Neural Networks
Lesson 4 - Unsupervised Learning

Prof. Michele Scarpiniti
INFOCOM Dpt. - “Sapienza” University of Rome

http://ispac.ing.uniroma1.it/scarpiniti/index.htm
michele.scarpiniti@uniroma1.it

Rome, 22 October 2009
Introduction

Unsupervised learning

Basis of Unsupervised Learning

Preprocessing

Self-organizing Map
Introduction

Basis of Unsupervised Learning
Unsupervised learning or blind signal processing is in opposition to the supervised learning where there exist a reference signal. A MIMO (multiple input multiple output) supervised example is depicted below: the aim is to develop a system minimizing a cost function based on the error computation.

- $d_j(n)$ is the $j$-th target signal;
- $x_j(n) = \mathcal{F}\{s(n)\} + v_j(n)$ is the $j$-th input signal, where $\mathcal{F}\{\cdot\}$ is the unknown system functional and $v_j(n)$ is the noise;
- $y_j(n)$ is the $j$-th reconstructed signal;
- $e_j(n) = d_j(n) - y_j(n)$ is the $j$-th error signal.
Unsupervised learning

An example of unsupervised system is depicted below: in this case the learning process is based on the optimization of a cost function $\mathcal{L}(x, y)$ evaluated on input and output of such a system. No a priori assumption on the environment are made.

Usually this cost function is based on statistical theory approach or information theoretic approach.

The evaluation of the parameters adaptation can be represented as

$$\Delta w = \frac{\partial \mathcal{L}(x, y)}{\partial w}$$
There appears to be something magical about blind signal processing; we are estimating the original source signals without knowing the parameters of the filtering processes. It is difficult to image that one can estimate this at all. In fact, without some a priori knowledge, one can usually estimate them up to certain indeterminacies. However, these indeterminacies in a great number of applications are not essential.

In order to eliminate indeterminacies, some a priori knowledge on the environment can be provided. In this case we use the term of semi-supervised learning or semi-blind learning.
Types of learning

Unsupervised learning is driven by several methodologies as seen previously. Other simpler learning rules are classified as follows:

1. **Hebbian learning**, when the weights correction $\Delta w_{ij}$ is proportional to the respective $i$-th input $x_i$ and the correspondence $j$-th output $y_j$

   $$\Delta w_{ij} = \mu x_i y_j$$

2. **Anti-Hebbian learning** when the weights correction $\Delta w_{ij}$ is anti-proportional to the respective $i$-th input $x_i$ and the correspondence $j$-th output $y_j$

   $$\Delta w_{ij} = -\mu x_i y_j$$

3. **Competitive learning** in this way only the active $w_{ij}$ are adapted, the inactive ones will be turned off $w_{ij} = 0$. 
Basis of Unsupervised Learning
Centering

The most basic and necessary preprocessing is to center $\mathbf{x}$, i.e. subtract its mean vector $\mathbf{m} = E\{\mathbf{x}\}$ so as to make $\mathbf{x}$ a zero-mean variable

$$\tilde{\mathbf{x}} = \mathbf{x} - \mathbf{m}$$

This preprocessing is made solely to simplify algorithms: it does not mean that the mean could not be estimated. After estimating the unknown parameters with centered data, we can complete the estimation by adding the mean vector $\mathbf{m}$ back to the centered estimated parameters.
More generally, normalization can be done with respect to the mean and variance of the feature values or with respect to the amplitude of the data. In the former, instances are normalized such that the normalized data have zero mean and unitary variance.

\[ x'_i = \frac{x_i - \bar{x}_i}{\sigma_i} \]

where \( x_i \) indicates the \( i \)-th feature of original data instance, \( \bar{x}_i \) is the mean value, and \( \sigma_i \) is the standard deviation of \( u_i \).

