

Table of Contents

CHAPTER VI- HEBBIAN LEARNING AND PRINCIPAL COMPONENT ANALYSIS	3
1. INTRODUCTION.....	4
2. EFFECT OF THE HEBB UPDATE	5
3. OJA'S RULE.....	16
4. PRINCIPAL COMPONENT ANALYSIS.....	20
5. ANTI-HEBBIAN LEARNING.....	27
6. ESTIMATING CROSSCORRELATION WITH HEBBIAN NETWORKS	30
7. NOVELTY FILTERS AND LATERAL INHIBITION	32
8. LINEAR ASSOCIATIVE MEMORIES (LAMs).....	37
9. LMS LEARNING AS A COMBINATION OF HEBB RULES.....	42
10. AUTOASSOCIATION	47
11. NONLINEAR ASSOCIATIVE MEMORIES	52
12. PROJECT: USE OF HEBBIAN NETWORKS FOR DATA COMPRESSION AND ASSOCIATIVE MEMORIES....	54
13. CONCLUSIONS	57
LONG AND SHORT TERM MEMORY	60
ASSOCIATIVE MEMORY	60
HEBBIAN AS GRADIENT SEARCH	61
INSTABILITY OF HEBBIAN	63
DERIVATION OF OJA'S RULE	63
PROOF OF EIGEN-EQUATION.....	64
PCA DERIVATION.....	65
DEFINITION OF EIGENFILTER.....	66
OPTIMAL LAMS	67
HEBB	67
UNSUPERVISED	67
SANGER	68
OJA.....	68
APEX	68
ASCII	68
SECOND ORDER	68
SVD	68
Eq.1	69
Eq.6	69
Eq.5	69
Eq.8	69
Eq.2	69
Eq.26	69
Eq.30	69
Eq.29	70
KOHONEN.....	70
Eq.36	70
STEPHEN GROSSBERG.....	70
Eq.7	70
Eq.11	70
Eq.4	71
Eq.27	71
Eq.19	71
Eq.34	71
DIAMANTARAS	71
DEFLATION.....	71
BALDI	71
HECHT-NIELSEN	72
KAY.....	72
Eq.18	72

ENERGY, POWER AND VARIANCE	72
PCA, SVD, AND KL TRANSFORMS.....	73
GRAM-SCHMIDT ORTHOGONALIZATION	76
SILVA AND ALMEIDA	77
INFORMATION AND VARIANCE	77
COVER AND THOMAS	78
FOLDIAK	78
RAO AND HUANG	78

Chapter VI- Hebbian Learning and Principal Component Analysis

Version 2.0

This Chapter is Part of:

Neural and Adaptive Systems: Fundamentals Through Simulation© by

Jose C. Principe

Neil R. Euliano

W. Curt Lefebvre

Copyright 1997 Principe

The goal of this chapter is to introduce the concepts of Hebbian learning and its multiple applications. We will show that the rule is unstable but through normalization is very useful. Hebbian learning is used to associate an input to a given output through a similarity metric. A single linear PE net trained with Hebbian rule finds the direction in data space where the data has the largest projection, i.e. such network transfers most of the input energy to the output.

This concept can be extended to multiple PEs giving rise to the principal component analysis (PCA) networks. These nets can be trained on-line and produce an output which preserve the maximum information from the input as required for signal representation. By changing the sign of the Hebbian update we also obtain a very useful network that decorrelates the input from the outputs, i.e. it can be used for finding novel information. Hebbian can be even related to the LMS learning rule showing that correlation is effectively the most widely used learning principle. Finally, we show how to apply Hebbian learning to associate patterns, which gives rise to a new and very biological form of memory called associative memory.

- 1. Introduction
- 2. Effect of the Hebb update
- 3. Oja's rule

- 4. Principal Component Analysis
- 5. Anti Hebbian Learning
- 6. Estimating crosscorrelation with Hebbian networks
- 7. Novelty filters
- 8. Linear associative memories (LAMs)
- 9. LMS learning as a combination of Hebb rules
- 10. AutoAssociation
- 11. Nonlinear Associative memories
- 12. Conclusions

[Go to next section](#)

1. Introduction

The neurophysiologist [Donald Hebb](#) enunciated in the 40's a principle that became very influential in neurocomputing. By studying the communication between neurons, Hebb verified that once a neuron repeatedly excited another neuron, the threshold of excitation of the later decreased, i.e. the communication between them was facilitated by repeated excitation. This means that repeated excitation lowered the threshold, or equivalently that the excitation effect of the first neuron was amplified (Figure 1).

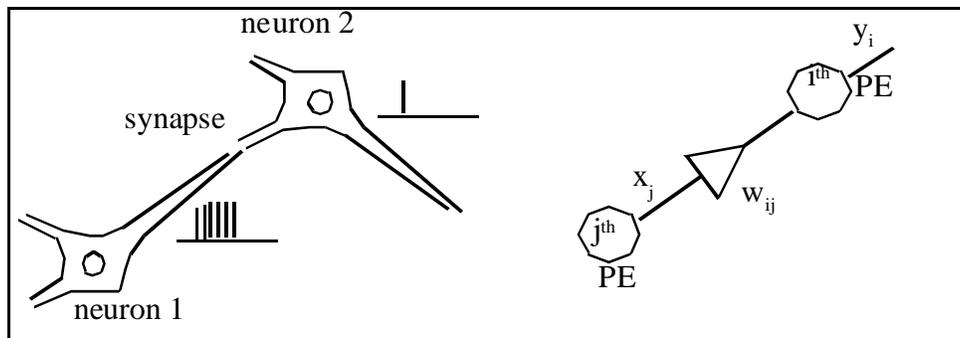


Figure 1. Biological and modeled artificial system

One can extend this idea to artificial systems very easily. In artificial neural systems,

neurons are equivalent to PEs, and PEs are connected through weights. Hence, Hebb's principle will increase the common weight w_{ij} when there is activity flowing from the j^{th} PE to the i^{th} PE. If we denote the output to the i^{th} PE by y_i and the activation of the j^{th} PE by x_j , then

$$\Delta w_{ij} = \eta x_j y_i \quad \text{Equation 1}$$

where η is our already known step size which controls what percentage of the product is effectively used to change the weight. There are many more ways to translate Hebb's principle in equations, but Eq. 1 is the most commonly used and is called *Hebb's rule*.

Unlike all the learning rules studied so far (LMS and backpropagation) there is *no desired signal* required in Hebbian learning. In order to apply Hebb's rule *only the input signal needs* to flow through the neural network. Learning rules that use only information from the input to update the weights are called **unsupervised**. Note that in unsupervised learning the learning machine is changing the weights according to some *internal rule* specified a priori (here the Hebb rule). Note also that the Hebb rule is local to the weight.

Go to the next section

2. Effect of the Hebb update

Let us see what is the net effect of updating a single weight w in a linear PE with the Hebb rule. *Hebbian learning* updates the weights according to

$$w(n+1) = w(n) + \eta x(n)y(n) \quad \text{Equation 2}$$

where n is the iteration number and η a stepsize. For a linear PE, $y = wx$, so

$$w(n+1) = w(n)[1 + \eta x^2(n)] \quad \text{Equation 3}$$

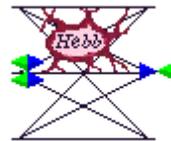
If the initial value of the weight is a small positive constant ($w(0) \sim 0$), irrespective of the

value of $\eta > 0$ and of the input sign, the update will always be positive. Hence, the weight value will increase with the number of iterations without bounds, irrespective of the value of η . This is unlike the behavior we observed for the LMS or backpropagation, where the weights would stabilize for a range of step sizes. Hence, Hebbian learning is intrinsically unstable, producing very large positive or negative weights. In biology this is not a problem because there are natural limitations to synaptic efficacy (chemical depletion, dynamic range, etc).

NeuroSolutions 1

6.1 Training with the Hebbian rule

In this example, we introduce the Hebbian Synapse. The Hebbian Synapse implements the weight update of Equation 2. The Hebbian network is built from an input Axon, the Hebbian Synapse and an Axon, so it is a linear network. Since the Hebbian Synapse, and all the other Unsupervised Synapses (which we will introduce soon), use an unsupervised weight update (no desired signal), they do not require a backpropagation layer. The weights are updated on a sample by sample basis.



This example shows the behavior of the Hebbian weight update. The weights with the Hebbian update will always increase, no matter how small the stepsize is. We have placed a scope at the output of the net and also opened a MatrixViewer to observe the weights during learning. The only thing that the stepsize does is to control the rate of increase of the weights.

Notice also that if the initial weight is positive the weights will become increasingly more positive, while if the initial weight is negative the weights become increasingly more negative.

NeuroSolutions Example

2.1. The multiple input PE

Hebbian learning is normally applied to single layer linear networks. Figure 2 shows a single linear PE with D inputs, which will be called the Hebbian PE. The output is

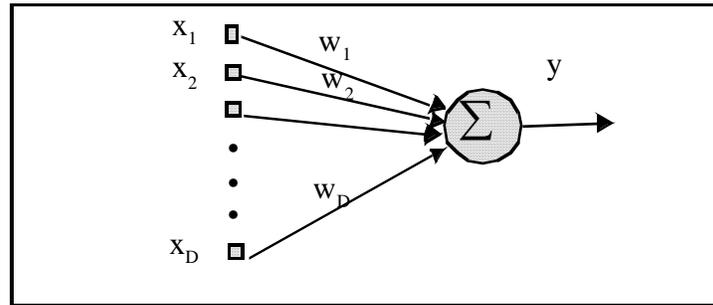


Figure 2. A D input linear PE

$$y = \sum_{i=1}^D w_i x_i$$

Equation 4

According to the Hebb's rule, the weight vector is adapted as

$$\Delta w = \eta \begin{bmatrix} x_1 y \\ \dots \\ x_D y \end{bmatrix}$$

Equation 5

It is important to get a solid understanding for the role of Hebbian learning, and we will start with a geometric interpretation. Eq. 4 in vector notation (vectors are denoted by bold letters) is simply

$$y = \mathbf{w}^T \mathbf{x} = \mathbf{x}^T \mathbf{w}$$

Equation 6

i.e. the transpose of the weight vector is multiplied with the input (which is called the *inner product*) to produce the scalar output y . We know that the inner product is computed as the product of the length of the vectors times the cosine of their angle θ ,

$$y = |\mathbf{w}| |\mathbf{x}| \cos(\theta)$$

Equation 7

So, assuming normalized inputs and weights, a large y means that the input \mathbf{x} is “close”

to the direction of the weight vector (Figure 3), i.e. \mathbf{x} is in the neighborhood of \mathbf{w} .

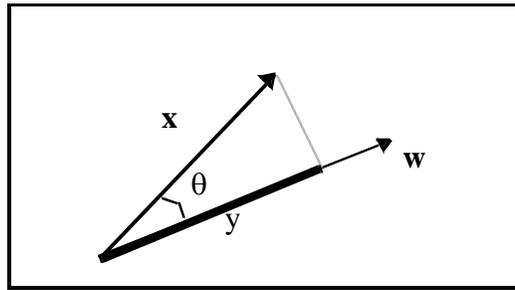


Figure 3. The output of the linear PE in vector space

A small y means that the input is almost perpendicular to \mathbf{w} (cosine of 90 degrees is 0), i.e. \mathbf{x} and \mathbf{w} are far apart. So the magnitude of y measures similarity between the input \mathbf{x} and the weight \mathbf{w} using the inner product as the similarity measure.

This is a very powerful interpretation. During learning the weights are exposed to the data and condense all this information in their value. This is the reason the weights should be considered as the *long-term memory of the network*. **long and short term memory**

The Hebbian PE is a very simple system that creates a similarity measure (the inner product, Eq. 7) in its input space according to the information contained in the weights. During operation, once the weights are fixed, a large output y signifies that the present input is “similar” to the inputs \mathbf{x} that created the weights during training. We can say that the output of the PE responds high or low according to the *similarity* of the present input with what the PE “remembers” from training. So, the Hebbian PE implements a type of memory that is called *an associative memory*

NeuroSolutions 2

6.2 Directions of the Hebbian update

This example shows how the Hebbian network projects the input onto the vector defined by its weights. We use an input which is composed of samples that fall in an ellipse in 2 dimensions, and allow you to select the weights. When you run the network, a custom DLL will display both the input (blue) and the projection of the input onto the weight vector (black) The default is to set the weights to [1,0]

which defines a vector along the x-axis. Thus you would be projecting the input onto the x-axis. Change the value of the weights which will rotate the vector. Notice that in any direction the output will track the input along that direction, i.e. the output is the projection of the input along that specified direction.

Notice also the Megascope display. When the input data circles the origin, the output produces a sinusoidal component in time since the projection increases and decreases periodically with the rotation. The amplitude of the sinusoid is maximal when the weight vector is $[1,0]$ since this is the direction that produces a larger projection for this data set.

If we release the weights, i.e. if they are trained with Hebbian learning the weights will exactly seek the direction $[1,0]$. It is very interesting to note the path of the evolution of the weights (it oscillates around this direction). Note also that they are becoming progressively larger.

NeuroSolutions Example

2.2. The Hamming Network as a primitive associative memory

This idea that a simple linear network embeds a similarity metric can be explored in many practical applications. Here we will exemplify its use in information transmission, where noise normally corrupts messages. We will assume that the messages are strings of bipolar binary values $(-1/1)$, and that we know what are the strings of the alphabet (for instance the ASCII code of the letters). A practical problem is to find from a given string of 5 bits received, which was the string sent. We can think of a n-bit string as a vector in n-dimensional space. The ASCII code for each letter can also be thought as a vector. So the question of finding the value of the received string is the same as asking which is the closest ASCII vector to the received string (Figure 4)? Using the argument above, we should find the ASCII vector in which the bit string produces the largest projection.

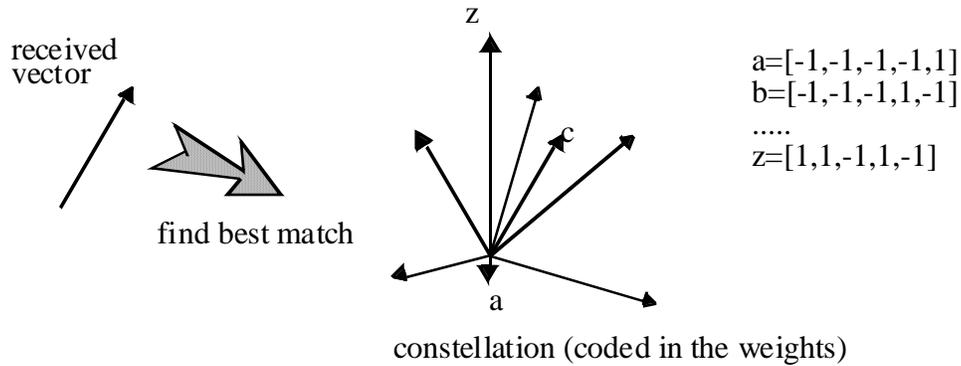


Figure 4. The problem of finding the best match to the received character in vector spaces

A linear network can be constructed with as many inputs as bits on an ASCII code (here we will only use 5 bits, although the ASCII code is 8 bits long) and a number of outputs equal to the size of the alphabet (here 26 letters). The weights of the network will be hard coded as the bit patterns of all ASCII letters. More formally, the inputs are vectors

$\mathbf{x} = [x_1, x_2, \dots, x_5]^T$, the output is a scalar and the weight matrix \mathbf{S} is built from rows that are our ASCII codes represented by $\mathbf{s}_i = [s_{i1}, s_{i2}, \dots, s_{i5}]$, with $i=1, \dots, 26$. The output of the network is $\mathbf{y} = \mathbf{S}\mathbf{x}$.

The remaining question is how to measure the distance between the received vector and each of the ASCII characters. Since the patterns are binary, one possibility is to ask how many bit flips are present between the received string and all the ASCII characters. One should assign the received string to the ASCII character that has the least number of bit flips. This distance is called the Hamming distance - HD (also known as the Manhattan norm or L1 norm).

When a character is received each output i of the network is the scalar product of the input with the corresponding row vector \mathbf{s}_i . This scalar product can be written as the total number of positions in which the vectors agree minus the number of positions they differ which is quantified as their HD. Since the number of positions they agree is $5 - \text{HD}$, we have

$$s_i \mathbf{x} = 5 - HD(s_i, x)$$

This equation states that if we add a bias equal to 5 to each of the outputs of our net, we can directly interpret the network output as an Hamming distance (to be exact the weights should be multiplied by 0.5, and the bias should be 0.5 to obtain the HD). A perfect match will provide an output of 5. So one just needs to look for the highest output to know what was the character that was sent.

