Chapter 3

Hebbian Learning

3.1 Simple Hebbian Learning

The aim of unsupervised learning is to present a neural net with raw data and allow the net to make its own representation of the data - hopefully retaining all information which we humans find important. Unsupervised learning in neural nets is generally realised by using a form of Hebbian learning which is based on a proposal by Donald Hebb who wrote:

When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

Neural nets which use Hebbian learning are characterised by making the activation of a unit depend on the sum of the weighted activations which feed into the unit. They use a learning rule for these weights which depends on the strength of the simultaneous activation of the sending and receiving neuron. With respect to the network depicted in Figure 3.1, these conditions are usually written as

$$y_i = \sum_j w_{ij} x_j \tag{3.1}$$

$$\text{and } \Delta w_{ij} = \alpha x_j y_i \tag{3.2}$$

the latter being the learning mechanism. Here y_i is the output from neuron i, x_j is the j^{th} input, and w_{ij} is the weight from x_j to y_i. α is known as the learning rate and is usually a small scalar which may change with time. Note that the learning mechanism says that if x_j and y_i fire simultaneously, then the weight of the connection between them will be strengthened in proportion to their strengths of firing.

It is possible to introduce a (non-linear) function into the system using

$$y_i = g(\sum_j w_{ij} x_j) \tag{3.3}$$

$$\text{and } \Delta w_{ij} = \alpha x_j y_i \tag{3.4}$$

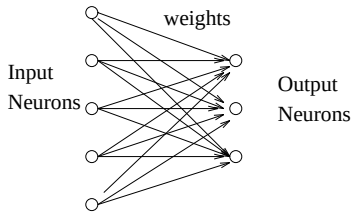

Figure 3.1: A one layer network whose weights can be learned by simple Hebbian learning.

for some function, $g()$; we will still call this Hebb learning.

Substituting Equation (3.1) into Equation (3.2), we can write the Hebb learning rule as

$$\begin{aligned} \Delta w_{ij} &= \alpha x_j \sum_k w_{ik} x_k \\ &= \alpha \sum_k w_{ik} x_k x_j \end{aligned} \tag{3.5}$$

If we take α to be the time constant, Δt and divide both sides by this constant we get

$$\frac{\Delta w_{ij}}{\Delta t} = \sum_k w_{jk} x_k x_j$$

$$\text{which, as } \Delta t \to 0 \qquad \text{is equivalent to} \tag{3.6}$$

$$\frac{d}{dt}\mathbf{W}(t) \propto \mathbf{C}\mathbf{W}(t) \tag{3.7}$$

where C_{ij} is the correlation coefficient calculated over all input patterns between the i^{th} and j^{th} terms of the inputs and $\mathbf{W}(t)$ is the matrix of weights at time t. In moving from the stochastic equation (3.5) to the averaged differential equation (3.7), we must place certain constraints on the process particularly on the learning rate α which we will not discuss in this course. We are using the notation

$$\frac{d}{dt}\mathbf{W} = \begin{pmatrix} \frac{dw_{11}}{dt} & \frac{dw_{12}}{dt} & \cdots & \frac{dw_{1n}}{dt} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{dw_{m1}}{dt} & \frac{dw_{m2}}{dt} & \cdots & \frac{dw_{mn}}{dt} \end{pmatrix} \tag{3.8}$$

The advantage of this formulation is that it emphasises the fact that the resulting weights depend on the second order statistical properties of the input data i.e. the covariance matrix is the crucial factor.

3.1.1 Stability of the Simple Hebbian Rule

At convergence the weights should have stopped changing. Thus at a point of convergence *if this point exists* $E(\Delta W)$ =0 or in terms of the differential equations $\frac{d}{dt}W = 0$.

However, a major difficulty with the simple Hebb learning rule is that unless there is some limit on the growth of the weights, the weights tend to grow without bound: we have a positive feedback loop - a large weight will produce a large value of y (Equation 3.1) which will produce a large increase in the weight (Equation 3.2). Let us examine mathematically the Hebb rule's stability:

Recall first that a matrix $\mathbf{A}$ has an eigenvector $\mathbf{x}$ with a corresponding eigenvalue λ if

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

In other words, multiplying the vector $\mathbf{x}$ or any of its multiples by $\mathbf{A}$ is equivalent to multiplying the whole vector by a scalar λ. Thus the direction of $\mathbf{x}$ is unchanged - only its magnitude is affected.

