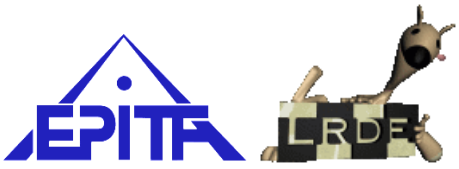
Neural Networks: Multi-Layer Perceptron and Hopfield Network

CSI 2004

Sylvain Berlemont, Nicolas Burrus, David Lesage, Francis Maes, Jean-Baptiste Mouret, Benoît Perrot, Maxime Rey, Nicolas Tisserand, Astrid Wang

The ability of neural networks to derive meaning from complicated or imprecise data make them a powerful tool to extract hidden correlations between patterns or to recognize noised patterns. This report is dedicated to the study of Multi-Layer Perceptrons (MLP) and Hopfield networks. In particular, two applications are detailed. MLP possibilities are illustrated through an image compression software and Hopfield networks are studied through a character recognizer. For both applications, theoretical principles, heuristic and algorithmic improvements are discussed thanks to various experiments.

Keywords
Multi-layer perceptron, Hopfield network, compression, pattern recognition



Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@epita.fr – http://www.lrde.epita.fr

Copying this document

Contents

Introduction

Inspired by animal brains, artificial neural networks simulate their biological equivalent. Through a set of mathematical operations, the behavior of a biological neuron is modeled: input signals are weighted, summed and thresholded to obtain an output signal. These output signals then become inputs for other neurons, creating a network. By processing a database of examples, neural networks may adapt themselves to recognize patterns or to classify data. Their ability to derive meaning from complicated or imprecise data make them a powerful tool to extract hidden correlations between patterns or to recognize noised patterns.

The aim of this study is to evaluate the potential of Multi-Layer Perceptrons (MLP) and Hopfield neural networks. To illustrate the different paradigms and algorithms manipulated, a concrete application was built for each network kind. We applied MLP to an image compression software, and Hopfield networks to an automatic character recognizer. Both applications were developed in CamL. The theoretical parts have been compiled from the neural networks class of EPITA, the books *Neural Networks, A Comprehensive Foundation* (13), *Neural Networks for Pattern Recognition* (4), *Réseaux de neurones, Méthodologie et applications* (8) and research papers that can be found in the bibliography.

The analysis of these two kinds of neural networks was driven in an experimental manner. When a new algorithm was studied then implemented, observations were made on its performance and the accuracy of learning and generalization that it leaded to. Theory was then confirmed by experimental results.

The first part of this report focuses on Multi-Layer Perceptrons. Various learning algorithms and improvement techniques are studied and the corresponding experimental results are analyzed. In particular, MLP application to image compression is detailed.

The second part presents a study of Hopfield networks applied to a character recognizer. Different learning rules are exposed a theoretical way and then discussed through experimental results.

Part I

Multi-layer Perceptron

Notations

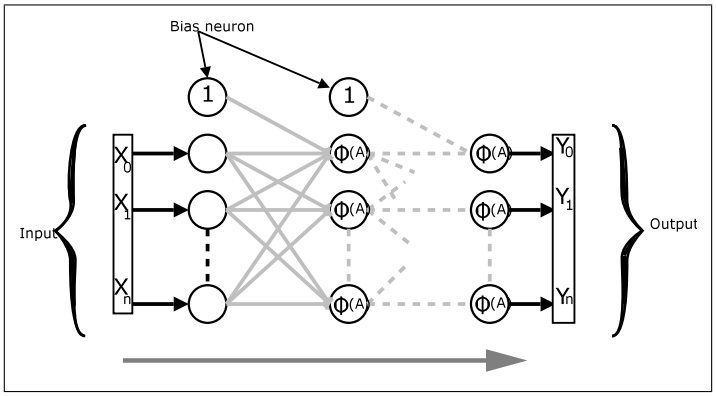


Figure 1: Typical topology of a multilayer peceptron.

n	number of layers
$nb(i)$	number of neurons on the layer i
	layers are numbered from 0 to $n-1$
$w_{c,i,j}$	weight of the connection between the neuron j of layer $c-1$
	towards the neuron i of the layer c
w_k	k-th row of the matrix $\mathbf{W}$
$N_{c,i}$	neuron i of layer c
$A_{c,i}$	activation of neuron $N_{c,i}$
$S_{c,i}$	output of $N_{c,i}$
D_i	desired output of neuron $N_{n-1,i}$
Y_i	output of $N_{n-1,i}$; $Y_i = S_{n-1,i}$
X_i	input i of the network. $X_i = S_{0,i}$
$\varphi_{c,i}(x)$	activation function of $N_{c,i}$
N	number of record in the database

For a neuron $N_{c,i}$, we have:

$$S_{c,i} = \varphi_{c,i}(A_{c,i}) \quad (1)$$

$$= \varphi_{c,i}\left(\sum_{j=1}^{nb(c-1)} w_{c,i,j} S_{c-1,j}\right) \quad (2)$$

Chapter 1

Useful numerical algorithms

These algorithms will be useful in the next part (Chapter 3) to speed up the compression process. They will be mainly used to initialize the weights of the neural network in a good configuration. This chapter aims at introducing their basic principles and analyzing their performance.

1.1 Principal Components Analysis (PCA)

Let $\mathbf{x}$ denote the n-dimensional input data vector. Without loss of generality, we can assume that $E[\mathbf{x}] = 0$. The aim of PCA is to find a set of m orthonormal vectors in an n-dimensional data space such that they will account for as much as possible of the variance of the data. Figure 1.1 shows examples of these vectors for a two-dimensional dataset.

Let $\mathbf{C} = E[\mathbf{x} \cdot \mathbf{x}^t]$ denote the data covariance matrix, it can be shown that the m orthonormal vectors are the m eigenvectors associated with the m largest eigenvalues of the matrix $\mathbf{C}$. Therefore the following algorithms are designed to find the eigenvectors of the matrix $\mathbf{C}$. The matrix $\mathbf{C}$ is symmetric and most methods will use this property, consequently it will be assumed that the input matrix of these algorithm is symmetric.

1.1.1 Power and deflation methods

Power method

The power method can be used to find the main eigenvalue and its associated eigenvector. It is often used in the deflation algorithm. Suppose that the eigenvalues of $\mathbf{C}$ are such that :

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n| \tag{1.1}$$

Let $\mathbf{v}$ a random vector, let us define the following reccurence:

$$\begin{cases} \mathbf{u}_{(k+1)} = \frac{\mathbf{C}\mathbf{u}_{(k)}}{<\mathbf{C}u_{(k)}, \mathbf{v}>} \\ l_{(k)} =< \mathbf{C}u_{(k)}, \mathbf{v} > \end{cases} \tag{1.2}$$

$$\text{until } |l_{(k)} - l_{(k-1)}| < \varepsilon$$

where $k = 0, \cdots$ and $u_{(0)}$ denotes a random vector.

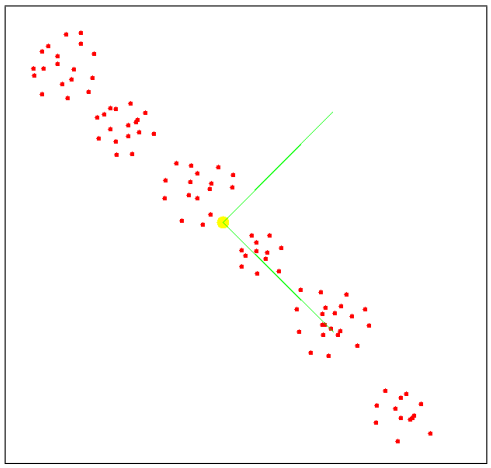


Figure 1.1: Two-dimensional dataset and associated eigenvectors.

It can be proved that:

$$\lambda_1 = \lim_{k \to \infty} l_{(k)} \tag{1.3}$$

$$\mathbf{x_1} = \lim_{k \to \infty} u_{(k)} \tag{1.4}$$

where x_1 is the normalized eigenvector associated with λ_1.

Deflation

Let us suppose that λ_1 and $\mathbf{x}_1$ have been computed before. The matrix $\mathbf{C}$ can be transformed into a new matrix $\mathbf{C}_{(1)}$ with the same eigenvalues that $\mathbf{C}$ excepted that λ_1 is replaced by a null value. The power method is the applied on the new matrix $\mathbf{C}_{(1)}$.

Let $\mathbf{w}$ such that $\|\mathbf{w}\| = 1$.

$$\mathbf{C}_{(1)} = \mathbf{C} - \lambda_1 \mathbf{x}_1 \mathbf{w}^t \tag{1.5}$$

Using the power method, we compute λ_2 and $\mathbf{v}_2$, where $\mathbf{v}_2$ is an eigenvector of $\mathbf{C}_{(1)}$. $\mathbf{x}_2$ can be computed as follows:

$$\mathbf{x}_2 = \mathbf{v}_2 + \frac{\lambda_1}{\lambda_2 - \lambda_1} < \mathbf{w}, \mathbf{v}_2 > \mathbf{x}_1 \tag{1.6}$$

This iteration can be repeated to find all the eigenvalues by computing $\mathbf{C}_{(n)}$ using λ_n and $\mathbf{x}_n$.

1.1.2 Neuron-based algorithms

Following (12) and (6), we chose to focus mainly on the Adaptive Learning Algorithm for Principal Component Analysis (ALA), which is described as both powerful and easy to implement. Another known algorithm is the Adaptive Principal component Extractor (APEX).

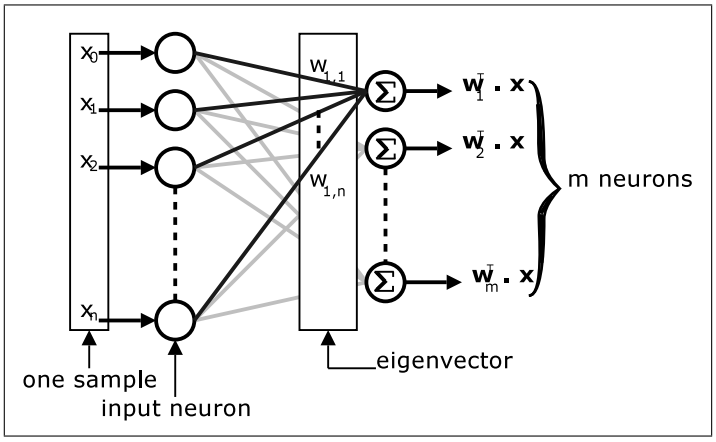


Figure 1.2: Network topology of the network used by GHA.

GHA

Oja (17) proposed a one-unit learning rule to find the first principal component direction vector, *i.e.* the first eigenvector of $\mathbf{C}$:

$$\Delta \mathbf{w}(t) = \eta(t)V(t)(\mathbf{x}(t) - V(t)\mathbf{w}(t)) \tag{1.7}$$

where $\eta(t)$ is the learning rate parameter, and $V(T)$ denote the output of a linear neuron such that:

$$V(t) = \mathbf{w}(t)^t \mathbf{x}(t)$$

Assuming that η is sufficiently small, Oja proved that the weight vector $\mathbf{w}(t)$ will asymptotically converge to the first normalized eigenvector of $\mathbf{C}$.

To find more than one eigenvector, Sanger proposed the Generalized Hebbian Algorithm (GHA) (22), which use a network of linear neurons instead of one linear neuron (figure 1.2). Let us denote by $\mathbf{W}$ the weight matrix and by $\mathbf{w}_i$ the i-th line of this matrix, the GHA is based on the following learning rule:

$$\Delta \mathbf{w_i}(t) = \eta(t)V_i(t)\left(\mathbf{x}(t) - \sum_{j=1}^{i} V_j(t)\mathbf{w_j}(t)\right),$$
$$j = 1, 2, \cdots, m \tag{1.8}$$

where $V_i(t) = \mathbf{w_i}(t)^t \mathbf{x}(t)$. $\mathbf{w}_i(t)$ converge to the first m eigen vector, *i.e.* the vectors associated with the i-th largest eigenvalue λ_i of the correlation matrix $\mathbf{C}$.

ALA for PCA

The learning process of the GHA can converge very slowly if eigenvalues are small, it can diverge if eigenvalues are large. Therefore selecting a good value for the learning rate η can be a difficult task. To address this problem, Chen and Chang (7) introduced an adaptive algorithm (ALA) for PCA. This algorithm uses the value of each eigenvalue to compute a good learning rate, consequently each eigenvector is associated with a different learning rate. This algorithm can be summarized as follows:

Step 1 Set weight vector $\mathbf{w_i}(0) \in R^n$ such that $\|\mathbf{w_i}\|^2 \ll 1/2^2$ and estimate of the eigenvalues $\hat{\lambda}_i = \delta$, for $i = 1, 2, \cdots, m$, where δ denotes a small positive number.

Step 2 Use the network to evaluate the output V_i for a random sample $\mathbf{x}(t)$:

$$V_i(t) = \mathbf{w_i}^t(t)\mathbf{x}(t), i \in 1, 2, \cdots, m \tag{1.9}$$

Step 3 Estimate the eigenvalues λ_i:

$$\hat{\lambda}_i(t) = \hat{\lambda}_i(t-1) + \gamma(t)\left(\frac{\mathbf{w_i}^t(t)\mathbf{x_i}(t)}{\|\mathbf{w_i}(t)\|^2} - \hat{\lambda}_i(t-1)\right) \tag{1.10}$$

where $\mathbf{x_i}(t) = \mathbf{x}(t) - \sum_{j=1}^{i-1} V_j(t)\mathbf{w_j}(t)$ and $\gamma(t)$ denote a value smaller than one an decreased to zero as t increases.

Step 4 Modify the weights $\mathbf{w}_i$. Let $\eta_i(t) = \beta_i(t)/\hat{\lambda}_i(t)$, where $\beta_i(t)$ is set to be smaller than $2(\sqrt{2}-1)$ and decreased to zero as t approaches ∞.

$$\mathbf{w_i}(t+1) = \mathbf{w_i}(t) + \eta_i(t)V_i(t)\left(\mathbf{x}(t) - \sum_{j=1}^{i} V_j(t)\mathbf{w_j}(t)\right) \tag{1.11}$$

Step 5 Check the length of $\mathbf{w}_i$ according to the rule :

$$\mathbf{w_i}(t+1) = \begin{cases} \frac{1}{\sqrt{2}}\frac{\mathbf{w_i}(t+1)}{\|\mathbf{w_i}(t+1)\|}, & \text{if } \|\mathbf{w_i}(t+1)\|^2 > \frac{1}{\beta_i(t+1)} + \frac{1}{2} \\ \mathbf{w_i}(t+1), & \text{otherwise} \end{cases} \tag{1.12}$$

This normalization process is required because the estimates of λ_i may be inaccurate during the initial period of the learning process.

Step 6 Go back to Step 2 until all the $\mathbf{w_i}$ are mutually orthonormal, *i.e.* until :

$$\sum_{i=0}^{m}\sum_{j=0}^{m} \mathbf{w}_i(t)^t \cdot \mathbf{w}_j(t) = 0 \tag{1.13}$$

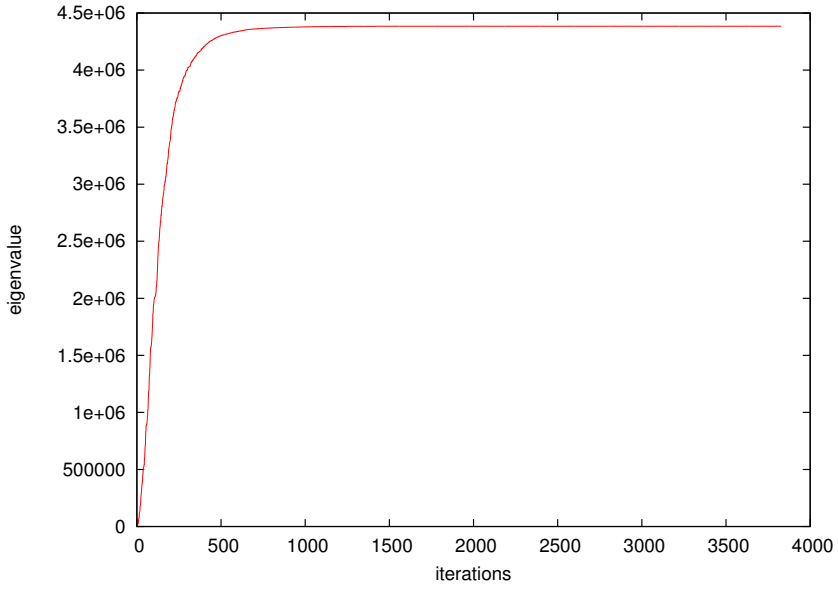


Figure 1.3: Time histories of the first eigenvalue λ_1 of data plotted on figure 1.1 during the execution of the ALA algorithm

1.1.3 QL/QR algorithm

One of most efficient known technique for finding eigenvalues and eigenvectors (20) of a symmetric matrix is the combination of the Householder reduction, which reduces a symmetric real matrix to a tridiagonal form, followed by the so called QR or a QL algorithm that can diagonalize a tridiagonal matrix within about $30n^2$ steps without eigenvectors. If eigenvectors are required then the number of operations grows to $3n^3$.

The QR or a QL method is an iterative method. But the orthogonal transformation employed preserves symmetry and tridiagonal form. So zeros stay zeroed. And this means that there are only $N-2$ off-diagonal elements to kill.

The Householder reduction on the other hand is a finite procedure, *i.e.*, not an iterative one at all. A symmetric matrix can be reduced to a tridiagonal form within a finite well defined number of steps: $N-2$ orthogonal transformations, where N denote the sizeof the matrix.

Proof and details about these methods can be found in (20). We only present here their basic concepts.

Householder method

Each transformation used in the Householder method annihilates the required part of a whole column and whole corresponding row. The basic ingredient is a Householder matrix $\mathbf{P}$ which has the form:

$$\mathbf{P} = \mathbf{1} - 2\mathbf{w}\cdot\mathbf{w}^t \tag{1.14}$$

where $\mathbf{w}$ is a real vector with $\|\mathbf{w}\|^2 = 1$ and $\mathbf{1}$ is the identity matrix. It can be proved that $\mathbf{P}$ is orthogonal. We can use any vector $\mathbf{u}$ in place of $\mathbf{w}$ if we normalize it at the same time:

$$\mathbf{P} = \mathbf{1} - \frac{2\mathbf{u}\cdot\mathbf{u}^t}{\mathbf{u}\cdot\mathbf{u}^t} = \mathbf{1} - \frac{\mathbf{u}\cdot\mathbf{u}^t}{H} \tag{1.15}$$

where $H = \frac{1}{2}\mathbf{u}\cdot\mathbf{u}^t$.

Suppose $\mathbf{x}$ is the vector composed of the first column of $\mathbf{C}$. Let us denote by $\mathbf{e}_1$ the unit vector $[1;0;\cdots;0]^t$, choosing $\mathbf{u}$ as $\mathbf{u} = \mathbf{x} \pm |\mathbf{x}|\mathbf{e}_1$ leads to:

$$\mathbf{P}\cdot\mathbf{x} = \pm|\mathbf{x}|\mathbf{e}_1 \tag{1.16}$$

This shows that the Householder matrix $\mathbf{P}$ acts on a given vector $\mathbf{x}$ to zero all its elements except the first one.

To reduce a symmetric matrix $\mathbf{C}$ to tridiagonal for, we choose the vector $\mathbf{x}$ for the first Householder matrix to be the lower $n-1$ elements of the first column. Then the lower $n-2$ elements will be zeroed. Thus the first Householder operator $\mathbf{P}_1$ is selected to rotate the sub-column of the first column:

$$\begin{pmatrix} c_{21} \\ c_{31} \\ \vdots \\ c_{n1} \end{pmatrix} \quad \text{onto} \quad \begin{pmatrix} c'_{21} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{1.17}$$

where the quantity c'_{21} is the magnitude of the vector $[c_{21},\cdots,c_{n1}]^t$. To accomplish that, the operator has to have the following form:

$$\mathbf{P}_1 = \left(\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \mathbf{P}_{(n-1)} & \\ 0 & & & \end{array}\right) \tag{1.18}$$

where $\mathbf{P}_{(n-1)}$ denotes a Householder matrix with dimensions $(n-1) \times (n-1)$.

By multiplying $\mathbf{C}$ by $\mathbf{P_1}$ from the left and the right, we get :

$$\mathbf{P_1} \cdot \mathbf{C} \cdot \mathbf{P_1} = \left(\begin{array}{c|ccc} c_{11} & c'_{12} & 0 & \cdots & 0 \\ \hline c'_{21} & & & & \\ 0 & & & & \\ \vdots & & & & \\ 0 & & & & \end{array}\right) \tag{1.19}$$

The second Householder matrix is going to look as follows:

$$\mathbf{P_2} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & P_{2_{(n-1)}} & & \\ 0 & 0 & & & \end{pmatrix} \tag{1.20}$$

In $n-2$ such steps the whole matrix will become triadiagonalized. (20) uses a more computationally-friendly formula that will not be detailed here.

QL/QR algorithm

The basic idea behind the QL algorithm is that any real matrix can be decomposed in the form:

$$\mathbf{C} = \mathbf{Q} \cdot \mathbf{L} \tag{1.21}$$

where $\mathbf{Q}$ is orthogonal and $\mathbf{L}$ is lower triangular. For a general matrix, the decomposition is constructed by applying Householder transformations to annihilate successive columns of $\mathbf{C}$ below the diagonal.

Let $\mathbf{C}' = \mathbf{L} \cdot \mathbf{Q}$. Since $\mathbf{Q}$ is orthogonal, we have $\mathbf{L} = \mathbf{Q}^t \cdot \mathbf{C}$ and therefore equation (1.21) becomes:

$$\mathbf{C}' = \mathbf{L} \cdot \mathbf{Q} = \mathbf{Q}^t \cdot \mathbf{C} \cdot \mathbf{Q} \tag{1.22}$$

Consequently the matrix $\mathbf{RQ}$ has the same eigenvalues than $\mathbf{C}$. This is called the $\mathbf{QL}$ transformation of matrix $\mathbf{C}$. It preserves the tridiagonal form and the symmetry of the matrix.

Algorithm now works as follows. Once you have a tridiagonal matrix $\mathbf{C}$:

1. find its $\mathbf{Q} \cdot \mathbf{L}$ decomposition
2. generate $\mathbf{C}_1 = \mathbf{L} \cdot \mathbf{Q}$
3. find the $\mathbf{Q_1} \cdot \mathbf{L_1}$ decomposition of $\mathbf{C_1}$
4. generate $\mathbf{C}_2 = \mathbf{L_1} \cdot \mathbf{Q_1}$
5. find the $\mathbf{Q_2} \cdot \mathbf{L_2}$ decomposition of $\mathbf{C_2}$
6. $\cdots$

and so on, until the off-diagonal elements vanish. The workload for this algorithm is $O(n^3)$ per iteration for a general matrix, but only $O(n)$ per iteration for a tridiagonal matrix.

The eigenvalues appear on the diagonal.