NeuroSolutions 3

6.3 Signal detection with Hamming networks

In this example we create the equivalent of an Hamming net which will recognize the binary ASCII of 5 letters (A,B,L,P,Z). The input to the network is the last 5 bits of each letter. For instance, A is -1,-1,-1,-1,1, B is -1,-1,-1,1,-1, etc.

Because we know ahead of time what the letters will be, we will set the weights to the expected ASCII code of each letter. But here we are not going to use the Hamming distance but the dot-product distance of Hebbian learning. According to the associative memory concept, when the input and the weight vector are same, the output of the net will be the largest possible. For instance, if -1,-1,-1,-1,1 is input to the network, the first PE will respond with the highest possible input. Single step through the data to see that in fact the net gives the correct response. Notice also that the other outputs are not zero since the distance between each weight vector and the input is finite (it depends on the Hamming distance between the input and weight vectors).

When noise corrupts the input, this network can be used to determine which letter the input was most likely to be. Noise will affect each component of the vector, but the net will assign the highest output to the weight vector that lies closer to the noisy input. When noise is small this still provides a good assignment. Increase the noise power to see when the system breaks down. It is amazing that such a simple device still provides the correct output most of the time when the variance of the noise is 2.

NeuroSolutions Example

Note that here we utilized the inner product metric intrinsic to the Hebbian network instead of the Hamming distance, but the result is very similar. In this example the weight matrix was constructed by hand due to the knowledge we have about the problem. In general the weight matrix has to be adapted, and this is where the Hebbian learning is important. Nevertheless this example shows the power of association for information processing.

2.3. Hebbian rule as correlation learning

There is a strong reason to translate Hebb's principle as in Eq. 1 . In fact, Eq. 1 prescribes a weight correction according to the *product* between the *j*th and the *i*th PE activations. Let us substitute Eq. 6 in Eq. 5 to obtain the vector equivalent

$$\Delta \mathbf{w}(n) = \eta y(n) \mathbf{x}(n) = \eta \mathbf{x}(n) \mathbf{x}^T(n) \mathbf{w}(n) \quad \text{Equation 8}$$

In on-line learning the weight vector is repeatedly changed according to this equation using a different input sample for each *n*. However in batch, after iterating over the input data of *L* patterns, the cumulative weight is the sum of the products of the input with its transpose

$$\mathbf{w}(L) = \mathbf{w}(0) \sum_{n=1}^L \mathbf{x}(n) \mathbf{x}^T(n) \quad \text{Equation 9}$$

Eq. 9 can be thought of as a sample approximation to the autocorrelation of the input data which is defined as $\mathbf{R}_x = E[\mathbf{x}\mathbf{x}^T]$ where $E[.]$ is the expectation operator (see Appendix). Effectively the Hebbian algorithm is updating the weights with a sample estimate of the autocorrelation function

$$\mathbf{w}(L) = \eta \hat{\mathbf{R}}_x \mathbf{w}(0) \quad \text{Equation 10}$$

Correlation is a well known operation in signal processing and in statistics, and it measures the **second order statistics** of the random variable under study. First and

second order statistics are sufficient to describe signals modeled as Gaussian distributions as we saw in Chapter II (i.e. first and second statistics is what is needed to completely describe the data cluster). Second order moments also describe many properties of linear systems, such as the adaptation of the linear regressor studied in Chapter I.

2.4. Power, Quadratic Forms and Hebbian Learning

As we saw the output of a linear network is given by Eq. 6. We will define the power

energy, power and variance at the output given the data set $\{\mathbf{x}(1), \mathbf{x}(2), \dots, \mathbf{x}(L)\}$ as

$$P = \frac{1}{L} \sum_{n=1}^L y^2(n) = \mathbf{w}^T \mathbf{R}_x \mathbf{w} \quad \text{where} \quad \mathbf{R}_x \approx \mathbf{R} = \frac{1}{L} \sum_{n=1}^L \mathbf{x}(n) \mathbf{x}^T(n) \quad \text{Equation 11}$$

11

P in Eq. 11 is a quadratic form, and it can be interpreted as a field in the space of the weights. Since \mathbf{R} is positive definite we can further say that this field is a paraboloid facing upwards passing through the origin of the weight space (Figure 5).

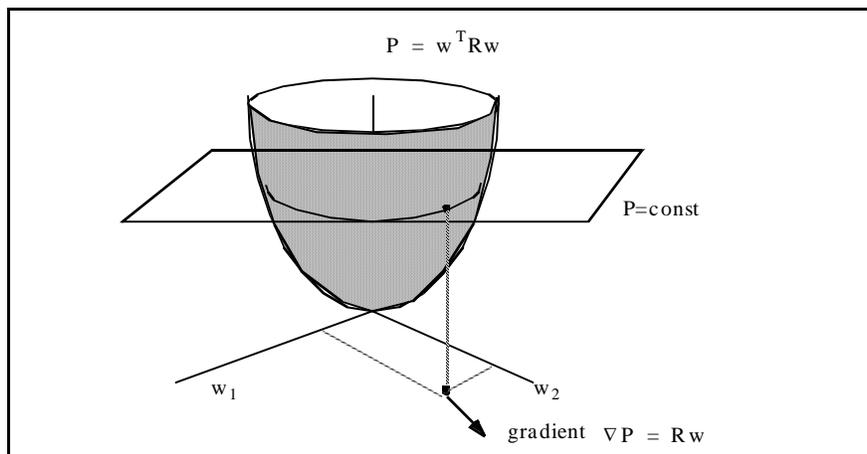


Figure 5. The power field as a performance surface

Let us take the gradient of P with respect to the weights

$$\nabla P = \frac{\partial P}{\partial \mathbf{w}} = 2\mathbf{R}\mathbf{w}$$

We can immediately recognize that this equation provides the basic form for the Hebbian

update of Eq. 10. If we recall the performance surface concept of Chapter I, we see immediately that the power field is the performance surface for Hebbian learning. So we conclude that when we train a network with the Hebbian rule we are doing gradient ASCENT (seeking the maximum) in the power field of the input data. The sample by sample adaptation rule of Eq. 8 is merely a stochastic version and follows the same behavior. Since the power field is unbounded upwards we can immediately expect that Hebbian learning will diverge, unless some type of normalization is applied to the update rule. **Instability of Hebbian** This is a shortcoming for our computer implementations because due to the limited dynamic range it will produce overflow errors. But there are many ways to normalize the Hebbian update.

NeuroSolutions 4

6.4 Instability of Hebbian

This example shows that the Hebbian update rule is unstable since the weights grow without bound. We use a simple 2D input example to show that the weight vector grows. We have opened a MatrixViewer to see the weights, and we also plot the tip of the weight vector in the ScatterPlot as a blue dot (think of the weight vector as going from the origin to the blue dot). Notice however, that the weight vector diverges always along the same direction. This is not by chance. Although unstable the Hebbian network is finding the direction where the output is the largest. The more you train the network, the larger the weights get. Repeat several times to observe the behavior we describe. So the Hebbian update is not practical.

NeuroSolutions Example

2.5 Data representations in multidimensional spaces

An important question is what does the direction of the gradient ascent represent? In order to understand the answer to this question we have to talk about data representations in multidimensional spaces. We normally collect information about the real world events with sensors. Most of the times the data to model a real world phenomenon is multidimensional, that is, we need several sensors (such as temperature,

pressure, flow, etc.). This immediately says that the state of the real world system is multidimensional, in fact a point in a space where the axes are exactly our measurement variables. In Figure 6 we show a two dimensional example. So the system states create a cloud of points somewhere in this measurement space.

An alternative to describe the cloud of points is to define a new set of axes that are “glued” to the cloud of points instead of with the measurement variables. This new coordinate system is called a *data dependent* coordinate system. From the Figure 6 we see that the data dependent representation moves the origin to the center (the mean) of our cloud of samples. But we can do more. We can also try to align one of the axes with the direction where the data has the largest projection. This is called the “*principal*” *coordinate system for the data*. For simplicity we also would like the principal coordinate system to be orthogonal (more on this later). Notice that the original (measurement) coordinate system and the principal coordinate system are related by a translation and a rotation, which is called an affine transform in algebra. If we know the parameters of this transformation we have captured a lot about the structure of our cloud of data.

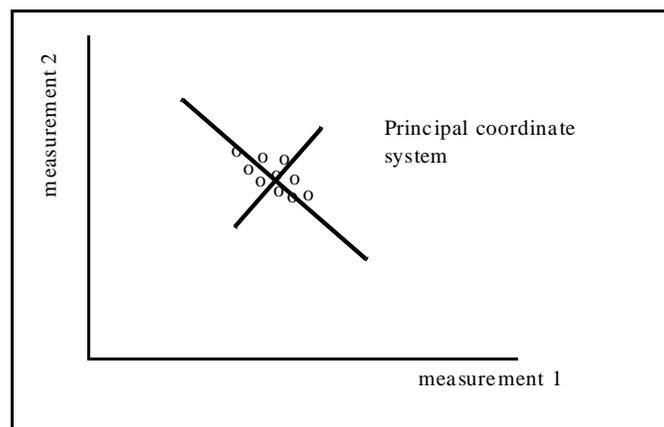


Figure 6. The principal coordinate system

What we gain with the principal coordinate system is knowledge about the structure of the data and versatility. We may say, I want to represent my data into a smaller dimensional space to simplify the problem, or to be able to visualize the data, etc.

Suppose that we are interested in preserving the variance of the cloud of points, since

variance is associated with information [Information and Variance](#) . To make the point clear let us try to find only a *single direction* (i.e. a one dimensional space) to represent most of the variance on our data. What direction should we use? If you think a bit, the principal coordinate system is the one that makes more sense, because we aligned one of the axes with the direction where the data has the largest variance. In this coordinate system we should then choose the axis where the data has the largest projected variance.

Now let us go back to the Hebbian network. The weights of the network trained with the Hebbian learning rule find the direction of the input power gradient. The output of the Hebbian network (the projection of the input into the weight vector) will then be the largest variance projection. Or in other words, the Hebbian network finds the axis of the principal coordinate system where the projected variance is the largest, and gives it as the output. What is amazing is that the simple Hebbian rule automatically finds this direction for us with a local learning rule!

So even though Hebbian learning was biologically motivated, it is a way of creating network weights that are tuned to the second order statistics of the input data. Moreover, the network does this with a rule that is local to the weights. We can further say that Hebbian extracts the most of the information about the input, since from all possible linear projections it finds the one that maximizes the variance at the output (which is a synonym of information for Gaussian distributed variables).

Go to the next section

3. Oja's rule

Perhaps the simplest normalization to Hebb's rule was proposed by [Oja](#) . Let us divide the new value of the weight in [Eq. 2](#) by the norm of the new weight vector connected to the PE, i.e.

$$w_i(n+1) = \frac{w_i(n) + \eta y(n)x_i(n)}{\sqrt{\sum_i (w_i(n) + \eta y(n)x_i(n))^2}}$$

Equation 12

We see that this expression effectively will normalize the size of the weight vector to one.

So if a given weight component increases, the others have to decrease to keep the weight vector at the same length. So weight normalization is in fact a constraint.

Assuming the step size small, Oja approximated the update of Eq. 12 by

$$w_i(n+1) = w_i(n) + \eta y(n)[x_i(n) - y(n)w_i(n)] = w_i(n)[1 - \eta y^2(n)] + \eta x_i(n)y(n)$$

Equation 13

producing the Oja's rule **derivation of Oja's rule**. Note that this rule can still be considered Hebbian update with a normalized activity $x_i(n) = x_i(n) - y(n)w_i(n)$. The normalization is basically a "forgetting factor" proportional to the output square (see Eq. 13).

This equation describes the fundamental problem of Hebbian learning. In order to avoid unlimited growth in the weights, we applied a forgetting term. This solves the problem of weight growth but creates another problem. If the pattern is not presented frequently it will be forgotten since the network forgets old associations.

NeuroSolutions 5

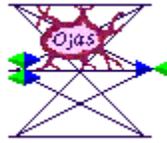
6.5 Oja's rule

This example introduces the Oja's Synapse (look at the synapse with the label Oja).

The network is still a linear network, but the Oja's synapse implements Oja's weight update described in Eq. 13. The overall network function is similar to the Hebbian network except that now the weights stabilize producing a vector in the direction of maximum change. The input data is the same as in the previous case.

Notice that now the weights of the single output network produce a vector oriented along the largest axis of the cloud of input samples (45 degrees). This is the direction which produces the largest possible output. Randomize the weights several times during learning to see that the network quickly finds this direction.

Depending upon the sign of the initial weights, the final weights will be both positive or both negative, but the direction does not change.



The stepsize controls now the speed of convergence. If the stepsize is too large, the iteration will blowup as in the gradient descent learning case. Large stepsizes also produce rattling of the final weights (note that the weights form a linear segment) which should be avoided. If the stepsize is too small, the process will slowly converge. The best is to start the adaptation large and anneal its value to a small constant to fine tune the final position of the weights. This can be accomplished with the scheduler.

NeuroSolutions Example

3.1 Oja's rule implements the principal component network

What is the meaning of the weight vector of a neural network trained with Oja's rule? In order to answer this question let us study a single linear PE network with multiple inputs (Figure 2) using the ideas of vector spaces. The goal is to study the projection defined by the weights created with the Oja's rule. We already saw that Hebbian finds the direction where the input data has the largest projection. But the weight vectors grows without limit. Now with Oja's rule we found a way to normalize the weight vector to 1. If you recall, vectors of length 1 are normally used for axes of coordinate systems. We should expect that this normalization would not change the geometric picture we developed for the Hebbian network. In fact, it is possible to show that Oja's rule finds a weight vector $\mathbf{w}=\mathbf{e}$ which satisfies the relation [proof of eigen-equation](#)

$$\mathbf{R}\mathbf{e}_0 = \lambda_0\mathbf{e}_0 \quad \text{Equation 14}$$

where \mathbf{R} is the autocorrelation function of the input data, and λ is a real value scalar. This equation was already encountered in Chapter I and tells us that \mathbf{e}_0 is an *eigenvector* of the autocorrelation function, since rotating \mathbf{e}_0 by \mathbf{R} (the left side) produces a vector

colinear with itself. We can further show that in fact λ_0 is the largest eigenvalue of \mathbf{R} so \mathbf{e}_0 is the eigenvector that corresponds to the largest eigenvalue. We should expect this since from the eigendecomposition theory we know that the scalar is exactly the variance of the projected data on the eigendirection, and Oja's rule seeks the gradient direction of the power field. We conclude that *training the linear PE with the Oja's algorithm produces a weight vector that is aligned with the direction in the input space where the input data cluster produces the largest variance (the largest projection).*

Figure 7 shows a simple case for 2D. It shows a data cluster (black dots) spread along the 45° line. The *principal axis* of the data is the direction in 2D space where the data has its largest power (projection variance). So imagine a line passing through the center of the cluster and rotate it so that the data cluster produces the largest spread in the line. For this case the direction will be close to 45°. The weight vector of the network of Figure 2 trained with Oja's rule coincides exactly with the principal axis, also called the principal component. The direction perpendicular to it (the *minor axis*) will produce a much smaller spread. For zero mean data, the direction of maximum spread coincides with the direction where most of the information about the data resides. The same thing happens when the data exists in a larger dimensionality space D , but we can not visualize it anymore.

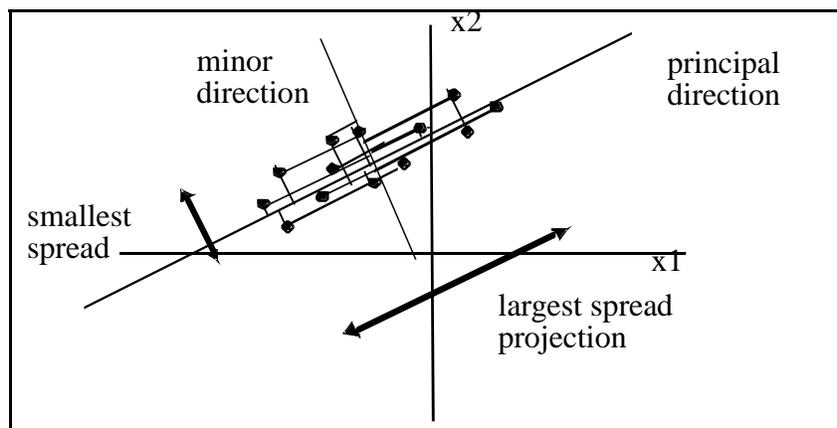


Figure 7. Projection of a data cluster onto the principal components

If you relate this figure with the NeuroSolutions example, the Oja's weight vector found

exactly the direction where the data produced the largest projection. This is a very important property because the simple one PE network trained with Oja's rule is extracting the most information that it can from the input, if we think that information is associated with power of the input. In engineering applications where the input data is normally corrupted by noise, this system will provide a solution that will maximize the signal-power (of the largest sinusoidal component) to noise-power ratio [definition of eigenfilter](#)

[Go to the next section](#)

4. Principal Component Analysis

We saw that Oja's rule found a unity weight vector that is colinear with the principal component of the input data. But how can we find still other directions where the data cluster has still appreciable variance? We would like to create more axes of the principal coordinate system mentioned in section 2.5. For simplicity we would like to create an orthogonal coordinate system (i.e. all the vectors are orthogonal to each other) with unit length vectors (orthonormal coordinate system). How can we do this? Principal Component Analysis answers this question.

