Chapter 7

Self-Organizing Neural Networks

7.1 Introduction

In supervised learning there is no teacher! "The purpose of an algorithm for self-organized (or unsupervised) learning is to discover significant features or patterns in the input data and todo the discovery without a teacher".

One often divide the algorithms for unsupervised learning into two subclasses:

- Hebbian learning algorithms. This is basically algorithms that will extract principal components.
- Competitive learning. This is algorithms that are related to various forms of clustering.
  - Competitive networks for vector quantization (clustering)
  - Self-organizing feature maps (SOFM)
  - Learning vector quantization (LVQ)

We will start by unsupervised Hebbian learning. But since this is much related to *principal component analysis*, we will start with that!

7.2 Principal Component Analysis (PCA)

7.2.1 Finding maximum variance directions

PCA is about finding directions in a data set where we have large variance (see Fig. 7.1). Suppose we have data $D = \{\mathbf{x}_n, \mathbf{d}_n\}_{n=1\cdots N}$ and $\mathbf{x} \in \mathbb{R}^P$. We will assume that the data has

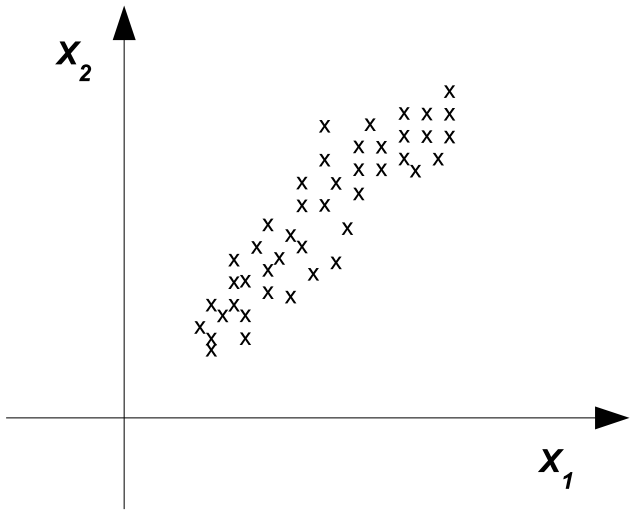

Figure 7.1: Find the axis onto which the projected data has the largest variance.

zero mean i.e.

$$\sum_{n=1}^{N} x_{nk} = 0 \ \ \forall k$$

If not, we can always subtract the mean from the data. Let $\mathbf{u}$ be a unit vector onto which $\mathbf{x}$ is projected. The projection a is given by

$$a = \mathbf{x}^T\mathbf{u} = \mathbf{u}^T\mathbf{x}$$

The mean of a is zero, because

$$< a >= \frac{1}{N}\sum_{n=1}^{N} \mathbf{u}^T\mathbf{x}_n = \frac{1}{N}\mathbf{u}^T \underbrace{\sum_{n}^{N} \mathbf{x}_n}_{=0} = 0$$

The variance of a is given by,

$$< (a- < a >)^2 >=< a^2 > \ = \ \frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}^T\mathbf{x}_n)(\mathbf{u}^T\mathbf{x}_n) = \tag{7.1}$$

$$= \ \frac{1}{N}\sum_{n=1}^{N}\mathbf{u}^T\mathbf{x}_n\mathbf{x}_n^T\mathbf{u} = \tag{7.2}$$

$$= \ \mathbf{u}^T \underbrace{\frac{1}{N}\sum_{n=1}^{N}\mathbf{u}^T\mathbf{x}_n\mathbf{x}_n^T}_{\mathbf{R}}\mathbf{u} = \mathbf{u}^T\mathbf{R}\mathbf{u} \tag{7.3}$$

$\mathbf{R}$ is the covariance matrix for the data and describes the degree of correlation between the different components of $\mathbf{x}$. Note that $\mathbf{R}$ is symmetric and $\mathbf{R}$ will be diagonal for completely uncorrelated data. So the variance of the projection a is given by $< a^2 >= \mathbf{u}^T\mathbf{R}\mathbf{u}$. We now want to maximize this variance for a given $\mathbf{u}$, lets call it $\mathbf{u}_1$. Clearly we need a way of constraining $\mathbf{u}_1$, to prevent $\mathbf{u}_1 \to \infty$, since $\mathbf{u}_1$ should be a unit vector with $\mathbf{u}_1^T\mathbf{u}_1 = 1$. A Lagrange multiplier, λ_1, can do the trick,

$$\text{maximize} \qquad \mathbf{u}^T\mathbf{R}\mathbf{u} + \lambda_1 \left(1 - \mathbf{u}_1^T\mathbf{u}_1\right)$$

By setting the derivative with respect to $\mathbf{u}_1$ equal to zero, we conclude that we have a stationary point when,

$$\mathbf{R}\mathbf{u}_1 = \lambda_1\mathbf{u}_1 \tag{7.4}$$

which means that $\mathbf{u}_1$ must be an eigenvector to $\mathbf{R}$ with eigenvalue λ_1. If we left-multiply Eqn. 7.4 with $\mathbf{u}_1^T$ we get

$$\mathbf{u}_1^T\mathbf{R}\mathbf{u}_1 = \lambda_1$$

thus, the variance of a is λ_1. Hence, we get the maximum variance when $\mathbf{u}_1$ is equal to the eigenvector having the largest eigenvalue, known as the first principal component. We can then iterate this to obtain all principal components by simply finding all eigenvectors and sort them according to decreasing eigenvalues, $\lambda_1 > \lambda_2 > \ldots > \lambda_P$.