(20) describes some refinements which lead to a "QL algorithm with implicit shifts". If the eigenvectors are required, the workload is about $3n^4$ operations.

1.1.4 Comparisons of the PCA methods

Speed

The figure 1.4 shows the typical amount of time required to find the first n eigenvalues of a 1024×1024 covariance matrix built using an 256×256 pixels image split into 8×8 blocks. Details about how this matrix has been generated from an image can be found in chapter 3. The amount of time to perform the same operation using the ALA is too large to be displayed on the same figure, it is plotted on figure 1.5.

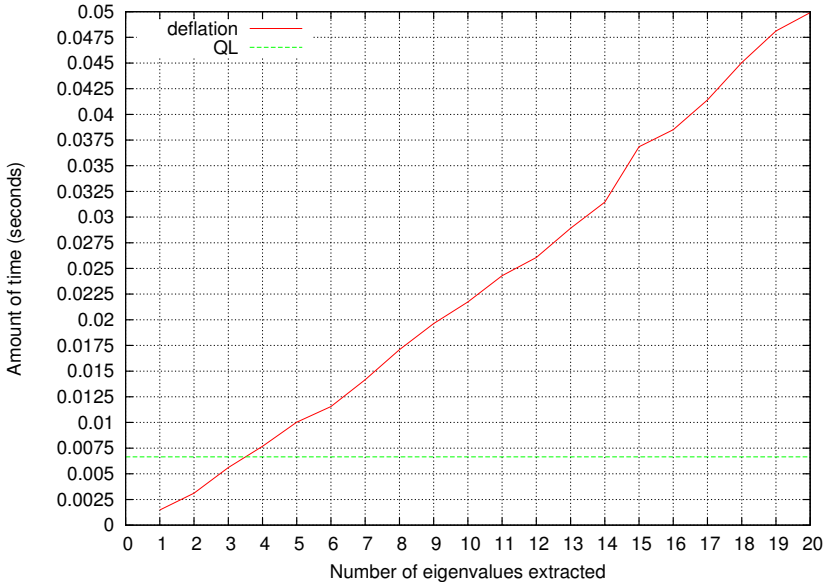


Figure 1.4: Amount of time required to find the first n eigenvalues and associated eigenvectors of a 1024×1024 matrix using the deflation and QL methods.

Results are conform to intuition. Since the QL method find all the eigenvalues at the same time, it requires the same amount of time to compute one than two hundred eigenvalues. In the other hand, the deflation method use the eigenvalues previously computed to find the next one. Consequently if computing an eigenvalue need an amount of time t and assuming that the same number of iterations is required for each eigenvalue, finding the nth one will need nt time units. Figure 1.4 shows that in our case the QL method is faster than the deflation if more than about 4 eigenvectors are needed.

The QL algorithm used in our test compute the eigenvectors and the eigenvalues – we need both since we use them for a PCA. We previously explained that the method can be significativly faster if only eigenvalues are required. The deflation method need the computation of eigenvectors, so no time will be saved if they are not needed. The QL method will per consequent be chosen if only eigenvalues are required.

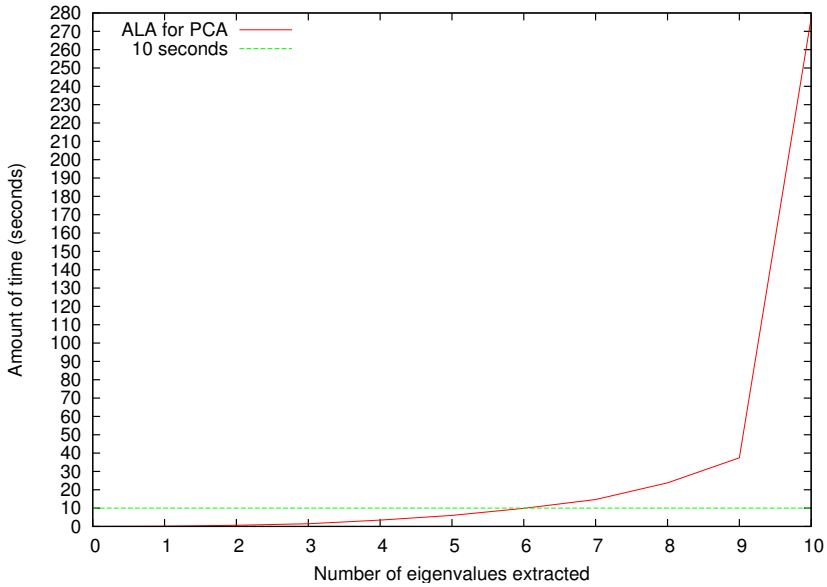


Figure 1.5: Amount of time required to find the first n eigenvalues and associated eigenvectors of a 1024×1024 matrix using the ALA for PCA

ALA execution to find eigenvectors take a lot more time than the previously described method. Its main advantage is its ability to be used *online*. This eliminates the need to compute – and store – the covariance matrix associated with the input data, which can be very large. Furthermore, in some practical applications of PCA the whole dataset is not known at the beginning of the algorithm.

Figure 1.5 shows that this algorithm is computationnaly useable in our test if less than six eigenvectors are needed, while it remains a lot slower than the deflation method.

Quality

These algorithms are iterative. Since a lot of iterative methods can accumulate errors – errors at step n are added to the ones at step $n+1$–, these algorithms may not be able to compute the exact eigenvalues and eigenvectors even if they run during an infinite amount of time. Please note that the observed errors are mainly caused by the computer implementation and the non-infinite precision used to represent real numbers.

It is difficult to estimate how close the found eigenvalues and eigenvectors are from the *real* values since the latter are usually not known.

Our first test is based on a fundamental properties of eigenvalues and eigevectors. Let us denote by $\lambda_1, \lambda_2, \cdots, \lambda_n$ the eigenvalues, and by $\mathbf{v}_{(\lambda_1)}, \mathbf{v}_{(\lambda_2)}, \cdots, \mathbf{v}_{(\lambda_n)}$ the associated eigenvectors.

Let $\mathbf{P}$ and $\mathbf{D}$ such that:

$$\mathbf{P} = \begin{pmatrix} v_{(\lambda_1)_1} & v_{(\lambda_2)_1} & \cdots & v_{(\lambda_n)_1} \\ \vdots & \vdots & \vdots & \vdots \\ v_{(\lambda_1)_n} & v_{(\lambda_2)_n} & \cdots & v_{(\lambda_n)_1} \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_n & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

We must have:

$$\mathbf{C} = \mathbf{PDP}^{-1} \tag{1.23}$$

For each algorithm, we compute $\mathbf{PDP}^{-1}$ using the found values and we compare the resulting matrix with $\mathbf{C}$. This leads to a total error computed as follows:

$$E = \frac{1}{n^2} \sum_{i=0}^{n} \sum_{j=0}^{n} |C_{ij} - (\mathbf{PDP}^{-1})_{ij}| \tag{1.24}$$

Using equation (1.24) we computed E for random symmetrical matrices of different sizes. Values are in $[0; 1]$. The QZ method implemented in the GSL (Gnu Scientific Library) has been included in order to compare our implementations with the algorithm currently used in scientific packages as Matlab, Mathematica, ... Figure 1.6 presents the results of this experiment.

Matrix size	QZ method	Deflation method	QL method
5	0.000000	0.000072	0.001522
10	0.000000	0.000113	0.051834
25	0.000000	0.000231	0.172451
50	0.000000	0.000377	0.307302
100	0.000000	0.000583	0.450959

Figure 1.6: Value of E (Equation 1.24) for random matrices of different size, with values between 0 and 1.

The QZ method seems to find perfect eigenvalues and eigenvectors. Using our implementations, the deflate method give more accurate values than the QL one.

Since the deflation method uses the n-th eigenvector to compute the $n + 1$-th one, we may expect that the last eigenvectors found using this method will be worse than the first ones. The previous experiment shown that the QZ method used in the GSL produces better results than the methods we implemented, consequently we choose to estimate how far each eigenvalue is from the real one by comparing each eigenvalue found with the one computed using the GSL. To that aim, we computed E_i such that:

$$E_i = \frac{\lambda_i}{\lambda_{i_{GSL}}} \tag{1.25}$$

Results for a 150×150 random symmetrical matrix are shown on figure 1.7. Values found using the deflation method seems inaccurate only for the last 30 eigenvalues, whereas our implementation of the QL method provides worse values.

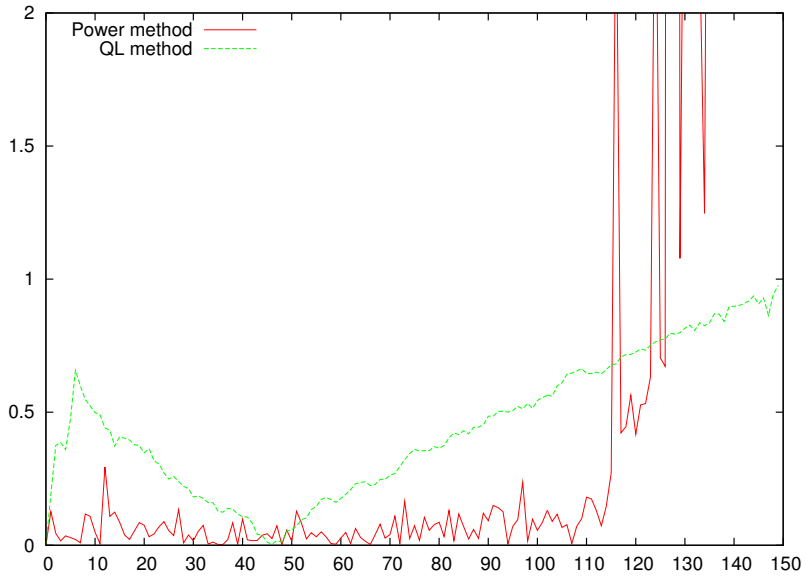


Figure 1.7: E_i for each eigenvalues of a 150×150 random symetrical matrix.

1.2 Gauss-Newton method

The Gauss-Newton method is an unconstrained optimization technique applicable to a function $\mathcal{E}(\mathbf{w})$ of some unknown weight vector $\mathbf{w}$ that is expressed as the sum of error squares:

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{N} e_i(p)^2 \tag{1.26}$$

where $e_i(p)$ denotes the error associated with each entry of the database. Proofs concerning this method can be found in (13).

This method can be used to compute the weight matrix associated with the last layer of a multi-layer perceptron; it can be especially useful if good weights have been computed on the previous layers. In this case, using (2) we have for each output neuron i:

$$\begin{aligned} e_i(p) &= Y_i(p) - D_i(p) && (1.27)\\ &= \varphi_{n-1}(A_{n-1,i}) && (1.28)\\ &= \varphi_{n-1}\left(\sum_{j=1}^{nb(n-1)} w_{n-1,i,j} S_{n-2,j}\right) - D_i && (1.29)\\ i &= 1, 2, \cdots, nb(n-1) \end{aligned}$$

Let $m = nb(n-2)$, the algorithm can be summarized as follows:

Step 1 Compute the Jacobian matrix J:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial e_1}{\partial w_0} & \frac{\partial e_1}{\partial w_1} & \cdots & \frac{\partial e_1}{\partial w_m} \\ \frac{\partial e_2}{\partial w_0} & \frac{\partial e_2}{\partial w_1} & \cdots & \frac{\partial e_1}{\partial w_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e_N}{\partial w_0} & \frac{\partial e_N}{\partial w_1} & \cdots & \frac{\partial e_N}{\partial w_m} \end{pmatrix} \tag{1.30}$$

Put differently:

$$\mathbf{J}_{p,k} = \frac{\partial e_p}{\partial w_k},\ p = 1, 2, \cdots, N;\ k = 1, 2, \cdots, m \tag{1.31}$$

If $e_i(p) = S_{n-1,i} - D_i(p)$ (1.29) then, if S_{n-2} is computed using the p-th training example, J can be computed as follows:

$$J_{p,k} = \varphi'_{n-1}(A_{n-1,i}) S_{n-2,k} \tag{1.32}$$

$$= \varphi'_{n-1} \left(\sum_{j=1}^{nb(n-1)} w_{n-1,k,j} S_{n-2,j} \right) S_{n-2,k} \tag{1.33}$$

Step 2 Compute the pseudo-inverse of $\mathbf{J}$.

$$\mathbf{P} = (\mathbf{J}^t(n)\mathbf{J}(n))^{-1}\mathbf{J}^t(n) \tag{1.34}$$

The Greville's algorithm can be used for this computation.

Step 3 Update weights for the neuron p:

$$\mathbf{w_p}(t+1) = \mathbf{w_p}(t) - \mathbf{P} \cdot \mathbf{e} \tag{1.35}$$

Step 4 Return to Step 1 until the error stops to decrease.

Chapter 2

MLP Learning Process

This chapter presents the study of different ways to improve the learning process of a multiple-layer perceptron. We first introduce the two variants of the base algorithm for the learning process, the back-propagation, and study their behavior. In a second part, we study several general techniques used to speed up the learning process. Since a quicker learning procedure does not effectively mean better results, we also present some general heuristics to improve the convergence of the network to satisfactory solutions. The last part is dedicated to the study of different learning algorithms designed to improve both learning speed and results quality.

2.1 Standard back-propagation algorithms

2.1.1 Stochastic back-propagation algorithm

Principle

The back-propagation algorithm find the set of weights which minimizes the error E. Extensive studies on this algorithm can be found in (13) and (4).

When a learning pattern is clamped, the activation values are forward-propagated to the output units, and the actual network output is compared with the desired output values; we usually end up with an error in each of the output neuron. The simplest method to reduce this error is to change weights in such a way that, on the next iteration, the error will be zero for this particular pattern. To that aim, weights are updated using a simple learning rule based on the gradient descent principle:

$$w_{cij}(t+1) = w_{cij}(t) - \epsilon \frac{\partial e(i)}{\partial w_{cij}} \tag{2.1}$$

where ϵ denotes a small positive real number called the *learning rate*.

Considering only neurons of the last layer (layer $n-1$), we may express this gradient as:

$$\frac{\partial e(i)}{\partial w_{n-1,i,j}} = \frac{\partial e(i)}{\partial A_{n-1,i}} \times \frac{\partial A_{n-1,i}}{\partial w_{n-1,i,j}} \tag{2.2}$$

$$\frac{\partial e(i)}{\partial A_{n-1,i}} = \frac{\partial e(i)}{\partial S_{k-1,i}} \times \frac{\partial S_{n-1,ii}}{\partial A_{n-1,i}} \tag{2.3}$$

$$= 2(S_{n-1,i} - D_i)\varphi'_{n-1}(A_{n-1}, i) \tag{2.4}$$

$$\frac{\partial A_{n-1,i}}{\partial w_{n-1,i,j}} = S_{n-2,j} \tag{2.5}$$

This leads to the final expression of the gradient associated with the output neurons:

$$\frac{\partial e(i)}{\partial w_{cij}} = 2(S_{n-1,i} - D_i)\varphi'_{n-1}(A_{n-1,i})S_{n-2,j} \tag{2.6}$$

The gradient associated with hidden neurons ($c \in [1, \cdots, n-2]$) can be computed using the following equations:

$$\frac{\partial e(i)}{\partial w_{cij}} = \frac{\partial e(i)}{\partial A_{ci}} \times \frac{\partial A_{c,i}}{\partial w_{cij}} \tag{2.7}$$

$$\frac{\partial A_{c,i}}{\partial w_{c,i,j}} = S_{c-1,j} \tag{2.8}$$

$$\frac{\partial e(i)}{\partial A_{ci}} = \sum_{k=1}^{nb(c+1)} \frac{\partial e(i)}{\partial A_{c+1,k}} \times \frac{\partial A_{c+1,k}}{\partial A_{ci}} \tag{2.9}$$

$$\frac{\partial A_{c+1,k}}{A_{ci}} = \frac{\partial A_{c+1,k}}{\partial S_{ci}} \times \frac{\partial S_{ci}}{\partial A_{ci}} \tag{2.10}$$

$$= w_{c+1,k,i} \times \varphi'_{ci}(A_{ci}) \tag{2.11}$$

The use of equations (2.8) and (2.11) yields

$$\frac{\partial e(i)}{\partial w_{cij}} = S_{c-1,j} \sum_{k=1}^{c+1} w_{c+1,k,i}\varphi'_{ci}(A_{ci}) \frac{\partial e(i)}{\partial A_{c+1,k}} \tag{2.12}$$

Learning rate influence

True gradient descent requires that infinitesimal steps are taken. For practical purpose, we choose a learning rate that is as large as possible without leading to oscillation.

Figure 2.1 shows the influence of the learning rate on the quadratic error. The benchmark used is detailed in Section 2.4.2.

A large value for ϵ leads quickly to a small error but the resulting curve highly oscillates. Intuitively, weights changes are so large that the next iteration of the algorithm will try to invert the last move. Using small values for ϵ reduces oscillations but increases the number of iterations required to reach a fixed error.

ϵ	final learning error	test error	classification test error
0.001	236.2	93.1	57
0.005	198.4	81.5	57
0.01	174.5	71.5	54
0.05	113.4	45.6	24
0.1	104.7	43.7	22
0.5	85.6	48.5	26

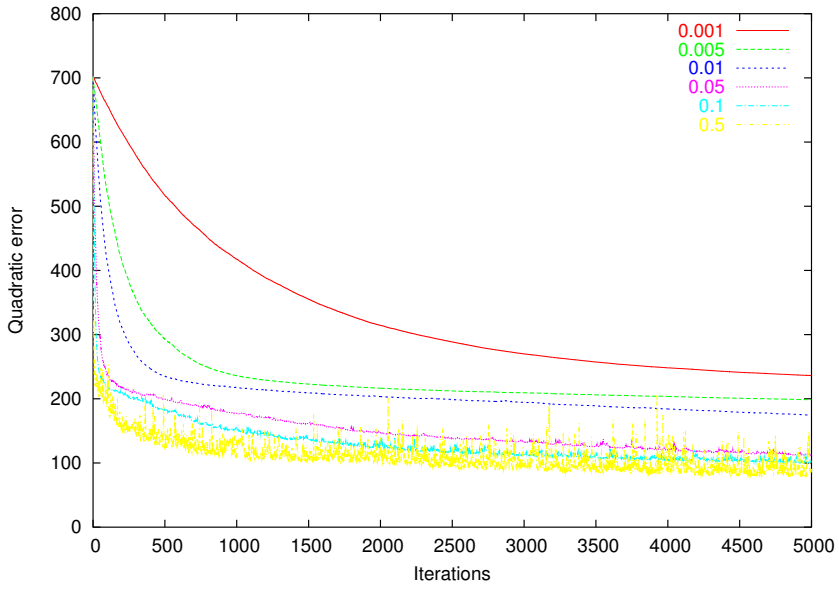


Figure 2.1: Stochastic back-propagation: Learning rate influence. The benchmark framework used is described in Section 2.4.2.

α	final learning error	test error	classification test error
0	236.8	93.2	57
0.1	232.2	91.8	57
0.2	228.3	90.6	57
0.4	220.0	88.5	57
0.6	212.4	86.5	57
0.8	198.3	81.9	57
0.9	174.9	72.0	54
0.95	202.2	82.1	57

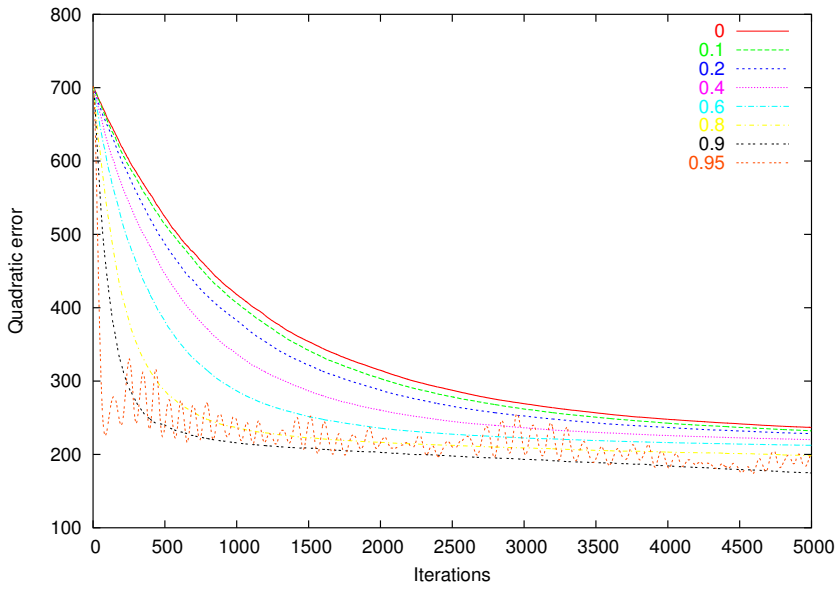


Figure 2.2: Stochastic back-propagation: Momentum influence. The benchmark framework used is described in Section 2.4.2

Momentum influence

One way to avoid oscillation at large learning rate is to make the current weight update dependent of the last weight update by adding a *momentum term*:

$$w_{cij}(t+1) = w_{cij}(t) + \Delta w_{cij}(t) \tag{2.13}$$

where

$$\Delta w_{cij}(t) = -\epsilon \frac{\partial e(i)}{\partial w_{cij}} + \alpha \Delta w_{cij}(t-1) \tag{2.14}$$

where α denotes a small positive number.

When two consecutive changes occur in the same direction, *i.e.* when $-\frac{\partial e(i)}{\partial w_{cij}}$ and $\Delta w_{cij}(t-1)$ have the same sign, the weight change will be larger than using the simple learning rule (Equation 2.1). By contrast, if their sign differs then $-\frac{\partial e(i)}{\partial w_{cij}}$ and $\Delta w_{cij}(t-1)$ will tend to cancel, and the effective learning rate will be small.

This adds inertia to the motion trough weight space and smoothes out the oscillations. Figure 2.2 illustrates the influence of the momentum on the convergence rates and on the error.

2.1.2 Batch back-propagation algorithm

Principle

Batch back-propagation is basically similar to stochastic back-propagation. The difference relies in the weight update. While in stochastic back-propagation an update step is performed after each single pattern is presented, in batch back-propagation weight updates are computed after a presentation of all training patterns (one epoch). Instead of using the error $e(p)$ as the stochastic back-propagation does, the batch version uses the error $E(p)$ computed on the whole dataset.

Fast adaptive versions of the back-propagation algorithm are almost all based on the batch back-propagation algorithm. Some of these versions are studied in Section 2.4.

Since $E = \sum_p e(p)$ (Equation 2.28), we have:

$$\frac{\partial E}{\partial w_{cij}} = \sum_p \frac{\partial e(p)}{\partial w_{cij}} \tag{2.15}$$

This leads to the weight adaptation rule:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \tag{2.16}$$

where

$$\Delta w_{ij}(t) = -\epsilon \times \frac{\partial E}{\partial w_{ij}} + \alpha \times \Delta w_{ij}(t-1) \tag{2.17}$$

$$= -\epsilon \times \sum_p \frac{\partial e(p)}{\partial w_{cij}} + \alpha \times \Delta w_{ij}(t-1) \tag{2.18}$$

The term $\frac{\partial e(p)}{\partial w_{cij}}$ can be computed using equation (2.6) and (2.12).