Principal Component Analysis or PCA for short is a very well known statistical procedure that has important properties. Suppose that we have input data of very large dimensionality (D dimensions). We would like to project this data to a smaller dimensionality space M ($M < D$), a step that is commonly called *feature extraction*. Projection will always distort somewhat our data (just think of a 3-D object and its 2-D shadow). Obviously we would like to do this projection to M dimensional space *preserving maximally the information* (variance from a representation point of view) about the input data. The linear projection that accomplishes this goal is exactly the PCA. [PCA, SVD, and KL transforms](#) .

PCA produces an orthonormal basis that is built from the eigenvectors of the input data autocorrelation function. The projections onto each basis are therefore the eigenvalues of

R. If one orders the eigenvectors by descending order of the eigenvalues, and we truncate them at M (with $M < D$) then we will project the input data to a linear space of (smaller) dimensionality M. In this space the projection onto each axis will produce the M largest eigenvalues, so there is no better linear projection to preserve the input signal power. The outputs of the PCA represent the input into a smaller subspace so they are called *features*. So PCA is the best linear feature extractor for signal reconstruction. The error e in the approximation when we utilize M features is exactly given by

$$e^2 = \sum_{i=M+1}^D \lambda_i$$

Equation 15

Eq. 15 tells that the *error power is exactly equal to the sum of the eigenvalues that were discarded*. For the case of Figure 7, the minimum error in representing the 2-D data set in an 1-D space is obtained when the principal direction is chosen as the projection axis. The error power is exactly given by the projection on the minor direction. If we decided to keep the projection in the minor direction, the error incurred would have been much higher. This method is called *subspace decomposition* and it is widely applied in signal processing and statistics to find the best sub-space of a given dimension that preserves maximally data information. There are well known algorithms that compute analytically the PCA, but they have to solve matrix equations ([Singular value decomposition](#)).

Can we build a neural network that implements PCA on-line, with local learning rules? The answer is affirmative. We have to use a linear network with multiple outputs (equal to the dimension M of the projection space) as in Figure 8.

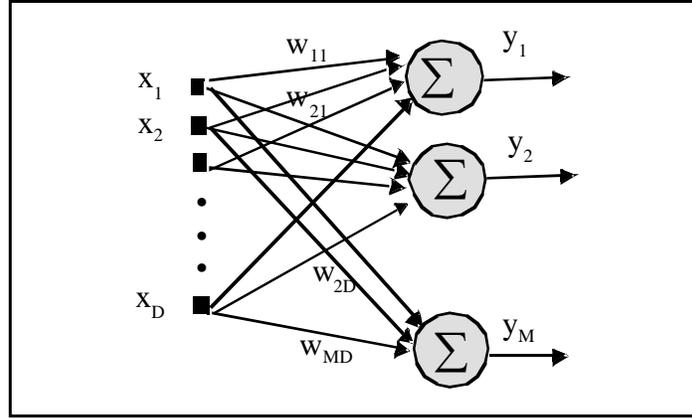


Figure 8. A PCA network to project the data from D to M dimensions.

The idea is very simple. First, we compute the largest eigenvector as done above with Oja's rule. Then we project the data onto a space perpendicular to the largest eigenvector and we apply the algorithm again to find the second largest principal component, and so on until order $M \leq D$. The projection onto the orthogonal space is easily accomplished by subtracting the output of all previous output components (after convergence) from the input. This method is called the *deflation method* and mimics the Gram-Schmidt orthogonalization procedure [Gram-Schmidt orthogonalization](#).

What is interesting is that the deflation method can be accomplished easily by modifying slightly Oja's learning rule as was first done by [Sanger](#). We are assuming that the network has M outputs each given by

$$y_i(n) = \sum_{j=1}^D w_{ij}(n)x_j(n) \quad i = 1, \dots, M \quad \text{Equation 16}$$

and D inputs ($M \leq D$). To apply Sanger's rule the weights are updated according to

$$\Delta w_{ij}(n) = \eta y_i(n) \left[x_j(n) - \sum_{k=1}^i w_{kj}(n)y_k(n) \right] \quad \text{Equation 17}$$

This rule resembles the Oja's update, but now the input to each PE is modified by subtracting the outputs from the preceding PEs times the respective weights. This implements the *deflation method*, after the system converges. The weight update of Eq.

17 is not local since we need all the previous network outputs to compute the weight update to weight w_{ij} . However, there are other rules that use local updates (such as the APEX algorithm [Diamantaras](#)).

As we can expect from Eq. 17 and the explanation there is a coupling between the modes, i.e. only after convergence of the first PE weights will the second PE weights converge completely to the eigenvector that corresponds to the second largest eigenvalue . There are other on-line algorithms for the same purpose, such as the lateral inhibition network and the recursively computed APEX, but for the sake of simplicity they will be omitted here.

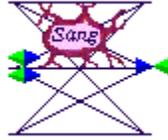
A two output PCA network will have weight vectors that correspond to the principal and minor component of Figure 5. The two outputs will correspond to the largest and smallest eigenvalues respectively. *The interesting thing about subspace projections is that in many problems the data is already restricted to a (unknown) subspace, so PCA can effectively perform data compression preserving the major features of the data.*

NeuroSolutions 6

6.6 Sanger's and PCA

This example introduces Sanger's rule (look at the synapse with the label Sang in the breadboard). Sanger's rule does Principal Component Analysis (PCA). The dimension M of the output determines the size of the output space, i.e. the number of eigenvectors and also the number of features used to represent the input data. PCA finds the M weight vectors which capture the most information about the input data. For instance, a 3 output Sanger's network will find 3 orthogonal vectors, the principal axis which captures the more information than any other vector in the input space, along with the two vectors which capture the second most and third most information. In this example, we take a high dimensional input, 8x8 images of the 10 digits, and project them onto their M Principal components. M is a variable that you can control by setting the number of outputs of the Sanger's network. The outputs of the PCA network are the features

obtained by the projection.



We then use a custom DLL to recreate the digits using only the M features. This DLL takes the output of the Sanger's network and multiplies it by the transpose of W , so it recreates a 64 output image. This image shows us how much of the original information in the input we have captured in the M dimensional subspace. When the two images are identical, we have preserved in the features the information contained in the input data.

The display of the eigenvectors (the PCA weights) is not easy since they are vectors in a 64 dimensional space. After convergence they are orthogonal. We can use the Hinton probe to visualize their value, but it is difficult to find patterns (in fact the signs should alternate more frequently towards the higher order meaning that finer details is being encoded). Try different values for the subspace dimension (M), and verify that PCA is very robust, i.e. even with just a few dimensions the reconstructed digits can be recognized.

A word of caution is needed at this point. The PCA finds the subspace that best represents the ensemble of digits, so the best discrimination among the digits in the subspace is not guaranteed with PCA. If the goal is discrimination among the digits then a classifier should be designed for that purpose. PCA is a linear representation mechanism, and only guarantees that the features contain the most information for reconstruction.

NeuroSolutions Example

The PCA decomposition is a very important operation in data processing because it provides knowledge about the hidden structure (latent variables) of the data. As such there are many other possible formulations for the problem. [PCA derivation](#)

4.1. PCA for data compression

PCA is the optimal linear feature extractor, i.e. *there is no other linear system that is able to provide better features for reconstruction*. So one of the obvious PCA applications is data compression. In data compression the goal is to be able to transmit as fewer bits per second as possible preserving as much as the source information as possible. So this means that we must “squeeze” in each bit as much information as possible from the source. We can model data compression as a projection operation where the goal is to find a set of basis that produce a large concentration of signal power in only a few components.

In PCA compression the receiver must know the weight matrix containing the eigenvectors since the estimation of the input from the eigenvalues is done by

$$\tilde{\mathbf{x}} = \mathbf{W}^T \mathbf{y} \quad \text{Equation 18}$$

The weight matrix is obtained after training with exemplars from the data to be transmitted. It has been shown that for special applications this step can be completed efficiently and is done only once. So the receiver can be constructed before hand. The reconstruction step requires $M \times D$ operations where D is the input vector dimension and M is the size of the subspace (number of features).

4.2. PCA features and classification

We may think that a system able to preserve optimally signal energy in a subspace should also be the optimal projector for classification. Unfortunately this is not the case. The reason can be seen in Figure 9. We have here represented two classes. When the PCA is computed no distinction is made between the samples of each class so the optimal 1-D projection for reconstruction (the principal direction) is along the x_1 axis. However it is easy to see that the best discrimination between these two clusters is along the x_2 axis which from the point of view of reconstruction is the minor direction.

So PCA chooses the projections to best reconstruct the data in the chosen subspace. This may or may not coincide with the projection for best discrimination. A similar thing

happened when we addressed regression and classification (first example of Chapter II). A linear regressor can be used as a classifier, but there is no guarantee that it produces the optimal classifier (which by definition minimizes the classification error).

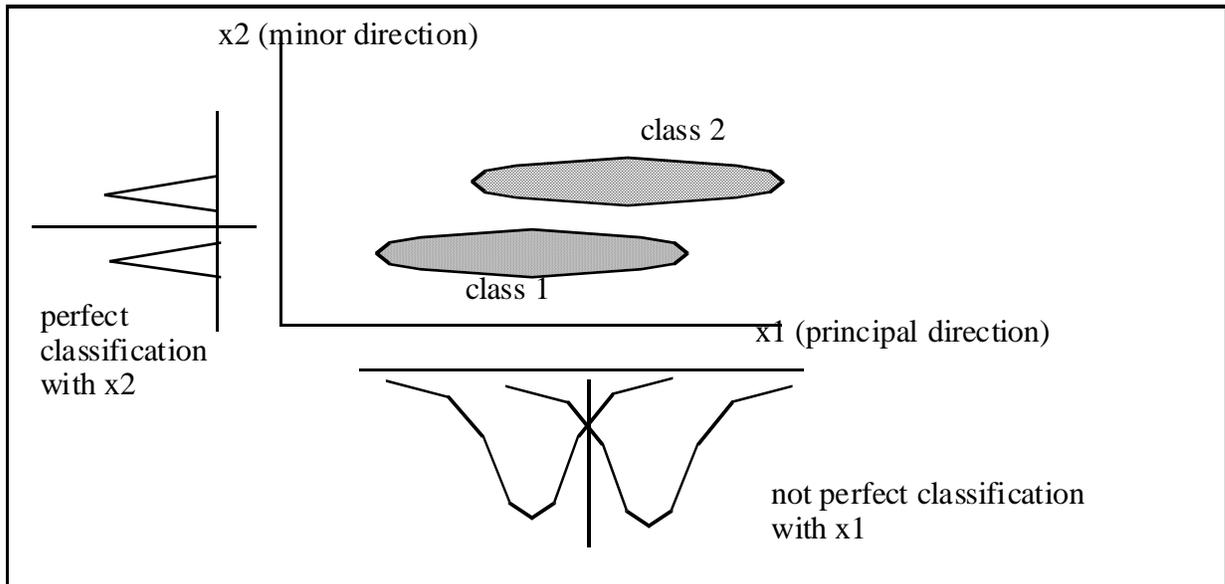


Figure 9. The relation between eigendirections and classification

However, PCA is appealing for classification since it is a simple procedure, and experience has shown that it normally provides good features for classification. But this depends upon the problem and there is no guarantee that classifiers based on PCA features work well.

NeuroSolutions 7

6.7 PCA for preprocessing

In this example we use PCA to find the best possible linear projection in terms of reconstruction and then we use a MLP to classify the data into one of 10 classes (the digits). Notice that in fact this problem was already solved in Chapter III with the perceptron and we obtained perfect classification using the input data directly.

The only way we can do a fair comparison is to limit the number of weights in the two systems to the same value and compare performance.

NeuroSolutions Example

Go to the next section

5. Anti-Hebbian Learning

We have seen that Hebbian learning discovers the directions in space where the input data has the largest variance. Let us do a very simple modification to the algorithm and include a minus sign in the weight update rule of Eq. 1 , i.e.

$$\Delta w_{ij} = -\eta x_j y_i \quad \text{Equation 19}$$

This rule is called the *anti-Hebbian* rule. Let us assume that we train the system of Figure 2 with this rule. What do you expect this rule will do?

The easiest reasoning is to recall that the Hebbian network maximizes the output variance by doing gradient ascent in the power field. Now with the negative sign in the weight update equation, the adaptation will seek the minimum of the performance surface, i.e. the output variance will be minimized. Hence, the output of the linear network trained with *anti-Hebbian will always produce zero output*, because the weights will seek the directions in the input space where the data cluster have a point projection. This is called the *null (or orthogonal) space of the data*. The network finds this direction by doing *gradient descent* in the power field.

If the data fills the full input space then the weights will have to go to zero. On the other hand, if the data exists in a subspace, the weights will find the directions where the data projects to a point. For Figure 7 anti-Hebbian will provide zero weights. However, if the data was one dimensional, i.e. along the 45 degree line, then the weights will be placed along the 135 degree line.

NeuroSolutions 8

6.8 Anti-Hebbian learning

In this example we use the Hebbian synapse with a negative stepsize to implement an anti-Hebbian network. The anti-Hebbian rule minimizes the output variance,

thus it will try to find a vector which is orthogonal to the input (the null space of the input) such that the projection of the data onto the weight vector is always zero.

There are two cases of importance. Either the data lies in a subspace of the input space in which case the zero output can be achieved by adapting the weight vector perpendicular to the subspace where the input lies. Or in the second case the input samples covers the full input space, so the only way to get a zero output is to drive the weights to zero.

Notice how fast the anti-Hebbian trains. If the data moves in the input space, notice that the weights are always finding the direction orthogonal to the data cluster.

NeuroSolutions Example

This behavior of anti-Hebbian learning can be translated as *decorrelation*, i.e. a linear PE trained with anti-Hebbian learning decorrelates the output from its input. We must realize that Hebbian and anti-Hebbian have complementary roles in projecting the input data, that are very important for signal processing. For instance the new high resolution spectral analysis techniques (such as MUSIC and ESPRIT [Kay](#)) are based on ways of finding the null space of the data and so they can be implemented on-line using anti-Hebbian learning. We will provide an example in Chapter IX.

5.1. Convergence of anti-Hebbian rule

Another interesting thing is that the convergence of the anti-Hebbian rule can be controlled by the step size, like in LMS or backpropagation. This means that if the step size is too large the weights will get progressively larger (diverge), but if the step size is below a given value the adaptation will converge. In fact from the fact that the power field is a paraboloid in weight space, we know it has a single minimum. Hence the situation is like gradient descent that we studied in Chapter I. What is the value under which the weights converge to finite values?

The anti-Hebbian update for one weight is

$$w(n+1) = w(n)(1 - \eta x^2(n)) \quad \text{Equation 20}$$

So, if we take expectations and project into the principal coordinate system as we did in Chapter I to compute the largest stepsize for the LMS, we can conclude that

$$w(n+1) = (1 - \eta\lambda)w(n) \quad \text{Equation 21}$$

which is stable if

$$\eta < \frac{2}{\lambda} \quad \text{Equation 22}$$

where λ is the eigenvalue of the autocorrelation function of the input. We can immediately see the similarity to the convergence of the LMS rule. For a system with multiple inputs the requirement for convergence has to be modified to

$$\eta < \frac{2}{\lambda_{\max}} \quad \text{Equation 23}$$

where λ_{\max} is the largest eigenvalue of the input autocorrelation function as for the LMS case.