Consider a one output-neuron network and assume that the Hebb learning process does cause convergence to a stable direction, $\mathbf{w}^*$; then if w_k is the weight vector linking x_k to y, at convergence,

$$0 = E(\Delta w_i^*) = E(yx_i) = E(\sum_j w_j x_j x_i) = \sum_j R_{ij} w_j$$

where R is the correlation matrix of the distribution. Now this happens for all i , so $\mathbf{R}\mathbf{w} = 0$. Now the correlation matrix, R, is a symmetric, positive semi-definite matrix and so all its eigenvalues are non-negative. But the above formulation shows that $\mathbf{w}^*$ must have eigenvalue 0. Now consider a small disturbance, ϵ, in the weights in a direction with a non-zero (i.e. positive) eigenvalue. Then

$$E(\Delta w*) = R(\mathbf{w}^* + \epsilon) = R\epsilon > 0$$

i.e. the weights will grow in any direction with non-zero eigenvalue (and such directions must exist). Thus there exists a fixed point at **W=0** but this is an unstable fixed point: if all weights happen to be zero a single small change from this will cause all the weights to move from zero. In fact, it is well known that in time, the weight direction of nets which use simple Hebbian learning tend to be dominated by the direction corresponding to the largest eigenvalue.

We will now discuss one of the major ways of limiting this growth of weights while using Hebbian learning and review its important side effects.

3.2 Weight Decay in Hebbian Learning

As noted in Section 3.1, if there are no constraints placed on the growth of weights under Hebbian learning, there is a tendency for the weights to grow without bounds. It is possible to renormalise weights after each learning epoch, however this adds an additional operation to the network's processing. By renormalising, we mean making the length of the weight vector always equal to 1 and so after each weight change we divide the weight vector by its length; this preserves its direction but makes sure that its length is 1. We then have the two stage operation:

$$\begin{aligned} \mathbf{w}_j &= \mathbf{w}_j + \mathbf{\Delta w}_j \\ \mathbf{w}_j &= \frac{\mathbf{w}_j}{\|\mathbf{w}_j\|} \end{aligned}$$

Another possibility is to allow the weights to grow until each reaches some limit, e.g. have an upper limit of w^+ and a lower limit of w^- and clip the weights when they reach either of these limits. Clearly a major disadvantage of this is that if all weights end up at one or other of these limits[1] the amount of information which can be retained in the weights is very limited.

A third possibility is to prune weights which do not seem to have importance for the network's operation. However, this is an operation which must be performed using non-local knowledge - typically which weights are of much smaller magnitude than their peers.

Hence, interest has grown in the use of decay terms embedded in the learning rule itself. Ideally such a rule should ensure that no single weight should grow too large while keeping the total weights on connections into a particular output neuron fairly constant. One of the simplest forms of weight decay was developed as early as 1968 by Grossberg and was of the form:

$$\frac{dw_{ij}}{dt} = \alpha y_i x_j - w_{ij} \tag{3.9}$$

It is clear that the weights will be stable (when $\frac{dw_{ij}}{dt} = 0$) at the points where $w_{ij} = \alpha E(y_i x_j)$. Using a similar type of argument to that employed for simple Hebbian learning, we can show that at convergence we must have $\alpha \mathbf{C}\mathbf{w} = \mathbf{w}$. Thus $\mathbf{w}$ would have to be an eigenvector of the correlation matrix of the input data with corresponding eigenvalue $\frac{1}{\alpha}$. We shall be interested in a somewhat more general result.

Grossberg went on to develop more sophisticated learning equations which use weight decay e.g. for his instar coding, he has used

$$\frac{dw_{ij}}{dt} = \alpha\{y_i - w_{ij}\}x_j \tag{3.10}$$

where the decay term is gated by the input term x_j and for outstar coding

$$\frac{dw_{ij}}{dt} = \alpha\{x_j - w_{ij}\}y_i \tag{3.11}$$

where the decay term is gated by the output term y_i. These, while still falling some way short of the decay in which we will be interested, show that researchers of even 15 years ago were beginning to think of both differentially weighted decay terms and allowing the rate of decay to depend on the statistics of the data presented to the network.

[1] This will certainly happen if simple Hebbian learning is used

3.3 Principal Components and Weight Decay

Miller and MacKay have provided a definitive study of the results of a decay term on Hebbian learning. They suggest an initial distinction between Multiplicative Constraints and Subtractive Constraints.

They define Multiplicative Constraints as those satisfying

$$\frac{d}{dt}\,\mathbf{w}(t) = \mathbf{C}\mathbf{w}(t) - \gamma(\mathbf{w})\mathbf{w}(t)$$

where the decay in the weights is governed by the product of a function of the weights, $\gamma(\mathbf{w})$, and the weights, $\mathbf{w}$(t), themselves. The decay term can be viewed as a feedback term which limits the rate of growth of each weight in proportion to the size of the weight itself while the first term defines the Hebbian learning itself.