Learning rate influence

Figure 2.3 shows the error curve for different learning rates. As in the stochastic back-propagation case, a too large learning rate leads to oscillations while a small one leads to a slower convergence. However, these oscillations are less frequent than in the stochastic case, mainly because summing all weight changes to compute Δw_{cij} is equivalent to add the mean of all changes. Oscillations are thus "amortized".

Momentum influence

Since almost no oscillations are visible on Figure 2.3, the momentum value does not affect a lot the results. Nonetheless, one can note that big values for the momentum slightly improve the convergence speed. Representative results can observed on Figure 2.4.

2.2 Speeding up the learning process

Changing the training algorithm remains the most popular way to accelerate the learning process. Relevant algorithms are presented in Section 2.4, but there also exist some techniques for improving learning speed independently of the training algorithm.

ϵ	final learning error	test error	classification test error
0.0001	166.2	68.2	53
0.0005	106.5	43.6	22
0.001	90.6	37.4	20
0.002	72.3	30.6	17
0.004	65.0	27.4	13

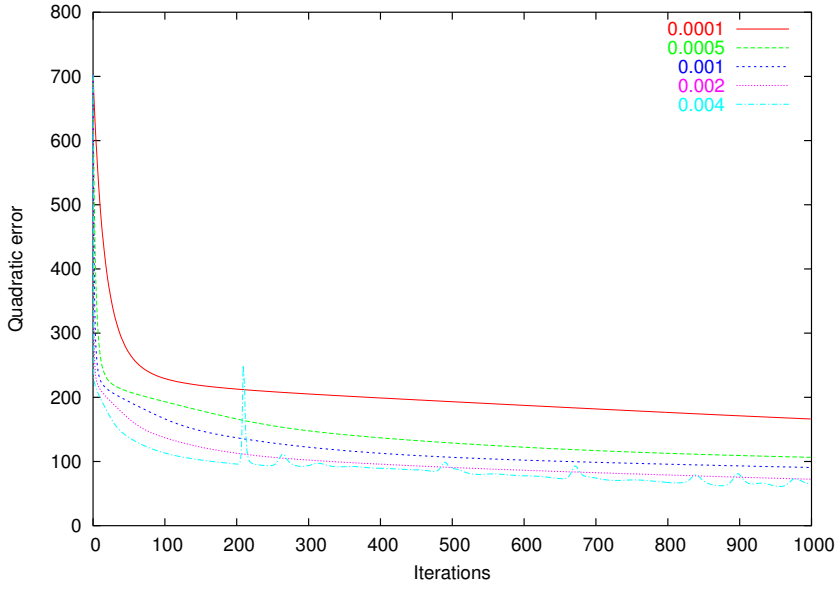


Figure 2.3: Batch back-propagation: Learning rate influence. The benchmark framework used is described in Section 2.4.2.

α	final learning error	test error	classification test error
0.	90.6	37.4	20
0.2	85.3	35.2	19
0.4	77.9	32.5	19
0.6	66.2	28.4	13
0.8	45.3	20.5	9

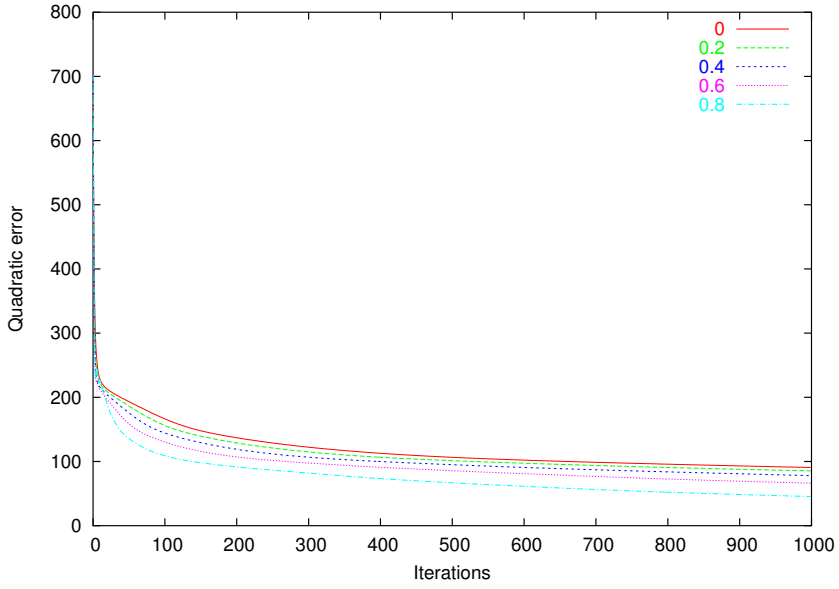


Figure 2.4: Batch back-propagation: Momentum influence. The benchmark framework used is described in Section 2.4.2.

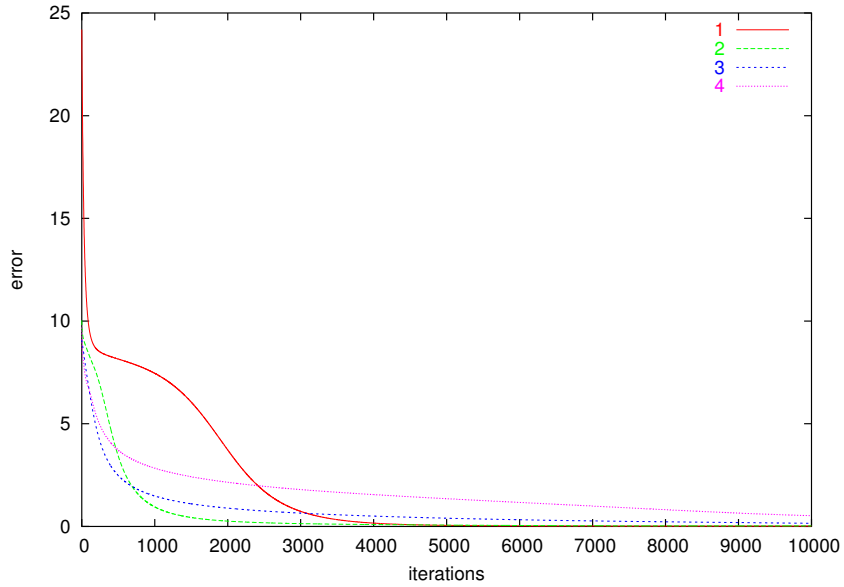


Figure 2.5: Different activation functions: influence of D

The test problem is the fit of the $sinus$ function. The learning algorithm used is a standard batch back-propagation.

2.2.1 Different activation functions

The usual activation function used for neural networks is a sigmoid between -1 and 1:

$$\varphi(x) = \frac{2}{1 + \exp(-D * x)} - 1$$

Usually, one uses $D = 1$ or $D = 2$ (the popular `tanh` function).

By increasing D, one sharpens the activation function. Some papers report that this can significantly accelerate the learning process (15), but there is no way to predict what value will be the best. Figure 2.5 presents the convergence curves obtained on an artificial test (fitting a sinus function) with a batch back-propagation algorithm and a variable D. Generally, increasing D speeded up the convergence for artificial tests (function fitting), but our experiments showed that this heavily depends on the application. For example, for real problems like medical classification ones (see Section 2.4.2), increasing D over 1 slowed down the convergence and even made the algorithm unstable.

Additionally, one can speed up the computation of the activation function by approximating it (3). These numerical approximation do not change the number of epoch needed but can save computation time.

2.2.2 Different error functions

Another existing strategy to speed up the learning process is to change the error expression. The usual error used is the quadratic one:

$$E = \sum_{p} \sum_{i \in OutputUnits} (Y_i - D_i)^2$$

One can express a generalization of this error function:

$$E = \sum_{p} \sum_{i \in OutputUnits} (Y_i - D_i)^{2^n}$$

Small values of n are valuable for problems with noisy data. Increasing n makes the "extreme" examples (i.e. the examples with the higher error) more and more preponderant. Since changing n modifies the computation of the gradient, it is difficult to evaluate properly the impact of this technique. Indeed, if the gradient is not computed the same way, we can not keep the same parameters (especially the learning rate) for the learning algorithm. An empiric observation is that n can be increased to speed up the learning process if the data are clean enough. With the "real" (i.e., not artificial) tests we made, increasing D over 1 always degraded the results and the stability of the learning process. Sometimes, the best value was even under 1.

2.2.3 Derivate fudge

Using the usual sigmoid as activation function, the derivate function is:

$$\frac{2 * D * \exp(-D * x)}{((1 + \exp(-D * x)) * (1 + \exp(-D * x)))}$$

In Figure 2.6, one can see that this derivate takes nearly null values for small negative and big positive inputs. Then, if the unit activation is very high or very small, weights will be modified very slowly. This can lead to very long refinement phases. To cope with this, Fahlman (9) proposes to add a small fixed value (generally 0.1) to the derivate in order to create artificial dynamics.

With the batch back-propagation algorithm, using a fudged derivate can slightly accelerate the convergence without degrading the test results. Naturally, too big fudge values can endanger the stabilization of the results by introducing too big artificial dynamics. The test presented in Figure 2.7 was performed on the same problem and with the same benchmarking rules than presented in Subsection 2.4.2.

The derivate fudge can be interesting for every learning rate-based algorithm, including back-propagation, quick back-propagation, delta-bar-delta, Silva-Almeida and SuperSAB. Indeed, all these algorithms can suffer from the so-called "flat-spot" phenomenon (9). With algorithms derived from the Manhattan rule (i.e. based on a momentum step) like the Rprop algorithm, we did not see any effective improvement.

2.3 Improving the results of the learning process

A quicker learning process does not necessarily imply better results. Indeed, the learning procedure consists only in minimizing the error on learning samples. On both artificial and real problems, the network is always trained on a limited set of samples. Then, we expect the network to be able to interpolate and/or extrapolate the other cases from this learning set. Finally, the generalization power of the network highly depends on the way it was trained. In practice, very low learning errors can hide catastrophic phenomenons like over-fitting.

This section presents different techniques for improving the results of the learning process. These heuristics and adaptive training algorithms are complementary, since advanced learning algorithms (see Section 2.4) can also help to improve the generalization results.

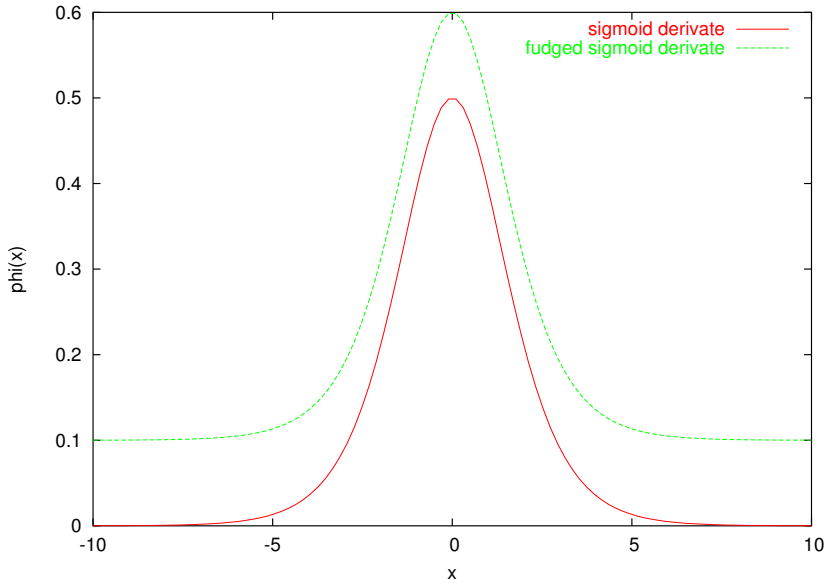


Figure 2.6: Derivate fudge

The derivate of the standard sigmoid can take very small values for small and big inputs. The derivate fudge technique consists in adding a constant value (0.1) to avoid infinitesimal weight steps.

fudge	final learning error	test error	classification test error
0.	106.5	43.6	2
0.1	96.7	40.0	21
0.2	89.9	37.2	20
0.5	93.5	42.2	21

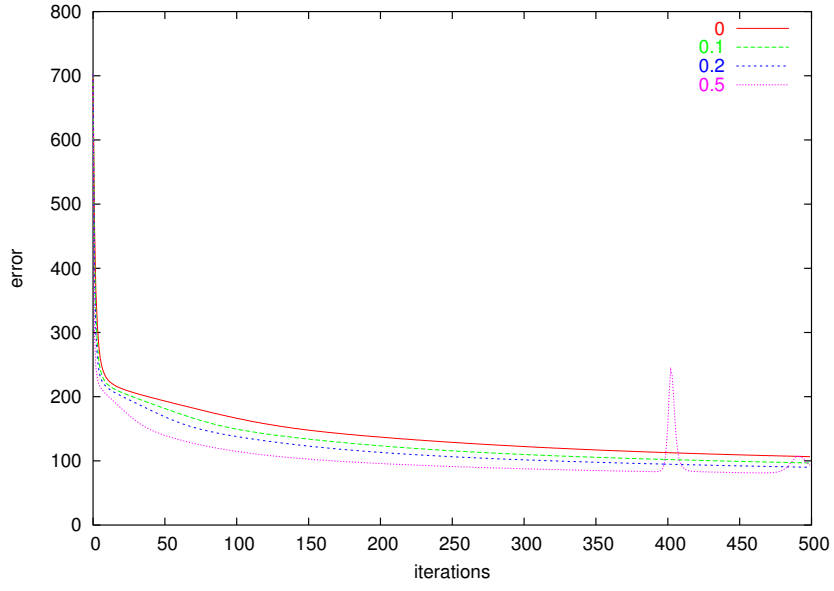


Figure 2.7: Influence of the derivate fudge on the batch back-propagation

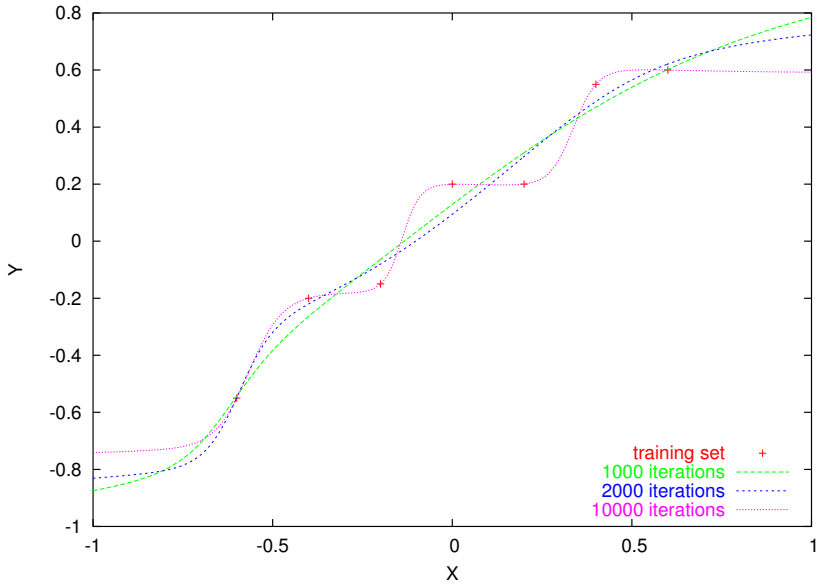


Figure 2.8: Over-fitting: influence of the number of training iterations
The aim of this test was to fit the function $f(x) = x + 0.1$ with a 1-50-1 MLP. To simulate coarsely a real-case problem, the training points given are alternatively over and under the theoretic line. The algorithm used to train the network is the Resilient Back-propagation. The curves present the result functions learned by the network w.r.t. the number of epoch.

2.3.1 Over-fitting phenomenon and bad generalization

In real problems, data are always degraded by a certain amount of noise. This makes a perfect learning impossible. Then, one can understand that learning "by heart" the exact training samples will not help in getting the better generalized results and can even degrade them.

Figure 2.8 presents a simple test to illustrate the over-fitting phenomenon. The best fit of the function $f(x) = x + 0.1$ is obtained after 1000 iterations of the Resilient Back-propagation algorithm (see Section 2.4.6). One can note that the algorithm tends naturally to the best linear approximation, at least during the first iterations of the algorithm. It is also interesting to notice how the algorithm is attracted by the first and last points. Progressively, the algorithm degrades the linear approximation, in order to reduce its learning error. After about 2000 iterations, the over-fitting phenomenon begins. The algorithm will attempt all the possible deformations to fit exactly the training patterns, at the risk of bad generalization. By computing complex, highly non-linear deformations, the learning process results in a network unusable if asked on anything else but the training samples.

2.3.2 Network size

A widely stated observation is that the best generalization results are obtained with a relatively small number of hidden units. Indeed, increasing the number of hidden units increases the computational power of the network. It becomes able to fit more complex functions and then to solve more complex problems. Unfortunately, too much hidden units increases the risk for the network to over-fit the learning samples and then to give bad generalization. Thus, a general recommendation is to keep the network as small as possible to avoid over-fitting problems. See Figure 2.9 for the results obtained on the fitting of the function $f(x) = x + 0.1$ with different numbers of

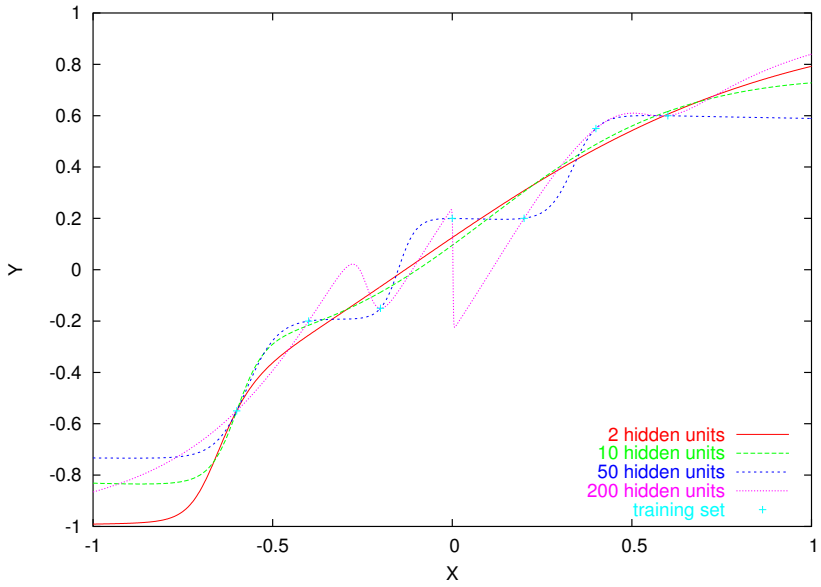


Figure 2.9: Over-fitting: influence of the number of hidden units
Fitting the function $f(x) = x + 0.1$ with a 1-50-1 MLP. To simulate coarsely a real-case problem, the training points given are alternatively over and under the theoretic line. The algorithm used to train the network is the Resilient Back-propagation. The curves present the result function learned by the network w.r.t. the number of hidden units.

hidden units.

2.3.3 Error threshold

Fahlman (9) proposes a new error computation scheme. He introduces a threshold of 0.1 under which the unit error is set to 0. By using this error criteria, units having a quadratic error less than 0.01 are not trained anymore. Such an error function is designed to improve the stabilization of the results. On the problem of fitting $f(x) = x + 0.1$ with a 1-50-1 MLP, a threshold reduces considerably the over-fitting phenomenon (see Figure 2.10. In our experiments on real problems, we did not note any significant melioration with this method.

2.3.4 Regularization and weight decay

The technique of regularization encourages a smooth network mapping by adding a penalty E_r to the error function:

$$E^* = E_a + \lambda E_r \tag{2.19}$$

where E_a is an error function (Section 2.2.2).

The most common form of regularizer E_r is called *weight decay* and consists in the sum of the weights elevated at a certain power of 2. Thus, *weight decay* penalizes large weights:

$$E_r = \sum_{c,i,j} (w_{cij})^{2^q},\ q \in \mathbb{N} \tag{2.20}$$

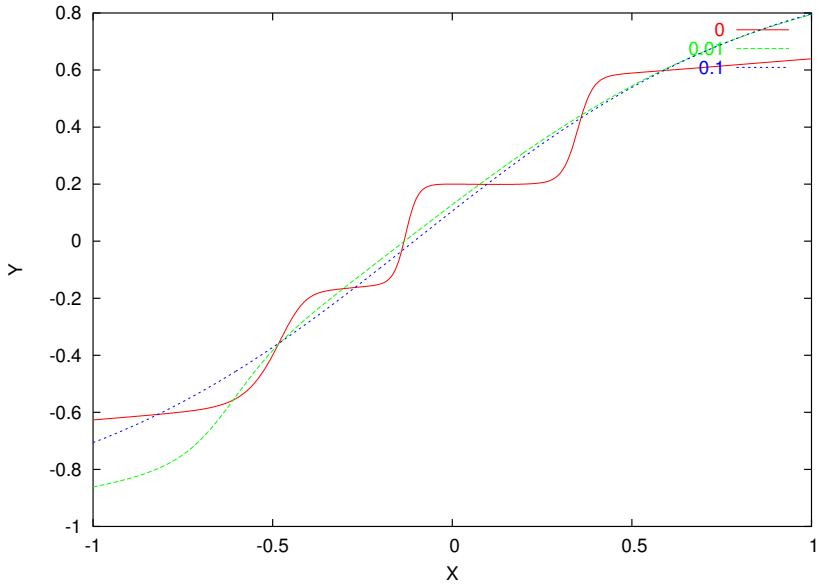


Figure 2.10: Error threshold influence
Fitting the function $f(x) = x + 0.1$ with a 1-50-1 MLP. To simulate coarsely a real-case problem, the training points given are alternatively over and under the theoretic line. The algorithm used to train the network is the Resilient Back-propagation. The curves present the result function learned by the network w.r.t. the error threshold.