NeuroSolutions 9

6.9 Stability of Hebbian

This example shows that the anti-Hebbian rule is stable for the range of values given by Eq. 23 when random data is utilized. Just change the stepsize to see what is the compromise rattling speed of convergence achieved with the anti-Hebbian. Since the weight update is sample by sample, when the data has deterministic structure divergence may occur at step sizes smaller than the ones predicted by Eq. 23. The same behavior was encountered in the LMS.

NeuroSolutions Example

[Go to the next section](#)

6. Estimating crosscorrelation with Hebbian networks

Suppose that we have two data sets formed by P exemplars of N dimensional data x_1, \dots, x_N and d_1, \dots, d_M , and the goal is to estimate the crosscorrelation between them. The crosscorrelation is a measure of similarity between the two sets of data which extends the idea of correlation coefficient (see [Appendix](#) and [Chapter I](#)).

In practice we are often faced with the question how similar is this data set to that other one. Crosscorrelation helps exactly to answer this question. Let us assume that the data samples are ordered by their indices. The crosscorrelation for index i, j is

$$r_{xd}(i, j) = \frac{1}{L} \sum_{k=1}^L x_{i,k} d_{j,k} \quad 0 < i < N, \quad 0 < j < M \quad \text{Equation 24}$$

where L is the number of patterns, N is the size of the input vector and M is the size of desired response vector. The fundamental operation of correlation is to cross multiply the data samples and add the contributions. Define the average operator

$$A[\mathbf{u}] = \frac{1}{L} \sum_{k=1}^L u_k \quad \text{Equation 25}$$

The crosscorrelation can then be defined as

$$r_{xd}(i, j) = A[\mathbf{x}_i \mathbf{d}_j] \quad \text{Equation 26}$$

where the vector $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{iP}]^T$ is built from the i^{th} sample of all the patterns in the input set (likewise for \mathbf{d}). The crosscorrelation matrix \mathbf{R}_{xy} is built from all possible shifts i, j , i.e

$$\mathbf{R}_{xd} = A \begin{bmatrix} \mathbf{x}_1 \mathbf{d}_1 & \mathbf{x}_1 \mathbf{d}_2 & \mathbf{x}_1 \mathbf{d}_N \\ \dots & \dots & \dots \\ \mathbf{x}_N \mathbf{d}_1 & \mathbf{x}_N \mathbf{d}_2 & \mathbf{x}_N \mathbf{d}_N \end{bmatrix} \quad \text{Equation 27}$$

The crosscorrelation vector used in regression ([Chapter I](#)) is just the first column of this matrix. Now let us relate this formalism to the calculations of a linear network trained with

Hebbian learning. Assume we have a linear network with N inputs \mathbf{x} and N outputs \mathbf{y}
(Figure 10)

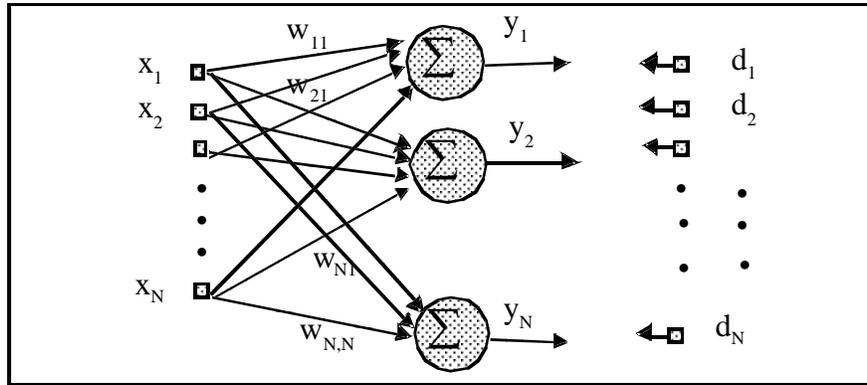


Figure 10. A multiple input multiple Hebbian network

In order to compute the cross correlation between \mathbf{x} and the data set \mathbf{d} , we will substitute the network output \mathbf{y} in the Hebbian rule by the data set \mathbf{d} , i.e.

$$\Delta w_{ij} = \eta x_j d_i \quad \text{Equation 28}$$

which implements what we call *forced Hebbian learning*. We can write the output y_j as Eq. 4 but now with two indices i and j

$$y_j = \sum_{k=1}^N w_{k,j} x_k \quad \text{Equation 29}$$

The weight $w_{i,j}$ when adapted with forced Hebbian learning takes the form

$$w_{i,j}(n+1) = w_{i,j}(n) + \eta x_j(n) d_i(n) \quad \text{Equation 30}$$

If $w_{ij}(0)=0$ after L iterations we get

$$w_{i,j}(L) = \eta \sum_{n=1}^L x_j(n) d_i(n) \quad \text{Equation 31}$$

So by comparing Eq.24 with Eq. 31 we conclude that the weight w_{ij} trained with forced Hebbian learning is proportional after P iterations to the crosscorrelation element r_{ij} . If $\eta=1/L$ and the initial conditions are zero this is exactly r_{ij} . Notice also that the elements of the crosscorrelation matrix are precisely the weights of the linear network (Eq. 27). For

this reason the linear network trained with forced Hebbian learning is called a *correlator* or a *linear associator*. Hence, *forced Hebbian learning is an alternate, on-line way of computing the crosscorrelation function between two data sets.*

NeuroSolutions 10

6.10 Forced Hebbian computes crosscorrelation

In this example we show how forced Hebbian learning simply computes the crosscorrelation of the input and desired output. We have a 3 input network which we would like to train with a desired response of 2 outputs. We have created a data set with 4 patterns. The crosscorrelation computed according to Eq. 24 is

$$r(0,0)= 0.5 ; r(0,1)=r(1,0)=0; r(1,1)=0.25; r(0,2)=0.5 ;r(1,2)=0.25$$

Let us use the Hebbian network and take a look at the final weights. Notice that we started the weights with a zero value, and stopped the network after 10 iterations of each batch (4 patterns) with a stepsize of 0.025 (1/4x10).

NeuroSolutions Example

There are two important applications of this concept that we will address in this chapter. One uses crosscorrelation with anti-Hebbian learning to find “what is different” between two data sets, and can be considered a novelty filter. The other is possibly even more important and is a memory device called an associative memory.

Go to the next section

7. Novelty Filters and Lateral Inhibition

Let us assume that we have two data sets x and d . Taking x as the input to our system, we want to create an output y as *dissimilar* as possible to the data set d (Figure 11). This function is very important in signal processing (decorrelation) and in information processing (uncorrelated features), and it seems to be at the core of biological information processing. We humans filter out with extreme ease what we know already from the sensor input (either visual or acoustic). This avoids information overload. It

seems that what we do first is to equalize the incoming information with what is expected, such that *unexpected things stand out*.

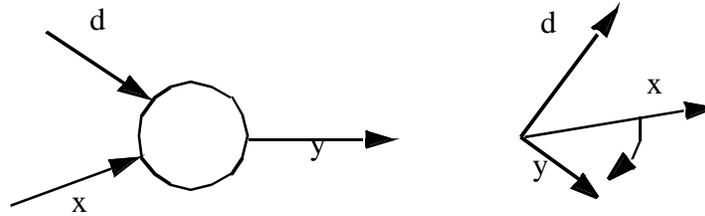


Figure 11. The function of a decorrelation system

We may think that the incoming data is represented by x , and what we already know is represented by d . So *novelty is the part of x that is not represented in d* . From a point of view of vector operations, this is equivalent to finding a rotation to x such that y is orthogonal to d . The system of Figure 10 with the learning rule of Eq. 19 where d substitutes y (i.e. $\Delta w = -\eta xd$) does exactly this job.

NeuroSolutions 11

6.11 Novelty filter with anti-Hebbian learning

In this example, we show an example of a novelty filter. We have created a three dimensional input signal which represents the output of a system under normal operating conditions. This system could be a car (outputs = velocity, acceleration, and turning angle), a power plant, or any other system. We will train the novelty filter on this data and the anti-Hebbian learning will learn it's null space – the vector where the input projection is always very close to zero. The weights are fixed at this point.

When the system changes slightly (abnormal system operation) and its output is fed to the trained novelty filter, the filter output is no longer close to zero because the new signal is no longer in the null space of the filter weights. This indicates that the system is no longer operating normally. We will change the parameters of the system midway in the experiment. From the filter output you should be able to pin point where the change occurred. Notice that the system output looks basically

unchanged throughout the segment, so it would be difficult to find the change in parameters.

NeuroSolutions Example

7.1 Lateral Inhibition

Another very useful strategy to decorrelate signals is to create lateral connections between PEs adapted with anti-Hebbian learning (see Foldiak). We will analyze the topology depicted in Figure 12. In the Figure, c is the lateral inhibition connection from y_i to y_j . We use the $+$ superscript to mean the pre-activity of the PEs. Note that

$$y_i = y_i^+$$

$$y_j = cy_i^+ + y_j^+$$

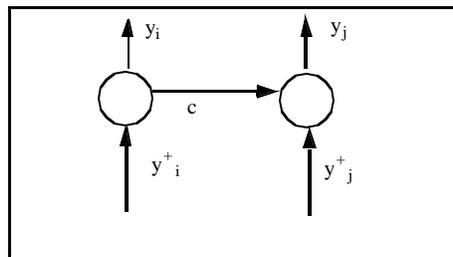


Figure 12. Lateral inhibition connections

The crosscorrelation between y_i and y_j is

$$R_{ij}(y_i, y_j) = c \sum_{n=1}^N [y_i^+(n)]^2 + \sum_{n=1}^N y_i^+(n) y_j^+(n)$$

If the power of y_i is greater than zero, then there is always a value

$$c = - \frac{\sum_{n=1}^N y_i^+(n) y_j^+(n)}{\sum_{n=1}^N [y_i^+(n)]^2}$$

which will decorrelate y_i and y_j , i.e. will make $R(y_i, y_j) = 0$. Notice that this value is the negative of the crosscorrelation between the i th and j th PE activations. So if we use the anti-Hebbian learning with a small stepsize, the outputs will be decorrelated. Notice that one of the characteristics of the PCA is that the outputs were orthogonal, i.e. the outputs were uncorrelated. The lateral inhibition is basically achieving the same thing, however

the variance of the outputs are not being constrained, nor the weight vectors.

The interesting thing about lateral inhibition is that it can provide an alternative method to construct networks that find the principal component space with a local learning rule, or even provide whitening transforms (i.e. a transform that not only orthogonalize the input data but also normalize the eigenvalues).

7.2 APEX model for PCA

Diamantaras has shown that the network of Figure 13 implements PCA when the weights are adapted according to

$$\begin{cases} \Delta w_i = \eta y_i(n)[x(n) - y_i(n)w_i] \\ \Delta c_{ji} = -\eta y_i(n)[y_j(n) + y_i(n)c_{ji}] \end{cases}$$

Note that the weight is adapted using Oja's rule, while the anti-Hebbian learning is used to adapt the lateral connections. Note that all the quantities are local to the weights, so the rule is actually local.

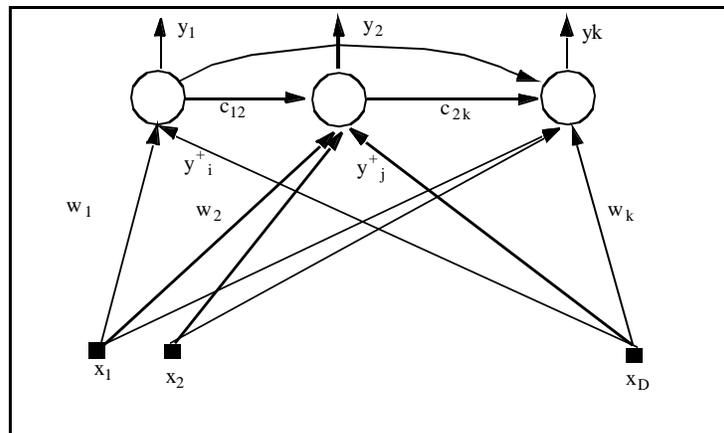


Figure 13. The APEX topology for PCA

7.3 Whitening Transform

A whitening transform is a very important linear transformation in adaptive systems, because it transforms any data described by an autocorrelation R with a large eigenvalue spread into an orthonormal matrix, i.e. a matrix with all the eigenvalues equal to a constant. For whitened data the LMS algorithm is as fast as Newton's method since the

eigenvalue spread is 1. So whitening the input data, will drastically improve the speed of linear learning systems using first order methods. We present now a topology and learning rule that will produce a whitening transform (see [Silva](#)).

The network that implements the whitening transform is as Figure 13. The idea of the algorithm is very similar to the Gram-Schmidt procedure (Figure 14), but it adapts all the vectors at the same time, yielding a symmetric adaptation structure. The adaptation rule reads

$$w_{ij}(n+1) = (1 + \eta)w_{ij}(n) - \eta \sum_{k=1}^D y_i(n)y_k(n)w_{kj}(n)$$

Notice that this formula specifies a weight update which is not local to the weights. However with lateral inhibition we can easily implement it in a single layer network (Figure 13). Notice that the sum over k can be implemented by bringing a lateral inhibition connection from the kth PE to the ith PE with a weight copied from the forward connections and connecting the jth input with the kth PE. Silva discusses another implementation and also proves the convergence of the algorithm. The interesting thing of this transformation is that it creates an orthonormal space at the output by equalizing the eigenvalues instead of by rotating the axis as done in PCA (Figure 14). This was reported much faster than PCA for a variety of problems (and PCA does not guarantee an orthonormal basis).

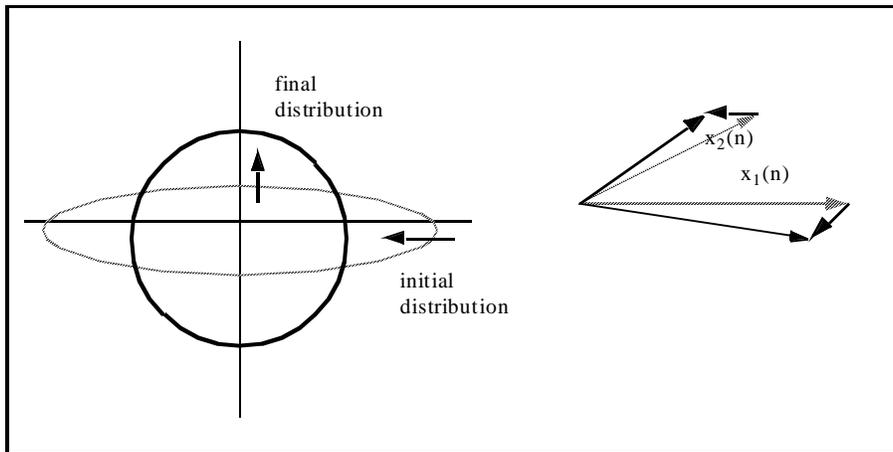


Figure 14. Whitening transform and the corresponding weight update

Go to the next section

8. Linear associative memories (LAMs)

Information processing requires memorization of information. In digital computers one memory location stores one bit of information, so the information is stored individually. An interesting question is to seek ways to store information in a more global way, i.e. have several PEs store many bit (data) patterns. And then ask the question what system is more efficient and robust to noise.

The linear associator also called a linear associative memory (LAM) provides an alternative computer memory paradigm. The research has strong ties to psychology since it is today pretty well accepted that the brain does not store each bit of information separately in each neuron. Many neurons (cell assembly) store many patterns.

The system of Figure 10 with the Hebbian rule (Eq. 1) can be used as an associative memory, i.e. a device that can be trained to associate an input \mathbf{x} to a response \mathbf{d} . Then in the absence of \mathbf{d} , \mathbf{x} can produce an output \mathbf{y} that resembles \mathbf{d} . So the question is how can information be stored globally and how can one retrieve it?

We will utilize the matrix notation for convenience. Let the input of N elements be denoted as a vector \mathbf{x} . Likewise \mathbf{y} is the N component output vector. The output being linear can be obtained as $\mathbf{y} = \mathbf{W}\mathbf{x}$ where \mathbf{W} is the weight matrix. The Hebbian learning constructs each weight according to Eq. 1, which can be written in matrix notation as the outer product, i.e. $\mathbf{W} = \mathbf{d}\mathbf{x}^T$. So, when the input \mathbf{x} is entered in the linear associator the output created by the system is

$$\mathbf{y} = \mathbf{d}\mathbf{x}^T \mathbf{x} \propto \mathbf{d} \quad \text{Equation 32}$$

which is proportional to the original output utilized in the training (remember that $\mathbf{x}^T \mathbf{x}$ is a constant equal to the length of the vector \mathbf{x}).