Subtractive Constraints are satisfied by equations of the form

$$\frac{d}{dt}\mathbf{w}(t) = \mathbf{C}\mathbf{w}(t) - \epsilon(\mathbf{w})\mathbf{n}$$

where the decay in the weights is governed by the product of a function of the weights ,$\epsilon(\mathbf{w})$, and a constant vector,$\mathbf{n}$, (which is often $\{1,1,..1\}^T$).

They prove that

- Hebb rules whose decay is governed by Multiplicative Constraints will, in cases typical of Hebb learning, ensure that the weights will converge to a stable point
- This stable point is a multiple of the principal eigenvector of the covariance matrix of the input data
- Hebb rules governed by Subtractive Constraints will tend to lead to saturation of the weights at their extreme permissible values.
- Under Subtractive Constraints, there is actually a fixed point within the permitted hypercube of values but this is unstable and is only of interest in anti-Hebbian learning(see below).
- If specific limits (w^+ and w^-) do not exist, weights under Subtractive Constraints will tend to increase without bound.

In summary then, Subtractive Constraints offer little that cannot be had from simple clipping of the weights at preset upper and lower bounds. Multiplicative Constraints, however, seem to give us not just weights which are conveniently small, but also weights which are potentially useful since

$$y_i = \sum_j w_{ij}x_j = \mathbf{w_i}.\mathbf{x}$$

where $\mathbf{w_i}$ is the vector of weights into neuron y_i and $\mathbf{x}$ is the vector of inputs. But,

$$\mathbf{w_i}.\mathbf{x} = |\mathbf{w_i}||\mathbf{x}|\cos\theta$$

where $|\mathbf{d}|$ is the length of $\mathbf{d}$ and θ is the angle between the 2 vectors.

This is maximised when the angle between the vectors is 0. Thus, if $\mathbf{w}_1$ is the weight into the first neuron which converges to the first Principal Component, the first neuron will maximally transmit information along the direction of greatest correlation, the second along the next largest, etc. In Section 2.5, we noted that these directions were those of greatest variance which from Section 2.3, we are equating with those of maximal information transfer through the system.

Given that there are statistical packages which find Principal Components, we should ask why it is necessary to reinvent the wheel using Artificial Neural Networks. There are 2 major advantages to PCA using ANNs:

1. Traditional statistical packages require us to have available prior to the calculation, a batch of examples from the distribution being investigated. While it is possible to run the ANN models with this method - "batch mode" - ANNs are capable of performing PCA in real-time i.e. as information from the environment becomes available we use it for learning in the network. We are, however, really calculating the Principal Components of a sample, but since these estimators can be shown to be unbiased and to have variance which tends to zero as the number of samples increases, we are justified in equating the sample PCA with the PCA of the distribution. The adaptive/recursive methodology used in ANNs is particularly important if storage constraints are important.

2. Strictly, PCA is only defined for stationary distributions. However, in realistic situations, it is often the case that we are interested in compressing data from distributions which are a function of time; in this situation, the sample PCA outlined above is the solution in that it tracks the moving statistics of the distribution and provides as close to PCA as possible in the circumstances.

   However, most proofs of convergence of ANNs which find Principal Components require the learning rate to converge to 0 in time and, in practice, it is the case that convergence is often more accurate when the learning rate tends to decrease in time. This would preclude an ANN following a distribution's statistics, an example of the well-known trade-off between tracking capability and accuracy of convergence.

We now look at several ANN models which use weight decay with the aim of capturing Principal Components. We will make no attempt to be exhaustive since that would in itself require a thesis; we do however attempt to give representative samples of current network types.

3.4 Oja's One Neuron Model

There were a number of ANN models developed in the 1980s which used Hebbian learning. The most important was Oja's.

Oja proposed a model which extracts the largest principal component from the input data. He suggested a single output neuron which sums the inputs in the usual fashion

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

His variation on the Hebb rule, though, is

$$\Delta w_i = \alpha(x_i y - y^2 w_i)$$

Note that this is a rule defined by Multiplicative Constraints ($y^2 = \gamma(w)$) and so will converge to the principal eigenvector of the input covariance matrix. The weight decay term has the simultaneous effect of making $\sum w_i^2$ tend towards 1 i.e. the weights are normalised.

However, this rule will find only the first eigenvector (that direction corresponding to the largest eigenvalue) of the data. It is not sufficient to simply throw clusters of neurons at the data since all will find the same (first) Principal Component; in order to find other PCs, there must be some interaction between the neurons. Other rules which find other principal components have been identified by subsequent research, an example of which is shown in the next Section.