The gradient $\frac{\partial E}{\partial w_{cij}}$ is now:

$$\frac{\partial E^*}{\partial w_{cij}} = \frac{\partial E_a}{\partial w_{cij}} + \lambda \frac{\partial E_r}{\partial w_{cij}} \tag{2.21}$$

$$= \frac{\partial E_a}{\partial w_{cij}} + \lambda 2^q (w_{cij})^{2^q - 1} \tag{2.22}$$

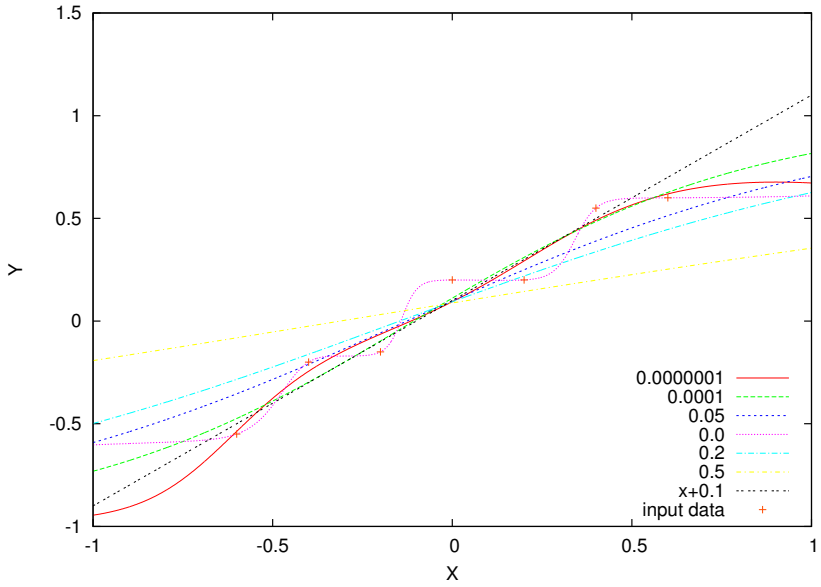


Figure 2.11: Influence of a regularization term, $q = 2$, variable λ
Fitting the function $f(x) = x + 0.1$ with a 1-50-1 MLP. To simulate coarsely a real-case problem, the training points given are alternatively over and under the theoretic line. The algorithm used to train the network is the Resilient Back-propagation. The curves present the result function learned by the network w.r.t. the value of the λ parameter.

The *weight decay* technique can be considered as a security for maintaining the stability of the convergence. In fact, it can slightly slow down the convergence, but it is also a way to cope efficiently with the over-fitting phenomenon. Figure 2.11 illustrates the influence of a regularization term λ on the problem of fitting the function $f(x) = x + 0.1$ with a 1-50-1 MLP. Some experiments presented in (11) show that the best results come from starting weight decay when the network reaches a minimum on the training set.

Schiffmann et al. (24) explains that this weight regularization is needed for some algorithms, like quick back-propagation (see Section 2.4.7). They suggest a simple decay:

$$\frac{\partial E^*}{\partial w_{ij}}(t) = \frac{\partial E_a}{\partial w_{ij}}(t) + \lambda * w_{ij}(t) \tag{2.23}$$

This computation scheme is strictly equivalent to the former one, with $q = 1$.

2.3.5 Weight Pruning and Cross-Validation

Another way to cope with over-fitting and to improve the generalization power of the network is the technique of weight pruning, usually mixed with cross-validation. The idea of weight pruning is to eliminate excess weights during the training process. Indeed, like it was exposed in Section 2.3.2, too numerous hidden units and then too numerous weights usually lead to over-fitting.

The cross-validation principle is to give an estimator of the prediction risk (19) used to estimate the expected performance of the network on future data (i.e. the generalization power). Since the predication risk cannot be computed directly, the cross-validation technique evaluates it. Cross-validation results in a new training scheme. The main idea is to divide the training set T into an

effective training set T' and a validation set V. The network is trained on T' and its generalization power is evaluated on V. This evaluation gives an estimation of the prediction risk and drives the training process to the best generalization. It used for example to perform some backtracking or some pruning of excess weights.

Utans and Moody (27) study different pruning algorithms. They point out different algorithms based on the cross-validation technique. Prechelt (19) gives a detailed overview of existing weight pruning techniques.

2.4 Learning algorithms

This section presents the study of different locally adaptive algorithms used for training multilayer perceptrons. Basically, they can be seen as extensions of the batch back-propagation algorithm presented in Section 2.1.2. Some of them are mainly based on a local adaptation of the learning rate, while others rather adapt the momentum term.

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t) \tag{2.24}$$

where $\Delta w_{ij}(t)$ is an adaptive learning step.

This section proposes the study of the following algorithms:

- algorithms based on an adaptation of the learning rate:

$$\Delta w_{ij}(t) = -\epsilon(t) \times \frac{\partial E}{\partial w_{ij}} + \alpha \times \Delta w_{ij}(t-1) \tag{2.25}$$

  - Silva-Almeida back-propagation;
  - super self-adaptive back-propagation (SuperSAB);
  - delta-bar-delta back-propagation.

- algorithms mainly based on the adaptation of the momentum term:

$$\Delta w_{ij}(t) = -\epsilon \times \frac{\partial E}{\partial w_{ij}} + \alpha(t) \times \Delta w_{ij}(t-1) \tag{2.26}$$

  - resilient back-propagation (RProp);
  - quick back-propagation (QuickProp).

For each algorithm, the influence of its parameters is studied. We also give some general considerations about performance, generalization quality and stability. We also give some pointers to other studies and point out commonly recommended settings.

2.4.1 Existing studies

Prechelt (18) presents `Proben1`, a study of different problems, artificial and real-world ones, treated with neural networks. He designs an entire benchmarking framework and especially stresses on benchmarking rules and test validity. Schiffmann et al. (24) focuses on the comparison of different techniques for optimizing back-propagation process. The different algorithms studied in this chapter are also discussed in this paper. Fahlman (9) stresses on back-propagation weaknesses and finally presents his Quick Back-Propagation algorithm to cope with this difficulties. Finally, other studies on specific applications can be found at the UCI repository for machine learning databases (5).

2.4.2 Benchmarking rules

Establishing relevant benchmarking rules for neural networks is a key point for proper scientific studies (18). The validity, the reproductibility and the comparability of the tests must be ensured.

Benchmark problem used

For this study, we preferred a real-world problem to an artificial one for algorithms not to be biased by the regularity of the problem. Like explained in (18), realistics problems are more representative of the algorithms' behavior than artificial ones, even if noise can be added to artificial tests.

The database used for this study is extracted from the UCI repository of machine learning databases (5). This `Thyroid` database aims at diagnosing thyroid hyper-, normal or hypofunction thanks to 21 factors. This problem is basically a classification problem between 3 classes. The class probabilities are 5.1%, 92.6% and 2.3% respectively. The database is composed of 972 samples.

For the purpose of this study, the database is separated into two databases:

- A training set of 729 samples (75% of the total database)
- A test set of 243 samples (25% of the total database)

The same training and test sets are used for the whole study. Training and test samples were chosen randomly, but with the property that the class probabilities of the original database remain the same in both training and test sets.

Neural network used

The network used is a 3-layer perceptron with the following characteristics:

- 21 input units, 10 units in the hidden layer, 3 output units;
- Sigmoid activation functions for all the units.

Error measures

The typical error used for evaluating both the training and the test parts is the global quadratic error, defined by:

$$E = \sum_{p} \sum_{i \in OutputUnits} (Y_i - D_i)^2$$

Where p is a pattern, Y_i the output of the unit i and D_i the corresponding component of the desired output vector for p.

We denote by $e(i)$ the error associated to one learning pattern of the database:

$$e(p) = \sum_{i \in OutputUnits} (Y_i - D_i)^2 \tag{2.27}$$

$$E = \sum_{p} e(p) \tag{2.28}$$

This error measure is valuated during the training process to evaluate the convergence speed. Afterwards, it is also computed on the test set in order to evaluate the generalization quality of the algorithm.

We also dispose of a classification error that indicates the number of test samples misclassified by the network.

Tests performed

The curves presented in this chapter illustrate the convergence speed for a given number of epoch. Network's weights are initialized randomly, but the same network initialization is used for all the test curves in order to ensure their reproducibility and comparability. Even if they come from a unique run, we ensure their representativity thanks to numerous unpublished tests.

The result tables proposed for each test give averaged results obtained through (at least 10) randomly initialized runs. They present the final learning error, the test error and the classification test error after a fixed number of iterations of the algorithm.

2.4.3 Silva-Almeida algorithm

Silva and Almeida (25) proposed the so-called "Silva-Almeida" algorithm that performs a learning rate adaptation by sign changes. When the gradient keeps its sign, the learning rate is increased by η_+ (acceleration rate > 1). At the contrary, if the gradient changes its sign, the learning rate is decreased by η_- times (0 < deceleration rate < 1). Small initial values for ϵ_0 has to be chosen.

This algorithm can additionally perform a fixed rate momentum update α.

Learning rates are adapted as follows:

$$\begin{array}{ll} \epsilon_{ij}(0) = \epsilon_0 & \\ \epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_+ & if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\ \epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_- & else \end{array}$$

Connection weights are updated classically:

$$\Delta w_{ij}(t) = -\epsilon_{ij}(t) * (\frac{\partial E}{\partial w_{ij}}(t)) + \alpha * \Delta W_{ij}(t-1)$$

In addition to this update rule, Silva and Almeida use a global backtracking strategy which restarts an update step if the total error increases. In this case, both acceleration and deceleration learning rates are halved.

Our experimentations revealed that it is preferable to bound learning rates by lower and upper limits (respectively ϵ_{min} and ϵ_{max}) in order to avoid over- and under-flow phenomenon.

α	final learning error	test error	classification test error
0.	16.4	8.3	2
0.2	13.6	7.8	2
0.4	10.6	7.6	2
0.6	10.9	7.8	2
0.8	8.3	4.9	2

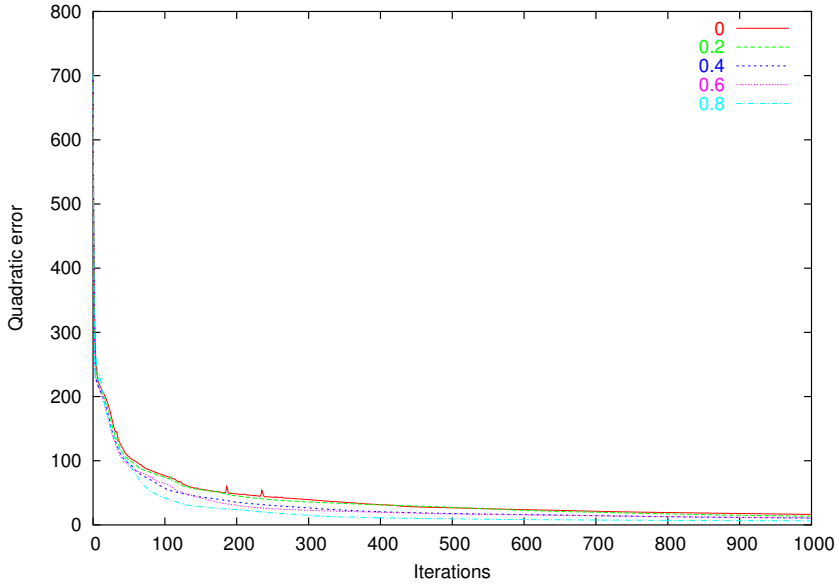


Figure 2.12: Silva-Almeida: Momentum influence

Recommended values for parameters

Silva and Almeida (25) recommend the following values:

$$\alpha = 0.2$$
$$\eta_+ = 1.1$$
$$\eta_- = 0.7$$
$$\epsilon_{max} = 5.$$
$$\epsilon_{min} = 0.00001$$

Momentum influence (α)

In the different tests we made, small momentum values had a little influence. Increasing its value produced a little acceleration. For some problems like this one, we managed to slightly improve the convergence speed thanks to very high momentum rates (see Figure 2.12).

Initial learning rate influence (ϵ_0)

Initial learning rate growth principally speeds up the beginning of the training process. After some iterations, the initial learning rate does not have a really big influence since the learning rate is auto-adjusted. One should note that too big values endanger the initial stability of the algorithm and slightly slow down the convergence in the stabilization and refinement phase. Depending on the problem, values between 0.0001 and 0.001 are recommended (see Figure 2.13).

ϵ_0	final learning error	test error	classification test error
0.00001	16.3	8.3	2
0.0001	16.6	8.8	2
0.001	24.8	11.9	2
0.01	26.7	13.2	2

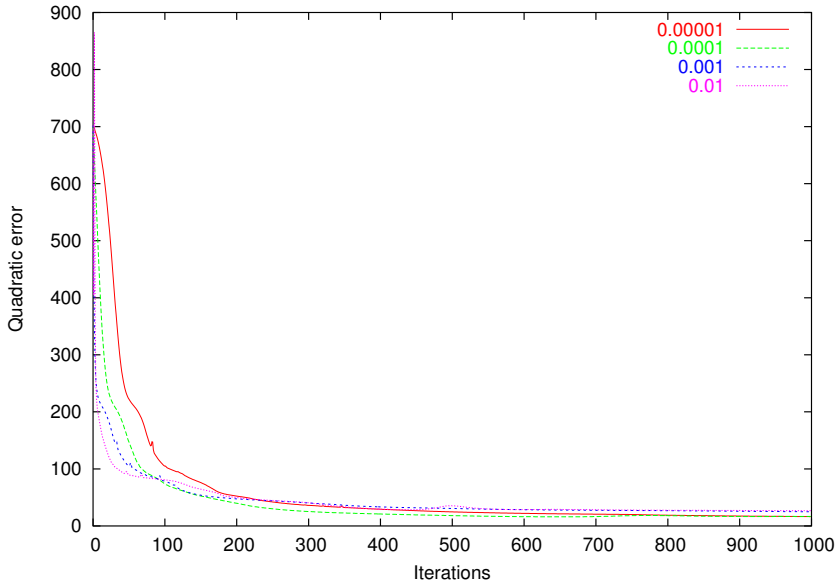


Figure 2.13: Silva-Almeida: Initial learning rate influence

Learning acceleration rate influence (η_+)

With a learning acceleration rate equal to 1, the Silva-Almeida algorithm can not accelerate. Then, it is somehow equivalent to a batch back-propagation with an ϵ_0 learning rate (see Figure 2.14). The difference is that Silva-Almeida can still decrease its learning rate and perform some back-tracking. Values slightly superior to 1 naturally accelerate the learning process, but at the risk of making the convergence unstable. Then, values under 1.2 are recommended (generally recommended value: 1.1).

Learning deceleration rate influence (η_-)

As shown in Figure 2.15, learning deceleration rate has a limited influence since all the tests we made did not involve numerous gradient sign changes. Anyway, having a too big deceleration rate forces the learning rate to stay too close to its old value, so that it can generate local perturbations and result degradation.

Discussion

Even if it is not the quickest algorithm we studied, the Silva-Almeida technique always gave very good results. It is finally one of the most natural auto-adaptive algorithm from a theoretical point of view, using a simple acceleration/deceleration rate on the learning rate. Some studies like (24) also report that this is the method that gives the best generalization results.

η_+	final learning error	test error	classification test error
1.0	90.6	37.4	20
1.05	21.1	10.2	2
1.1	24.8	11.9	2
1.15	23.6	12.8	3
1.2	37.8	17.6	5

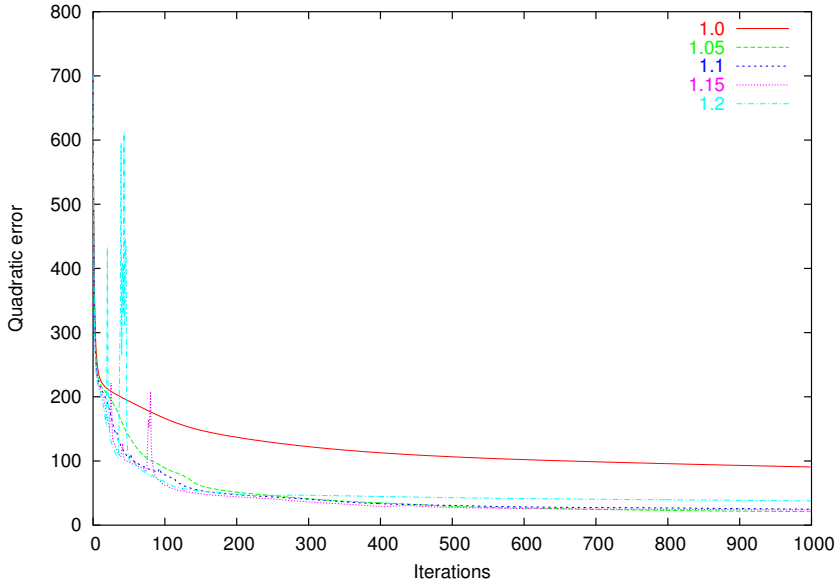


Figure 2.14: Silva-Almeida: Learning acceleration rate influence

η_-	final learning error	test error	classification test error
0.1	27.5	13.5	4
0.4	21.4	10.1	3
0.7	24.8	11.9	2
0.8	38.3	12.8	7

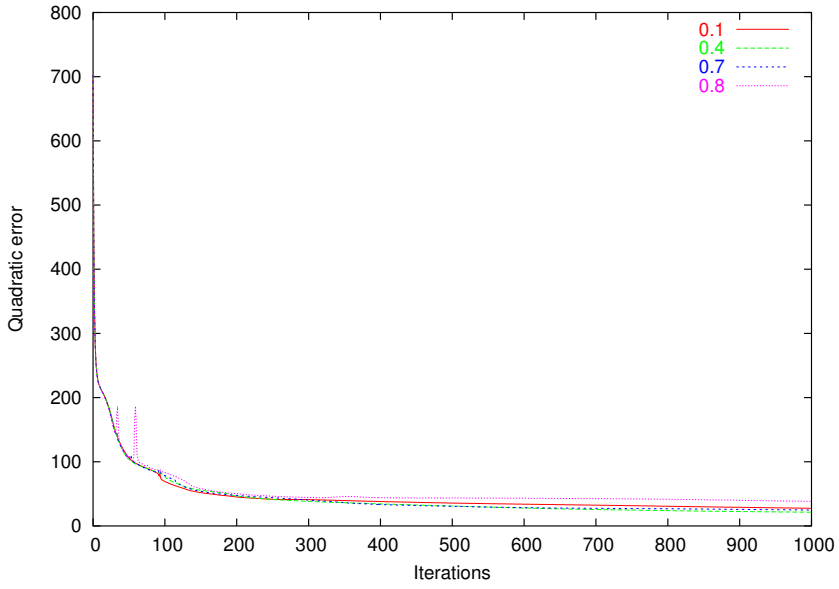


Figure 2.15: Silva-Almeida: Learning deceleration rate influence

2.4.4 Super Self-Adaptive back-propagation algorithm (SuperSAB)

T. Tollenaere's SuperSAB algorithm (26) is quite similar to Silva and Almeida's approach. The SuperSAB algorithm proposes a modified update rule to perform local backtracking instead of global backtracking (see also Subsection 2.4.6 on Rprop backtracking scheme).

Learning rate is adapted the same way than Silva-Almeida:

$$\begin{aligned}
&\epsilon_{ij}(0) = \epsilon_0 \\
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_+ \quad if\ \tfrac{\partial E}{\partial w_{ij}}(t) * \tfrac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_- \quad else
\end{aligned}$$

Where ϵ_0 is the initial learning rate, η_+ the acceleration rate and η_- the deceleration rate.

Weight update is modified this way:

$$\begin{aligned}
&\Delta w_{ij}(t) = -\epsilon_{ij}(t) * \tfrac{\partial E}{\partial w_{ij}}(t) + \alpha * \Delta w_{ij}(t-1) \quad if\ \tfrac{\partial E}{\partial w_{ij}}(t) * \tfrac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\
&\Delta w_{ij}(t) = -\Delta w_{ij}(t-1);\ \Delta w_{ij}(t) = 0 \quad else
\end{aligned}$$

If a gradient sign change is detected, the last weight step is reverted and the current weight update is set to 0 in order to inhibit the momentum term at the next iteration.

Like exposed in (24), we experienced some stability problems with the original algorithm. In fact, this instability is due to too important learning rate growth. Then, the solution was to fix a proper upper limit ϵ_{max} to the learning rate. The new learning rate update rule is:

$$\begin{aligned}
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_+ \quad if\ \tfrac{\partial E}{\partial w_{ij}}(t) * \tfrac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \bigwedge \epsilon_{ij}(t-1) \leq \epsilon_{max} \\
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_- \quad else
\end{aligned}$$

α	final learning error	test error	classification test error
0.	17.7	8.7	2
0.2	19.2	8.3	2
0.4	16.1	8.8	2
0.6	13.1	8.4	2
0.8	8.3	4.5	2

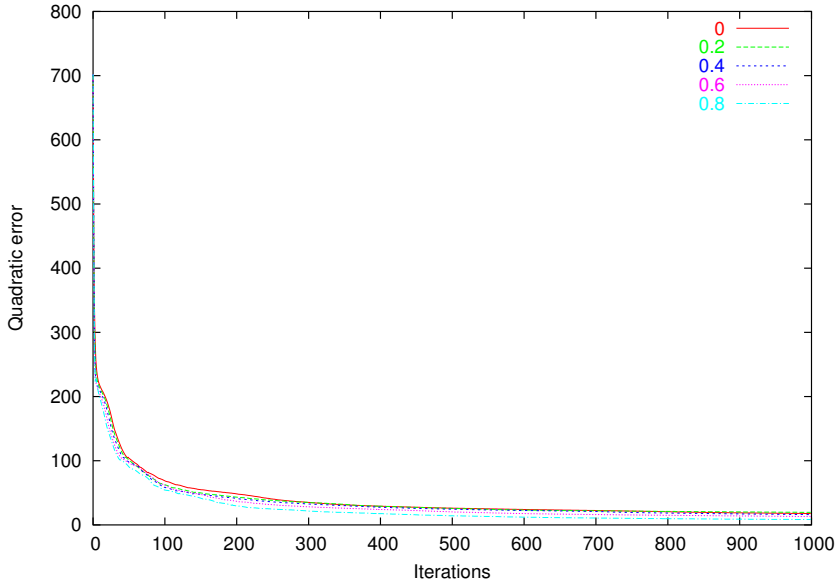


Figure 2.16: SuperSAB: Momentum influence

Recommended values for parameters

Existing papers generally recommend the same settings than for Silva-Almeida parameters, since these two algorithms are very close:

$$\epsilon_0 = 0.001$$
$$\alpha = 0.2$$
$$\eta_+ = 1.1$$
$$\eta_- = 0.7$$
$$\epsilon_{max} = 5.$$

Momentum influence (α)

Standard momentum values brings generally about no real improvement of the convergence speed (see Figure 2.16). It slightly speeds up the convergence in the stabilization and refinement, but not significantly. In addition, papers reports problems with too big momentums (26).

Maximum learning rate influence (ϵ_{max})

On classical tests like the one we present, it is difficult to see the influence of the ϵ_{max} value since learning rate limitation is needed in only few and limited cases. The principal observation we

ϵ_{max}	final learning error	test error	classification test error
1.	19.7	10.6	2
5	19.2	8.3	2
100	21.1	9.9	2
∞	$\emptyset$	$\emptyset$	$\emptyset$

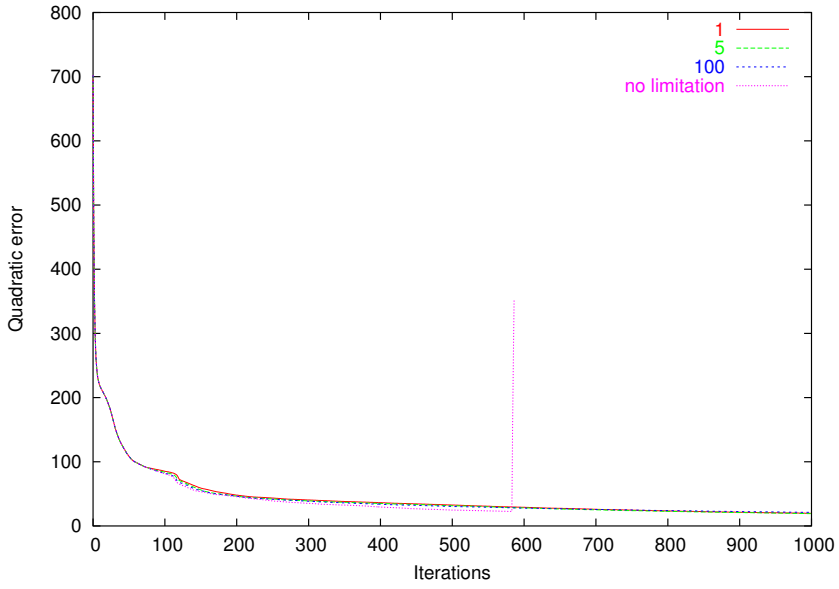


Figure 2.17: SuperSAB: Maximum learning rate influence

made is that this upper limit is necessary to get the algorithm working (see Figure 2.17). Without limitation, we did not get the algorithm working properly on any problem.