The interesting question is what happens when more than one input vector is stored in

the memory? Can we still recover each one of the inputs or is the output contaminated by the other inputs?

NeuroSolutions 12

6.12 LAM application

In this example we use a linear associative memory (LAM) to associate area codes (3 digits) with prices for long distance phone calls (2 digits). When we input an area code, we would like the network to output the correct price for the corresponding long distance phone call rate. So during training we will use hetero-association to train the LAM. We have encoded the area codes and rates as binary digits (12 and 8 bits respectively). Hence this LAM will have 12 inputs and 8 outputs.

We have created input files which contain a set of 3 binary encoded area codes and prices. We have also added a custom DLL which will allow us to display a sequence of binary digits as the equivalent number. Once the network is trained, we can present the area code and the system will produce at the output the corresponding long distance call rate.

It is interesting to ask where is the information stored. The answer is in the weights, throughout the network. This is rather different from the storage we use in digital computers where the memory is addressed. If one loses the address the item stored can never be recovered. Here we are recalling the output by providing the input (i.e. the content of the memory, so these memories are called content addressable).

Content addressable memories are very robust. Just go with a matrix editor and zero one (or several weights), and observe that the output barely changes (notice that the numbers displayed are subject to an encoding, so they only change when there are drastic modifications in one of the bits). If one bit was lost in the address or content of a computer memory, the original content would be

impossible to retrieve (except if coding - which is redundancy- was used).

Another interesting thing is that these memories cover the input space with a similarity measure (the inner product metric as we have seen). For information in the form of numbers this is not that important since numeric information is normally precise. But for names, words, concepts, etc. similarity makes a lot of sense. (is his name Gary, Cary, Gerry, Larry, ???). To see this property of LAMs let us just change one of the input digits and see that the output is basically unchanged. These are nice properties of LAMs which make them very good models for human memory in cognitive science.

NeuroSolutions Example

8.1. Crosstalk in LAMs

Let us assume that we have K input-output vector pairs $\mathbf{x}_k \rightarrow \mathbf{d}_k$. The associative memory is trained by repeated presentation of each input, so using the principle of superposition the final weight matrix is the sum of the individual weight matrices

$$\mathbf{W} = \sum_{k=1}^K \mathbf{W}_k \quad \text{Equation 33}$$

where each $\mathbf{W}_k = \mathbf{x}_k \mathbf{x}_k^T$. Now when an input vector \mathbf{x}_l is presented to the network its output is

$$\mathbf{y} = \mathbf{W}\mathbf{x}_l = \mathbf{d}_l \mathbf{x}_l^T \mathbf{x}_l + \sum_{k=1, k \neq l}^K \mathbf{d}_k \mathbf{x}_k^T \mathbf{x}_l \quad \text{Equation 34}$$

The associative memory output is built up from two terms. The first, which is the true output for the input \mathbf{x}_l is added with a term that is called the *crosstalk* because it measures how much the other outputs interfere with the true one. But if the crosstalk term is small, Eq. 34 tells us that in fact the associative memory is able to retrieve the pattern that corresponded to \mathbf{x} during training (the association).

The crosstalk is a function of how similar the input \mathbf{x}_i is to the other inputs \mathbf{x}_k . This can be better understood in a geometric setting. Assume that the input patterns are vectors in a N dimensional vector space. The output of the linear associator, being a product of a matrix by a vector (Eq. 34), rotates the input \mathbf{x}_i to obtain \mathbf{y} . The goal is to obtain a rotation that produced the expected association to \mathbf{d}_i . What Eq. 34 is saying is that the actual output \mathbf{y} is constructed by two terms. The first is the desired output \mathbf{d}_i scaled by the length of \mathbf{x}_i , and a sum of contributions that depend on the inner product of all the other input patterns \mathbf{x}_k with \mathbf{x}_i . Figure 15 shows the construction for two vectors only.

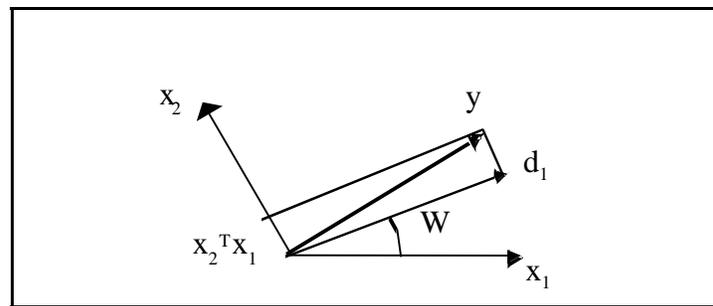


Figure 15. Output with crosstalk

If the inputs are all orthogonal, the inner product of \mathbf{x}_i and \mathbf{x}_k is zero (zero crosstalk), and the linear associator produces perfect recall. However, if the input patterns are not orthogonal, each one of the \mathbf{y}_k in the sum is multiplied by the projection of the input vector k on l , which can add up to a large number, rotating and changing the length of the true output \mathbf{d} . If the crosstalk term is comparable to the first term, then the linear associator will produce an output that has nothing to do with the expected response \mathbf{d} .

This analysis brings immediately the concept of *storage capacity*, which is defined as the maximum number of patterns that can be stored and recalled without degradation.

Associative memories, unlike computer memories have finite storage capacity. We know that in a space of dimension N , there are only N possible orthogonal directions, so *perfect recall is limited to a number of patterns equal to the size of the input space* (length of the input vector). In practical conditions the inputs may not be orthogonal to each other. So, if orthogonality is not enforced, the crosstalk term may be large even for a number of

patterns less than N . But it is always possible (although computationally expensive) to project a set of N vectors onto an N dimensional orthogonal basis (as we saw with the PCA). In fact, we do not need to perform PCA, we just need to find a spanning orthogonal set of vectors which is possible with simpler algorithms. So using such a preprocessor one can say that perfect recall can be achieved for a *number of patterns equal to the size of the input layer*. Therefore, the *storage capacity* of the linear associator equals N . When the number of patterns is larger than the space dimensionality a severe degradation of performance can be expected.

NeuroSolutions 13

6.13 LAM and crosstalk

This example is exactly the same as before but now we have added more patterns which happen to be correlated (non-orthogonal). This will produce crosstalk. Run the network and observe that now the output values do not correspond to the desired response for two reasons. First the outputs that were zero have now non zero values (watch the size of the bars) and the desired values are not met anymore for some patterns. The errors get worse when the number of ones in the patterns increase, and also when more patterns are included. This is the problem of the crosstalk.

NeuroSolutions Example

This analysis gives the theoretical basis for associative memories. When we train such a system with a set of input-output vectors using the Hebbian learning, the network will produce an output similar to the individual output, provided the number of patterns is less than the input space dimensionality. Orthogonalization of the patterns may have to be performed to achieve perfect recall.

LAMs are very different from computer memories. They are content addressable and global, while computer memories are location addressable and local. Hence they have very different properties: computer memory is precise (no crosstalk), it has no limitation of

size (just increase the width of the address bus), but it is brittle. Once a bit is in error the full memory system breaks down (which requires error correction). On the other hand, LAMs are very robust to errors in the weights, but they suffer from limited storage and crosstalk. They have also the wonderful property of association, i.e. the pattern that is closest to the input is recalled. How often have you wished to have the property of association when retrieving information from a computer database....

In Chapter XI we will see another type of associative memory with recurrent connections that is able to clean crosstalk to a certain extent, clean the noise from the input, or even complete patterns partially occluded by feeding back the output to the input several times. In each iteration a better approximation of the stored pattern is obtained so the system can self-correct errors. The most famous of these recurrent memories is the Hopfield network.

Go to the next section

9. LMS learning as a combination of Hebb rules

The LMS rule studied in Chapter I can be created by a combination of Hebbian type rules between the desired response and the learning system input. In fact, if we recall the LMS rule

$$\Delta w_{ij} = \eta \varepsilon_j x_i \quad \text{Equation 35}$$

and note that the error ε can be expressed by

$$\varepsilon_j = d_j - y_j \quad \text{Equation 36}$$

we get

$$\Delta w_{ij} = \eta (d_j x_i - y_j x_i) \quad \text{Equation 37}$$

i.e. the LMS rule is a combination of an Hebbian term between the desired response and the input, and an anti-Hebbian term between the PE output and its input. The first term

substitutes the system output by the desired response, so it is the forced Hebbian term. So LMS is a combination of a forced Hebbian and anti-Hebbian rules.

We can interpret the LMS adaptation as a compromise between two different Hebbian forces: the forced Hebbian term that makes the output similar to the desired response, and the anti-Hebbian term that tries to decorrelate the input with the system output. The forced Hebbian term does gradient ascent on the performance surface and will be unstable as we saw above. The anti-Hebbian term decorrelates the input from the output and drives the output to zero, allowing a range of step sizes to produce convergence to the minimum of the performance surface. The anti-Hebbian term is what controls the convergence of the LMS algorithm since the product of desired and input responses is independent of the weights. So it is understandable that the range of stepsizes for convergence for the anti-Hebbian and LMS is the same.

An important conclusion is that the Hebbian principle of correlation is also present in supervised learning. This simple derivation also calls our attention to the fact that the learning principles studied so far in neurocomputing (Hebbian, LMS, and backpropagation) are based on correlation learning (or compositions of correlation learning).

We can alternatively think that the LMS is a “smart” forced Hebbian learning rule which at the same time approximates the system output to the desired response as Hebbian does, but does so without being unstable (for a range of stepsizes) due to the anti-Hebbian component. Hence we can expect that the LMS will improve the forced Hebbian in the same way as Oja improved the Hebbian learning.

9.1. Improving the performance of linear associative memories (OLAMs)

An alternative to orthogonalization of the input patterns is to use different learning rules during training. We can interpret the individual output pattern as the desired response for the linear associator, and then train it with the error

$$\varepsilon = \mathbf{d} - \mathbf{y} = \mathbf{d} - \mathbf{W}\mathbf{x} \quad \text{Equation 38}$$

This equation should remind us of the supervised learning procedure used in regression (Chapter 1), which lead to the design of the LMS algorithm. So supervised learning can be applied to train the linear associator for hetero-association. The output pattern becomes the desired response. Note that with LMS training the weights are being modified at each iteration by

$$\Delta \mathbf{W}(n) = \eta \varepsilon(n) \mathbf{x}^T(n) = \eta \mathbf{d}(n) \mathbf{x}^T(n) - \eta \mathbf{y}(n) \mathbf{x}^T(n) \quad \text{Equation 39}$$

The first term is the desired forced Hebbian update which is combined with a term that decorrelates the present output y from the input (the anti-Hebbian term). If we compare Eq. 39 with Eq.28 we can conclude that the anti Hebbian term reduces the crosstalk term at each iteration. So training the associative memory with LMS is an efficient way to improve its performance in terms of reduced crosstalk for correlated input patterns. A LAM trained with LMS is called an Optimal Linear Associative Memory (OLAM).

NeuroSolutions 14

6.14 Optimal LAMs

This example still uses the same basic network and files as the previous, but now we trained the LAM using LMS. Notice the difference in the breadboard (the backpropagation plane). Observe the network during training and watch the response approximate the ideal response obtained with the orthogonal patterns. If we train enough and the number of patterns is less than the size of the space, the ideal response will be obtained.

NeuroSolutions Example

One issue that is worth raising is why are we interested in using forced Hebbian to train associative memories when LMS works better? From an engineering point of view optimal LAMs should be utilized. It turns out that the Hebbian paradigm has been utilized by cognitive scientists to study models of human memory. The mistakes associative

memories make have the same general character of the human memory deficiencies.

Associative memories trained with forced Hebbian become rather bad when the density of ones in the patterns is high, i.e. *they work reasonably well only for sparse patterns*. We can understand this since when the patterns have sparse nonzero values (e.g., only 5 bits equal to one in 50 bit long patterns) they are very approximately orthogonal, so there is little crosstalk. It turns out that the human brain has so many neurons that very probably the encoding in the human memory is also sparse, so Hebbian learning makes sense. Moreover there is physiological evidence for the Hebbian learning while the biological implementation of LMS is unclear at this stage.

9.2 LAMs and Linear Regression

You may have noticed the similarity of topologies between the LAM of Figure 10 and the linear regression problem we studied in Chapter I. The marked difference here is that we are interested in multiple-input, multiple-output linear topologies, while in Chapter I we only studied the multiple-input single output case. But the desired response in multiple regression can also be a vector, in which case the topology for regression becomes exactly that of Figure 10. The desired response is effectively the forced response in LAMs. So the difference has to be found in other aspects.

You may recall that in Chapter I we have used the LMS algorithm to find the optimal regression coefficients while in LAMs we utilized the Hebbian learning. But now that we also propose to utilize the LMS to optimally find the LAM weights even this difference is watered down. So what is the difference if any, when we use LMS to train a LAM or a linear regressor?

The difference is very subtle. In linear regression we want to pass a single optimal hyperplane by ALL the desired samples, while in the LAMs we want to output a response which is as close as possible to EACH of the true forced responses. But the mapping system is the same, so as you might expect, it can not provide the two goals under general conditions.

This is where the number of exemplars comes into play. Notice that in LAMs we just saw that the number of exemplars must be less than the size of the input layer to guarantee a small crosstalk. In linear regression the opposite happens. We normally want (and have) more patterns than the size of the input layer of the regressor. So the real difference between a LAM and a regressor is the amount of data, which provides two distinct solutions to the problem. The solution to the LAM is the unconstrained case (more equations than data), while the regression solution is over-constrained (more data than equations). The solution obtained by the LMS for associative memories in fact is one of the infinitely many solutions (R is not full rank because we have less data than dimensions). It is interesting that the storage capacity quantifies the dividing line between the unconstrained and the constrained case. [Optimal LAMs](#)

This gives also a new insight into our terminology of “memorization” when we discussed generalization in Chapter IV and V. The linear regressor can either provide memorization or to generalize the statistical properties of the input, desired response pairs. We see that the distinctive factor is the number of input samples. We described here practical applications for the two conditions. However, if we want to create a regressor and the data is less than the number of input dimensions, LMS will provide an associative memory, not a regressor!!! This clearly shows the risk and the weakness of MSE learning, and emphasizes the importance of capacity control (optimal hyperplanes) discussed in the support machine theory section. We conclude that the existence of crosstalk is critical for generalization.

A similar thing happens to a nonlinear system. If we train it for function approximation with a small number of samples (either for nonlinear regression or classification) we may end up with an associative memory!!! And it will never generalize well. We can then expect that there are many nonlinear associative memory topologies we have not talked about.....

Go to the next section

10. AutoAssociation

There are basically two types of associative memories, the hetero-associators and the autoassociators. As we have just seen, hetero-association or simply association is the process of providing a link between two distinct sets of data (faces with names).

Heteroassociation was just described and it is the most widely used associative memory paradigm. *Autoassociation links a data set with itself.* You may wonder where autoassociation will be useful. It turns out that autoassociation can be used for input reconstruction, noise removal and data reduction.

In autoassociation the output pattern is equal to the input (substitute \mathbf{d} by \mathbf{x} in Figure 10), and the system is trained either with forced Hebbian or with LMS. If we substitute \mathbf{d} by \mathbf{x} in Eq.24 we see that the crosscorrelation function becomes the autocorrelation function, and so the weight matrix of Eq. 27 becomes the autocorrelation matrix of the input

$$\mathbf{W} = \mathbf{xx}^T \quad \text{Equation 40}$$

So when a pattern is presented to the input and no crosstalk is present the autoassociator produces an output

$$\mathbf{Wx} = \mathbf{xx}^T \mathbf{x} \rightarrow \mathbf{Rx} = \lambda \mathbf{x} \quad \text{Equation 41}$$

since $\mathbf{x}^T \mathbf{x}$ is a constant equal to the length of the input vector. If you recall, this is exactly the condition for a vector to be an eigenvector of a matrix, so we conclude that the *autoassociator is performing an eigen-decomposition* of the autocorrelation function, i.e. the outputs will be the eigenvalues of the autocorrelation function, and the weights are the associated eigenvectors. [Hebbian as gradient search](#)

The problem is that if we train the system with the forced Hebbian learning and the inputs are not orthogonal, there will be crosstalk. However, if the learning rule is the LMS, the crosstalk will be decreased to virtually zero (one can show the solution exists unlike the case of hetero-association).

Notice that in the topology of Figure 10 there is no flexibility in the reconstruction. We can produce a more powerful network called an autoencoder (or autoassociator) if we include an extra layer of linear PEs as in Figure 16. The network is normally trained with backpropagation (although the PEs are linear) since there is no desired signal at the hidden layer. We normally impose a constraint that the top matrix $\mathbf{W}_2 = \mathbf{W}_1^T$. Under this constraint we can show that the network will operate in the same way as the PCA network studied in section 4. The signals z_i are effectively the eigenvalues, and their number selects the size of the reconstruction space.