3.4.1 Derivation of Oja's One Neuron Model

If we have a one neuron model whose activation is modelled by

$$y = \sum_{j} w_j x_j \tag{3.12}$$

and use simple Hebbian learning with renormalisation

$$\begin{aligned} w_j(t+1) &= w_j(t) + \alpha y(t) x_j(t) \\ w_j(t+1) &= \frac{w_j(t) + \alpha y(t) x_j(t)}{\{\sum_k (w_k(t) + \alpha y(t) x_k(t))^2\}^{\frac{1}{2}}} \end{aligned}$$

If α is very small, we can expand the last equation as a power series in α to get

$$w_j(t+1) = w_j(t) + \alpha y(t)(x_j(t) - y(t) w_j(t)) + O(\alpha^2) \tag{3.13}$$

where the last term, $O(\alpha^2)$ denotes terms which contain a term in the square or higher powers of α which we can ignore if $\alpha << 1$.

Therefore we can look at Oja's rule as an approximation to the simple Hebbian learning followed by an explicit renormalisation.

3.5 Recent PCA Models

We will consider 3 of the most popular PCA models. It is of interest to begin with the development of Oja's models over recent years.

3.5.1 Oja's Subspace Algorithm

The One Neuron network reviewed in the last section is capable of finding only the first Principal Component. While it is possible to use this network iteratively by creating a new neuron and allowing it to learn on the data provided by the residuals left by subtracting out previous Principal Components, this involves several extra stages of processing for each new neuron.

Therefore Oja's Subspace Algorithm provided a major step forward. The network has N output neurons each of which learns using a Hebb type rule with weight decay. Note however that it does not guarantee to find the actual directions of the Principal Components; the weights *do* however converge to an orthonormal basis of the Principal Component Space. We will call the space spanned by this basis the Principal Subspace. The learning rule is

$$\Delta w_{ij} = \alpha(x_j y_i - y_i \sum_k w_{kj} y_k) \tag{3.14}$$

which has been shown to force the weights to converge to a basis of the Principal Subspace.

Two sets of typical results from an experiment are shown in Table 3.2. The results shown in Table 3.2 are from a network with 5 inputs each of zero mean random Gaussians, where x_1's variance is largest, x_2's variance is next largest, and so on. Sample input data is shown in Table 3.1. You should be able to see that there is more spread in the data the further to the left you look. All data is zero mean.

Therefore, the largest eigenvalue of the input data's covariance matrix comes from the first input, x_1, the second largest comes from x_2 and so on. The advantage of using such data is that it is easy to identify the principal eigenvectors (and hence the principal subspace). There are 3 interneurons in the network and it can be seen that the 3-dimensional subspace corresponding to the first 3 principal components has been identified by the weights. There is very little of each vector outside the principal subspace i.e. in directions 4 and 5. The left matrix represents the results from the interneuron network[2], the right shows Oja's results. The lower ($W^T W$) section shows that the product of any two weights vectors is 0 while the product of a weight vector with itself is 1.Therefore the weights form an orthonormal basis of the space (i.e. the weights are at right angles to one another and of length 1). The upper (W) section shows that this space is almost entirely defined by the first 3 eigenvectors.

[2] which is equivalent to the subspace algorithm (see later)

First	Second	Third	Fourth	Fifth
-4.28565	-3.91526	3.13768	0.0433859	-0.677345
4.18636	3.43597	2.81336	-1.08159	-1.63082
-6.56829	0.849423	6.22813	-2.35614	-0.903031
5.97706	6.2691	-1.70276	-4.28273	0.626593
-2.54685	2.1765	-4.60265	2.34825	0.00339159
4.48306	-3.4953	3.50614	-2.22695	0.107193
-3.92944	-0.0524066	-4.75939	-0.816988	-1.10556
-1.37758	-2.22294	-0.108765	1.19515	1.84522
0.849091	0.189594	-3.75911	0.597238	1.73941
2.60213	-0.952078	-0.542339	0.58135	0.459956
6.21475	-0.48011	-1.31189	-2.50365	-0.809325
-4.33518	2.53261	1.47284	-4.52822	1.6673
10.1211	-4.96799	3.61302	0.00288919	0.48462
1.1967	3.71773	0.214127	0.105751	-0.343055
-8.72964	8.72083	1.2801	-1.41662	1.21766
10.8954	-7.03958	-2.00256	-2.27068	-2.1738
-2.1017	-0.779569	3.09251	1.51042	-2.11619
1.63661	2.40136	-4.79798	0.190769	-0.225283
-0.736219	0.389274	1.65305	1.79372	-0.133088
2.51133	-3.50206	-2.2774	2.13589	1.01751

Table 3.1: Each column represents the values input to a single input neuron. Each row represents the values seen by the network at any one time.