We can now make the transformation of our data vector $\mathbf{x}$,

$$a_j = \mathbf{u}_j^T\mathbf{x} \qquad j = 1, ..., P$$

or if we collect all eigenvectors in a matrix $\mathbf{U}$,

$$\mathbf{a} = \mathbf{U}^T\mathbf{x}$$

We call a_j the principal components. We can also express $\mathbf{x}$ in the new coordinate system with unit vectors $\mathbf{u}_j$, as

$$\mathbf{x} = (\mathbf{U}^T)^{-1}\mathbf{a} = \mathbf{U}\mathbf{a} = \sum_j^P a_j\mathbf{u}_j \tag{7.5}$$

7.2.2 Dimensionality reduction

We can truncate the above representation of $\mathbf{x}$,

$$\mathbf{x}' = \sum_{j=1}^{M} a_j\mathbf{u}_j$$

The error

$$\mathbf{e} = \mathbf{x} - \mathbf{x}' = \sum_{j=M+1}^{P} a_j \mathbf{u}_j$$

It is easy to show that $\mathbf{e}^T \mathbf{x}' = 0$, meaning that $\mathbf{e}$ and $\mathbf{x}'$ are orthogonal. The total variance of the error vector (sum of variance for each component) is then given by

$$\sum_{j=M+1}^{P} \lambda_j$$

In a practical example one would compute the eigenvalues and eigenvectors of the covariance matrix $\mathbf{R}$ and then only keep the first M components $(a_1, a_2, ..., a_M)$ as the new input vector (instead of $\mathbf{x}$). Note:

- It is a linear transformation. Only linear dependencies among the input components will be captured.
- No easy interpretation of a_j.

Figure 7.2 shows some 2-D data sets and the corresponding eigenvalues.

7.2.3 Self-organizing network that does PCA

We will now look at a 2-layer network with one output using a Hebbian-like update rule. It will extract the first principal component.

The updating rule for the weights are (Hebb rule),

$$\omega_i(t+1) = \omega_i(t) + \eta y(t) x_i(t) \qquad i = 1, ..., P$$

Here t serves as an iteration index. At each iteration a new pattern $\mathbf{x}$ is selected from the training data set, denoted by $\mathbf{x}(t)$ giving the corresponding output $y(t)$. There is nothing here that prevents the weights from growing unbounded. To prevent this we can introduce normalization.

$$\omega_i(t+1) = \frac{\omega_i(t) + \eta y(t) x_i(t)}{\sqrt{\sum_j (\omega_j(t) + \eta y(n) x_j(t))^2}} \qquad i = 1, ..., P$$

Now Taylor expand for small η (to first degree),

$$\omega_i(t+1) = \omega_i(t) + \eta y(t) \left[x_i(t) - y(t) \omega_i(t) \right]$$

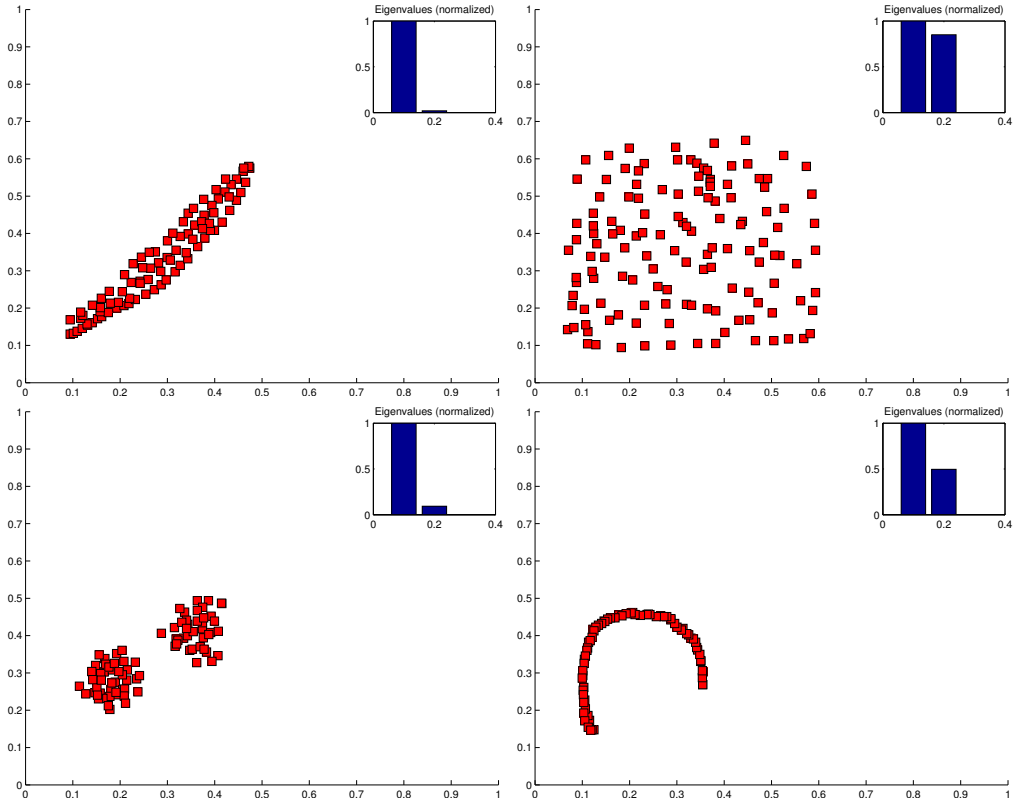

Figure 7.2: Eigenvalues for some 2-D data sets.

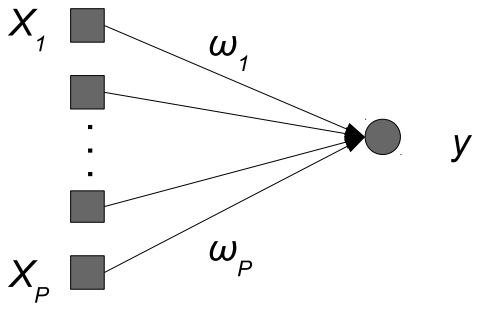

Figure 7.3: Network for extracting the first principal component.