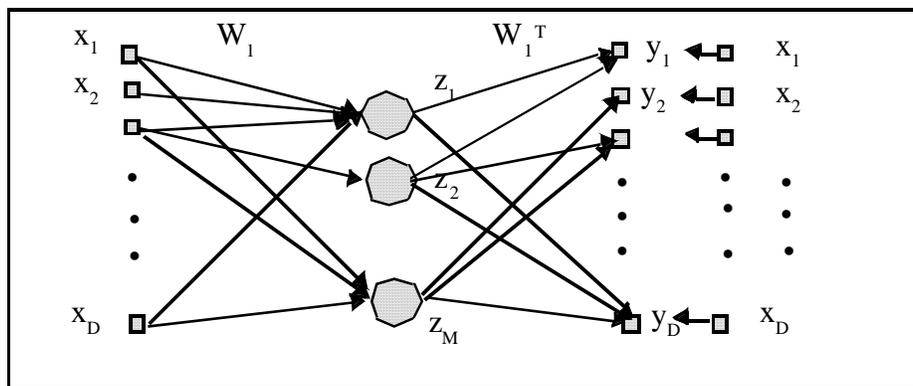


Figure 16. Autoassociator with $\mathbf{W}_2 = \mathbf{W}_1^T$

With this constraint we can alternatively train the network using LMS to determine the top layer weights and then copy them to the transpose locations (reversal of the indices) in the input layer. For this case the weight update using the LMS rule for the top layer weights is

$$\Delta w_{ij}(n) = \eta (x_i(n) - w_{ij}(n) z_j(n)) z_j(n) \quad \text{Equation 42}$$

which can be recognized as the Oja's rule. The autoassociator is a linear system, so it has been analytically studied in depth (Baldi). We know now that the performance surface of the autoassociator is non-convex with saddle points but does not have local minima. This means that the convergence to the global minima can be ensured with the control of the learning rates.

It is possible to even lift the constraint of the transpose between the input and the output weight matrices, and simply train the network with backpropagation (we can not use straight LMS since we have a hidden layer). One can show that in this case the PCA solution is not always obtained, although the system still performs autoassociation, and the solution found by the hidden PEs exists always in the principal component space (but the outputs of the bottleneck layer are not necessarily orthogonal Baldi). The interesting thing is that in some cases the *autoassociator with no constraints on W_2 is able to find projections that seem to preserve better the individuality of each input class*, which makes it better for classification. However, no linear solution will be able to provide a better reconstruction error than the PCA.

NeuroSolutions 15

6.15 Autoassociator and PCA

This problem is a duplication of the reconstruction of digits using PCA, but now we will use an autoassociator trained with backpropagation. Notice the architecture with the hidden layer (called the bottleneck layer). This network effectively computes the PCA when the second weight matrix is restricted to be the transpose of the first weight matrix. In order for the system to train well we have added a minor amount of noise to the input. In this example there is no constraint in the weight matrices.

Experiment with the number of the PEs in the bottleneck layer and compare the accuracy in the digits obtained with this autoassociator and the PCA with the same subspace. Notice that the reconstruction error is higher than PCA, but the digits seem to be better discriminated. Use a MLP with the confusing matrix to quantify this hint.

NeuroSolutions Example

10.1. Pattern Completion/noise reduction properties of the autoassociator

Another interesting property of the autoassociator is the pattern completion property that

is very useful for noise reduction and recovery of missing data. Suppose that a segment \mathbf{x}^l of an input vector \mathbf{x} is lost (for instance during transmission). Let us see if we can recover the full vector after passing it through the autoassociator. The part of \mathbf{x} that is lost is orthogonal to what was kept \mathbf{x}^k , so this is equivalent to decomposing \mathbf{x} into two orthogonal components $\mathbf{x} = \mathbf{x}^k + \mathbf{x}^l$, for instance

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ x_3 \\ x_4 \end{bmatrix}$$

Now if we write the weight matrix \mathbf{W} as a function of the lost and kept part

$$\mathbf{W} = (\mathbf{x}^k + \mathbf{x}^l)(\mathbf{x}^k + \mathbf{x}^l)^T \quad \text{Equation 43}$$

the output becomes

$$\mathbf{y} = (\mathbf{x}^k + \mathbf{x}^l)(\mathbf{x}^k + \mathbf{x}^l)^T \mathbf{x}^k \quad \text{Equation 44}$$

One can show using the orthogonality of \mathbf{x}^k and \mathbf{x}^l that the output is

$$\mathbf{y} = (\mathbf{x}^k + \mathbf{x}^l)\alpha \quad \text{Equation 45}$$

where α is a scalar ($\alpha = (\mathbf{x}^k)^T \mathbf{x}^k$), i.e. the true output \mathbf{x} is obtained. The same argument can be utilized to show that the autoassociator filters out noise. These are very important properties for data transmission.

NeuroSolutions 16

6.16 Autoassociator and pattern completion

In this example, we show how an autoassociator can be used for pattern completion. If the autoassociator is trained with noisy inputs, then it will eventually learn the important parts of the input pattern. Then, after training, if we input patterns which are noisy or incomplete (e.g. digits with missing segments),

the autoassociator will reconstruct the correct image because it has enough information from the input pattern to correctly reconstruct the output pattern.

NeuroSolutions Example

10.3 Supervised versus nonsupervised training

An interesting observation from the autoassociator's discussion is that we reached the same solution with very different learning paradigms: for the PCA we used unsupervised learning, but for the autoassociator we used a supervised procedure (the LMS rule) on a linear architecture with a transpose constraint ($W_2 = W_1^T$) and a desired response $d(n)=x(n)$. *The conclusion is that supervised learning using minimization of the L2 criterion defaults to unsupervised (Hebbian) learning when the desired signal is equal to the input,*

We should ask what is the real difference between supervised and unsupervised learning. Until now we stated that it was the existence of the desired response that made the difference, but this example of the autoassociator proved us wrong. So we have to qualify further the differences.

A learning system adapts its coefficients from the environment using one or several sources of information. In unsupervised learning, the only source of information from the environment is the input. In supervised learning, there are more than one source of information, the input and the desired response. *But for the learning to be qualified supervised, the information contained in the desired response must be different from the input source.* Otherwise, as we just saw, supervised learning defaults to an unsupervised solution.

A further question is the efficiency of both learning strategies. It may be that even if we want to conduct unsupervised learning, a supervised training rule is preferable for more efficient extraction of information from the input signal (provided we choose appropriately the desired response, e.g. $d(n)=x(n)$). *We submit that in this context supervised learning*

is more efficient than unsupervised learning. This is reasonable since the desired signal plays a specific goal in supervised learning, and we now know efficient algorithms to search the performance surface (the gradient descent rule). We saw above that the autoassociator of Example 17 trained with backpropagation trains much faster than the PCA network of Example 6. The other practical condition for which supervised learning defaults to unsupervised is prediction as we will encounter in Chapter X. Others may exist.

Go to the next section

11. Nonlinear Associative memories

Up to now we covered only linear associative memories or LAMs. But there is no reason to limit ourselves to linear PEs. In fact, when the PEs are nonlinear more robust performance is normally obtained. Some new designs are even able to automatically provide a normalized output when the input is normalized, which simplifies learning. The topology of a nonlinear associative memory is shown in Figure 17.

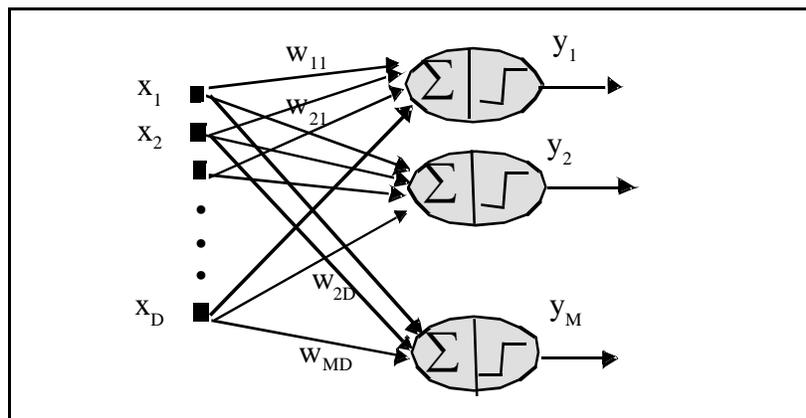


Figure 17. A nonlinear associative memory (NLAM)

Note that the nonlinear PE only affects the output of the memory, so *Hebbian learning of Eq. 1 has exactly the same form for nonlinear networks*. One important advantage of bringing in the nonlinearity is to threshold the output of the LAM. *For binary encoded data the output can be cleaned to a certain extent from the crosstalk error*. In fact we can see

using Eq. 34 that mistakes occur only when the crosstalk term is larger in magnitude than the threshold used to make the binary assignment. The nonlinear LAM is more robust to noise. Equivalently, if the input is contaminated by noise the output can be noise free which is impossible with the LAM.

NeuroSolutions 17

6.17 Nonlinear Associative Memories

Here we will be using one of the previous breadboards but now the output PE will be nonlinear. The big advantage of the nonlinearity is that it can threshold the errors (crosstalk) if it is below the level to make the decision (which normally is set at half dynamic range). Which this means is that if the true output was a zero, but the crosstalk was .4 (between 0 and 1), the output is still 0, the correct response. Since this is done at the output, one can either think that the nonlinearity is part of the network, or it is simply an external read out.

When we implement this type of network in NeuroSolutions and train it with LMS we have to make sure that the error is passed through a linear backprop component to mimic the effect of Hebbian learning, otherwise the final weights will differ from the linear solution. We can see that the system cleans up totally its outputs, so it provides a better memory.

NeuroSolutions Example

These are some of the advantages of the nonlinearity. However the vector space interpretation for the outputs is lost due to the nonlinearity. We can no longer for instance talk about eigenfilters, or PCA. However the network may in fact perform better than the linear counterpart in some applications. In the autoassociator when the bottleneck layer is built from nonlinear PEs the result has been shown to be still PCA, i.e. the linear solution is obtained. However, if the network becomes multilayer the nonlinear network may perform better. These are presently active areas of research.

Go to Next Section

12. Project: Use of Hebbian Networks for Data Compression and Associative memories

Data Compression

In data compression we have a source of data, a communication channel and a receiver. Communication channels have an usable bandwidth, i.e. for a given error rate the number of bits per second - the *bit rate*- has an upper bound. The goal is to be able to transmit as fewer bits per second as possible preserving as much as possible about the source information. So this means that we must squeeze in each bit as much information as possible from the source. We can see immediately the prominent role of PCA for data compression. Moreover, we can model data compression as a projection operation where the goal is to find a set of basis that produce a large concentration of signal power in only a few components.

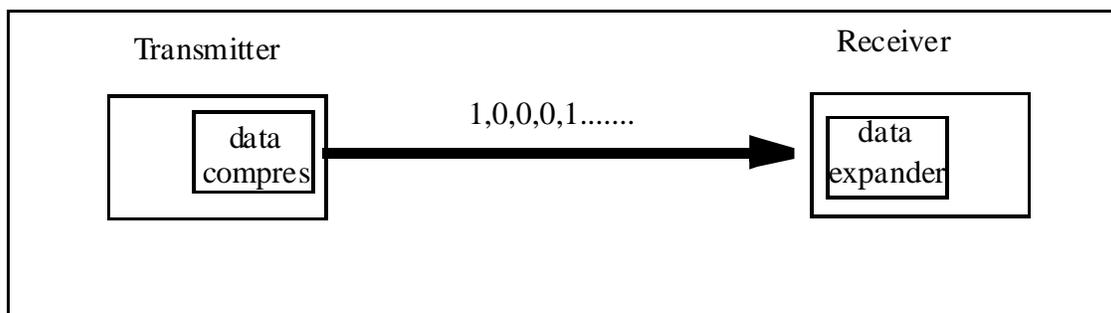


Figure 18. Data transmission with compression

Practically data compression has been based on simpler schemes where the projection vectors are fixed functions instead of being signal dependent as in the PCA. A good example is the cosine basis of JPEG called the Discrete Cosine Transform (DCT) (see [Rao](#)). But notice that there is no need for such constraints since the determination of the optimal projection is an off-line operation, so we can strive for optimal decompositions, as

long as the reconstruction can be done efficiently.

In PCA compression the receiver must know the weight matrix containing the eigenvectors since the estimation of the input from the eigenvalues is done by Eq.18 . The weight matrix is obtained after training with exemplars from the data to be transmitted. It has been shown that for special applications this step can be completed efficiently and is done only once. But in general a given set of coefficients for given signal types (i.e. in images a set of coefficients for people faces, out-door natural scenes, buildings, etc.) will provide better results. But notice that the new image coding schemes such as MPEG already provide this type of labeling. So the receiver can be constructed before hand. The reconstruction step requires $M \times N$ operations where N is the input vector dimension and M is the size of the subspace (number of features).

NeuroSolutions 18

6.18 Data compression with PCA

We've already shown data compression before with PCA. But here we will treat the breadboard in more realistic terms. We have included one extra synapse and an extra axon between the output of the bottleneck layer and the reconstruction layer to show clearly the transmitter at left and the receiver. The extra synapse is depicting the communication channel.

With the PCA the compressor has first to be trained and its weights transmitted to the receiver (which we have done with a DLL), but this is needed only once after the weights converge. Run the network and experiment with the number of features.

Next let us include a noise source at the receiver to mimic the noise in the communication channel. Notice that the PCA encoding is very immune to white, zero mean noise. Effectively the eigenvectors work as lowpass filters so the noise is averaged out.

NeuroSolutions Example

Associative Memory

Associative memories are one of the most widely used applications of Hebbian networks. In particular in the cognitive sciences, LAMs are used due to the analogies between associative memories and mammalian memory (. In general, when the size of the input vectors are much larger than the number of patterns to be stored, this type of memory provides an effective way of associating input patterns with output patterns. The systems train fast and there is no local minima, so they are practical.

Image processing is such an application due to the large input vector of a normal image. In fact a $N \times N$ image is a point in a N^2 dimensional space. So we can store many image to image associations in a matrix of weights. In these cases we may not even need all the weights for perfect recall. This project explores the size of the weight matrix for association in image processing. Due to the size of the systems involved, you may need a fast computer for training.

NeuroSolutions 19

6.19 OLAMs and arbitrary connections

In this example we will use a Linear Associative Memory trained with the LMS rule (an OLAM) to associate facial images of three people with images of their names. In order to reduce network complexity, we will use the arbitrary synapse to reduce the number of weights in the system. A fully connected weight matrix would contain over 400,000 weights (48x48 pixel input and 7x30 pixel output). We will use roughly 20,000 weights which will give us more than enough power to solve the problem. Remember that we only have three images which is much less than the capacity of the network.

NeuroSolutions Example

Go to the next section

13. Conclusions

This chapter studied linear networks adapted with Hebbian learning and similar rules (Oja and Sanger) which are in principle unsupervised learning types. We showed that such networks can be used for data representation also called *feature extraction*, since they project high dimensional data to smaller dimensionality output spaces. Hence PCA networks can be used as data *preprocessors* for other connectionist topologies such as the MLP.

There are analytic procedures to compute PCA, so one may think that this class of networks can be easily substituted by mathematical operations, which is true, but does not address the implementation issues which are important in practical cases. Here all the learning rules were implemented sample by sample and eventually with local algorithms, so they are well suited for on-line distributed implementations. When the matrices are ill-conditions the numerical solutions fail, while the adaptive solutions provide one of the many possible solutions. Convergence speed is normally affected.

Another application of linear networks trained with forced Hebbian is as associative memories. We saw that associative memories work with similar principles to human memory since the memory is contained in the interconnection weights (pattern of activity). They are content addressable (it is enough to input the data to get the recall) unlike computer memories which require an address to retrieve the data. They are also robust to noise and to failure in the components. On the other hand they have limited storage.

We also presented other interesting views such as linking supervised and unsupervised learning. We pointed out the fact that LMS can be thought of as a composition of forced Hebbian and anti-Hebbian, which shows that the learning rules studied so far explore only correlation about the input patterns (or second order statistics about the data clusters).