W			W		
0.249	**0.789**	**0.561**	**0.207**	**-0.830**	**0.517**
0.967	**-0.234**	**-0.100**	**-0.122**	**0.503**	**0.856**
-0.052	**-0.568**	**0.821**	**0.970**	**0.241**	-0.003
0.001	0.002	0.016	-0.001	0.001	0.001
-0.001	0.009	0.005	0.000	0.000	-0.001
W^TW			W^TW		
1.001	0.000	0.000	**1.000**	0.000	0.000
0.000	**1.000**	0.000	0.000	**1.000**	0.000
0.000	0.000	**1.000**	0.000	0.000	**1.000**

Table 3.2: Results from the simulated network and the reported results from Oja *et al.* The left matrix represents the results from the negative feedback network (see next Chapter), the right from Oja's Subspace Algorithm. Note that the weights are very small outside the principal subspace and that the weights form an orthonormal basis of this space. Weights above 0.1 are shown in bold font.

W			W		
1.000	-0.036	-0.008	**1.054**	-0.002	-0.002
0.036	**0.999**	-0.018	0.002	**1.000**	0.001
0.010	0.018	**1.000**	0.003	-0.002	**0.954**
-0.002	-0.002	0.016	-0.001	0.001	-0.002
0.010	0.003	0.010	0.001	-0.001	0.000
W^TW			W^TW		
1.001	0.000	0.000	**1.111**	0.000	0.000
0.000	**1.000**	0.000	0.000	**1.000**	0.000
0.000	0.000	**1.000**	0.000	0.000	**0.909**

Table 3.3: Results from the interneuron network (left) and from Oja (right).
Both methods find the principal eigenvectors of the input data covariance matrix. The interneuron algorithm has the advantage that the each vector is equally weighted.

One advantage of this model compared with some other networks is that it is completely homogeneous i.e. the operations carried out at each neuron are identical. This is essential if we are to take full advantage of parallel processing.

The major disadvantage of this algorithm is that it finds only the Principal Subspace of the eigenvectors not the actual eigenvectors themselves.

3.5.2 Oja's Weighted Subspace Algorithm

The final stage is the creation of algorithms which find the actual Principal Components of the input data. In 1992, Oja *et al* recognised the importance of introducing asymmetry into the weight decay process in order to force weights to converge to the Principal Components. The algorithm is defined by the equations

$$y_i = \sum_{j=1}^{n} w_{ij} x_j$$

where a Hebb-type rule with weight decay modifies the weights according to

$$\Delta w_{ij} = \alpha y_i (x_j - \theta_i \sum_{k=1}^{N} y_k w_{kj})$$

Ensuring that $\theta_1 < \theta_2 < \theta_3 < ...$ allows the neuron whose weight decays proportional to θ_1 (i.e. whose weight decays least quickly) to learn the principal values of the correlation in the input data. That is, this neuron will respond maximally to directions parallel to the principal eigenvector, i.e. to patterns closest to the main correlations within the data. The neuron whose weight decays proportional to θ_2 cannot compete with the first but it is in a better position than all of the others and so can learn the next largest chunk of the correlation, and so on.

It can be shown that the weight vectors will converge to the principal eigenvectors in the order of their eigenvalues. The algorithm clearly satisfies Miller and Mackay's definition of Multiplicative Constraints with $\gamma(w_i) = \theta_i \sum_k y_k w_{ki} x_i$. To compare the results with Oja's Weighted Subspace Algorithm, we repeat ed the above experiment with the algorithm. The results are shown in Table 3.3; the left set is from the negative feedback network (next Chapter), the right from Oja's Weighted Subspace Algorithm.

Clearly both methods find the Principal eigenvectors. We note that the interneuron results have the advantage of equally weighting each eigenvector.

3.5.3 Sanger's Generalized Hebbian Algorithm

Sanger has developed a different algorithm (which he calls the "Generalized Hebbian Algorithm") which also finds the actual Principal Components. He also introduces asymmetry in the decay term of his learning rule:

$$\Delta w_{ij} = \alpha(x_j y_i - y_i \sum_{k=1}^{i} w_{kj} y_k) \tag{3.15}$$

Note that the crucial difference between this rule and Oja's Subspace Algorithm is that the decay term for the weights into the i^{th} neuron is a weighted sum of the first i neurons' activations. Sanger's algorithm can be viewed as a repeated application of Oja's One Neuron Algorithm by writing it as

$$\Delta w_{ij} = \alpha([x_j y_i - y_i \sum_{k=1}^{i-1} w_{kj} y_k] - y_i^2 w_{ij}) \tag{3.16}$$

We see that the central term comprises the residuals after the first j-1 Principal Components have been found, and therefore the rule is performing the equivalent of One Neuron learning on subsequent residual spaces. So that the first neuron is using

$$\Delta w_{1i} = \alpha(x_i y_1 - y_1^2 w_{1i}) \tag{3.17}$$

while the second is using

$$\Delta w_{2i} = \alpha([x_i y_2 - y_2 w_{1i} y_1] - y_2^2 w_{2i}) \tag{3.18}$$

and so on for all the rest.