In the case of data normalization, instances are simply divided by the max entry value so that all feature values are restricted to a \([-1, 1]\) range. Moreover, if great magnitude differences exist between the individual feature values, a logarithmic transformation, for example, can be used to reduce the dynamic range of the feature values.
Outliers reduction

Outliers is a term used in statistics to define an anomalous or aberrant value, in a set of observations. The outliers values are numerically distant from the rest of the data collected (for example, by sampling process), so that the statistics derived from samples containing outliers may be misleading. Preprocessing should also include outliers removal when possible. A simple way, is to assume Gaussian or near-Gaussian distribution model for the data defined as

\[
p_x(x) = \frac{1}{(2\pi)^p \sqrt{\Delta K_x}} e^{-\frac{(x-\bar{x})^T K_x^{-1}(x-\bar{x})}{2}}
\]

where \( K_x \) is the covariance matrix and \( \Delta K_x \) its determinant. In this case, is possible to use the Mahalanobis distance, defined as the exponent of the multivariate Gaussian distribution

\[
M = (x - \bar{x})^T K_x^{-1} (x - \bar{x})
\]

If this distance \( M \) is larger than a given threshold, usually some multiple of the standard deviation of the data, that instance can be considered as an outliers.
Further preprocessing

If the data consists of time-signals, some band-pass filtering may be very useful. If we filter linearly the observed signals $x_i(t)$ we obtain new signals, say $x^*_i(t)$. Now, time filtering of $x$ corresponds to multiplying $x$ from the right by a matrix, let us call it $M$. This gives

$$x^* = Mx$$

More in general the pre-filtering can be made nonlinearly

$$x^* = M(x)$$
In several applications, such as classification, it is important to perform a feature extraction from a $N$-dimensional space to an $M$-dimensional one, with $M \leq N$.

A classical approach to feature selection is the Principal Component Analysis (PCA), known also as Karhunen-Loève Transform (KLT) or Hotteling Transform (HT).

The PCA highlights the statistical regularities of the data by projecting the input space, originally of size $N$, on a space of size $M$, with $M \leq N$, called the output space, and making a reduction of the dimensionality by maintaining, almost unchanged, the useful information content.
Principal Component Analysis (PCA)

The Principal Component Analysis (PCA) aims to find an optimal linear transformation $y = Wx$, determining the maximal variance directions, as shown in the following figure.

In other words, the PCA transforms a large number of input data into a set of components statistically uncorrelated (features) and sorted according to their information content.
Principal Component Analysis (PCA)

Given \( \mathbf{x} \in \mathbb{R}^N \), with \( \mathbf{x}(t) = [x_1(t), x_2(t), \ldots, x_N(t)]^T \), probability density function (pdf) \( p_x(\mathbf{x}) \) and autocorrelation matrix \( \mathbf{R}_{xx} = E \{ \mathbf{x} \mathbf{x}^T \} \), let us pose \( \mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N \) the eigenvectors of the autocorrelation matrix \( \mathbf{R}_{xx} \), so that \( \mathbf{w}_1 \) corresponds to the biggest eigenvalue \( \lambda_1 \) and so on until \( \mathbf{w}_N \) corresponding to the smallest eigenvalue \( \lambda_N \). Then:

**Definition**

The \( k \)-th **principal component** of the input vector \( \mathbf{x} \) is equal to the normalized eigenvector \( \mathbf{w}_k \) corresponding to the \( k \)-th eigenvalue \( \lambda_k \) of the correlation matrix \( \mathbf{R}_{xx} \), when eigenvalues are in descending order. That is the following equation

\[
\mathbf{R}_{xx} \mathbf{w}_k = \lambda_k \mathbf{w}_k, \quad \text{with} \quad k = 1, 2, \ldots, N;
\]

is satisfied by \( \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_N \geq 0 \) and \( \| \mathbf{w}_k \|^2 = 1 \).
Principal Component Analysis (PCA)

Fundamental is the following

**Theorem**

The $k$-th **principal direction** of the input vector $\mathbf{x}$ given by the normalized eigenvector $\mathbf{w}_k$, is such that the variance has a maximum value towards that direction.