In vector notation,

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \eta y(t)\left[\mathbf{x}(t) - y(t)\boldsymbol{\omega}(t)\right]$$

Now, replace $y(t)$ by

$$y(t) = \mathbf{x}(t)^T\boldsymbol{\omega}(t) = \boldsymbol{\omega}(t)^T\mathbf{x}(t)$$

resulting in,

$$\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}(t) + \eta \left[\mathbf{x}(t)\mathbf{x}(t)^T\boldsymbol{\omega}(t) - \boldsymbol{\omega}(t)^T\mathbf{x}(t)\mathbf{x}(t)^T\boldsymbol{\omega}(t)\boldsymbol{\omega}(t)\right]$$

Assume now that $\boldsymbol{\omega}(t) \to \mathbf{q}_o$ as $t \to \infty$ (it can be shown). Large t here means that we iterate over the training set many times. If $\boldsymbol{\omega}(t)$ converges to $\mathbf{q}_o$ it implies, in an average sense,

$$< \mathbf{x}\mathbf{x}^T\mathbf{q}_o - \mathbf{q}_o^T\mathbf{x}\mathbf{x}^T\mathbf{q}_o\mathbf{q}_o >= 0$$

where $< \cdot >$ means expectation or sample average over the training data set. Remember the definition of the covariance matrix $\mathbf{R}$ (Eqn. 7.1),

$$\begin{aligned} & < \mathbf{x}\mathbf{x}^T > \mathbf{q}_o - \mathbf{q}_o^T < \mathbf{x}\mathbf{x}^T > \mathbf{q}_o\mathbf{q}_o = 0 \\ \Leftrightarrow \quad & \mathbf{R}\mathbf{q}_o = \mathbf{q}_o^T\mathbf{R}\mathbf{q}_o\mathbf{q}_o \end{aligned}$$

This can be written,

$$\mathbf{R}\mathbf{q}_o = \lambda_o\mathbf{q}_o \tag{7.6}$$

with

$$\lambda_o = \mathbf{q}_o^T\mathbf{R}\mathbf{q}_o = \mathbf{q}_o^T\lambda_o\mathbf{q}_o = \lambda_o||\mathbf{q}_o||^2 \tag{7.7}$$

Equation 7.6 shows that $\mathbf{q}_o$ must be an eigenvector of $\mathbf{R}$ and from Eqn. 7.7 it follows that $\mathbf{q}_o$ is a normalized eigenvector. One can also show that $\lambda_o = \lambda_{\max}$. So, we have a neural network that can extract the first principal component.

Sanger network

If we want a network that can extract more principal components we just add more output nodes. Figure 7.4 shows a network that can extract the first M principal components. Each of the output nodes computes,

$$y_i(\mathbf{x}_n) = \sum_k^P \omega_{ik}x_{nk}$$

Here n is index for data points. The update rule for the weight, that will extract the first M principal components of the input data is called Sanger's rule,

$$\omega_{ik} \to \omega_{ik} + \eta \left[y_i(\mathbf{x}_n)x_{nk} - y_i(\mathbf{x}_n)\sum_j^i \omega_{jk}y_j(\mathbf{x}_n) \right] \qquad i = 1, .., M$$

The weight vectors $\boldsymbol{\omega}_i$ will converge to the eigenvectors corresponding of the covariance matrix. This update rule is given as online updating, but it can also be used in a block or batch fashion.

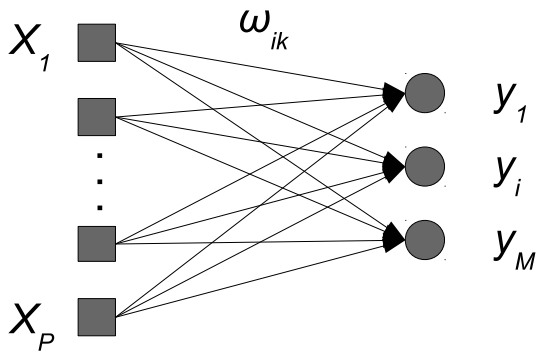

Figure 7.4: Sanger network for extracting the first M principal components.

APEX networks

Another algorithm for doing PCA is the APEX network (Adaptive Principal components EXtraction). This architecture has lateral inhibition for the output nodes (see Fig. 7.5. The

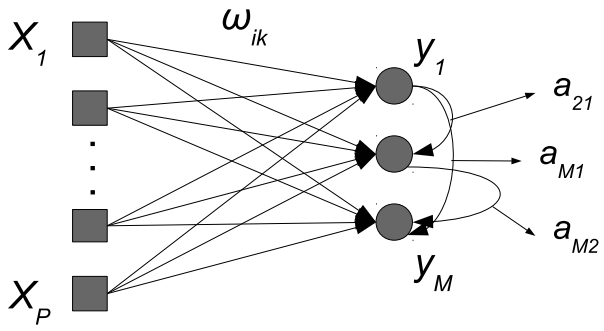

Figure 7.5: APEX network for extracting the first M principal components.

output is given by,

$$y_j(\mathbf{x}_n) = \boldsymbol{\omega}_j^T \mathbf{x}_n + \mathbf{a}_j^T \mathbf{y}_{j-1}(\mathbf{x}_n)$$

where $\mathbf{a}_j$ are all lateral weights feeding into output node j, and $\mathbf{y}_{j-1}(\mathbf{x}_n)$ is defined as

$$\mathbf{y}_{j-1}(\mathbf{x}_n) = (y_1(\mathbf{x}_n), y_2(\mathbf{x}_n), ..., y_{j-1}(\mathbf{x}_n))$$