Initial learning rate influence (ϵ_0)

Like expected, initial learning rate especially influences the early steps of the algorithm by speeding it up. Over the recommended value of 0.001, we generally did not obtain significant improvement and even experienced critical stability difficulties in the early iterations (see Figure 2.18).

Learning acceleration rate influence (η_+)

With a learning acceleration rate equal to 1, the SuperSSAB algorithm can not accelerate (see Figure 2.19). It is only able to decrease its $epsilon_0$ initial learning rate. Values superior to 1 significantly accelerate the learning process, but at the risk of making the convergence unstable, especially during the early iterations. Then, values under 1.2 are recommended (generally recommended value: 1.1).

Learning deceleration rate influence (η_-)

As shown in Figure 2.20, a little learning deceleration rate can slightly slow down the convergence by decreasing too much the learning rate after a gradient sign change. Anyway, having a too big deceleration rate forces the learning rate to stay too close to its old value, so that it can generate local perturbations and result degradation.

ϵ_0	final learning error	test error	classification test error
0.00001	20.4	9.9	2
0.0001	19.9	8.9	2
0.001	19.2	8.3	2
0.01	21.8	10.5	2
0.012	24.8	11.5	3

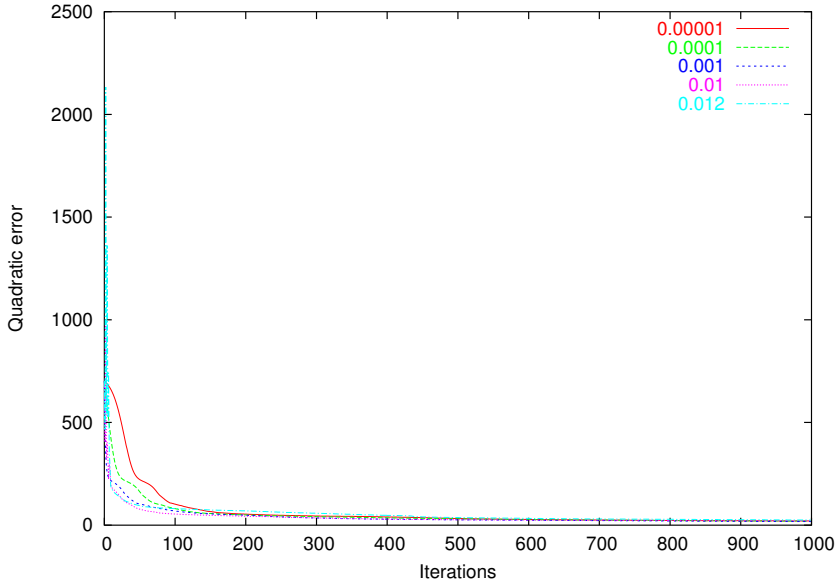


Figure 2.18: SuperSAB: Initial learning rate influence

η_+	final learning error	test error	classification test error
1.0	90.6	37.4	20
1.05	20.5	10.3	2
1.1	21.1	9.9	2
1.2	19.2	8.3	2
1.5	12.9	7.4	2
2	25.3	11.8	3

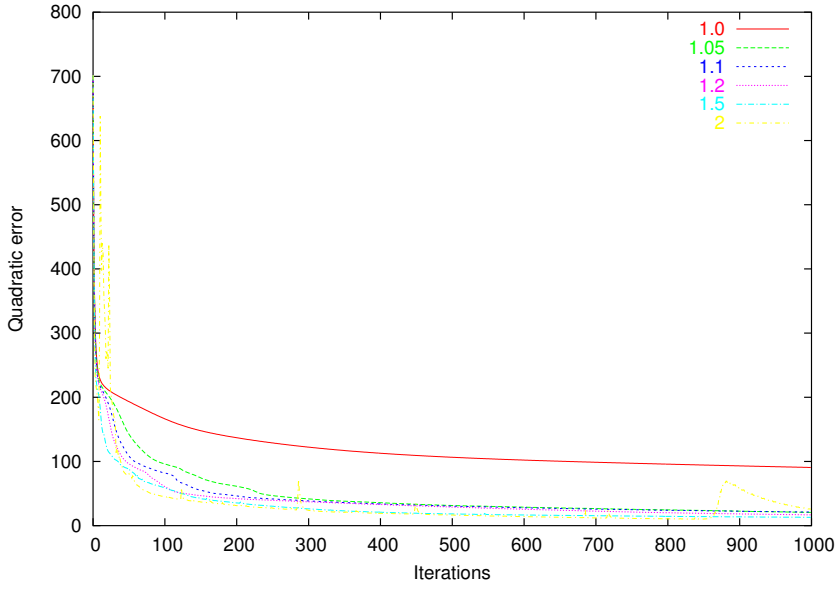


Figure 2.19: SuperSAB: Learning acceleration rate influence

η_-	final learning error	test error	classification test error
0.1	30.2	14.4	2
0.4	22.7	10.3	2
0.7	19.2	8.3	2
0.9	41.1	18.5	5

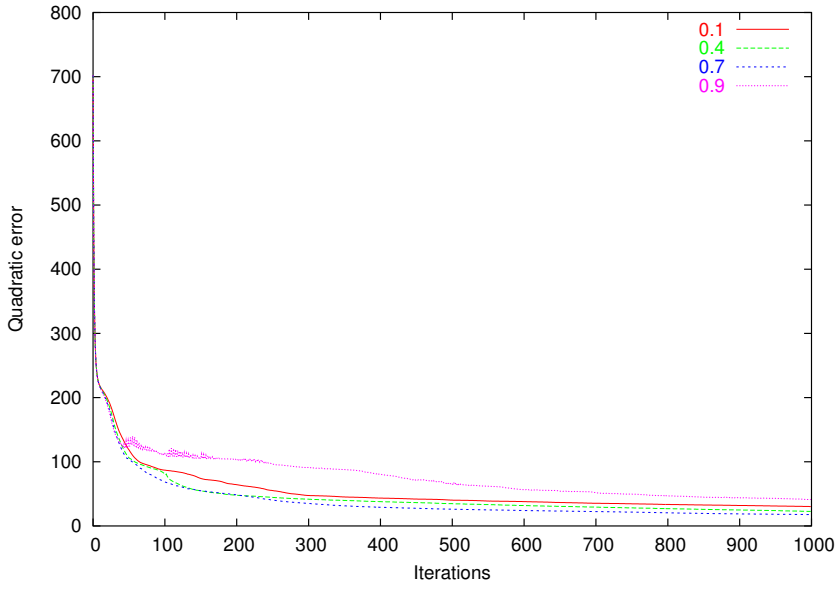


Figure 2.20: SuperSAB: Learning deceleration rate influence

Discussion

The original algorithm, as described in (26), does not include a maximum learning rate (η_{max}). In this version, the convergence was really chaotic, so that a proper η_{max} became unavoidable.

The SuperSAB algorithm remains very close to Silva-Almeida's approach. The local backtracking scheme it introduces appeared to improve very slightly the convergence speed without degrading the generalization power of the network. Tollenaere (26) reports some applications where its algorithm was among the best one. Unfortunately, both speed and generalization gains over Silva-Almeida were not significant for all our experiments.

2.4.5 Delta-Bar-Delta algorithm

Delta-Bar-Delta is another algorithm based on a local learning rate adaptation (1). This algorithm controls its learning rate bu observing the sign changes of an exponential averaged gradient. Instead of increasing the learning rate by multiplying it by an acceleration rate, Delta-Bar-Delta technique adds a constant step η_+. The learning rate is classically decelerated by a deceleration rate η_-.

$$
\begin{aligned}
&\epsilon_{ij}(0) = \epsilon_0 \\
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) + \eta_+ \quad if\ \tfrac{\partial E}{\partial w_{ij}}(t) * \tilde{\delta_{ij}}(t-1) > 0 \\
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) * \eta_- \quad if\ \tfrac{\partial E}{\partial w_{ij}}(t) * \tilde{\delta_{ij}}(t-1) < 0 \\
&\epsilon_{ij}(t) = \epsilon_{ij}(t-1) \quad else
\end{aligned}
$$

α	final learning error	test error	classification test error
0.	23.9	11.7	2
0.2	20.0	10.2	2
0.6	11.8	6.1	2
0.8	19.8	11.7	2

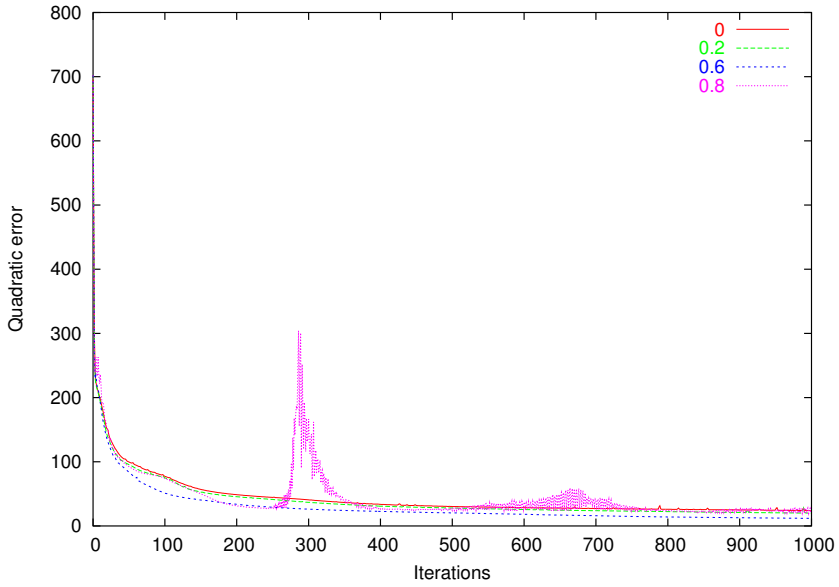


Figure 2.21: Delta-Bar-Delta: Momentum influence

Where $\tilde{\delta}_{ij}(t)$ represents the exponential averaged gradient:

$$\tilde{\delta}_{ij}(t) = (1-\phi) * \frac{\partial E}{\partial w_{ij}}(t) + \phi * \tilde{\delta}_{ij}(t-1)$$

Recommended parameters

$$\epsilon_0 = 0.001$$
$$\alpha = 0.2$$
$$\eta_+ = ???$$
$$\eta_- = 0.7$$
$$\phi = 0.7$$

Since η_+ is an acceleration step, it heavily depends on the application. Then, it is difficult to find a proper η_+. For the `thyroid` problem, we used $\eta_+ = 0.001$.

Momentum influence (α)

Increasing the momentum induces a small acceleration of the convergence process, but can also create critical stability problems (see Figure 2.21).

ϵ_0	final learning error	test error	classification test error
0.0001	20.9	11.0	2
0.001	20.0	10.2	2
0.01	24.1	11.5	3
0.02	26.2	13.3	3

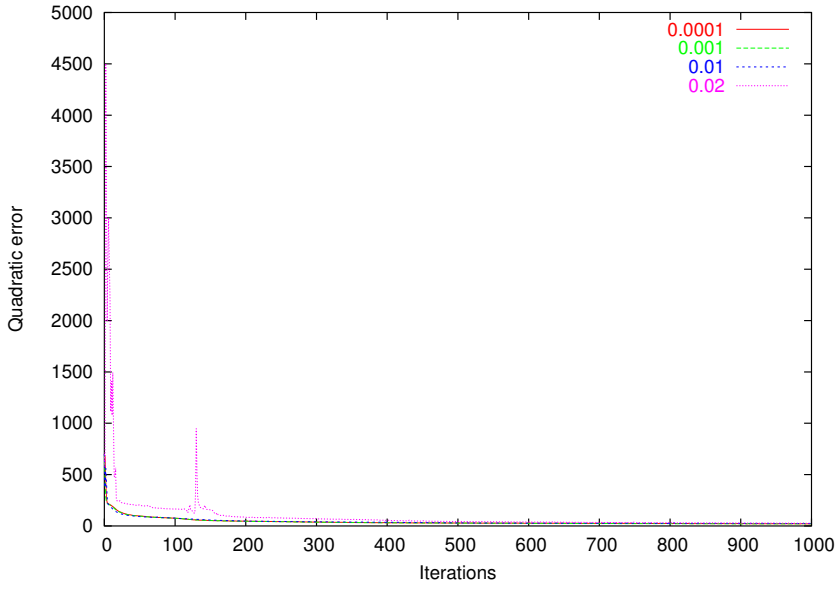


Figure 2.22: Delta-Bar-Delta: Initial learning rate influence

Initial Learning rate influence (ϵ_0)

Like every other auto-adaptive learning rate algorithm, the Delta-Bar-Delta algorithm slightly benefits from the growth of the initial learning rate. Our experimentations showed us that one should take care of too big values that can create important perturbations in the early iterations (see Figure 2.22). Since Delta-Bar-Delta acceleration is based on an acceleration step instead of an acceleration rate for the other algorithms, it seems more sensitive to too big values than Silva-Almeida and SuperSAB algorithms.

Learning acceleration step influence (η_+)

The η_+ parameter is probably the most difficult parameter to set because it is a fixed acceleration step. Figure 2.23 illustrates the influence of η_+ on our medical classification problem. We managed to improve the convergence speed by increasing η_+ up to a critical limit over which the algorithm became locally unstable. The problem is that this limit highly depends on the application so that a recommended η_+ can not be given (24).

Learning deceleration rate influence (η_-)

Figure 2.24 shows that the deceleration rate does not have a big influence under a certain limit. With too big values, the algorithm can not decelerate enough, so that important local perturbations appear.

η_+	final learning error	test error	classification test error
0.0001	18.2	9.0	2
0.0005	20.3	10.1	2
0.001	20.0	10.2	2
0.005	25.4	12.2	3

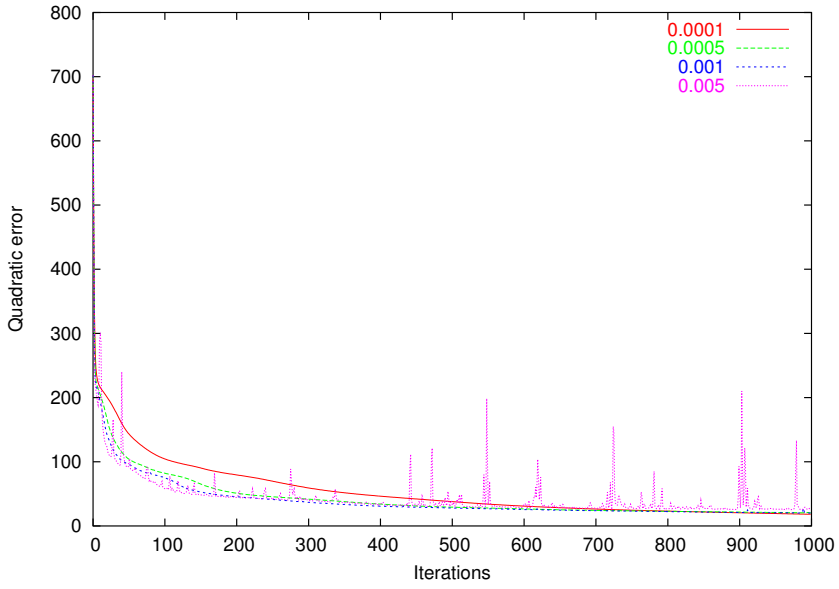


Figure 2.23: Delta-Bar-Delta: Learning acceleration step influence

Exponential average gradient parameter (ϕ)

The ϕ parameter represents the importance of the last exponential averaged gradient $\tilde{\delta}_{ij}(t-1)$ in the current exponential averaged gradient $\tilde{\delta}_{ij}(t)$. Basically, if ϕ is set to 0, $\tilde{\delta}_{ij}(t)$ is equal to the actual gradient and the Delta-Bar-Delta algorithm is equivalent to Silva-Almeida's approach. Unfortunately, our experimentations did not reveal the benefit of this exponential averaged gradient. Small values of ϕ (i.e. the current gradient is predominant) give very good convergence speeds, at least in the early steps. Increasing this value, we observed a very little melioration. When the last exponential gradient becomes too predominant, the algorithm becomes unstable in the stabilization and refinement process. If ϕ is set to 1, the averaged gradient $\tilde{\delta}_{ij}(t)$ is basically the last gradient, so that gradient sign changes are not correctly detected anymore. This considerably degrade the training results (see Figure 2.25).

Discussion

The exponential averaged gradient of the Delta-Bar-Delta technique did not prove to be a real improvement, even if we observed a better stabilization of the convergence on some tests. The results were quite good without outperforming Silva-Almeida and SuperSAB algorithms. One should also note that because of its unusual acceleration step η_+, this algorithm's setup highly depends on the final application. In practice, this results in some difficulties in obtaining the optimal behavior.

η_-	final learning error	test error	classification test error
0.2	20.4	11.6	3
0.4	17.1	9.2	2
0.6	19.8	10.5	2
0.8	24.8	12.0	3
0.9	35.2	15.4	3

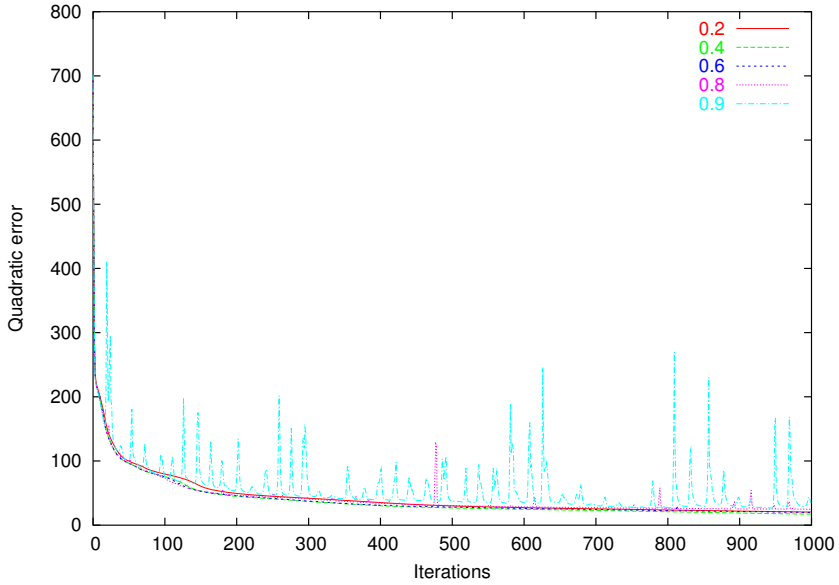


Figure 2.24: Delta-Bar-Delta: Learning deceleration rate influence

ϕ	final learning error	test error	classification test error
0.	26.4	11.6	2
0.2	26.5	9.2	2
0.4	24.5	10.5	2
0.6	22.0	12.0	2
0.8	21.1	11.4	2
0.9	26.1	13.2	3
1	90.6	37.4	20

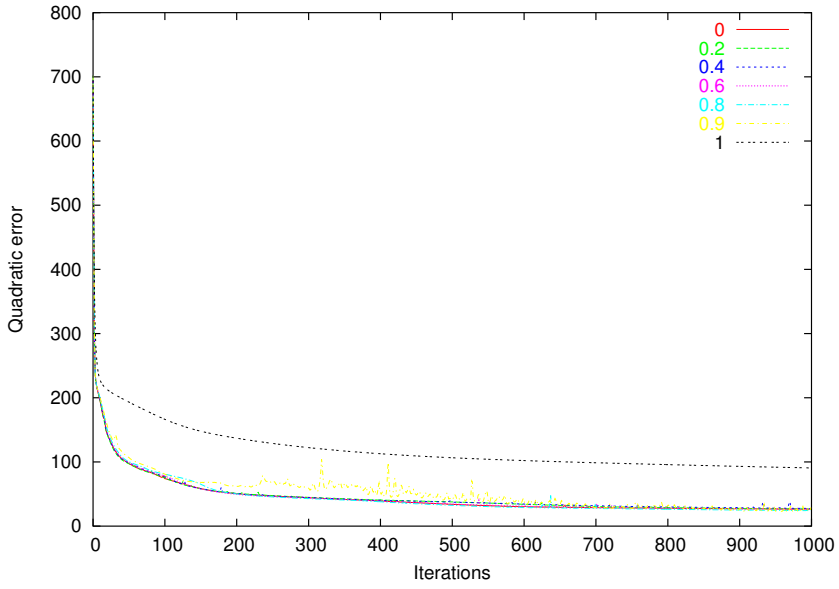


Figure 2.25: Delta-Bar-Delta: Exponential average gradient parameter

2.4.6 Resilient back-propagation algorithm

The Resilient backpropagation algorithm (RProp) can be described as an adaptive version of the Manhattan update rule (21). In contrast to all the other algorithms presented in this report, the Rprop algorithm is not influenced by the intensity of the gradient. It is based on an adaptive update step controlled by the sign changes of the gradient.

Step update rates (η_+ for acceleration rate and η_- for deceleration rate) are controlled by upper and lower limits (Δ_{max} and Δ_{min}) in order to avoid oscillations, over- and underflow problems. A small initial value for update step ($\Delta_{ij}(0)$) has to be fixed.

Basically, the update rate is multiplied by η_+ (> 1) when the gradient keeps it sign and multiplied by η_- (< 0) when its sign changes:

$$\begin{array}{ll}
\Delta_{ij}(t) = \Delta_{ij}(t-1) * \eta_+ & if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) > 0 \\
\Delta_{ij}(t) = \Delta_{ij}(t-1) * \eta_- & if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) < 0 \\
\Delta_{ij}(t) = \Delta_{ij}(t-1) & else \\
\Delta_{ij}(t) = \Delta_{max} & if\ \Delta_{ij}(t) \geq \Delta_{max} \\
\Delta_{ij}(t) = \Delta_{min} & if\ \Delta_{ij}(t) \leq \Delta_{min}
\end{array}$$

Weights are then updated the following way:

$$w_{ij}(t+1) = w_{ij}(t) - sign(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t)$$

Optimizations of the original algorithm

Riedmiller and Braun (21) proposes a local backtracking scheme for improving the Rprop algorithm. If a gradient sign change is detected, the update step is decelerated as usual, but the last weight update is reverted and the next adaptation of the update step is skipped. This can be implemented as follows:

$$\begin{array}{ll}
w_{ij}(t+1) = w_{ij}(t) - sign(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t) & if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\
w_{ij}(t+1) = w_{ij}(t) - \Delta w_{ij}(t-1);\ \frac{\partial E}{\partial w_{ij}}(t) = 0 & else
\end{array}$$

In the case of a gradient sign change, we set the gradient value to 0. If we assume that `sign(x)` returns 0 if x is null, this skips the next update of Δ_{ij}. This local backtracking is particularly efficient for difficult, irregular problems (21).