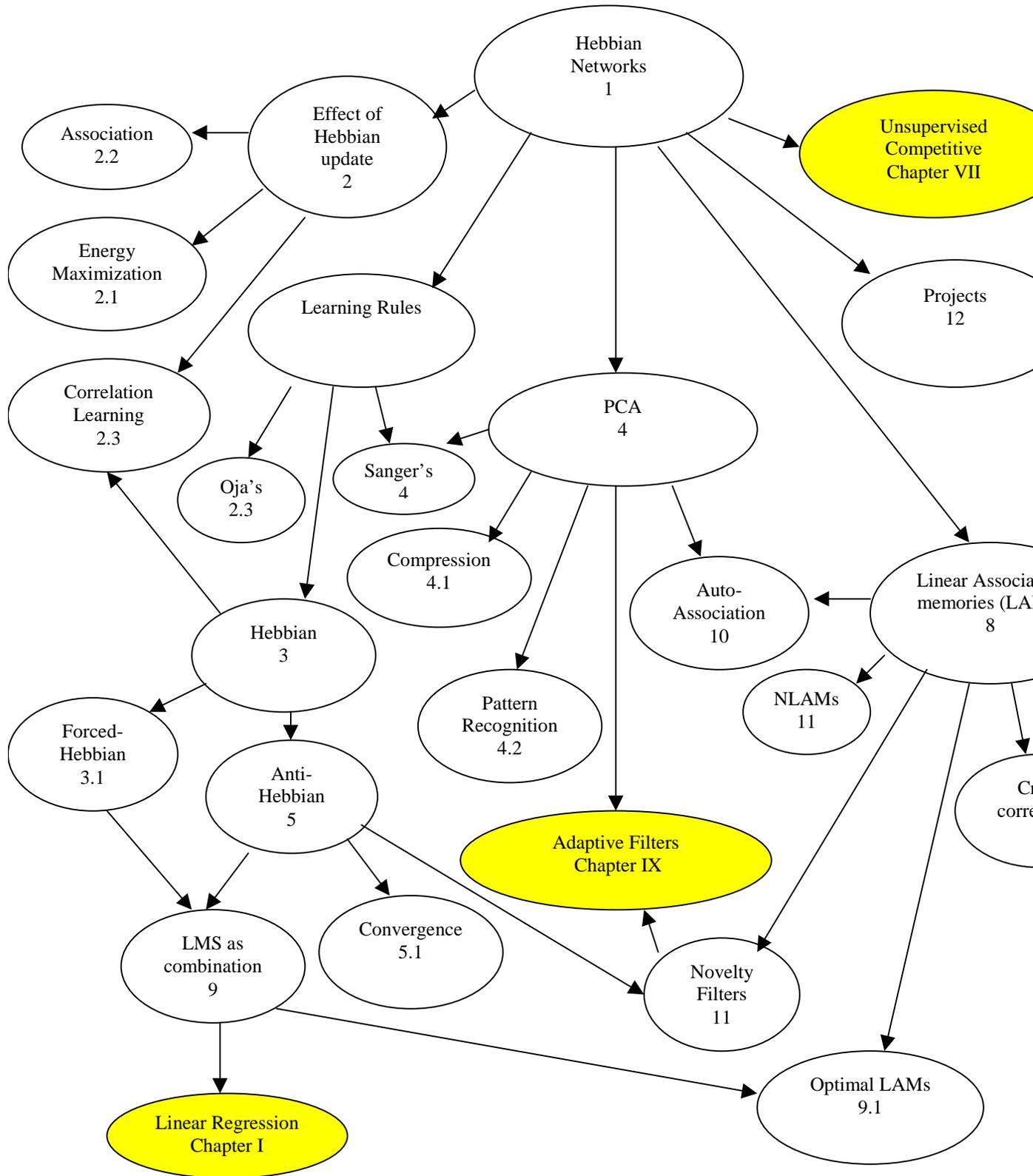
Hebbian networks are very useful in many engineering applications and they train rather quickly, so they are well suited to on-line applications.

NeuroSolutions Examples

- 6.1 Training with the Hebbian rule
- 6.2 Directions of the Hebbian update
 - 6.3 Signal detection with Hamming networks
- 6.4 Instability of Hebbian
 - 6.5 Oja's rule
- 6.6 Sanger's and PCA
- 6.7 PCA for preprocessing
- 6.8 Anti-Hebbian learning
- 6.9 Stability of Hebbian
- 6.10 Forced Hebbian computes crosscorrelation
- 6.11 Novelty filter with anti-Hebbian learning
- 6.12 LAM application
- 6.13 LAM and crosstalk
- 6.14 Optimal LAMs
- 6.15 Autoassociator and PCA
- 6.16 Autoassociator and pattern completion
- 6.17 Nonlinear Associative Memories
- 6.18 Data compression with PCA
- 6.19 OLAMs and arbitrary connections

Concept Maps for Chapter VI

Chapter VI



[Go to Next Chapter](#)

[Go to the Tables of Contents](#)

long and short term memory

Long term memory refers to the storage of information from the past. Since the weights are adapted with the input information their value corresponds to all the data that has been presented to the network. Hence they represent the long term memory of the network.

It is convenient to also consider the activation in the PEs as short-term memory. So far the short term memory is instantaneous since the activations of the PEs discussed so far only depend upon the current data sample. But later in Chapter IX we will consider other network topologies where the activations depend upon a few samples of the past.

[Return to Text](#)

associative memory

We are very familiar with the concept of memory in digital computers, where a set of bits (0 and 1) are stored exactly in a memory location in the address space of our computer. The computer memory is an organization of such locations that is accessed by the processor by an address, and is therefore called *location-addressable*. One can think that the computer memory is a filing cabinet, with each folder containing the data. The processor access the data by *searching the tag* of the folder. This is the reason why computer memory is location addressable and local.

Associative memories are very different and in a lot of ways resemble our own memory. They are *content-addressable* and global. Content addressable means that the recall is not done through the address location, but through the content. During *retrieval* of

information with an associative memory, no address is used, just the input data. When one of the inputs used in training is presented to the Hebbian PE, the output is the pattern created with the storage algorithm during training (we will show this later). The memory is also global in the sense that all the weights in a distributed fashion contain the memory information, and the weights are shared by all the memories that eventually are stored in the system. This is unlike the computer memory where the data is contained locally and independently in each location.

Associate memories are therefore more robust to destruction of information than computer memories. However, their capacity is limited by the number of inputs (as we will see in this chapter), unlike computer memories where the size of the data path is independent of the number of memory locations.

[Return to Text](#)

Hebbian as gradient search

To confirm this just differentiate [Eq. 11](#) with respect to \mathbf{W} to obtain

$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{W}^T \mathbf{R} + \mathbf{R} \mathbf{W} = 2\mathbf{R} \mathbf{W}$$

due to the Toeplitz properties of \mathbf{R} . So gradient ascent would change the weights according to

$$\Delta \mathbf{W} = \eta \frac{\partial J}{\partial \mathbf{W}} = \eta \mathbf{R} \mathbf{W}$$

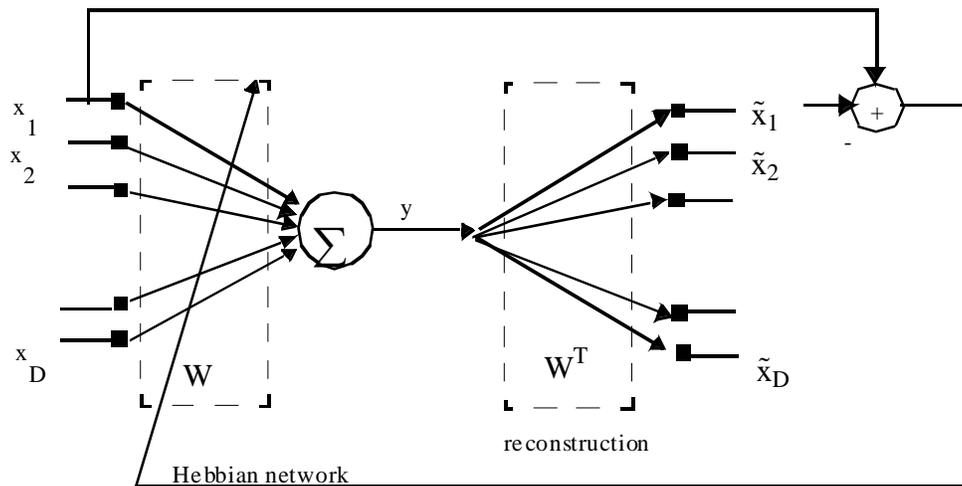
which is exactly what we presented for the Hebbian rule (apart of the 2 which is included in the stepsize). Note that gradient ascent goes in the direction of the gradient, so there is no minus sign in the weight update as we included in the LMS rule.

This view is also interesting because it helps us interpret Hebbian in a supervised learning context. In fact, if we have a performance criterion exterior to the network it is

equivalent to think of an error, hence of a desired response. So what is the implicit desired response in Hebbian learning? It is the input signal itself, and the minimization is

to try to reconstruct \mathbf{x} from $\tilde{\mathbf{x}}$ which is the projection of \mathbf{y} into \mathbf{W} , i.e. $\tilde{\mathbf{x}} = \mathbf{W}^T \mathbf{y}$. A

Figure will clarify this procedure



This means that the criterion is the mean square difference between the input and the projected output i.e.

$$J = E(\mathbf{d} - \mathbf{y})^2 = E(\mathbf{x} - \tilde{\mathbf{x}})^2$$

If we substitute the definition of $\mathbf{y} = \mathbf{W}\mathbf{x}$ we get

$$J = E(\text{tr}[(\mathbf{x} - \tilde{\mathbf{x}})(\mathbf{x} - \tilde{\mathbf{x}})^T]) = \text{tr}(\mathbf{R}_x) - \text{tr}(\mathbf{W}\mathbf{R}_x\mathbf{W}^T)$$

since $\text{tr}(E[\mathbf{W}^T \mathbf{W} \mathbf{x} \mathbf{x}^T \mathbf{W}^T \mathbf{W}]) = \text{tr}(\mathbf{W}\mathbf{R}_x\mathbf{W}^T)$

So now we have a more refined definition of what Hebbian is accomplishing from a vector space point of view. In fact Hebbian can be either interpreted as maximizing the variance of \mathbf{y} (the projection variance Eq. 11), or minimizing the reconstruction error between the input and its version obtained after projecting the output \mathbf{y} on the weight vector (transposed).

[Return to Text](#)

Instability of Hebbian

Let us write the Hebbian update as

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

Applying the expectation operator we get

$$\mathbf{W}(n+1) = (\mathbf{I} + \eta \mathbf{R}_x) \mathbf{W}(n)$$

where \mathbf{R} is the autocorrelation function of the input, and \mathbf{I} is the identity matrix. The stability of this iterative equation is determined by the characteristic roots of the matrix $\mathbf{I} + \eta \mathbf{R}$. Since \mathbf{R} is positive definite, all the roots will be positive hence the iteration will diverge for any value of η .

[Return to Text](#)

derivation of Oja's rule

Let us define a normalized (to unity length) weight vector \bar{w} at each iteration

$$\bar{w}(n+1) = w(n) + \eta y(n)x(n)$$

with

$$w(n+1) = \frac{\bar{w}(n+1)}{\|\bar{w}(n+1)\|}$$

where $\|\cdot\|$ is the length of the vector (square root of the sum of the square components).

The adaptation of this normalized weight using Hebbian learning is

$$\bar{w}(n+1) = \bar{w}(n) + \eta y(n) \frac{y(n)}{\bar{w}(n)} = 1 + \eta y^2(n)$$

since the weights are normalized to 1. So substituting for $w(n+1)$ we can write

$$w(n+1) = \bar{w}(n+1) [1 + 2\eta y^2(n) + \eta^2 y^4(n)]^{-1/2}$$

where we approximated the inverse square root of the norm by its power expansion.

Truncating terms of order higher than 1 in η (if η is small all these terms will be practically zero), we obtain

$$w(n+1) = \bar{w}(n+1)(1 - \eta y^2(n))$$

Now substituting the definition of the normalized weight and again discarding terms that depend on powers of η , we get finally

$$w(n+1) = w(n) + \eta(y(n)x(n) - y^2(n)w(n))$$

as in the text. See [Oja](#).

[Return to Text](#)

proof of eigen-equation

In order to enhance the readability of the equations we will use matrix notation and rewrite Oja's rule as

$$\frac{\mathbf{w}(n+1) - \mathbf{w}(n)}{\eta} = \mathbf{y}(n)\mathbf{x}(n) - \mathbf{y}^2(n)\mathbf{w}(n)$$

The differential equation that corresponds to this difference equation is

$$\frac{d\mathbf{w}(t)}{dt} = \mathbf{R}_x \mathbf{w}(t) - [\mathbf{w}(t)^T \mathbf{R}_x \mathbf{w}(t)] \mathbf{w}(t)$$

So any solution of this equation has to be an eigenvector \mathbf{e}_i of \mathbf{R} (see for instance [Diamantaras and Kung](#)). Now writing $\mathbf{w}(t)$ as a linear combination of its basis vectors \mathbf{e}_i ,

$$\mathbf{w}(t) = \sum_i \alpha_i(t) \mathbf{e}_i$$

, one can further show that the weights adapted with the Oja's rule

converge with probability one to either \mathbf{e}_1 or $-\mathbf{e}_1$ (i.e. to the eigenvector that corresponds to the largest eigenvalue of \mathbf{R}). This is the reason a linear network adapted with Oja's rule is sometimes called the maximum eigenfilter.

[Return to text](#)

PCA derivation

Let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p]$ represent a set of data with p samples, where $\mathbf{x}_i \in R^n$. Without loss of generality we will assume that the data is zero mean. The PCA problem is to find a vector $\mathbf{w} \in R^n$ which maximizes the ratio

$$J = \frac{\mathbf{w}^T \mathbf{R} \mathbf{w}}{\mathbf{w}^T \mathbf{w}}$$

where $\mathbf{R} = \mathbf{X}\mathbf{X}^T$ is the data scattering matrix. Analyzing the expression for J we can conclude that the norm of \mathbf{w} is irrelevant for the solution, so we can keep it constant at $\|\mathbf{w}\| = 1$. If we want to use the gradient descent procedure to maximize J we have to compute the gradient with respect to the weights, which gives

$$\nabla_{\mathbf{w}} J = \frac{1}{\|\mathbf{w}\|^2} \sum_i y_i \left(\mathbf{x}_i - \frac{y_i}{\|\mathbf{w}\|^2} \mathbf{w} \right)$$

If we keep the norm equal to one, this expression defaults to the Oja's rule. So maximizing the output variance will produce Oja's rule. If we want to show that maximizing output variance yields the PCA decomposition, we can alternatively start by analyzing each one of the components of the output. Let us start with the first output,

$$y_1 = \mathbf{w}_1^T \mathbf{x}$$

Its variance is

$$E\{y_1\} = \mathbf{w}_1^T \mathbf{R}_x \mathbf{w}_1$$

Now the Rayleigh-Ritz theorem guarantees that $\mathbf{w}_1 = \mathbf{e}_1$ and $y_1 =$ largest eigenvalue λ_1 yields the maximum for the variance provided $\|\mathbf{w}_1\| = 1$. The same argument can be applied to the other components with the added constraint that the weights have to be orthogonal to the previous weights.

Hence we have shown that maximizing the constrained output variance provides an ordered eigenvalue decomposition of the input correlation matrix. The directions are the

eigenvectors of R and the projections on each the corresponding eigenvalues. PCA does provide a way to analyze the structure of the correlation matrix of the input data.

With this view, if the number of outputs is less than the number of inputs, the projection will preserve maximally the energy of the input. This is a powerful technique for signal representation in very large dimensional spaces.

[Return to Text](#)

definition of eigenfilter

An eigenfilter is associated with the eigen-decomposition we studied in Chapter V. Recall that there we were looking for natural bases to decompose functions. Here we will be looking at ways of naturally decomposing data clusters. The eigenfunctions are the bases from which the functions are exactly constructed by a finite weighted sum (the projection theorem). So they are the most efficient way to decompose any function.

Oja's rule when applied to the linear PE network implements a decomposition that finds the weights corresponding to the principal component direction. This direction maximizes the projection of the input data cluster. In order to find this direction the input data has to be projected by a "filter" matched to the data, hence the name maximum eigenfilter. We should think of the weights of the network as the bases (as we did in Chapter V), and the network output as the scalar in the projection theorem.

There is a very important concept hidden here. When we use data collected from sensors (measurements), the representation space is given by our measurements. This space may not be the best to capture the relevant properties of the data. One goal is to find a representation space that is meaningful for data analysis. The principal directions embody exactly this idea since it is the data that it is telling us what are the basis to represent them well.