The update rule for the APEX network is,

$$\begin{aligned} \boldsymbol{\omega}_j &\rightarrow \boldsymbol{\omega}_j + \eta\big[y_j(\mathbf{x}_n)\mathbf{x}_n - y_j(\mathbf{x}_n)^2\boldsymbol{\omega}_j\big] \\ \mathbf{a}_j &\rightarrow \mathbf{a}_j + \eta\big[y_j(\mathbf{x}_n)\mathbf{y}_{j-1}(\mathbf{x}_n) - y_j(\mathbf{x}_n)^2\mathbf{a}_j\big] \end{aligned}$$

Autoencoder

We have talked about the autoencoder before. Here we will just say that a single hidden layer autoencoder, as in figure 7.6, will perform a PCA analysis. Remember that the encoder

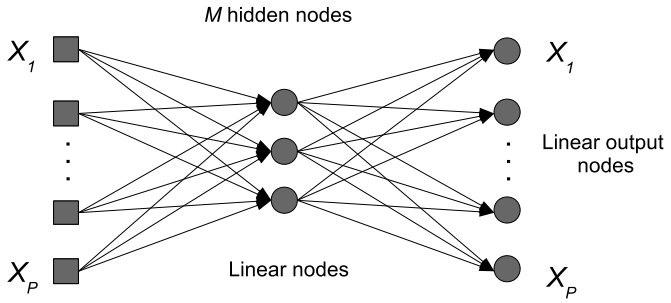

Figure 7.6: Autoencoder network. The inputs are mapped onto themselves using a hidden layer of fewer nodes than the number of inputs.

network can be trained using the following error function,

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_{n}^{N}\sum_{i}^{M}\big(y_i(\mathbf{x}(n),\boldsymbol{\omega}) - x_i(n)\big)^2$$

This network will extract the M first principal components!

7.3 Competitive learning

In competitive learning we have an architecture similar to the PCA networks (see Fig. 7.7). One difference is that we define a "winner" j^* among the C output nodes, usually as

$$j^* = \arg\max_j h(||\mathbf{x} - \boldsymbol{\omega}_j||) \tag{7.8}$$

for some function $h(x)$. A common choice is $h(x) = -x$, meaning that the winning node is simply the one having the weight vector being closest to the input data. Competitive learning is very much used for clustering. Before we present the competitive network, lets describe the standard *k-means* clustering method.

k-means clustering

1. **Initialization**
   Decide how many clusters to look for (k). Define initial positions for the k clusters (see Fig. 7.8).

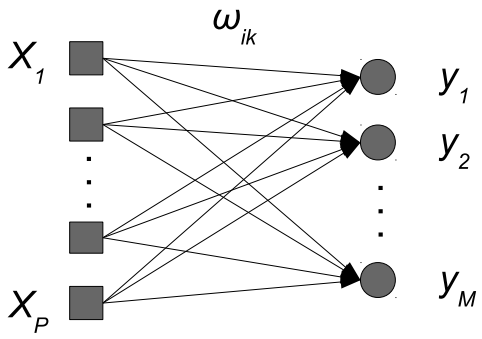

Figure 7.7: The network architecture used for competitive learning.

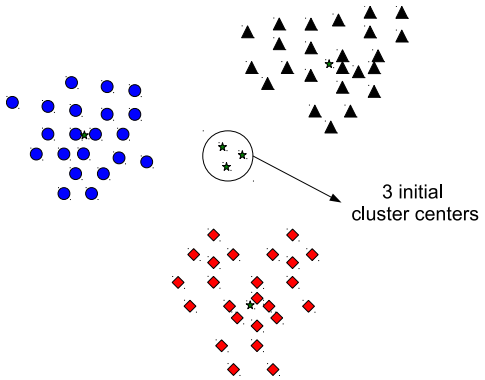

Figure 7.8: Clustering example. Here 3 initial cluster centers are updated according to the k-means procedure, to finally discover the clusters, i.e. move to the center of each cluster.

The algorithm now alternates between two steps:

2. **Assignment**
   Each data point is assigned to the nearest cluster. Nearest here often means the Euclidian distance between the data point and the cluster center. If $\mathbf{m}_j^{(n)}$ are the cluster centers at iteration n and $S_j^{(n)}$ is the set of all data points belonging to cluster center j at iteration n. Then we can write the assignment step as:

$$S_i^{(n)} = \left\{ \mathbf{x} : ||\mathbf{x} - \mathbf{m}_i^{(n)}||^2 < |\mathbf{x} - \mathbf{m}_j^{(n)}||^2, j = 1, ..., k \right\}$$

3. **Update**

Update all cluster centers to be the mean of all data points part of the cluster,

$$\mathbf{m}_j^{(n+1)} = \frac{1}{|S_j^{(n)}|} \sum_{\mathbf{x}_k \in S_j^{(n)}} \mathbf{x}_k$$

The above procedure can be viewed as batch updating. One can also imagine an online procedure where cluster centers are updated after only "presenting" one data point. For this approach to work one need to introduce a learning rate, meaning that a cluster center is moved only a small distance towards the data point. As for neural network training a block update approach is also possible.

7.3.1 Competitive learning for clustering

For the competitive network (see Fig. 7.7) the output nodes are typically set by the following rule,

$$y_j = \begin{cases} 1 & \text{if } j \text{ is the winning node} \\ 0 & \text{otherwise} \end{cases} \tag{7.9}$$

where the winning node is determined according to Eqn. 7.8. Given a training data set $\{\mathbf{x}(1), ..., \mathbf{x}(N)\}$, the training of a competitive network is then accomplished using the following procedure:

1. Determine the size of the network, i.e. the number of output nodes.
2. Initialize all weight vectors $\boldsymbol{\omega}_j$ (cluster centers) randomly, but close the the center of mass of the data set.
3. Randomly pick an input pattern μ from the data set.
   3.1 Find the winning output node j^*.
   3.2 Update the weight vector $\boldsymbol{\omega}_{j^*}$ according to
   $$\boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big)$$
4. Repeat step 3 until no more changes occur, including a necessary lowering of the learning rate η.