The autocorrelation matrix $\mathbf{R}_{xx}$ can be decomposed (EVD), as

$$
\mathbf{R}_{xx} = \mathbf{W}^T \mathbf{\Lambda} \mathbf{W}
$$

where $\mathbf{\Lambda} = \\
\begin{bmatrix}
\lambda_1 & 0 & \cdots & 0 \\
0 & \lambda_2 & & \vdots \\
\vdots & \ddots & \ddots & 0 \\
0 & \cdots & 0 & \lambda_N \\
\end{bmatrix}
$ and $\mathbf{W}$ is the matrix containing the eigenvectors $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N]^T$. 
**Principal Component Analysis (PCA)**

The matrix $\mathbf{W}$ is an orthogonal matrix: $\mathbf{WW}^T = \mathbf{W}^T\mathbf{W} = \mathbf{I}$, that is

$$w_i^T w_j = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

Then follows that

$$w_i^T \mathbf{R}_{xx} w_j = \begin{cases} \lambda_i & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}$$

Using this relation and given $\mathbf{y} = \mathbf{Wx}$, we can say that

$$\mathbf{R}_{yy} = \mathbf{WR}_{xx} \mathbf{W}^T = \mathbf{\Lambda}$$

The output covariance matrix $\mathbf{R}_{yy}$ is diagonal with components in descending order. This implies that the PCA **linearly uncorrelates** the outputs. The variance of each component of the output covariance matrix corresponds to the input eigenvalue. The variance of the $N$ components is

$$\sum_{j=1}^{N} \sigma_j^2 = \sum_{j=1}^{N} \lambda_j$$

where $\sigma_j^2$ is the variance of the $j$-th principal component $y_j$. 
Principal Component Analysis (PCA): dimensional reduction

The projection of the input vector $\mathbf{x}$ on the unitary vector $\mathbf{w}_j$ identifies the $j$-th principal direction

$$y_j = \mathbf{w}_j^T \mathbf{x}$$

This relation is known as the analysis formula. In order to reconstruct the original vector $\mathbf{x}$ from projections $y_j$, we can

1. combine the projections $y_j$ into a vector $\mathbf{a}$

   $$\mathbf{a} = [a_1, a_2, \ldots, a_N] = [\mathbf{w}_1^T \mathbf{x}, \mathbf{w}_2^T \mathbf{x}, \ldots, \mathbf{w}_N^T \mathbf{x}] = \mathbf{W}^T \mathbf{x}$$

2. multiply this vector for the orthogonal matrix $\mathbf{W}$

   $$\mathbf{x} = \mathbf{Wy} = \sum_{j=1}^{N} \mathbf{w}_j y_j$$

This relation is known as the synthesis formula.
Principal Component Analysis (PCA): dimensional reduction

In many applications there is the need of a dimensional reduction (called also subspace decomposition). For this purpose, let us pose $\lambda_1, \lambda_2, \ldots, \lambda_M$ the first (greater) $M$ eigenvalues of the autocorrelation matrix $R_{xx}$, then we can truncate to $M$ the expansion

$$x' = \sum_{j=1}^{M} \mathbf{w}_j y_j,$$

with $M < N$

The approximation error $\mathbf{e}$ can be defined as $\mathbf{e} = \mathbf{x} - \mathbf{x}'$, and it is evaluated as

$$\mathbf{e} = \sum_{j=M+1}^{N} \mathbf{w}_j y_j$$

The orthogonality principle is valid in this case too

$$\mathbf{e}^T \cdot \mathbf{x}' = \sum_{i=M+1}^{N} \mathbf{w}_i y_i \sum_{j=1}^{M} \mathbf{w}_j y_j = \sum_{i=M+1}^{N} \sum_{j=1}^{M} \mathbf{w}_i \mathbf{w}_j y_i y_j = 0$$
As an example we report an handwritten number representation. In the leftmost column, some normalized $32 \times 32$ digit images. Such images, when scanned row by row, can be represented as 1024-dimensional vectors. In the second column, mean of the training set and in the remaining columns, reproductions by adding projections in the principal directions up to dimension 1, 2, 5, 16, 32, and 64. It can be noted that there is no evident difference between the last two columns.
Linear Discriminant Analysis (LDA)

The PCA provides the best representation of the data in a lower dimensional space. Its main drawback, however, is that no guarantee exists that the classes in the transformed space are better separated than they are in the original space. PCA, in fact, does not take class information into consideration. Ensuring that the classes are best separated in the transformed space is better handled by the linear discriminant analysis (LDA). The transformation, for the projection from the original feature space onto a lower dimensional feature space is defined as

\[ y = Q^T x \]

The determination of the \( Q \) matrix can be no very simple. The figure shows the axes transformation obtained with the LDA. It can be noted that in this new projection coordinates, the “square” and “circle” features are separable.
Factor analysis (FA)