[Return to text](#)

Optimal LAMs

In fact the LMS is an approximated method to train a LAM optimally. One can show that the optimal LAM weights have to meet the solution

$$\mathbf{W}^* = \mathbf{Y}\mathbf{X}^{-1}$$

which exists as long as the inverse of \mathbf{X} exists (here \mathbf{X} and \mathbf{Y} are the matrices constructed from the full training set). This means that the patterns must be linearly independent (instead of orthogonal as required for the Hebb training). If we have less patterns than inputs the optimal solution is not unique. We can show that in this case ([Kohonen](#))

$$\mathbf{W}^* = \mathbf{Y}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T = \mathbf{Y}\mathbf{X}^+$$

which involves the computation of \mathbf{X}^+ the pseudo-inverse of \mathbf{X} . There is a method to compute \mathbf{W}^* recursively (Greville's theorem) using a nonlocal algorithm that resembles the LMS. The LMS with a small stepsize is a good approximation to this recursive algorithm. See also [Hecht-Nielsen](#).

[Return to Text](#)

Hebb

Donald Hebb, The organization of Behavior: a neurophysiological theory, Wiley, 1949.

unsupervised

a learning rule is called unsupervised if the adaptation of the weights utilize only one source of external information (the input). In supervised learning more than one external source is utilized to adapt the weights. These are the input and the desired signal which is normally utilized as the target response.

Sanger

Terry Sanger, Optimal unsupervised learning in a single layer linear feedforward neural network, *Neural Networks*, 12, 459-473, 1989.

Oja

Erki Oja, A simplified neuron model as a principal component analyzer, *J. of Mathematical Biology*, 15, 239-245, 1982.

APEX

see *Principal Component Neural Networks* by Diamantaras and Kung, Wiley, 1996, page 90.

ASCII

American standard code for information interchange . It has become a standard for coding of characters into binary strings.

second order

are measured by the covariance function. We saw that a Gaussian is fully described by the mean and variance for 1-D and the mean vector and the covariance for multi-D.

SVD

or SVD for short is an analytical procedure that computes the orthogonal decomposition of data. See *Matrix computations* by Golub and Van Loan , Johns Hopkins U. Press.

Eq.1

$$\Delta w_{ij} = \eta x_j y_i$$

Eq.6

$$y = \mathbf{w}^T \mathbf{x} = \mathbf{x}^T \mathbf{w}$$

Eq.5

$$\Delta \mathbf{w} = \eta \begin{bmatrix} x_1 y \\ \dots \\ x_D y \end{bmatrix}$$

Eq.8

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

Eq.2

$$w(n+1) = w(n) + \eta x(n) y(n)$$

Eq.26

$$r_{xd}(i, j) = \frac{1}{P} \sum_{k=1}^P x_{i,k} d_{j,k} \quad 0 < i < N, \quad 0 < j < M$$

Eq.30

$$\Delta w_{ij} = \eta x_j d_i$$

Eq.29

$$\mathbf{R} = A \begin{bmatrix} \mathbf{x}_1 \mathbf{d}_1 & \mathbf{x}_1 \mathbf{d}_2 & \mathbf{x}_1 \mathbf{d}_N \\ \dots & \dots & \dots \\ \mathbf{x}_N \mathbf{d}_1 & \mathbf{x}_N \mathbf{d}_2 & \mathbf{x}_N \mathbf{d}_N \end{bmatrix}$$

Kohonen

Teuvo Kohonen, Self-organization and associative memory, Springer Verlag, 1984.

Eq.36

$$\mathbf{y} = \mathbf{W} \mathbf{x}_l = \mathbf{d}_l \mathbf{x}_l^T \mathbf{x}_l + \sum_{k=1, k \neq l}^K \mathbf{d}_k \mathbf{x}_k^T \mathbf{x}_l$$

Stephen Grossberg

is a very influential neural network researcher that proposed many biologically plausible neural network architectures.

See for instance Natural Intelligence, MIT Press, 1992.

Eq.7

$$y = |\mathbf{w}| |\mathbf{x}| \cos(\theta)$$

Eq.11

$$J = E[y]^2 = \mathbf{w}^T \mathbf{R}_x \mathbf{w}$$

Eq.4

$$y = \sum_{i=1}^D w_i x_i$$

Eq.27

$$r_{xd}(i, j) = A[\mathbf{x}_i \mathbf{d}_j]$$

Eq.19

$$\Delta w_{ij} = -\eta x_j y_i$$

Eq.34

$$\mathbf{y} = \mathbf{W}\mathbf{x}_l = \mathbf{d}_l \mathbf{x}_l^T \mathbf{x}_l + \sum_{k=1, k \neq l}^K \mathbf{d}_k \mathbf{x}_k^T \mathbf{x}_l$$

Diamantaras

Diamantaras and Kung, Principal component analysis networks, Wiley, 1996.

deflation

is a method of computing the principal components that reminds us of the Gram Schmidt orthogonalization procedure, i.e. first compute the principal direction, and subtract it from the data before computing the next principal direction.

Baldi

see Baldi and Hornik, Neural networks and principal component analysis: learning from

examples without local minima”, Neural networks 1, 53-58, 1989

Hecht-Nielsen

NeuroComputing, Addison Wesley, 1990.

Kay

Modern Spectral Analysis, Prentice Hall, 1988.

Eq.18

$$\tilde{\mathbf{x}} = \mathbf{W}^T \mathbf{y}$$

energy, power and variance

From the statistical point of view, the energy of a 1-D signal $x(n)$ is related to its variance.

In fact the energy of a stationary signal $x(n)$ with variance σ^2 and mean m is

$$E = E[x^2(n)] = \sigma^2 + m^2$$

where $E[\cdot]$ is the expectation operator. If $m=0$ then the energy is equal to the variance,

$$E = \sigma^2 \text{ so the energy is related to the second order statistics of the signal.}$$

The power P (or short term energy) is defined as the energy in a finite window, or for a finite number of samples. So the power is also related to an estimation of the second order statistics with finite data. The condition of zero mean is normally assumed in the discussion.

[Return to Text](#)

PCA, SVD, and KL transforms

We have to cover briefly the mathematics of principal component analysis (PCA) to fully understand and apply the concept.

PCA and Singular Value Decomposition (SVD) are intrinsically related. Let us start with the SVD because it is an algebraic operation applicable to any matrix. The goal of SVD is to diagonalize any matrix, i.e. to find a rotation where only the diagonal elements are nonzero.

Consider the matrix Z with M rows and N columns ($M \times N$). For every such matrix there are two orthonormal matrices, U ($M \times M$) and V ($N \times N$) and a pseudodiagonal matrix

$$D = \text{diag}\{\sigma_1, \dots, \sigma_p\} \quad (M \times N) \text{ where } P = \min\{M, N\} \text{ such that}$$

$$Z = UDV^T \quad \text{or} \quad Z = \sum_{i=1}^P \sigma_i u_i v_i^T$$

The vectors u and v are called the left and right singular vectors of Z , while the s are called the singular values of Z .

SVD is intrinsically related to the eigendecomposition of a matrix. In fact, if we postmultiply by $Z^T U$ we obtain $Z Z^T U = U D V^T Z^T U = U D D^T$. Likewise we can show that $Z^T Z V = V D^T D$. Now $D D^T$ and $D^T D$ are square diagonal matrices, and so the vectors u and v are the eigenvectors of the matrices $Z Z^T$ and $Z^T Z$ respectively,

$$\begin{aligned} Z Z^T u_i &= \sigma_i^2 u_i & i = 1, \dots, M \\ Z^T Z v_i &= \sigma_i^2 v_i & i = 1, \dots, N \end{aligned}$$

Now let us define PCA. Consider a vector $x = [x_1, \dots, x_D]^T$ with mean zero, and covariance $R = E[xx^T]$ which is a symmetric matrix ($D \times D$). PCA produces a linear transformation of the data $y = Wx$ to a subspace of size $M \leq D$ where the columns of W form an orthonormal basis. PCA has a very nice property: it minimizes the mean square error between the projected data (to a subspace M) and the original data. The

reconstructed data from the projections is a vector $\hat{x} = W^T y = W^T W x$. So PCA minimizes

$$J = E\{\|x - \hat{x}\|^2\} = \text{tra}(R) - \text{tra}(WRW^T)$$

where $\text{tra}(\cdot)$ means the trace of the matrix. The trace of WRW^T is effectively the variance of y , i.e.

$$\text{tra}(WRW^T) = \sum_{i=1}^M y_i^2$$

So the minimization of J implies the maximization of the variance of y , which is also the variance of the estimated projection. So this provides still another interpretation for PCA: PCA is the linear projection that maximizes the variance (power) of the projection to a subspace.

What is interesting is to analyze the characteristics of the PCA projection, i.e. the structure of W . If the eigenvalues of R $\{\lambda_1, \dots, \lambda_N\}$ are ordered in descending order of the eigenvalues $\{\lambda_1, \dots, \lambda_N\}$, we can show that

$$\min J = \sum_{i=M+1}^D \lambda_i \quad \text{tra}(WRW^T) = \sum_{i=1}^M \lambda_i$$

These two equations basically state that if we project with PCA to a subspace of dimension M we preserve the variance given by the sum of the first M (principal) eigenvalues. The error can also be easily obtained by adding the $D-M-1$ (minor) eigenvalues.

The projections are called the principal components of x . They are statistically uncorrelated

$$E\{y_i, y_j\} = e_i^T R e_j = 0$$

and their variances are equal to the eigenvalues of R ,

$$E\{y\} = e_i^T R e_i = \lambda_i$$

and are arranged by descending order of variance.

Since R is a symmetric matrix, then we can say that it can always be decomposed as

$$R = e\Lambda e^T = \sum_{i=1}^D \lambda_i e_i e_i^T$$

where Λ is a diagonal matrix with entries λ_i , the eigenvalues, and e_i are the eigenvectors. This equation shows the special function that eigenvectors represent for a matrix. They diagonalize it, i.e. they represent the direction in space where we can compute the entries of the matrix using only scalar operations. Alternatively, once the eigenvectors and eigenvalues are known, we can construct R with scalar operations! This means we have found the structure of the data.

We see that the PCA is actually operating with the eigenstructure of R, hence its importance. In general only the data is known, not R. Even when R is known, normally its eigenstructure is not quantified. But when we perform PCA we discover the dependencies on the data, and we can even project it to a subspace to simplify the analysis losing the least of variance.

Now the equivalence between SVD and PCA should be clear. In fact, if Z is square and symmetric, then the two orthogonal matrices U and V become the same, and SVD becomes equivalent to PCA.

Lastly we would like to define the Karhunen-Loeve transform (KLT). This transform was originally developed to study decompositions of continuous time signals. But for finite duration (D) discrete signals, it can be formulated in the following way:

Consider the stationary random process $x(n)$ with zero mean and autocorrelation

$R = E[x(n-k)x(n-l)] = R(l,k)$. The KLT is defined as the set of basis $u_i(n)$ that satisfy the relation

$$\sum_{k=0}^{D-1} R(l,k) u_i(k) = \lambda_i u_i(l) \quad i, l = 0, \dots, D-1$$

We can write this expression in matrix form to read $Ru_i = \lambda_i u_i$, and we immediately

recognize the eigenvalue equation involving the (time) autocorrelation of the data. So KLT and PCA yield the same solution for the case of finite duration discrete signals.

[Return to Text](#)

Gram-Schmidt orthogonalization

The question is very simple. Given a set of vectors $\{x_1, \dots, x_m\}$ spanning a space S (i.e. the space of all their linear combinations), can we find a rotation that will orthogonalize all the vectors and preserve the span?

The solution was proposed many years ago by Gram-Schmidt in the form of a recursive procedure. Let us start with one of the vectors and make

$$v_1 = \frac{x_1}{\|x_1\|}$$

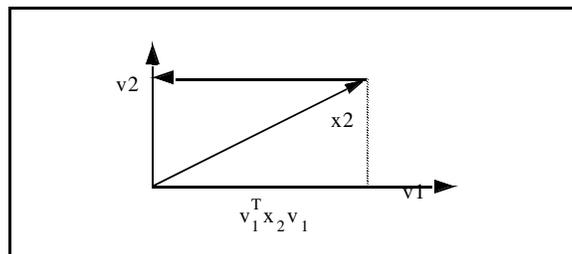
Then find a direction orthogonal to the subspace defined by the k vectors already orthogonalized and normalize, which yields

$$\tilde{v}_{k+1} = \left[\prod_{i=1}^k (I - v_i v_i^T) \right] x_{k+1}$$

which due to the previously orthogonalized vectors yields

$$\tilde{v}_{k+1} = x_{k+1} - \sum_{i=1}^k (v_i^T x_{k+1}) v_i \quad \text{and} \quad v_{k+1} = \frac{\tilde{v}_{k+1}}{\|\tilde{v}_{k+1}\|}$$

It is interesting to look at this equation in a figure for the case of two vectors



This method is in fact a deflation procedure, because it is removing the contributions of the previous vectors into the current direction. To see this we first have to define a projector as a matrix such that $Z^2=Z$. We immediately see that ZZ^T is a projector, so

when applied to any vector it will project it to the span of Z. Now, the form $I-ZZ^T$ is also a projector, and it will project any vector to the orthogonal space of Z (also called the null space of Z). So Gram-Schmidt uses an iterative projection to the orthogonal space of each vector, so it is a deflation procedure. (see [Diamantaras](#)).

[Return to Text](#)

Silva and Almeida

A distributed decorrelation algorithm, in Gelenbe (Ed.), Neural Networks, Advances and Applications, North Holland, 1991.

Information and Variance

Extracting information from data is what learning is all about. Here we are using a layman Concept of information, but we can also provide a technical definition. Shannon in a seminal paper proposed the following definition for entropy

$$H(X) = -\sum_{k=1}^N p(x_k) \log(p(x_k)) = -E\{\log(p(x))\}$$

where p_k are the probabilities of set of messages $\{x_1, \dots, x_N\}$ occurring with probabilities p_1, \dots, p_N . The idea is the following: if we know what the message is ($p_k=1$), the information it carries is zero. On the other hand, if its content is unexpected (small p_k), then the amount of information the message carries is rather large. This definition translates well our intuition, although Shannon utilized an axiomatic approach to derive his definition. Shannon's entropy definition has been the cornerstone to create efficient and reliable communication systems (see [Cover](#)), and it is also quite important in statistics and learning.

We can note that entropy uses the full information about the probability density function

about the data, but normally we do not know this information. It turns out that if the data is Gaussian distributed, i.e.

$$p(x) = \frac{1}{\sqrt{(2\pi)\sigma}} e^{-\frac{1}{2}\left(\frac{x-m}{\sigma}\right)^2}$$

then only two numbers, the mean and the variance are sufficient to describe the pdf of the data. This means that only the first and second order moments are different from zero, all the others are identically zero. Therefore, for Gaussian distributed data, the entropy can be written

$$H(x) = 0.5 \log(2\pi\sigma) + 0.5 \log E\left\{\left(\frac{x-m}{\sigma}\right)^2\right\}$$

or that it is proportional to the variance of the data. So for Gaussian distributed variables information is synonym of variance.

[Return to Text](#)

Cover and Thomas

Elements of Information Theory, Wiley, 1991.

Foldiak

Adaptive network for optimal linear feature extraction, Proc. Int. J. Conf. Neural Networks, vol I, 401-405, 1989.

Rao and Huang

Techniques and Standards for Image Video & Audio Compression, Prentice Hall, 1996

Index

1	
1. Introduction	4
10. AutoAssociation	32
11. Nonlinear Associative memories	35
12. Conclusions	38
12. Project.....	36
Use of Hebbian Networks for Data Compression and Associative memories	36
2	
2. Effect of the Hebb update	5
2.3. Oja's rule	12
4	
4. Principal Component Analysis	
Chapter5.....	14
5	
5. Anti Hebbian Learning	19
6	
6. LMS learning as a combination of Hebb rules	29
7	
7. Estimating the crosscorrelation between two data sets with Hebbian networks.....	21
8	
8. Decorrelation filters.....	23
9	
9. Linear associative memories (LAMs).....	26
C	
Chapter VI- Hebbian Learning and Principal Component Analysis	3
Chapter5	5, 12, 19, 21, 23, 26, 29, 32, 35
E	
energy	51
power and variance	51
G	
Gram-Schmidt orthgonalization.....	54
I	
Information and Variance	55
P	
PCA	52, 53
SVD	
and KL transforms	52
PCA derivation.....	45