Again, the above procedure is online learning and can easily be converted into a batch version where all weight updates are averaged over the full data set before doing the actual

weight update. The batch version is identical to k-means clustering if the winning node is determined based on the distance between weight vectors and data points. Once the competitive network has been trained (i.e. the clusters have been determined) it can be used in a test situation where the winning node signals cluster belonging for any new data point.

7.3.2 Learning vector quantization (LVQ)

Since competitive learning is about finding clusters in the data, it is to some extent similar to classification problems. We just have to label the different clusters with appropriate class labels. There is a supervised version of the above competitive learning algorithm, called *Learning Vector Quantization* (LVQ). Here we again have both input data $\{\mathbf{x}_1, ..., \mathbf{x}_N\}$ and target labels $\{\mathbf{d}_1, ..., \mathbf{d}_N\}$. The basic idea of LVQ is to label each of the outputs in the competitive network with a class label and introduce a modified learning rule that depends on whether the class of the winning node is correct or not.

The LVQ learning rule is:

$$\boldsymbol{\omega}_{j^*} \rightarrow \begin{cases} \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to the same class} \\ \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} - \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to different classes} \end{cases}$$

We can summarize the LVQ classification network as follows:

1. Determine the size of the network, i.e. the number of output nodes.
2. Initialize all weight vectors $\boldsymbol{\omega}_j$ (cluster centers) randomly, preferably such that they cover most of the input space.
3. Label all output nodes with the available target class labels. It can be an even distribution or follow the prior class probabilities.
4. Randomly pick an input pattern μ from the data set.

   3.1 Find the winning output node j^*.

   3.2 Update the weight vector $\boldsymbol{\omega}_{j^*}$ according to

$$\boldsymbol{\omega}_{j^*} \rightarrow \begin{cases} \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ “agrees”.} \\ \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} - \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ “disagrees”.} \end{cases}$$

5. Repeat step 3 until no more changes occur, including a necessary lowering of the learning rate η.

There are modified learning rules for LVQ, called LVQ2.1 and LVQ3, but the underlying idea is as presented above. Two examples of LVQ classification can be seen in Fig. 7.9.

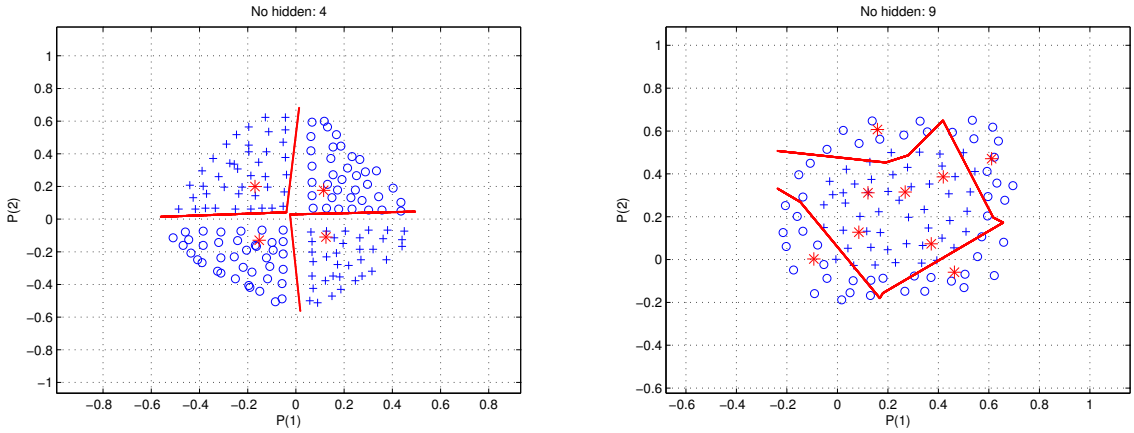

Figure 7.9: Examples of decision boundaries produced by the LVQ network. (Left) The red "*" are the cluster centers found by the LVQ algorithm. Two of then belong to the "+" class and two of the to the "ring" class. (Right) This problem requires many more output nodes, if we compare to an MLP. Again, the red "*" marks the found cluster centers by the LVQ algorithm.

7.4 Self-organizing feature maps

From Haykin: "The principal goal of the self-organizing feature mapping algorithm developed by Kohonen (1982) is to transform an incoming signal pattern of arbitrary dimension into a one- or two-dimensional discrete map, and to perform this transformation adaptively in a topological ordered fashion."

The output nodes are placed in a 1- or 2-dimensional grid where each node has a predefined number of neighbors (see Fig. 7.10). Each of the inputs are connected to each of the output nodes, the same principle for all other feed-forward architectures we have used. Figure 7.11 shows a self-organizing feature map (SOFM) for a 2-dimensional square output grid.