Factor analysis is a statistical method used to describe variability among observed variables in terms of fewer unobserved variables called factors. The observed variables \( y = [y_1, y_2, \ldots, y_N]^T \) are modeled as linear combinations of the factors, plus “error” terms.

\[
y = \Lambda x + e
\]

where \( x \) is a \( K \)-dimensional zero-mean unit-variance multivariate Gaussian vector with elements corresponding to hidden (or latent) factors, \( \Lambda \) is a \( D \times K \) matrix of parameters, known as the factor loading matrix, and \( e \) is a \( N \)-dimensional zero-mean multivariate Gaussian noise vector with diagonal covariance matrix \( \Psi \). Defining the parameters of the model to be \( \theta = (\Psi, \Lambda) \), by integrating out the factors, one can readily derive that

\[
p_y(y | \theta) = \int_{-\infty}^{\infty} p_x(x | \theta)p_x(x | x, \theta) \, dx = \mathcal{N}(0, \Lambda\Lambda^T + \Psi)
\]

where \( \mathcal{N}(\mu, \Sigma) \) refers to a multivariate Gaussian density with mean \( \mu \) and covariance matrix \( \Sigma \).
Factor analysis (FA)

- **Factor analysis** is an interesting model for several reasons. If the data is very high dimensional ($N$ is large) then even a simple model like the full-covariance multivariate Gaussian will have too many parameters to reliably estimate or infer from the data.

- By choosing $K < N$, factor analysis makes it possible to model a Gaussian density for high dimensional data without requiring too many parameters. Moreover, given a new data point, one can compute the posterior over the hidden factors, $p_x(x|y, \theta)$; since $x$ is lower dimensional than $y$ this provides a low-dimensional representation of the data.

- **Factor analysis** is related to principal component analysis (PCA) but not identical. Because PCA performs a variance-maximizing rotation of the variable space, it takes into account all variability in the variables. In contrast, factor analysis estimates how much of the variability is due to common factors ("communality").
The **EM algorithm** is an algorithm for estimating ML parameters of a model with latent variables. Consider a model with observed variables \( y \), hidden/latent variables \( x \), and parameters \( \theta \). We can lower bound the log likelihood for any data point as follows

\[
L(\theta) = \ln p_y(y | \theta) = \ln \int_{-\infty}^{\infty} p_x(x, y | \theta) \, dx =
\]

\[
= \ln \int_{-\infty}^{\infty} q_x(x) \frac{p_x(x, y | \theta)}{q_x(x)} \, dx =
\]

\[
\geq \int_{-\infty}^{\infty} q_x(x) \ln \frac{p_x(x, y | \theta)}{q_x(x)} \, dx \overset{\text{def}}{=} F(q, \theta)
\]

where \( q_x(x) \) is some arbitrary density over the hidden variables, and the lower bound holds due to the concavity of the log function (this inequality is known as **Jensen’s inequality**).

The lower bound \( F \) is a functional of both the density \( q_x(x) \) and the model parameters \( \theta \). For a data set of \( N \) data points \( y_1, \ldots, y_N \) this lower bound is formed for the log likelihood term corresponding to each data point, thus there is a separate density \( q_x^{(n)}(x) \) for each point and \( F(q, \theta) = \sum_n F^{(n)}(q^{(n)}, \theta) \).
Expectation-maximization algorithm (EM)

The basic idea of the Expectation-Maximization (EM) algorithm is to iterate between optimizing this lower bound as a function of $q$ and as a function of $\theta$. We can prove that this will never decrease the log likelihood. After initializing the parameters somehow, the $k$-th iteration of the algorithm consists of the following two steps:

1. **E-step**: optimize $F$ with respect to the distribution $q$ while holding the parameters fixed

   $$q_k(x) = \arg \max_{q_x(x)} F(q, \theta)$$

2. **M-step**: optimize $F$ with respect to the parameters $\theta$ while holding the distribution over hidden variables fixed

   $$\theta_k = \arg \max_{\theta} F(q, \theta)$$
Factor Graphs (FG)

Many applications that involve inference and learning in signal processing, communication and artificial intelligence can be cast into a graph framework. Factor graphs are a type of network that can be studied and solved by propagating belief messages with the sum/product algorithm.

