercise 3. 7
Implement the Vector Quantization Image Compression System using the Frequency-Sensitive Competitive Learning.

1. Test the VQ image compression system on a sample image available in MATLAB.
2. Discuss briefly the differences between image coding systems based on competitive learning (VQ) and the Generalised Hebbian learning.

□

3.3 Self-Organizing Feature Maps

Exercise 3. 8
Implement the Kohonen's algorithm for self-organizing neural networks which performs feature mapping from a two-dimensional input space onto:

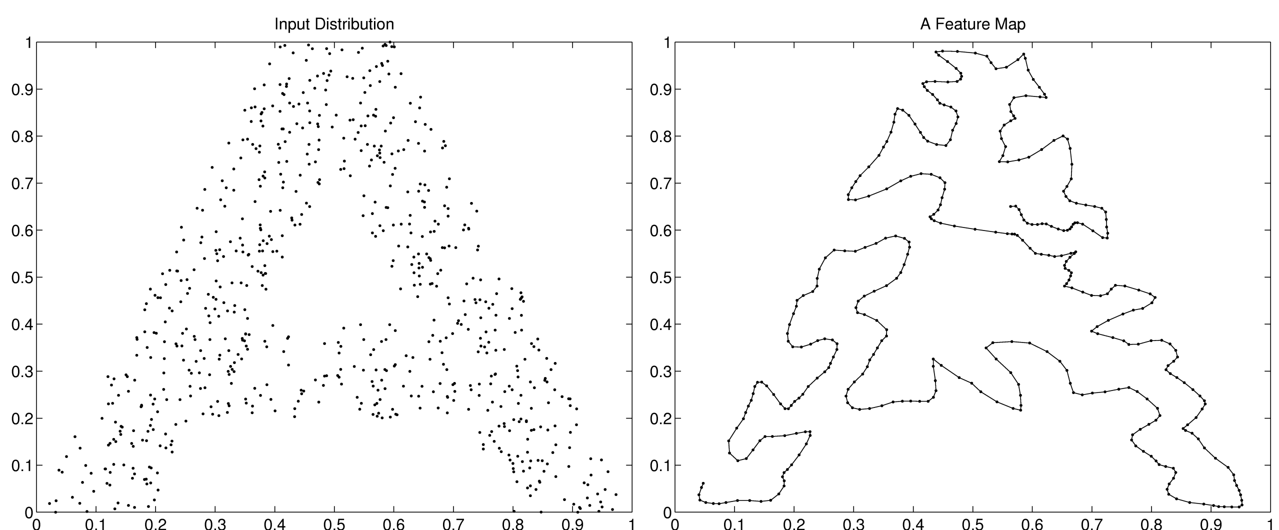
1. a one-dimensional feature space, and
2. a two-dimensional feature space.

The two-dimensional input space should have a shape of one of the following characters:

A, C, E, F, H, J, K, L, M, N, X, Y, Z

Use (studentID)mod13 as the pointer to select your character from the above list.

The input space and the resulting one-dimensional feature map might look as follows:



Refer to the script `$CSE5301/Mtlb/sofm.m` for assistance, but

- Project the point on the a sphere and
- use the appropriate similarity measure.
- Plot the two-dimensional feature map using the "pcolor" MATLAB function,

□

Exercise 3. 9
Write a MATLAB script that implements a data clustering algorithm using SOFMs, and study its behaviour.

- Generate randomly located 4 pairs of clusters of 2-D points, each cluster with a different variance and containing 8 points. Clusters forming a pair should be relatively close to each other.
- Use a 2×3 and a 3×3 SOFM to cluster the data, that is, to approximate distribution of data

□

Submission

In **your report/submission** include answers/solutions to all exercises. Include

- Brief comments regarding the demo files,
- Relevant derivations, equations, scripts and plots with suitable comments.