An important difference between competitive networks and SOFM networks is the *neighborhood* function that is used by the SOFM. It is important in order to have a topology preserving map. The neighborhood function Λ_{jj^*} uses the distance between output node j and j^* to derive its output. As for the competitive network a winning node is found among all output nodes, typically the node having the smallest Euclidian distance to the input

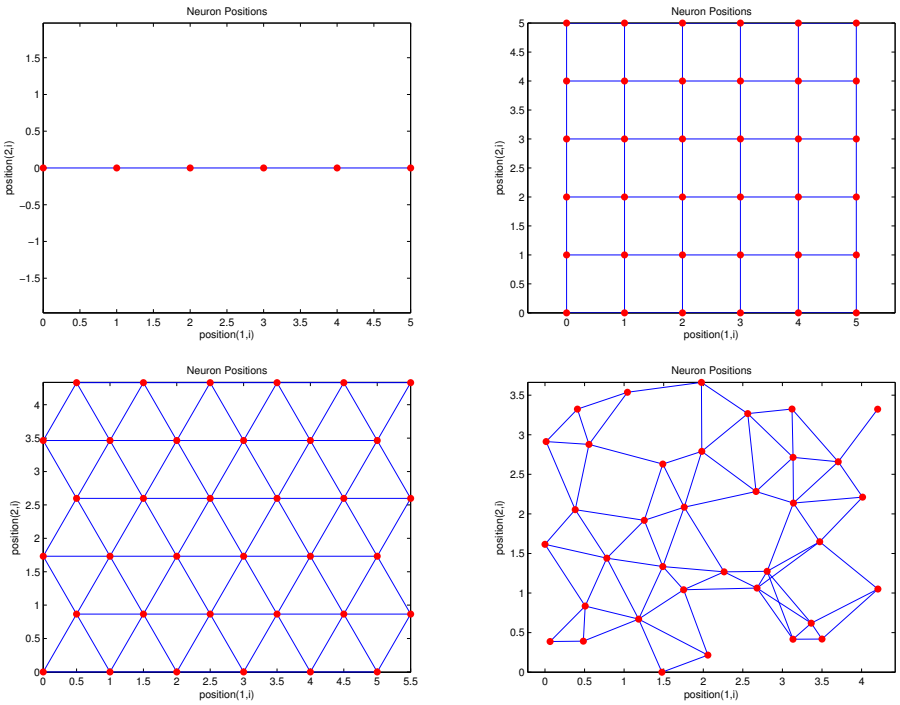

Figure 7.10: Examples of output grids for the self-organizing feature map. A 1-dimensional grid where each node have 2 neighbors, one 2-dimensional square grid where each node has 4 neighbors, a hexagonal grid where each node has 6 neighbors and finally a random grid. (All figures produced in Matlab).

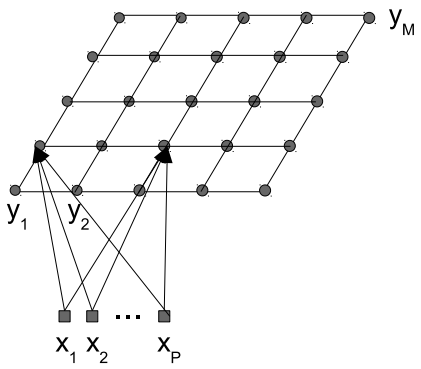

Figure 7.11: An example of a SOFM where a P dimensional input $\mathbf{x}$ is mapped onto a 2-dimensional square grid.

vector (see Eqn. 7.8. The update equation for the SOFM is,

$$\boldsymbol{\omega}_j \rightarrow \boldsymbol{\omega}_j + \eta \Lambda_{jj^*} \big(\mathbf{x}_\mu - \boldsymbol{\omega}_j \big) \tag{7.10}$$

where j^* is the winning node for $\mathbf{x}_\mu$. Note that for the SOFM all weights are updated, not just the winning node. It is the neighborhood function that determines how many (or how mush) of the neighboring nodes that are updated. Examples of neighborhood functions are:

$$(1) \quad \Lambda_{jj'} = \begin{cases} 1 & \text{if } d(j,j') \leq d_o \\ 0 & \text{otherwise} \end{cases} \tag{7.11}$$

$$(2) \quad \Lambda_{jj'} = e^{-d(j,j')^2/2\sigma^2} \tag{7.12}$$

where $d(j,j')$ is the distance between node j and j' on the grid. Again there are several possible choices for $d(j,j')$, such as Euclidian distance or link distance, where the latter is just the smallest number of links connecting j and j' on the grid. We can summarize the SOFM as,

1. Choose the number of output nodes and the layout (1- or 2-dimensional and grid).
2. Initialize all weight vectors $\boldsymbol{\omega}_j$, typically randomly around center-of-mass for the input data.
3. Repeat until all $\boldsymbol{\omega}_j$ have converged:
   - 3.1 Randomly pick an input pattern μ from the data set.
   - 3.2 Find the winning output node j^*.
   - 3.3 Update the weight vector $\boldsymbol{\omega}_{j^*}$ according to
     $$\boldsymbol{\omega}_j \rightarrow \boldsymbol{\omega}_j + \eta \Lambda_{jj^*} \big(\mathbf{x}_\mu - \boldsymbol{\omega}_j \big)$$
   - 3.4 It is often necessary to dynamically change d_o/σ and η to find good "solutions" and ensure convergence.

7.4.1 Properties of the SOFM

Haykin (chapter 9.4) lists an number of properties of the SOFM:

1 **Approximation of the input space**
The feature map Φ, represented by the set of synaptic weight vectors $\{\boldsymbol{\omega}_j\}$ in the output space $\mathcal{A}$, provides a good approximation of input space $\mathcal{X}$.

2 **Topological ordering**
The feature map Φ computed by the SOFM algorithm is topologically ordered in the sense that the spatial location of a neuron in the lattice (grid) corresponds to a particular domain or feature of the input patterns.

3 **Density matching**
The feature map Φ, reflects variations in the statistics of the input distribution: Regions in the input space $\mathcal{X}$ from which sample vectors $\mathbf{x}$ are drawn with a high probability of occurrence are mapped onto larger domains of the output space, and therefore with better resolution than regions in $\mathcal{X}$ from which sample vectors $\mathbf{x}$ are drawn with a low probability of occurrence.