However, note that the asymmetry which is necessary to ensure convergence to the actual Principal Components, is bought at the expense of requiring the j^{th} neuron to 'know' that it is the j^{th} neuron by subtracting only j terms in its decay. It is Sanger's contention that all true PCA rules are based on some measure of deflation such as shown in this rule.

3.6 The InfoMax Principle in Linsker's Model

Linsker has developed a Hebb learning ANN model which attempts to realise the InfoMax principle - the neural net created should transfer the maximum amount of information possible between inputs and outputs subject to constraints needed to inhibit unlimited growth. Linsker notes that this criterion is equivalent to performing a principal component analysis on the cell's inputs.

Although Linsker's model is a multi-layered model, it does not use a supervised learning mechanism; he proposes that the information which reaches each layer should be processed in a way which maximally preserves the information. That this does not, as might be expected, lead to an identity mapping, is actually due to the effect of noise. Each neuron "responds to features that are statistically and information-theoretically most significant". He equates the process with a Principal Component Analysis.

Linsker's network is shown in Figure 3.2. Each layer comprises a 2-dimensional array of neurons. Each neuron in layers from the second onwards receives input from several hundred neurons in the previous layer and sums these inputs in the usual fashion. The region of the previous layer which sends input to a neuron is called the receptive field of the neuron and the density of distribution of inputs from a particular region of the previous layer is defined by a Gaussian distribution i.e. we can imagine two layers of neurons with the a neuron in the second layer receiving most activation from those neurons which lie directly below it and less activation from those neurons further away from directly beneath it. At the final layer, lateral connections within the layer are allowed.

The Hebb-type learning rule is

$$\Delta w_{ij} = a(x_j - E(x))(y_i - E(y)) + b$$

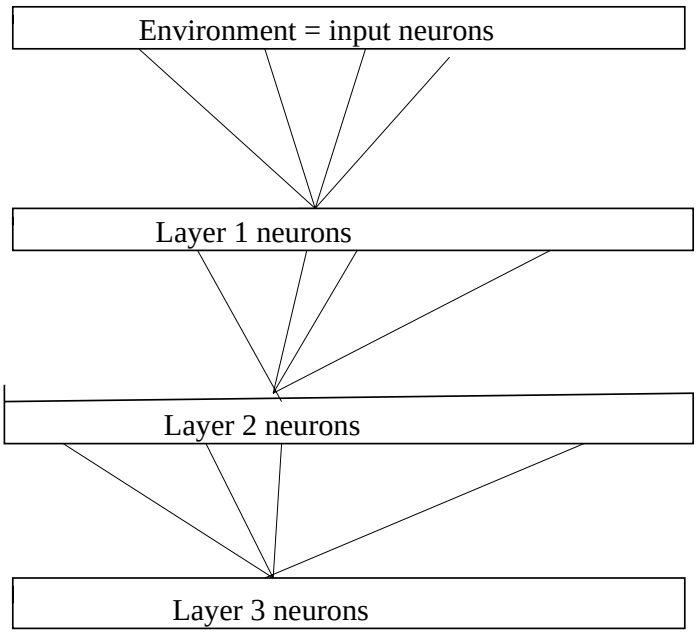

Figure 3.2: Linsker's model

where a and b are constants.

In response to the problem of unlimited growth of the network weights, Linsker uses a hard limit to the weight-building process i.e. the weights are not allowed to exceed w^+ nor decrease beyond w^- where $w^- = -w^+$.

Miller and MacKay have observed that Linsker's model is based on Subtractive Constraints, i.e.

$$\Delta w_{ij} = a x_j y_i - a E(x) y_i - a E(y)(x_j - E(x))$$

Both y_i and $E(y)$ are functions of w, but in neither case are we multiplying these by w itself. Therefore, as noted earlier, the weights will not tend to a multiple of the principal eigenvector but will saturate at the bounds (w_{ij}^+ *or* w_{ij}^-) of their permissible values.

Because the effects of the major eigenvectors will still be felt, there will not be a situation where a weight will tend to w^- in a direction where the principal eigenvector has a positive correlation with the other weights. However, the directions of the weight matrix will, in general, bear little resemblence to any eigenvector of the correlation matrix. The model will not, in general, enable maximal information transfer through the system.

Linsker showed that after several layers, this model trained on noise alone, developed

- center-surround cells - neurons which responded optimally to inputs of a bright spot surrounded by darkness or vice-versa
- bar detectors - neurons which responded optimally to lines of activity in certain orientations

Such neurons exist in the primary visual cortex of mammals. They have been shown to respond even before birth to their optimal inputs though their response is refined by environmental influences i.e. by experience.