Igel and Hüsken (14) proposes another small optimization to the Rprop algorithm. He introduces a global backtracking test to soften the local backtracking. In the case of a gradient sign change, the last weight update is reverted only if the global energy has grown up. Thus, it allows the algorithm to accept weight changes even if a gradient sign change occurred. Thanks to this technique, Igel reports interesting improvement of the convergence speed. Unfortunately, our experimentations did not reflect such impressive melioration.

Recommended values for the parameters

Literature recommends the following parameters:

$$\begin{array}{l}
\Delta_{max} = 50. \\
\Delta_{min} = 0.0001 \\
\Delta_0 = 0.0125 \\
\eta_+ = 1.2 \\
\eta_- = 0.5
\end{array}$$

Δ_{min}	final learning error	test error	classification test error
0.00001	6.0	4.3	1
0.0001	5.5	3.3	1
0.001	15.2	8.5	2
0.01	55.7	23.9	10
0.05	176.3	61.2	16

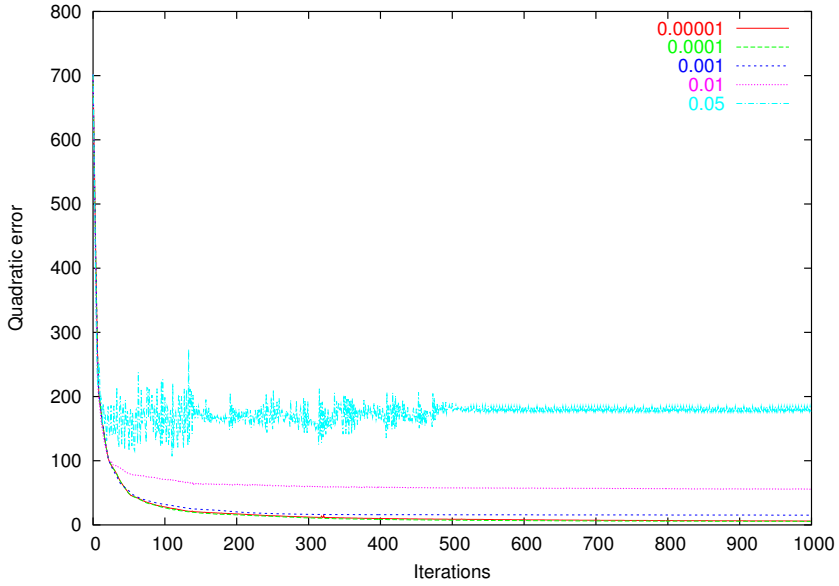


Figure 2.26: Rprop: Minimum update rate influence

When not indicated, these parameters are used for the following tests.

Maximum update step influence (Δ_{max})

In all the tests we made, the Rprop algorithm kept a remarkable stability w.r.t its update step. Then, it was extremely rare that it had to be limited. At the contrary, a reasonably big Δ_{max} (for example, the standard value 50) is necessary for the algorithm to keep its acceleration properties. We also observed that decreasing this value could increase the stability of the algorithm for some "difficult" problems.

Minimum update step influence (Δ_{min})

The same way than for Δ_{max}, the Rprop algorithm rarely reached so little update step that a Δ_{min} was necessary to maintain some dynamics. Figure 2.26 shows that a very small Δ_{min} value is valuable for continuing to converge when refining the final result. Higher Δ_{min} values will maintain an artificial agitation that will result in worse results or even in critical instability.

Update rate initialization influence (Δ_0)

Figure 2.27 illustrates the influence of the initialization value of the update rate on the convergence speed. Δ_0 especially affects the initial convergence speed: with increasing Δ_0, learning

Δ_0	final learning error	test error	classification test error
0.0001	5.1	5.5	1
0.001	6.1	5.8	1
0.01	7.3	6.2	1
0.1	9.8	8.0	1

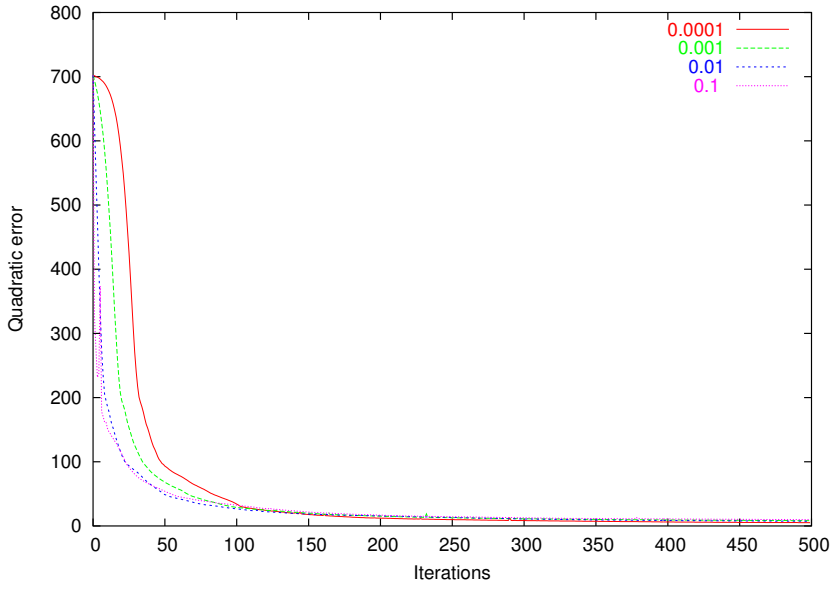


Figure 2.27: Rprop: Update rate initialization influence

error decreases more quickly at the beginning, but too big values endanger the stability of the early steps of the algorithm. With simple, regular problems, it was possible to increase considerably Δ_0 in order to improve the initial convergence speed. Anyway, recommended value was always close to the optimum.

Acceleration rate influence (η_+)

η_+ has naturally an important influence on the convergence speed, as shown in Figure 2.28. With η_+ equal to 1, the algorithm can not accelerate. It keeps its initial learning rate Δ_0 and only adapt it by decreasing it. With values greater than 1, the algorithm accelerates significantly. Increasing the acceleration rate quickly comes with no gain, since the algorithm is accelerating too much. Then, deceleration (and backtracking if available) iterations occur immediately to moderate the update rate. The algorithm proved to be remarkably stable w.r.t to the growth of η_+, especially in its backtracked versions. Since increasing η_+ quickly comes with about no gain, it is recommended to keep standard value (1.2) in order to maintain a perfect stability.

Deceleration rate influence (η_-)

Figure 2.29 sums up the typical observations we made about η_- influence. Too small values slightly slow down the convergence by decreasing too much the update rate after a gradient sign change. At the contrary, too big values endanger the convergence by forcing the update rate to stay too close to its old value.

η_+	final learning error	test error	classification test error
1	54.3	23.7	8
1.1	12.2	6.3	2
1.2	11.3	6.6	2
1.5	9.9	6.1	2

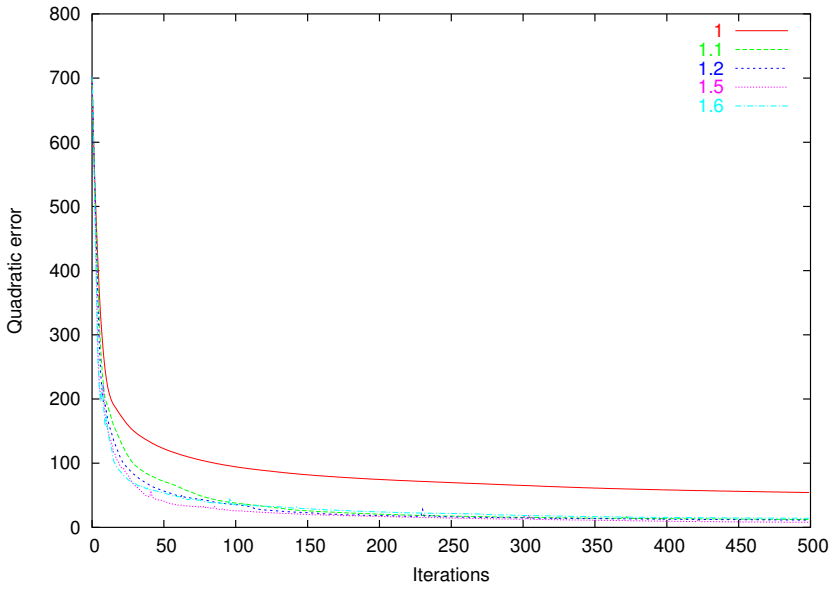


Figure 2.28: Rprop: Acceleration rate influence

η_-	final learning error	test error	classification test error
0.1	8.2	5.4	1
0.5	6.1	5.8	1
0.95	10.8	8.8	3

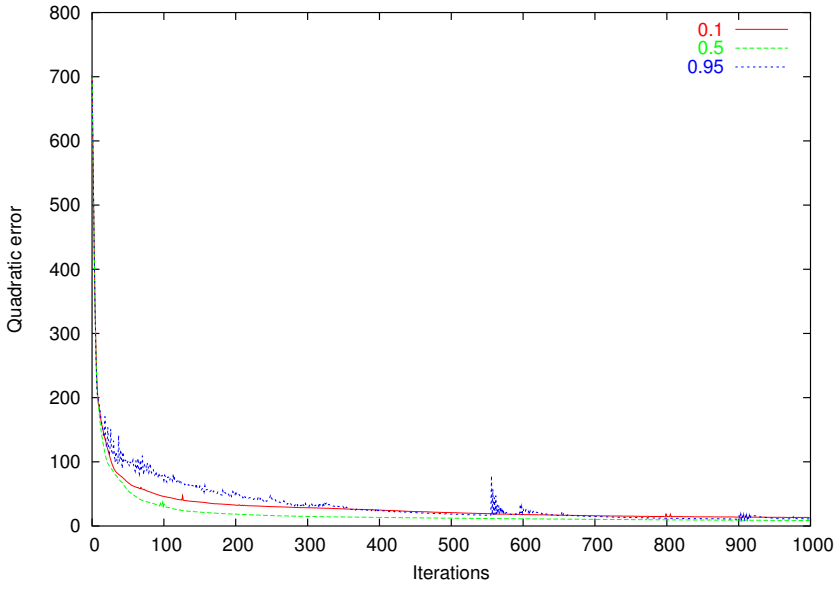


Figure 2.29: Rprop: Deceleration rate influence

Discussion

Basically, this is the algorithm that comes with the simplest idea: accelerating a momentum step, totally independently of the gradient intensity, only driven by the gradient sign changes. Finally, this simple idea makes the Rprop the best first-order, auto-adaptive method in terms of convergence speed and robustness w.r.t. its parameters. In about all our tests, we obtained the quickest convergence with the Rprop without modifying any of its "standard" parameters and without any additional technique. Unfortunately, this convergence performance can make the over-fitting phenomenon more visible (see Section 2.3.1). The optimizations on the backtracking scheme (14) give an interesting way of stabilizing the results without managing to avoid totally over-fitting. The best generalization network is obtained very quickly, so that over-fitting starts after very few iterations. Then, we recommend to use this algorithm in collaboration with one or several of the techniques presented in Section 2.3.

2.4.7 Quick back-propagation algorithm

This algorithm was first proposed by Fahlman (9). It is a second-order method, based heuristically on Newton's technique. Basically, this algorithm relies on a second-order approximation of the optimal weight step:

$$\tilde{\alpha}(t) = \Delta w_{ij}(t) = \frac{\frac{\partial E}{\partial w_{ij}}(t)}{\frac{\partial E}{\partial w_{ij}}(t-1) - \frac{\partial E}{\partial w_{ij}}(t)} \Delta w_{ij}(t-1)$$

Fahlman (9) develops the assumptions that make this approximation heuristically valuable. In real situations, this update rule makes the algorithm particularly unstable. Some control is then required.

First, if the step computed by this formula is too large, infinite or uphill on the current gradient, this momentum has to be limited. Schiffmann et al. (24) proposes to control the momentum term as follows:

$$\begin{array}{ll} \alpha(t) = \alpha_{max} & if\ \tilde{\alpha}(t)\ infinite \\ & \bigvee \tilde{\alpha}(t) > \alpha_{max} \\ & \bigvee \tilde{\alpha}(t) * \Delta w_{ij}(t-1) * \frac{\partial E}{\partial w_{ij}}(t-1) > 0 \\ \alpha(t) = \tilde{\alpha}(t) & else \end{array}$$

A learning rate is also necessary to start the training or restart it after a gradient sign change:

$$\begin{array}{ll} \epsilon_{ij}(t) = \epsilon_0 & if\ \frac{\partial E}{\partial w_{ij}}(t) * \Delta w_{ij}(t-1) < 0 \\ & \bigvee \Delta w_{ij}(t-1) = 0 \\ \epsilon_{ij}(t) = 0 & else \end{array}$$

A weight decay (see Subsection 2.3.4) can also be necessary to get the Quickprop algorithm working properly.

Recommended values for the parameters

$$\begin{array}{l} \alpha_{max} = 1.75 \\ \epsilon_0 = 0.001 \\ decay = 0.00001 \end{array}$$

When not indicated (i.e. when the parameter is not the one studied), these values are used for the following tests.

decay	final learning error	test error	classification test error
0.000001	12.3	8.3	2
0.00001	9.7	5.4	2
0.0001	14.0	7.0	2
0.001	11.5	7.1	2
0.01	17.4	10.0	2
0.1	31.4	15.1	5

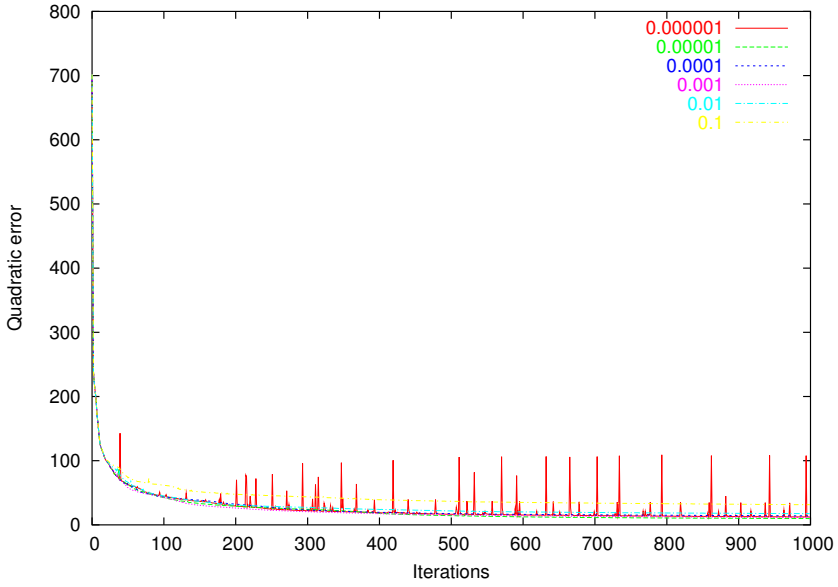


Figure 2.30: QuickProp: Weight decay influence

Weight decay influence ($decay$)

The need for a weight decay has been verified. In fact, with lots of tests, no convergence has been obtained without weight decay. Figure 2.30 presents convergence curves obtained with different values of decay. It confirms the relevance of the 0.0001 standard value. It also shows that a too small decay value results in the instability of the algorithm and that a too big value degrades results by maintaining artificially a too high gradient.

Learning Rate influence (ϵ_0)

All our experimentations have confirmed the standard value for the learning rate. Small values have little impact since quickprop algorithm is mainly based on the momentum approximation. At the contrary, big values endanger the stability after a zero update and considerably degrade the results. See Figure 2.31 for typical results.

Maximum momentum influence (μ)

Constraining the momentum under small values (< 1), the convergence is considerably slowed down. With too big values (> 2) the algorithm becomes locally unstable (see Figure 2.32). The standard value of 1.75 was valuable for about all our experimentations.

ϵ_0	final learning error	test error	classification test error
0.0001	8.5	6.9	2
0.001	9.7	5.4	2
0.005	33.8	15.7	5
0.008	40.6	27.5	4

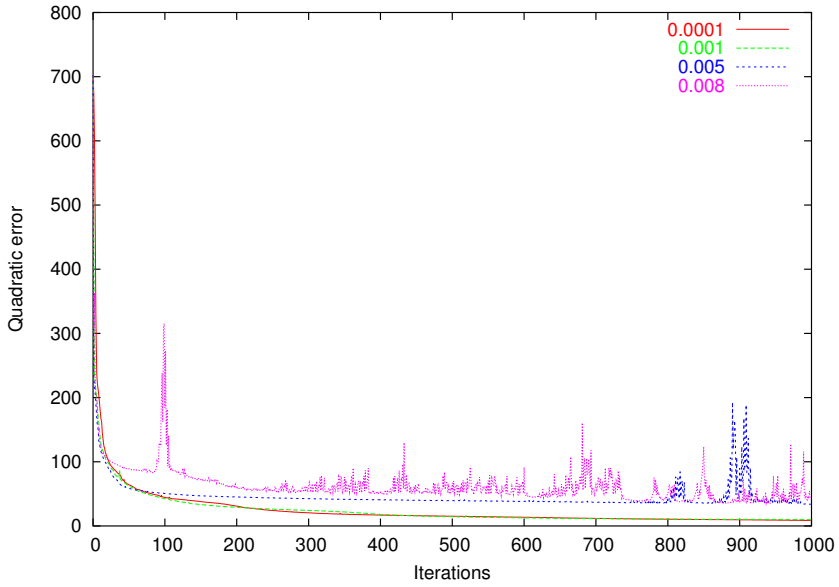


Figure 2.31: QuickProp: Learning rate influence

Discussion

Quickprop's performance is generally close to Rprop's results, without outperforming them. Quickprop's adaptation rule, based on an loosely second-order approximation of the best momentum, is much more complex than the adaptive step of the Resilient back-propagation. This approximation is even unstable, so that lots of control tests must be done to ensure the convergence. This algorithm is more difficult to setup and less robust w.r.t. its parameters. Moreover, it generally needs additional techniques like weight decay to work properly. Finally, all these practical drawbacks make Quickprop less convenient than Rprop. Though, see (9) for some applications where this algorithm is worth to be preferred to Rprop.

2.4.8 Cascade correlation algorithm

Fahlman and Lebiere have presented a new learning architecture called cascaded correlation algorithm in (10). This algorithm comes with an entire training scheme that starts with a minimal network with only one hidden unit. Then, the algorithms tries to add new hidden units and trains them using one of the algorithms presented before. With such an adaptation of the network topology during the training process, this algorithm has proved to be one of the most efficient one, for both training speed and generalization (10; 24) but is not directly comparable to the other algorithms presented. This algorithm

This algorithm can also be viewed as a solution to the problem of choosing the size of the network (see Section 2.3.2) but some expertise is needed to get the algorithm converging to a relatively small topology (10).

μ	final learning error	test error	classification test error
0.5	72.3	30.6	17
1	12.0	5.5	2
1.75	9.7	5.4	2
3	14.3	7.2	3

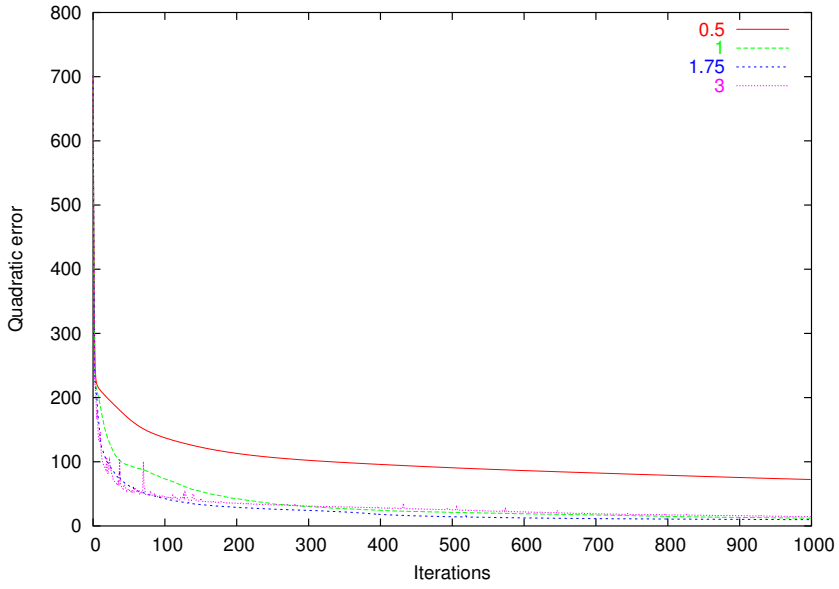


Figure 2.32: QuickProp: Maximum momentum influence

2.4.9 Other existing algorithms

There exist many other ways to speed up MLP learning process. This study focussed on locally adaptive algorithms, but algorithms based on a global adaptation of the learning rate have proved to give good results, without outperforming the former ones. We can cite angle driven and conjugate gradient methods, studied in detail in (24).

2.4.10 Comparison and discussion

Learning speed

If we except the cascade-correlation algorithm not studied in this report, the resilient back-propagation appeared to be the quickest algorithm for all the tests we performed. The convergence speed of the quick back-propagation was almost always comparable to Rprop one. The other algorithms studied were slightly slower but remained significantly faster than classical batch back-propagation.

Results quality and generalization power

All the auto-adaptive algorithms presented significantly outperform the standard batch back-propagation in terms of convergence speed. A more interesting result is the way they also lead to good generalization.