4 **Feature selection**
Given data from an input space, the self-organizing map is able to select a set of best features for approximating the underlying distribution.

The following numerical examples will provide some insight into these properties.

7.4.2 Some SOFM examples

For all examples presented in the figures below (7.12-7.12) the input data is 2-dimensional (continuous) and the output grid of the SOFM is either 1-D or 2-D. The weight vectors (after training) are shown as grey dots and there is a red line between nearest neighbors.

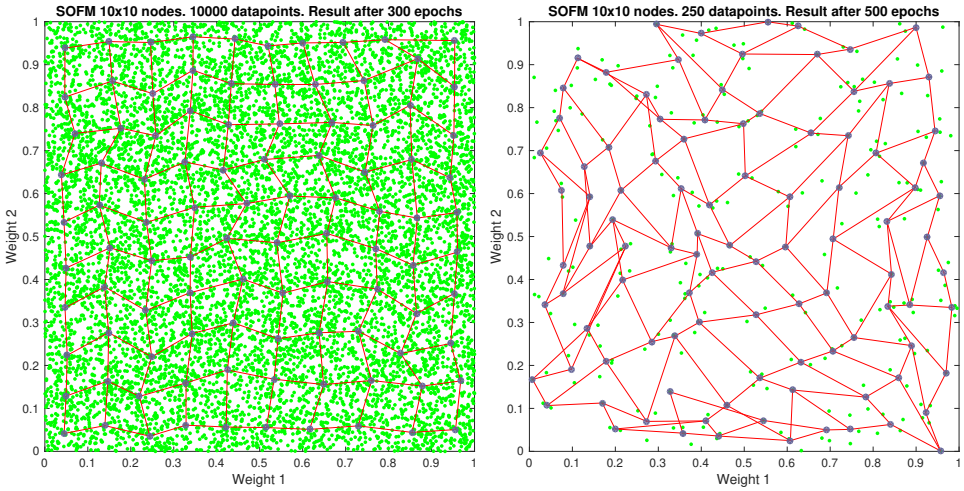

Figure 7.12: (left) Approximation of unit square to a 10x10 square grid. (right) Unit square now sampled with 250 data points, again 10x10 square grid.

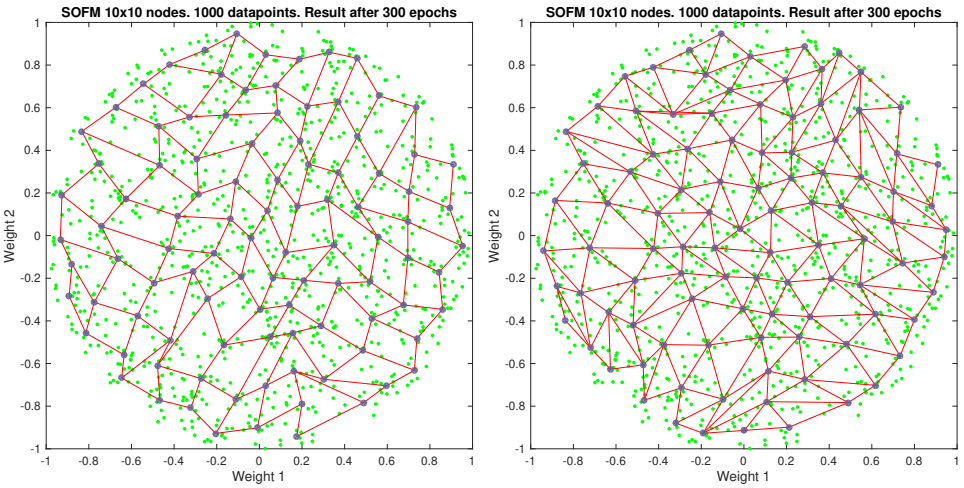

Figure 7.13: (left) Approximation of unit circle to a 10x10 square grid. (right) Approximation of unit circle to a 10x10 hexagonal grid. The unit circle is sampled with 1000 data points.

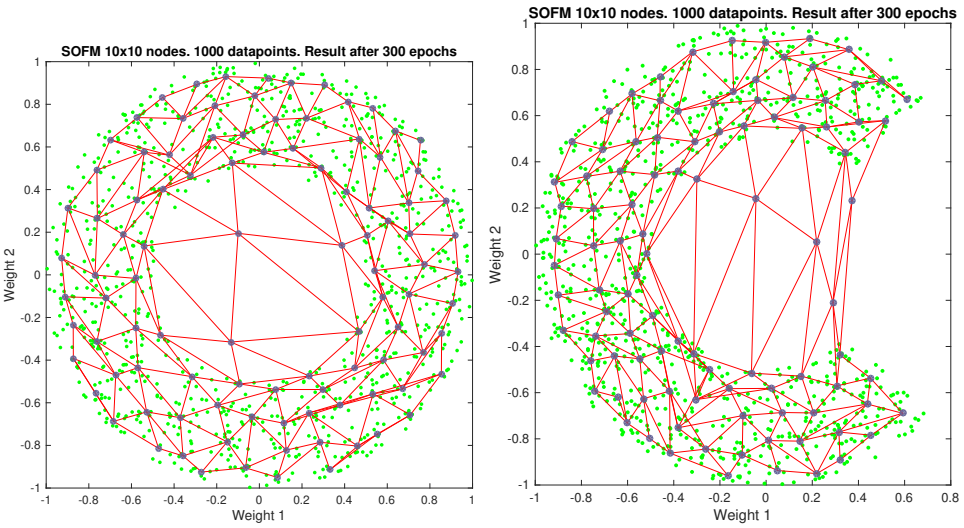

Figure 7.14: Parts of the unit circle approximated by a 10x10 hexagonal grid.