3.7 Regression

Regression comprises finding the best estimate of a dependent variable, y, given a vector of predictor variables, **x**. Typically, we must make some assumptions about the form of the predictor

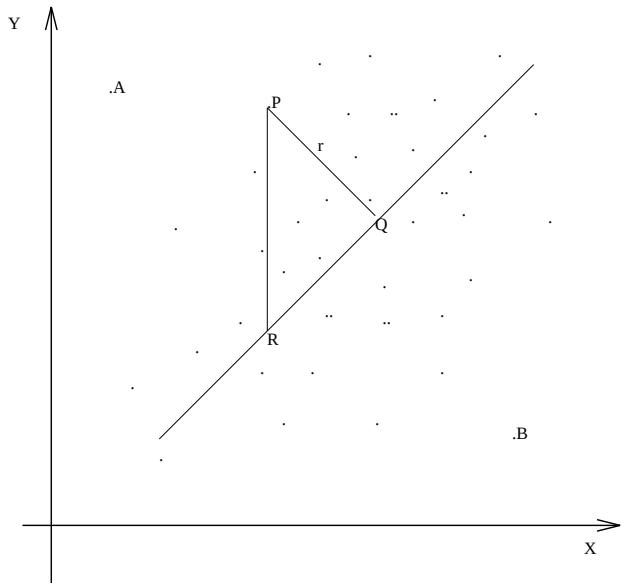

Figure 3.3: The vertical lines will be minimised by the Least Squares method. The shortest distances, r_i, will be minimised by the Total Least Squares method.

surface e.g. that the surface is linear or quadratic, smooth or disjoint etc.. The accuracy of the results achieved will test the validity of our assumptions.

This can be more formally stated as: let (X,Y) be a pair of random variables such that $X \in R^n, Y \in R$. Regression aims to estimate the response surface,

$$f(x) = E(Y|X = x) \tag{3.19}$$

from a set of p observations, $\mathbf{x}_i, y_i, i = 1, ..., p$.

The usual method of forming the optimal surface is the Least (Sum of) Squares Method which minimises the Euclidean distance between the actual value of y and the estimate of y based on the current input vector, $\mathbf{x}$. Formally, if we have a function, f, which is an estimator of the predictor surface, and an input vector, $\mathbf{x}$, then our best estimator of y is given by minimising

$$E = \min_f \sum_i^N (y_i - f(\mathbf{x}_i))^2 \tag{3.20}$$

i.e. the aim of the regression process is to find that function f which most closely matches y with the estimate of y based on using f() on the predictor, $\mathbf{x}$, for all values (y,$\mathbf{x}$).

For a linear function of a scalar x, we have $y = mx + c$, and so the search for the best estimator, f, is the search for those values of m and c which minimise

$$E_1 = \min_{m,c} \sum_i (y_i - mx_i - c)^2$$

For each sample point in Figure 3.3, this corresponds to finding that line which minimises the sum of the vertical lengths such as PR from all actual y-values to the best-fitting line, $y = mx + c$.

However, in minimising this distance, we are making an assumption that only the y-values contain errors while the x-values are known accurately. This is often not true in practical situations in which, for example, which variable constitutes the response variable and which the predictor variables is often a matter of choice rather than being a necessary feature of the problem. Therefore, the optimal line will be that which minimises the distance, r, i.e. which minimises the shortest distance from each point, (x_i, y_i) to the best fitting line. Obviously, if we know the

relative magnitude of the errors in x and y, we will incorporate that into the model; however here we assume no foreknowledge of the magnitudes of errors. Thus, we are seeking those values of m and c which minimise

$$E_2 = \min_{m,c} \sum_i r_i^2 = \min_{m,c} \sum_i \frac{(y_i - mx_i - c)^2}{1 + m^2}$$

This is the so-called Total Least Squares method. Because of the additional computational burden introduced by the non-linearity in calculating E_2, TLS is less widely used than LS although the basic idea has been known for most of this centuary.

3.7.1 Minor Components Analysis

Xu *et al.* have shown that the TLS fitting problem can be solved by performing a Minor Component Analysis of the data: i.e. finding those directions which instead of containing maximum variance of the data contain minimum variance. Since there may be errors in both y and x we do not differentiate between them and indeed incorporate y into the input vector $\mathbf{x}$. Therefore we reformulate the problem as: find the direction $\mathbf{w}$ such that we minimise E_2 i.e.