Algorithm	final learning error	test error	classification test error
Stochastic back-propagation	170.2	68.7	54
Batch back-propagation	85.3	35.2	19
Delta-Bar-Delta	22.6	12.5	3
Quick back-propagation	9.7	5.4	2
Resilient back-propagation	4.9	4.7	1
Silva-Almeida	16.4	8.3	2
Super Self-Adaptive back-propagation	17.7	8.7	2

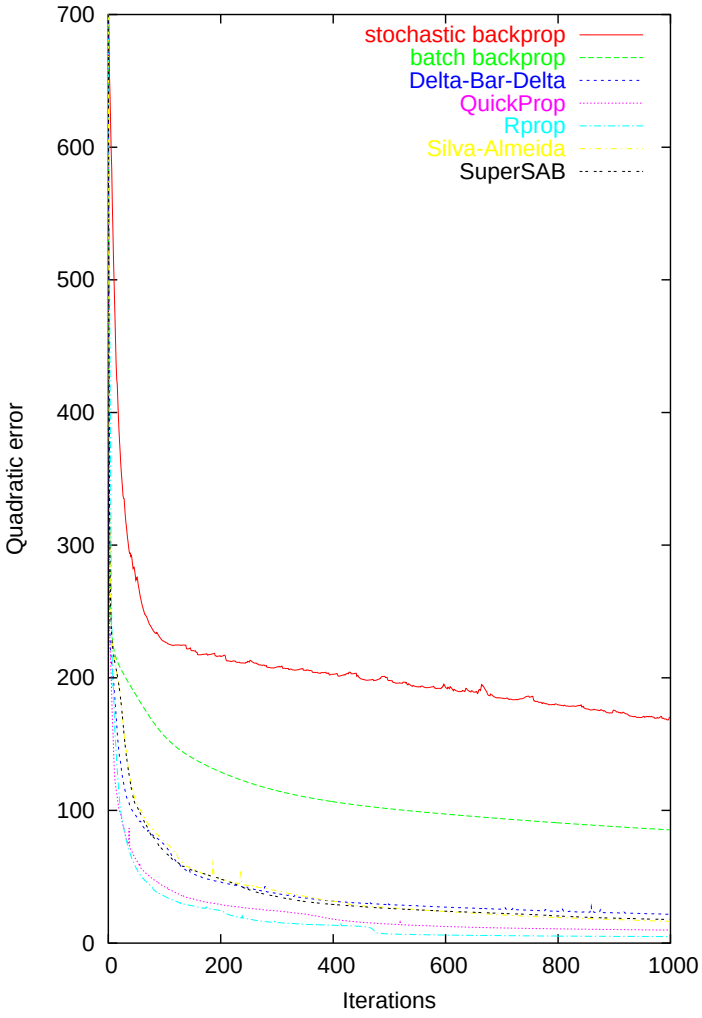


Figure 2.33: Comparison of the different algorithms

The tests published are fully representative of the general observations we made. For all the locally adaptive algorithms tested, quadratic test error and classification test error were correlated to the final learning error. With appropriate settings and limited number of iterations, none of these algorithms engenders over-fitting phenomenons. Since all these algorithms aim to minimize the training error, one should understood that they will inevitably lead to over-fitting if they run too many iterations. Like it was already explained in Section 2.3.1, this category of algorithms will naturally tend to the best generalization before beginning over-fitting. Different techniques, already exposed in Section 2.3, have to be used in collaboration with these algorithms in order to benefit from their learning performance and keep a good generalization power.

Very little difference has been observed between the algorithms w.r.t the final generalization power of the network. Anyway, articles report that Silva-Almeida and SuperSAB algorithms can sometimes lead to better generalization. (24; 25; 26).

Ease of use and robustness to variations of parameters

Since all the algorithms presented here are auto-adaptive, they all present good stability properties. Anyway, some of these algorithms were more difficult to setup than others. It was the case of the Delta-Bar-Delta technique that uses an acceleration step instead of an acceleration rate. The Quickprop algorithm was also difficult to setup and required some additional techniques like weight decay to work properly. Since it is based on a heuristic approximation, one should take care of setting the good control values (α_{max} and $decay$) for maintaining the stability of the learning process when the approximation fails. Finally, the resilient back-propagation appeared to be the most stable algorithm w.r.t. its parameters. Indeed, it is the only one presented here that relies entirely on an adaptive learning step independent from the gradient intensity. With the standard values proposed in this chapter, the algorithm worked fine on all the tests we performed. It was even possible to significantly accelerate the convergence for some highly regular problems.

Chapter 3

Application to image compression

3.1 Framework

There are two types of compression algorithm: loss-less data compression, that preserves all the information, and lossy data compression, that throws away some non-essential information. The choice of such algorithms depends on the considered data type. Considering binary or text files, where each bit of information is essential, a loss-less compression (such as Huffman's) must be used. On the other hand, when manipulating sound, image or video data, which are already lossy digitalizations of analogue phenomena, a lossy compression algorithm may be prefered.

Images are noisy signals where an important quantity of information can be pruned without significant appearance damage. From then on, we try to reduce image's entropy defined as:

$$H = -\sum_i \log_2(p_i) x_i$$

where p_i refers to the probability of appearance of the x_i pixel. Entropy represents the quantity of information (in terms of minimum number of bits needed to encode an image). Sparse pixels (the noise) represent a significant part of that quantity. The main goal is to delete those pixels to reduce image's entropy and thus compress it with data loss but without significant quality loss.

Various destructive compression methods exist. For example, we can quote the discrete cosine transform (DCT) used in *jpeg* file format or neural network approaches. This last framework will be ours in the rest of that study.

3.2 Data analysis

In this section, we will analyze data that are contained in an image.

Consider the cut of and $N \times M$ image into n blocks of $h \times w$ size:

$$X = \begin{pmatrix} x_{1,1} & \cdots & \cdots & \cdots & x_{1,h\times w} \\ \vdots & \ddots & & & \\ \vdots & & x_{i,j} & & \\ \vdots & & & \ddots & \\ x_{n,1} & & & & x_{n,h\times w} \end{pmatrix}$$

where $x_{i,j}$ is the j-th pixel of the i-th block. Section 3.2.4 will study the interest and the importance of such cutting up.

In the following, an image is considered as n observations of a $h \times w$ dimension random variable.

3.2.1 Linear correlation

For a convenient illustration, the image is cut in blocks of only two pixels, but the results may be generalized to upper dimensions. In order to determine whether it exists a link between two adjacent pixels in an image, we represent each observation i as a point with coordinates (x_i, y_j) in a Cartesian basis. The points cloud' shape is given as follows:

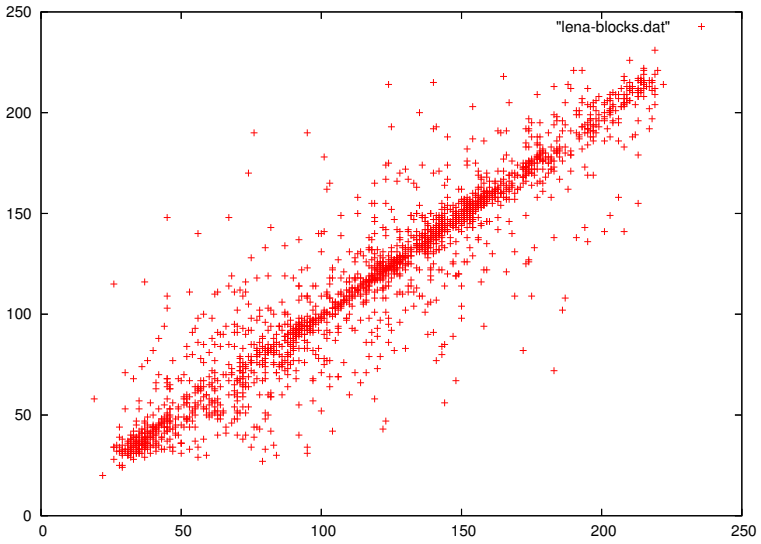


Figure 3.1: A cloud of 1×2 blocks

We notice that we are in the case of a strong linear correlation between block's pixels ; we admit that this tendency remains true for dimensions greater than 2.

All observations are located near the equation's line $y = x$. This linear correlation can be explained by the fact that neighbor pixels within a block have a very close grey level, which tend to be equal. Observations that are far from that line are blocks containing noisy pixels.

To study more precisely this linear relation between pixel of a same block, we can represent the linear correlation matrix. We consider a 64 dimension space (image is cutting up into 8×8 blocks). Figure 3.2 represents this matrix.

The height represents the absolute value of linear coefficient (timed by 100) between two components of the random variable (thereby, between two pixels of a block). We notice a periodic variation shaped as *waves*. Those *waves* can be explained by the fact that two nearby pixels tend to have the same level of intensity. The more distant pixels are to each other, the less their intensity are similar. *Pick* are the consequence of a line by line pixel traversal process during the block's construction.

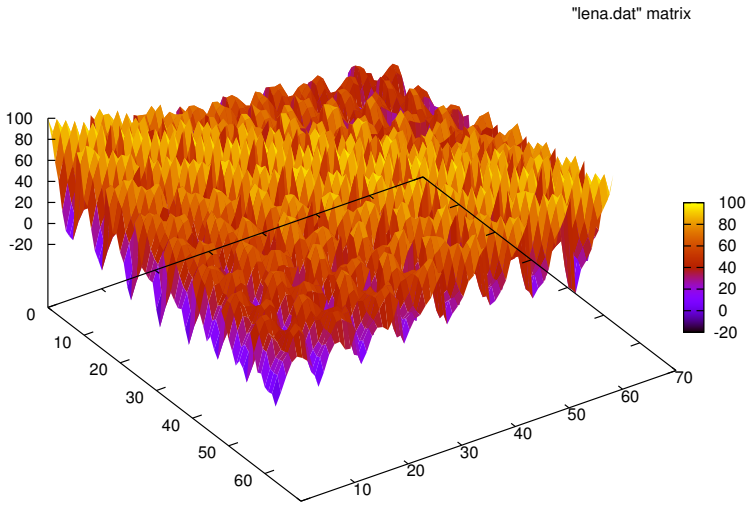


Figure 3.2: linear correlation matrix between pixels

3.2.2 Principle components analysis

In this section, we consider an image cut up into 4×4 blocks. From then on, each observation is an element of $\mathbb{R}^{16}$.

In the last paragraph, we observed that pixels have a strong linear correlation, that is to say, we could foresee the intensity of a pixel by considering the intensity of other pixels within a block.

The objective of a principle components analysis is to obtain an approximative representation of the n blocks cloud into a sub-space of smaller dimension. More precise information on this method can be found in (23). The goal is to found a new basis such that it axes *explain* better the cloud of blocks. These axes are the eigen vectors of X's variance-covariance matrix. The quantity defined as:

$$Z_i = \frac{\lambda_i}{\sum\limits_j \lambda_j} \times 100$$

Where λ_i is the i-th eigenvalue, defined the percent of explained variance of the blocks cloud. The array below gives values of Z for several eigen values:

	Eigen value	Z (%)	cumulated Z (%)
1	3064.100	74.807	74.807
2	478.837	11.690	86.497
3	172.723	4.216	90.714
4	129.585	3.163	93.878
⋮	⋮	⋮	⋮
13	4.676	0.114	99.827
14	3.395	0.082	99.910
15	2.395	0.058	99.968
16	1.280	0.031	100.000

Only three components are needed to represent more than 90% of cloud's inertia. The more we consider a large number of axes, the more we represent the whole data.

3.2.3 Back to the compression issue

The main objective of a compression with data loss is to decrease the quantity of information; in an image, it amounts to prune noise. In our framework, noise is located into blocks that don't have a homogeneous intensity (intensity pixels are very far from each other).

To understand well the interest of a PCA in pruning noise, let consider figure 3.1 where we worked on 1×2 blocks. The diagram below shows the two axes of the new base q_1 and q_2:

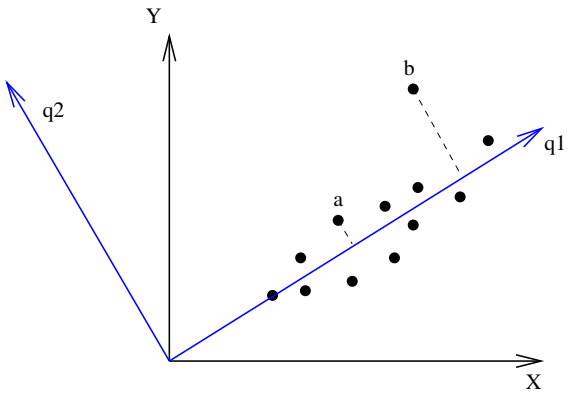


Figure 3.3: blocks cloud's projection upon the first eigen vector

Each point of coordinates (x_i, y_j) in the Cartesian basis can be written as :

$$x = < x|q_1 > \times q_1 + < x|q_2 > \times q_2$$

If we consider just one principal component q_1, x will be orthogonally projected upon q_1. Therefore, blocks near q_1, meaning blocks that are homogeneous, undergo minor changes (point A). On the other hand, points that are far from q_1 (noisy blocks) will turn to homogeneous blocks (point B). For that reason, considering only a small number of principle axes will contribute towards turn blocks into homogeneous ones and then, prune noise and reduce significantly the quantity of information contained into the image.

3.2.4 Size of blocks

Notice that the more size's block is smaller, the more they become homogeneous (within variance decreases). The array below presents several results of explained variance of the blocks cloud by the first axis, regarding different sizes' block. (the input image is a 64×64 one).

size's block	Z (%)
64×64	undefined
32×32	17.376
16×16	28.834
8×8	51.481
4×4	74.807
2×2	89.920
1×2	96.880
1×1	100.000

In the data compression framework, the case where blocks are compounded with only one pixel isn't very useful because there is only one axis, which resume the whole information, and it cannot be deleted (the ability of compression is null). Concerning the other cases, the more the blocks are smaller and the more the first axis represents blocks cloud better. However, the compression rate, which is proportional to the reduction of dimension data' space, decrease with size's block: It is more interesting going from a 64 dimension space to single one than going from a 2 dimension space to a single one. Indeed, if we consider only one principle axis for a 32×32 cutting up image, it allows us to hardly compress image ($\mathbb{R}^{1024} \leftarrow \mathbb{R}$). Nevertheless, this compression dramatically damages the image (only 17% of information is remained). Thereby, it exists a compromise between size's blocks and the desired compression rate. Unfortunately, there are only empirical methods to estimate these parameters.

Notice that size's blocks also influences on the ability of generalization. Generalization is presented in further section.

3.3 Multi Layer Perceptron for image compression

3.3.1 Network's topology

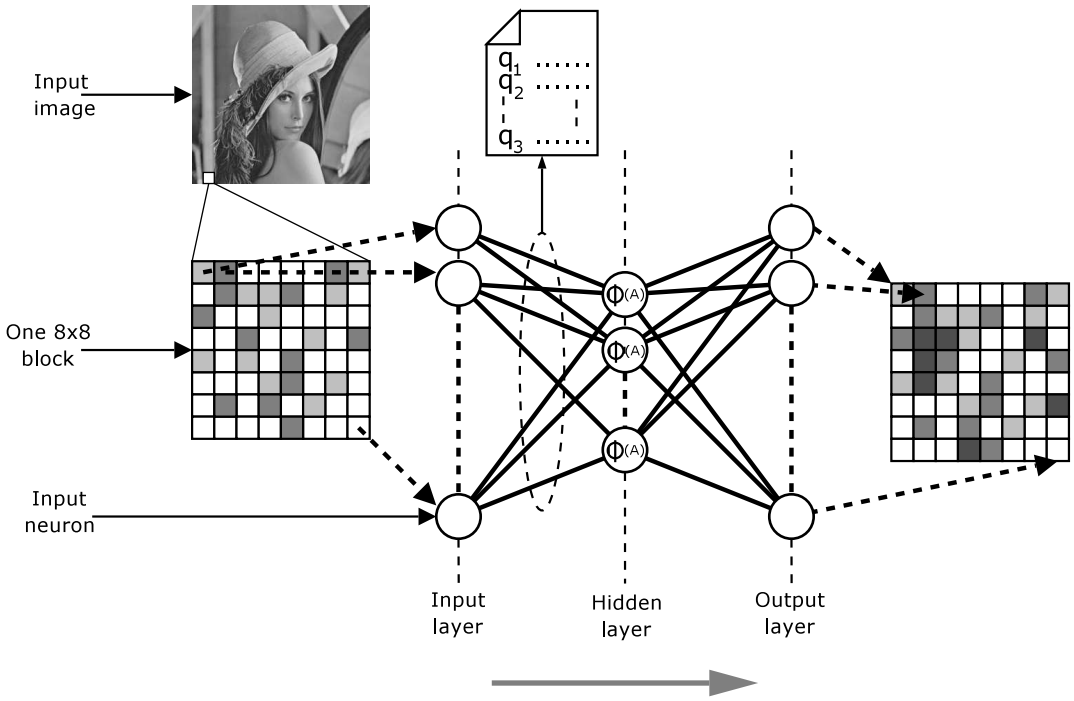


Figure 3.4: Topology of the network used.

We use a two-layer network. The layers are organized as follows:

- each neuron of the input layer corresponds to a pixel of the input block (thus 64 neurons for 8x8 blocks);
- the neurons of the hidden layer represent the components on which the input blocks will be projected;
- the output layer is similar to the input layer, except than its neurons corresponds to the decompression of the compressed block.

The weights towards the hidden layer correspond to the principal components (q_i). In the compressed image file, we will store only the projection of the blocks on the principal components ($x_i.q_i$). On the header of the file, the q_i vectors are stored to permit the decompression. Thus, the number of q_i specifies the compression ratio.

The weights towards the output layer perform the decompression, and the error is then evaluated by comparing the resulting pixels with the input, using the usual quadratic error.

We use the standard activation rule $A_i = x_i.q_i$, with x_i the input value vector and q_i the weight vector for the neuron i. Basically, the identity function is a good candidate for the output function Φ and is used in the remaining part of this report.

3.4 Experiments

In the following tests, we consider this typical use case of image compression:

- greyscale photo images;
- 8x8 blocks;
- compression ratio close to 8 (which approximately corresponds to 8 neurons on the hidden layer).

We experimented several aspects of image compression with neural networks. First, we describe the results we got for the learning phase by studying the influence of the initialization, the back-propagation algorithm, the number of neurons on the hidden layer and the size of the blocks. Then the results obtained by applying the trained network on another image are discussed.

3.4.1 Learning

Initialization

To compare the different initialization algorithm, we use the resilient propagation algorithm. Similar results were observed with other algorithms. Three initializations are discussed:

- random initialization: all the weights are initialized randomly between -0.3 and 0.3;
- initialization of some weight vectors of the first layer (two or three) with PCA;
- initialization of some weight vectors of the first layer (two or three) with PCA and initialization of the same vectors on the second layer with Gauss-Newton.

Since the network will converge to the PCA of the image blocks, initializing it with a PCA give the optimal weight values. It should be noticed that initializing the second layer weights also is essential. If they are initialized randomly, the back-propagation algorithm will modify both the weights of the second and the first layer during its gradient descent. Hence, the well initialized values will be destroyed at the beginning of the process. This will even slow down the initial convergence rate, since the values of the initialized weights will be important (probably higher than $[-0.3; 0.3]$) and structured.

These statements are confirmed by Figure 3.5, which compares the convergence speed of the three algorithm by analyzing the error in function of the number of iterations. The PCA+Gauss-Newton initialization is by far the more efficient, requiring only a few iterations to reach the optimal error.

Back-propagation algorithms comparison

The results observed for image compression are almost the same as in Section 2.4.10, as shown in Figure 3.6. The resilient back-propagation remains the best one, this is why we chose it for the experiments of this chapter.

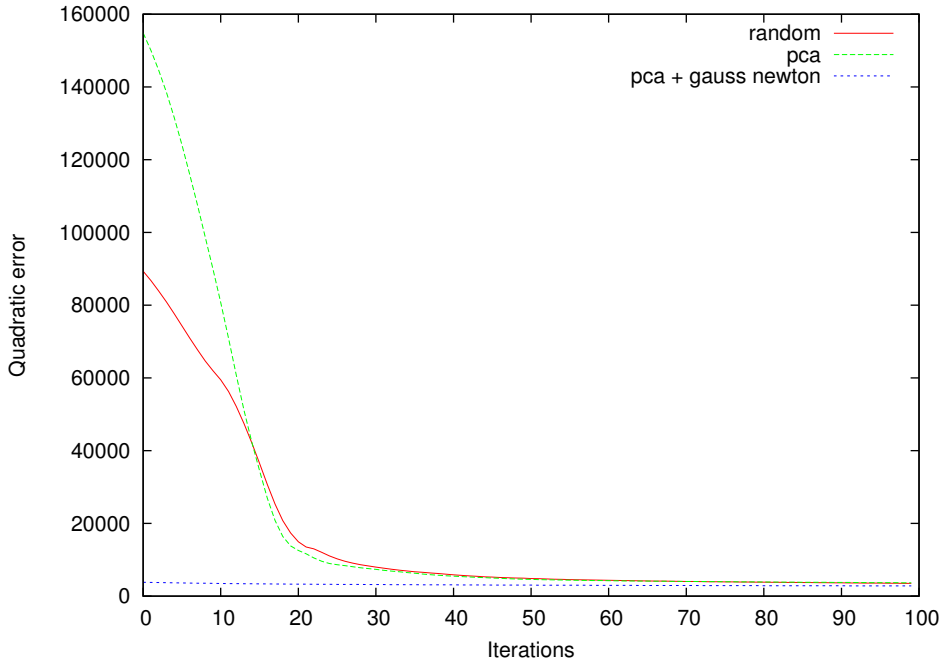


Figure 3.5: Comparison of initialization algorithms

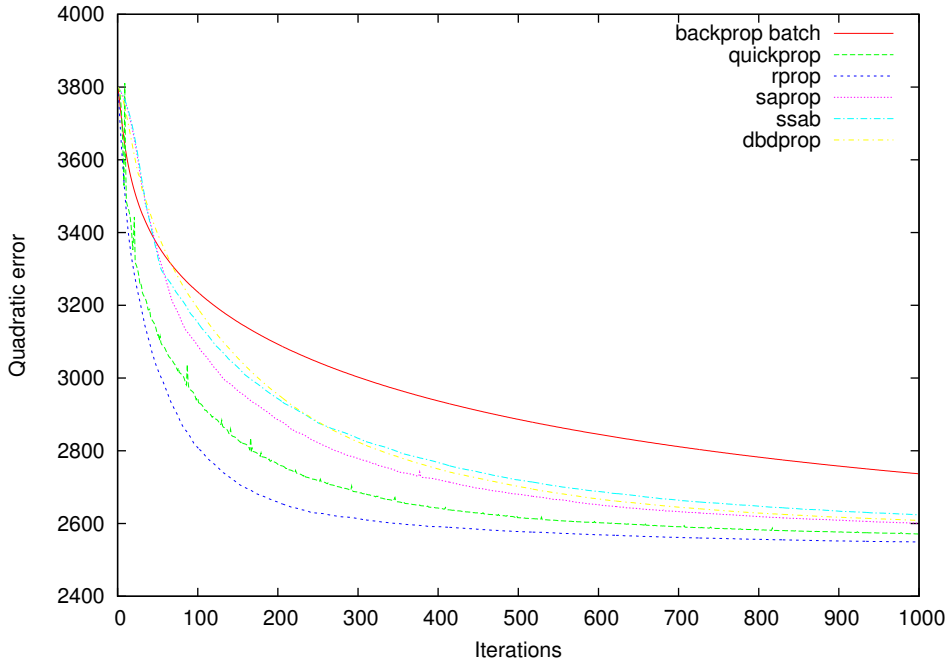


Figure 3.6: Comparison of learning algorithms

Influence of the number of principal components

On the one hand, the more principal components (neurons of the hidden layer) we choose, the more we get a good image quality. On the other hand, the less principal components we keep the more we get a good compression power. The evolution of the ratio $\frac{quality}{numberofeigenvectors}$ is not linear. Indeed, the first eigen vectors express much more information than the last ones. Figure 3.7 illustrates this for a typical greyscale image. We notice that the quality grows exponentially up to 8 eigen vectors, reaching a quality of 0.95 and then evolute slowly. This means the first 8 eigen vectors are highly representative of an image, thus adding more components has less importance.