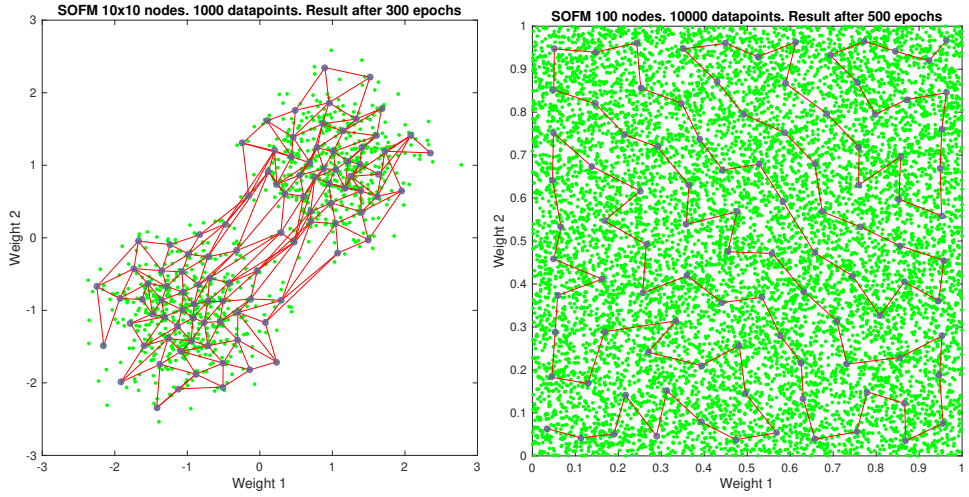

Figure 7.15: (left) Two normal distributed data sets approximated by a 10x10 hexagonal grid. (right) Unit square approximated by 100 nodes in a 1-dimensional SOFM.

7.5 Summary of self-organizing networks

PCA
Expand inputs

$$\mathbf{x}_n = \sum_j a_{nj}\mathbf{u}_j \qquad \text{with} \qquad R\mathbf{u}_j = \lambda_j \mathbf{u}_j$$

dimensionality reduction:

$$(x_{n1}, x_{n2}, ..., x_{nP}) \curvearrowright (a_{n1}, a_{n2}, ..., a_{nM})$$

with $M << P$.

Sanger network for extracting PCA eigenvectors

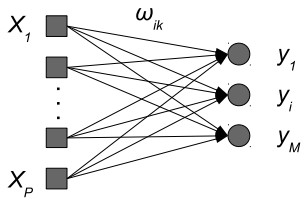

Figure 7.16: Sanger network.

$$y_i(n) = \sum_k^P \omega_{ik} x_{nk}$$

and

$$\omega_{ik} \to \omega_{ik} + \eta \left[y_i(\mathbf{x}_n) x_{nk} - y_i(\mathbf{x}_n) \sum_j^i \omega_{jk} y_j(\mathbf{x}_n) \right] \qquad i = 1, .., m$$

Autoencoder (auto-association)

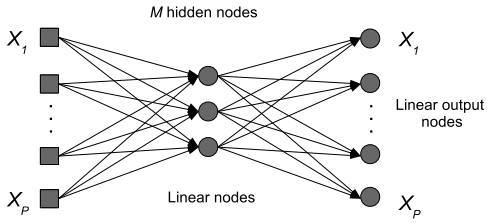

Figure 7.17: Autoencoder network.

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_{n}^{N}\sum_{i}^{M}\big(y_i(\mathbf{x}_n, \boldsymbol{\omega}) - x_{ni}\big)^2$$

$\boldsymbol{\omega}_j$ are the eigenvector of $\mathbf{R}$.

Competitive network for clustering

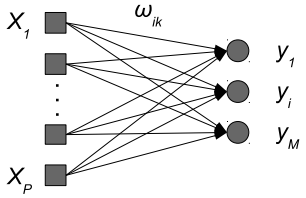

Figure 7.18: Competitive network.

Winning node:

$$j^* = \arg\max_j h(||\mathbf{x} - \boldsymbol{\omega}_j||) \tag{7.13}$$

Output value:

$$y_j = \begin{cases} 1 & \text{if } j \text{ is the winning node} \\ 0 & \text{otherwise} \end{cases} \tag{7.14}$$

Update rule:

$$\boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big)$$

LVQ
The LVQ learning rule:

$$\boldsymbol{\omega}_{j^*} \rightarrow \begin{cases} \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} + \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to the same class} \\ \boldsymbol{\omega}_{j^*} \rightarrow \boldsymbol{\omega}_{j^*} - \eta\big(\mathbf{x}_\mu - \boldsymbol{\omega}_{j^*}\big) & \text{if } j^* \text{ and } \mathbf{x}_\mu \text{ belongs to different classes} \end{cases}$$

SOFM

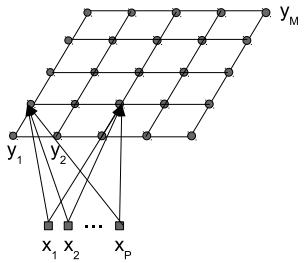

Figure 7.19: SOFM.

Winning node:

$$j^* = \arg\max_j h(||\mathbf{x} - \boldsymbol{\omega}_j||) \tag{7.15}$$

Output value:

$$y_j = \begin{cases} 1 & \text{if } j \text{ is the winning node} \\ 0 & \text{otherwise} \end{cases} \tag{7.16}$$

Update rule:

$$\boldsymbol{\omega}_j \rightarrow \boldsymbol{\omega}_j + \eta\Lambda_{jj^*}\big(\mathbf{x}_\mu - \boldsymbol{\omega}_j\big)$$

Example of Λ_{jj^*}:

$$\Lambda_{jj'} = e^{-d(j,j')^2/2\sigma^2}$$