$$\begin{aligned} E_2 &= \min_{\mathbf{w}} \frac{(\mathbf{w}.\mathbf{x} + c)^2}{\mathbf{w}^2} \text{ over all inputs } \mathbf{x} \\ &= \min_{\mathbf{w}} \sum_{i=1}^{N} \frac{(\mathbf{w}.\mathbf{x}_i + c)^2}{\mathbf{w}^2} \\ &= N \min_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{R} \mathbf{w} + 2c\mathbf{w}^T \mathbf{E}(\mathbf{x}) + c^2}{\mathbf{w}^T \mathbf{w}} \end{aligned}$$

where $R = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x_i} \mathbf{x_i}^T$, the autocorrelation matrix of the data set and $E(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i$, the mean vector of the data set. Since, at convergence, $\frac{dE_2}{d\mathbf{w}} = 0$, we must have

$$R\mathbf{w} + cE(\mathbf{x}) - \lambda \mathbf{w} = 0 \tag{3.21}$$

where $\lambda = \frac{\mathbf{w}^T R \mathbf{w} + 2c\mathbf{w}^T E(\mathbf{x}) + c^2}{\mathbf{w}^T \mathbf{w}}$. Now we wish to find a hyperplane of the form

$$\mathbf{w}.\mathbf{x} + c = 0$$

So, taking expectations of this equation we have $c = -\mathbf{w}.\mathbf{E}(\mathbf{x})$ which we can substitute

$$C\mathbf{w} - \lambda \mathbf{w} = 0 \tag{3.22}$$

where now $\lambda = \frac{\mathbf{w}^T C \mathbf{w}}{\mathbf{w}^T \mathbf{w}}$ where C is the covariance matrix $= R - E(\mathbf{x}\mathbf{x}^T)$. From this we can see that every eigenvector is a solution of the minimisation of E_2.

Using a similar technique to that used earlier to show that only the greatest Principal Component was stable for Oja's one neuron rule, we can now show that only the smallest Principal Component is stable for this rule.

As an example of the network in operation, we show in the first line of Table 3.4 the converged values of the weights of an MCA network when sample points are drawn from the line and both x and y coordinates are subject to noise. Clearly the algorithm has been successful.

For the lines shown in Table 3.4, points were drawn uniformly from only the first (both x and y positive) quadrant of the distribution determined by the line in each case. The first 3 lines show the direction to which the network converged when the distribution was affected by only white noise in both x and y direction drawn from N(0,0.05). Clearly the degree of accuracy of the convergence depends very greatly on the relative proportion of the amount of variance due to the length of the distribution from which points were drawn and the white noise. In the third case, the noise was of the same order as the variance due to the spread of points on the line and the convergence was severly disrupted.

Actual Distribution	Direction Found	Outliers
3x + 2y = 10	0.300x + 0.200y = 1	none
3x + 2y = 1	2.970x + 2.018y = 1	none
3x + 2y = 0.1	24.3x + 23.7y =1	none
3x + 2y = 1	3.424x + 1.714y = 1	1% in y direction
3x + 2y = 1	2.459x + 2.360y = 1	1% in x direction

Table 3.4: Directions converged to when the points from the distribution were disturbed by noise drawn from N(0,0.05)

3.8 Your Practical Work

3.8.1 Annealing of Learning Rate

The mathematical theory of learning in Principal Component Nets requires the learning rate to be such that $\alpha_k \geq 0, \sum \alpha_k^2 < \infty, \sum \alpha_k = \infty$. In practise, we relax these requirements somewhat and we find that we can generally find an approximation to the Principal Components when we use a small learning rate.

However for more accurate results we can anneal the learning rate to zero during the course of the experiment. Different annealing schedules have been tried - e.g. to subtract a small constant from the learning rate at each iteration, to multiply the learning rate by a number < 1 during the course of each iteration, to have a fixed learing rate for the first 1000 iterations and then to anneal it and so on. All of these have been successfully used in practise and it is reccommended that your simulations use one of these methods.

3.8.2 The Data

The theory of Principal Components is most easily applied when we have Gaussian distributions. Not all compilers come with a built-in Gaussian distribution but most usually have a means of creating samples from a uniform distribution from 0 to 1: e.g. in C on the Unix workstations, we typically use drand48(). Donald Knuth has an algorithm which provides a means of creating a pseudo-Gaussian distribution from a uniform distribution. The C++ code for doing so is shown below:

```
#define A1  3.949846138
#define A3  0.252408784
#define A5  0.076542912
#define A7  0.008355968
#define A9  0.029899776

double cStat::normalMean(double mean,double stdev)
{
        int j;
        float r,x;
        float rsq;

        r=0;
        for(j=0;j<12;j++) r += drand48();
        r = (r-6)/4;

        rsq = r*r;
        x = ((((A9*rsq+A7)*rsq + A5)*rsq + A3)*rsq+A1)*r;
```

```
        return mean+x*stdev;
}
```

This is a function which takes in two doubles (the mean and standard deviation of the distribution you wish to model) and returns a single value from that distribution. It uses 5 values, $A_1 - A_9$, which are constant for the life of the program. You should call the function with e.g.
x[0] = aStat.normalMean(0,7);
x[1]= aStat.normalMean(0,6); etc