In a factor graph (FG) there are two kinds of nodes, variable nodes and factor nodes (called also as function nodes), usually denoted as open circles and filled dots. Like an undirected model, the factor graph represents a factorization of the joint probability distribution: each factor is a non-negative function of the variables connected to the corresponding factor node.

Thus for the following factor graph we have:

$$p_x (x_1, x_2, x_3) = f_1 (x_1) f_2 (x_1, x_2) f_3 (x_1, x_2) f_4 (x_2, x_3)$$

A main application of factor graphs are probabilistic models.
A self-organizing map (SOM) or self-organizing feature map (SOFM), commonly also known as Kohonen network, is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional, discretized representation of the input space of the training samples, called a map. Self-organizing maps are different than other artificial neural networks in the sense that they use a neighborhood function to preserve the topological properties of the input space.

- Like most artificial neural networks, SOMs operate in two modes: training and mapping.

  1. **Training** builds the map using input examples. It is a competitive process, also called vector quantization.
  2. **Mapping** automatically classifies a new input vector.

- A self-organizing map consists of components called nodes or neurons. Associated with each node is a weight vector of the same dimension as the input data vectors and a position in the map space. The usual arrangement of nodes is a regular spacing in a hexagonal or rectangular grid. The self-organizing map describes a mapping from a higher dimensional input space to a lower dimensional map space. The procedure for placing a vector from data space onto the map is to find the node with the closest weight vector to the vector taken from data space and to assign the map coordinates of this node to our vector.
Self-organizing map (SOM)

- The goal of learning in the self-organizing map is to cause different parts of the network to respond similarly to certain input patterns. This is partly motivated by how visual, auditory or other sensory information is handled in separate parts of the cerebral cortex in the human brain.

- The weights of the neurons are initialized either to small random values or sampled evenly from the subspace spanned by the two largest principal component eigenvectors. With the latter alternative, learning is much faster because the initial weights already give good approximation of SOM weights.

- The network must be fed a large number of example vectors that represent, as close as possible, the kinds of vectors expected during mapping. The examples are usually administered several times.
Self-organizing map (SOM)

The training utilizes competitive learning. When a training example is fed to the network, its Euclidean distance to all weight vectors is computed. The neuron with weight vector most similar to the input is called the best matching unit (BMU). The weights of the BMU and neurons close to it in the SOM lattice are adjusted towards the input vector. The magnitude of the change decreases with time and with distance from the BMU. The update formula for a neuron with weight vector \( w_n \) is

\[
    w_{n+1} = w_n + \mu(n) \Theta(v, n) (x_n - w_n)
\]

where \( \mu(n) \) is a monotonically decreasing learning coefficient and \( x_n \) is the input vector. The neighborhood function \( \Theta(v, n) \) depends on the lattice distance between the BMU and neuron \( v \). In the simplest form it is one for all neurons close enough to BMU and zero for others, but a Gaussian function is a common choice, too.
Self-organizing map (SOM)

- At the beginning when the neighborhood is broad, the self-organizing takes place on the global scale. When the neighborhood has shrunk to just a couple of neurons the weights are converging to local estimates.

- This process is repeated for each input vector for a (usually large) number of cycles $N$.

- During mapping, there will be one single winning neuron: the neuron whose weight vector lies closest to the input vector.

Algorithm

1. Randomize the map’s nodes’ weight vectors;
2. Grab an input vector;
3. Traverse each node in the map
   1. Use Euclidean distance to find similarity between the input vector and the map’s node’s weight vector;
   2. Track the node that produces the smallest distance (this node is the best matching unit, BMU);
4. Update the nodes in the neighborhood of BMU by pulling them closer to the input vector in eq. (1);
5. Increment $n$ and repeat from 2 while $n < N$. 
References

S. Haykin.
Neural Networks, a comprehensive foundation.

I.T. Jolliffe.
Principal Component Analysis.

H.-A. Loeliger.
An Introduction to Factor Graphs.

T. Kohonen.
Self-Organizing Maps.

T. Kohonen.
The Self-Organizing Map.

T. Kohonen.
Engineering Applications of the Self-Organizing Map.