These observations give a first justification of the choice of 8 eigen vectors, leading to a compression ratio close to $1:8$. Other empirical justifications are given further in this chapter.

Figure 3.8 illustrates these results with several images compressed using an increasing number of principal components.

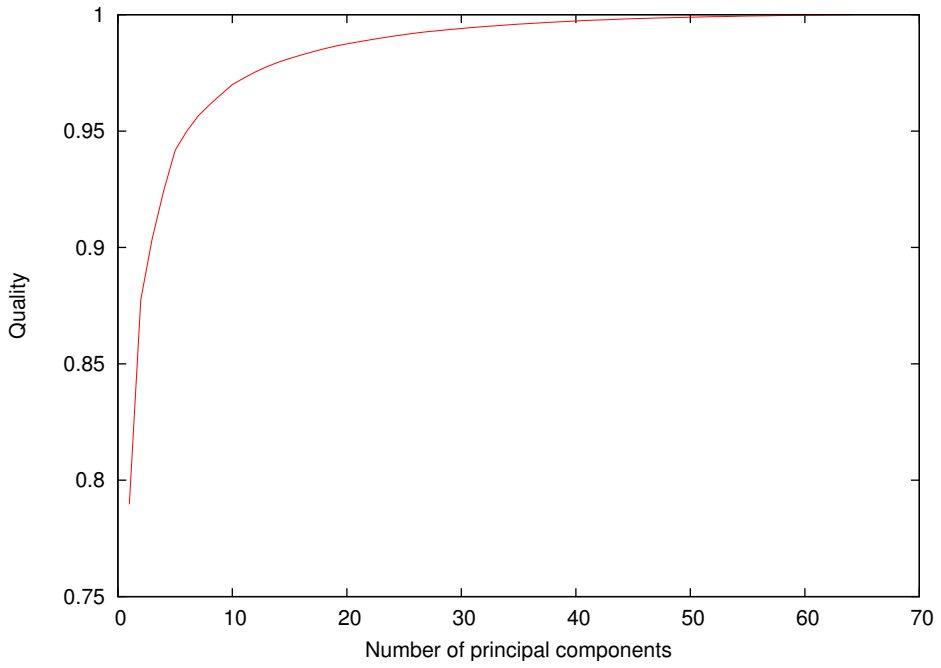


Figure 3.7: Evolution of the quality in function of the number of principal components

Block size influence

In this chapter, we always considered 8x8 blocks. On the one hand, it may be interesting to choose smaller blocks to increase the correlation between the pixels within a block. On the other hand, too small blocks may not reproduce the global image properties by analyzing too reduced portions of it. Figure 3.9 shows the quality of the compressed image at a fixed ratio of 1:8 in function of the size of the blocks. The number of principal components to keep was computed with the following equation:

$$ncomp * (bsize + \frac{imgsize}{bsize}) = \frac{imgsize}{ratio}$$

Figure 3.8: Original image and compressed images with 4, 6, 8, 10 and 12 eigen vectors

with $ncomp$ the number of components, $imgsize$ the size of the image and $bsize$ the size of the block in pixels. This equation takes in consideration the fact that we need to store the eigen vectors in the compressed image to perform the decompression.

The best compromise for a greyscale image appears to be 8x8 blocks. This size keeps a good correlation between pixels while preserving the properties of the image.

3.4.2 Generalization on other images

The principle of PCA compression is to analyze the correlation between neighbors pixels. There is no fundamental reason for two structurally similar images (eg: photographs) to have different pixel correlations. Thus, we can hope to keep a good quality by compressing an image using eigen vectors computed on another image. This section reports our experiments about this process.

Image characteristics comparison

A first point to determine whether generalization could be good or not is to analyze the differences of pixel correlations between grey-scale photographs. Figure 3.10 shows the evolution of the correlation in function of the distance between pixels for three images. The overall evolution remains almost the same.

The similarities of pixel correlations between different images is confirmed by the eigen vectors. Figure 3.11 confronts the eigen vectors of two images. The eigen vectors are almost similar even if their order may vary.

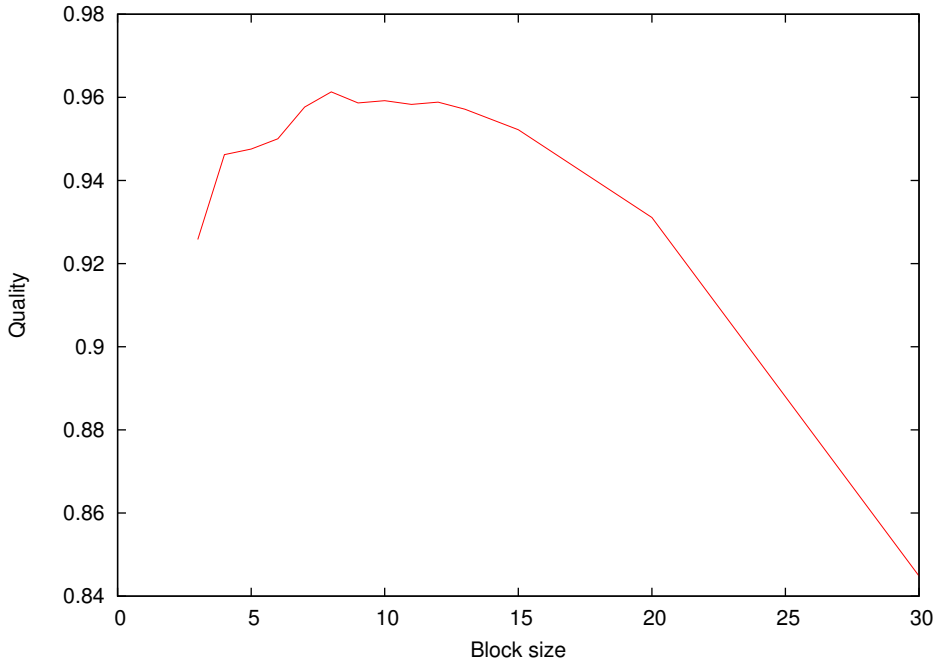


Figure 3.9: Blocks' size influence

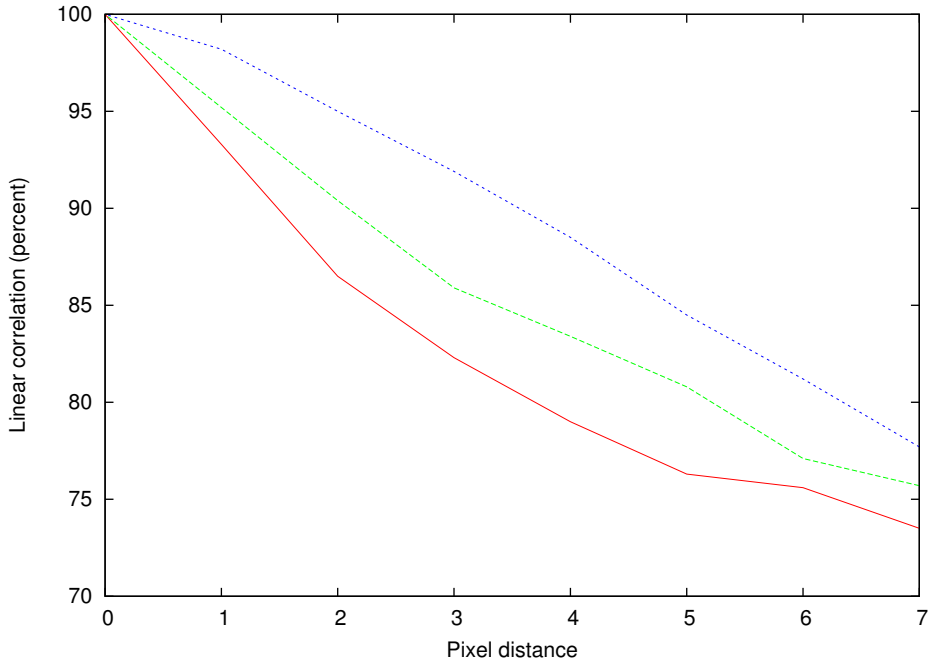


Figure 3.10: Linear correlation in three greyscale photographs

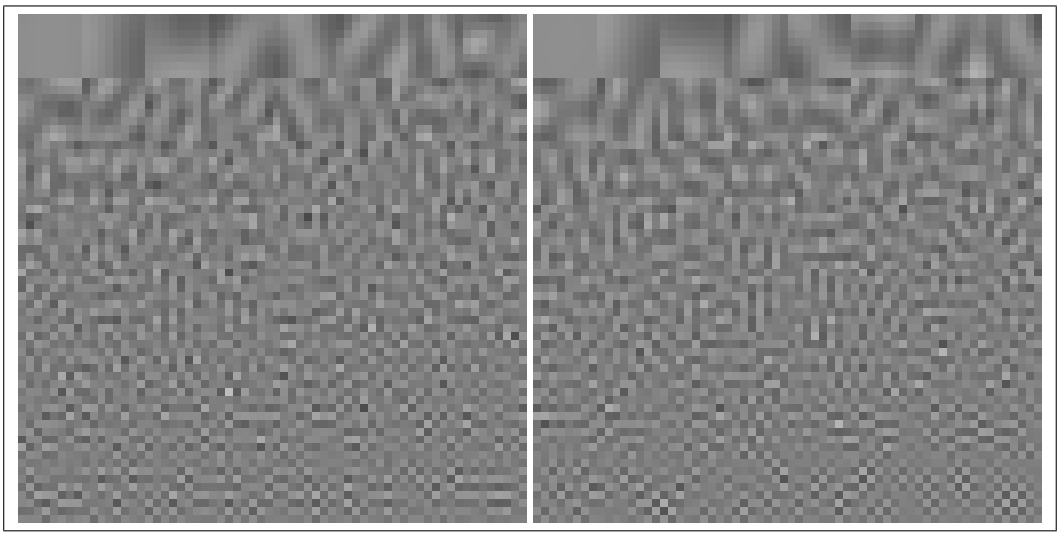


Figure 3.11: Eigen vectors for two greyscale photographs
Each vector is a block of 8x8 pixels. The vectors are sorted with the English reading order.

These observations give great hopes to get a good generalization power. Of course, these observations were made under the assumption of homogeneous, regular images. Particular cases such as sketches or noise are studied at the end of this section.

Quality

To evaluate the quality of the generalization, we compared the error of an image compressed by direct learning (learning was performed on its own blocks) with an image compressed using eigen vectors performed on another image. Figure 3.12 illustrates that the image computed by generalization is barely worse than the directly learned compression. The error increases with the number of eigen vectors selected, confirming the intuition that main components are more similar than insignificant components.

Blocks' size influence

We concluded that 8x8 blocks were the best compromise for direct image learning. It is not necessarily so for image generalization. The question is to find the scale where two image appears to be the closest. Small blocks are likely to have similar structure between images, but we face the same problems as direct learning with too small blocks. On the other side, since the correlation decreases with pixel distance, big blocks are particular to one image and does not represent a generic structure.

This behavior is illustrated by Figure 3.13. 8x8 blocks is still the best choice.

3.4.3 Compression of other kind of images

Up to now, we studied only grey-scale photographs. One may wonder what the results would be with really noisy images, sharp images (such as sketches) or regular images (computer generated graphics). This chapter aims at giving some concrete results about these kinds of image.

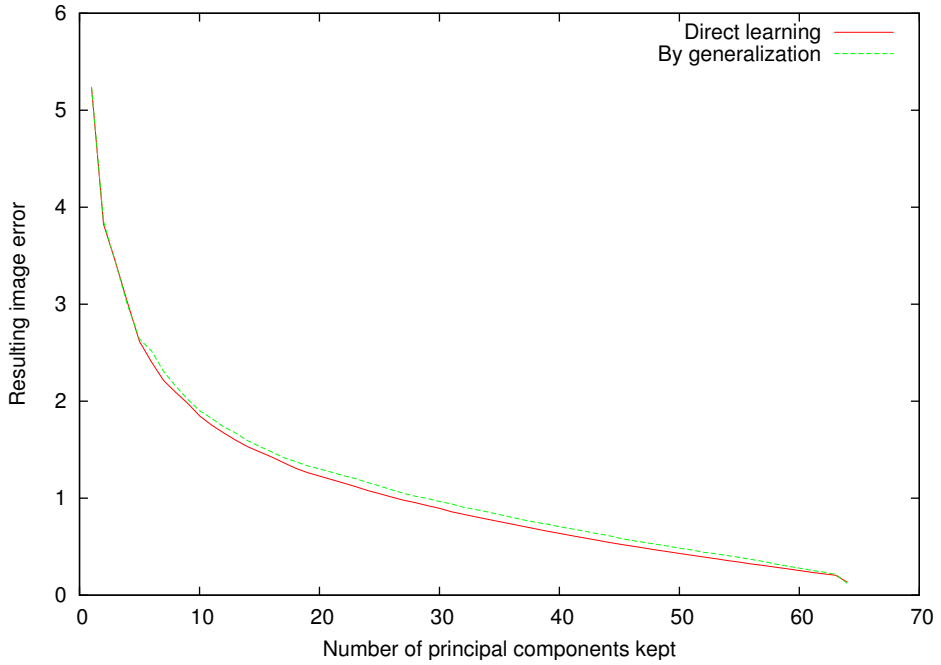


Figure 3.12: Generalization error depending on the number of principal components kept for two greyscale images

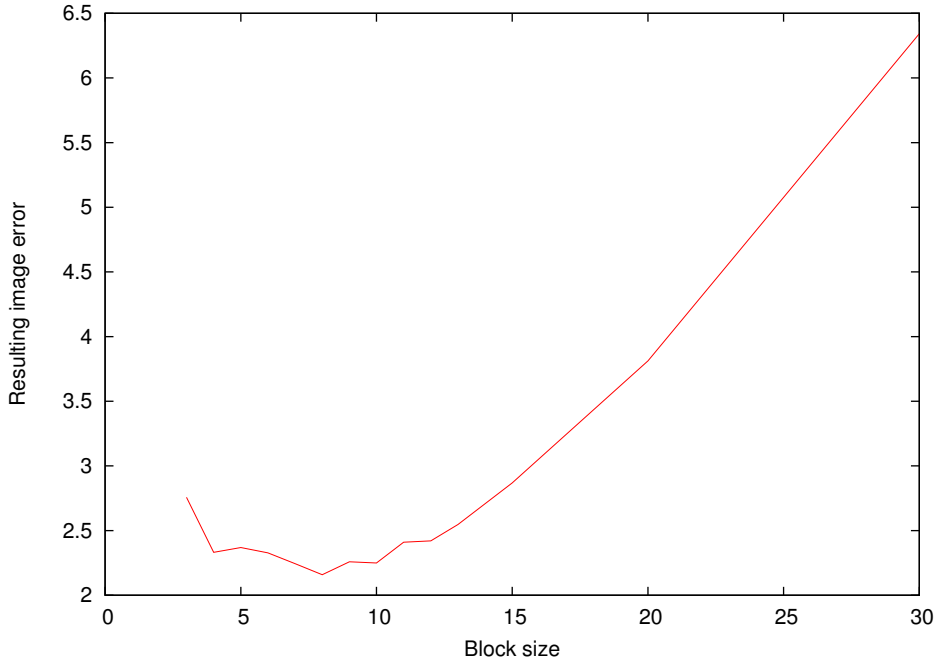


Figure 3.13: Generalization error depending on the size of the blocks (with a fixed ratio of 1:8)

Noisy images

Noise cannot be represented by a linear correlation. Thus, the noise cannot be conserved in the compressed image. The more an image is noisy, the less the compressed image will be close to the original image, as showed in Figure 3.14.

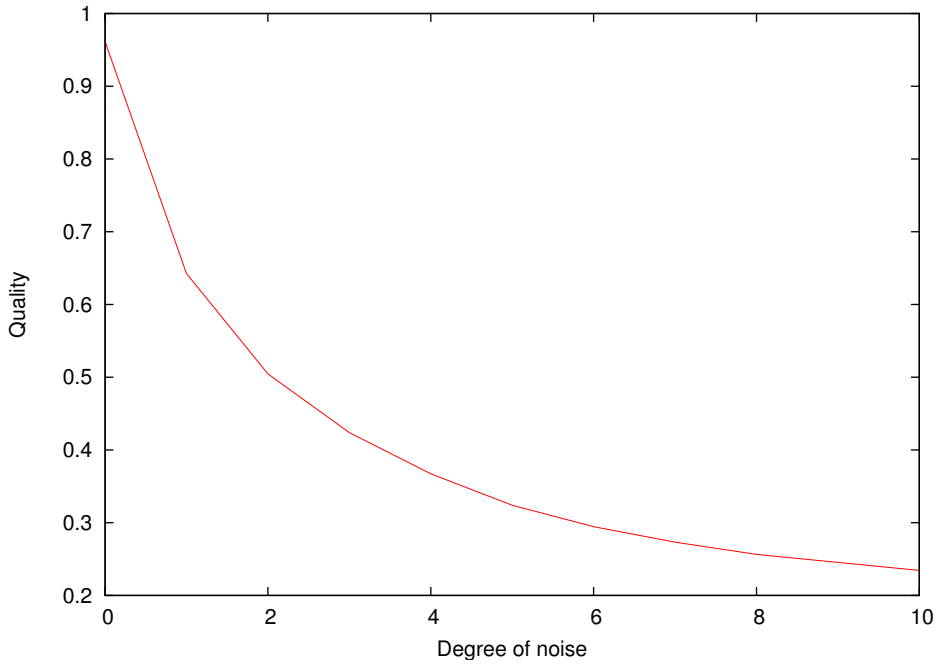


Figure 3.14: Evolution of the fidelity of the compressed image in function of the degree of noise.

Sharp and regular images

Images with sharp shapes (such as sketches) are not homogeneous. Only really near pixels are correlated, thus the compression rate will not be interesting.

On the opposite, regular images will have a great correlation even for distant pixels, allowing the compression to use less eigen vectors to reach the same quality.

These statements are illustrated in Figure 3.15 and Figure 3.16.

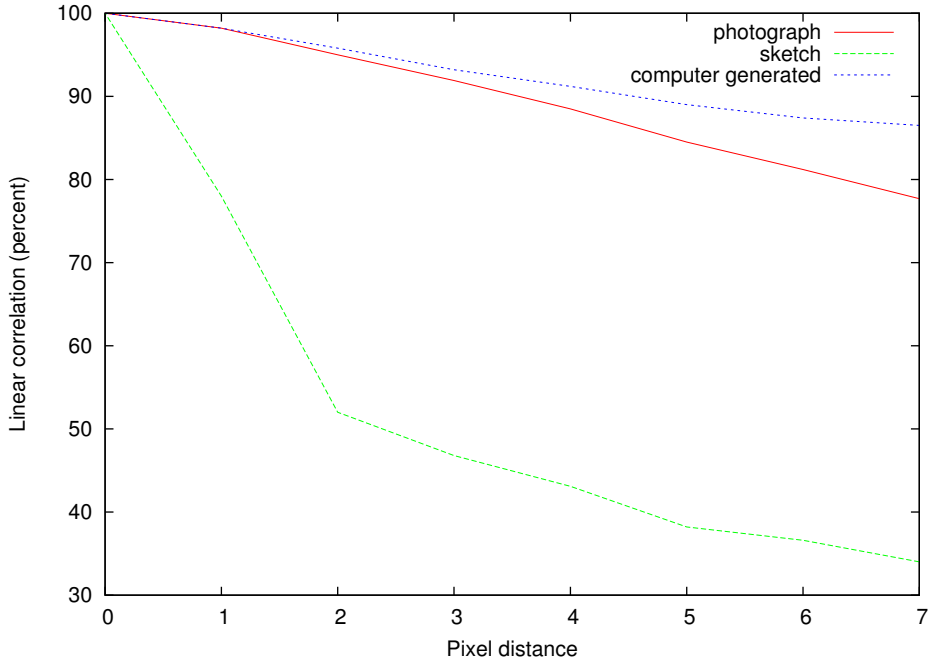


Figure 3.15: Linear correlation for a photograph and a sketch

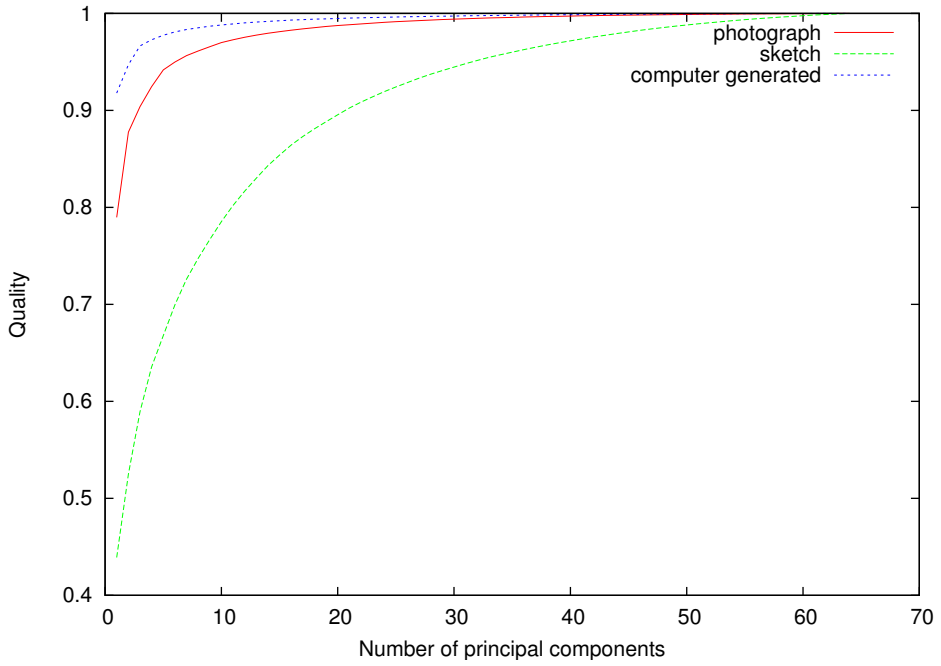


Figure 3.16: Evolution of the quality of a photograph and a sketch in function of the number of principal components

Chapter 4

Conclusion

In this report, improvements of the learning process have first been studied. Two families of algorithms have been tested and compared. The first one regroups algorithms based on an adaptation of the learning rate. It includes Silva-Almeida back-propagation, super self-adaptive back-propagation (SuperSAB) and delta-bar-delta back-propagation. The second family is composed by algorithms that try to adapt the momentum term. Its main members are the resilient back-propagation (RProp) and the quick back-propagation (QuickProp). For each algorithm, parameters influence and general techniques to speed up the learning process have been presented.

From our experiments, conducted on a classification benchmark, best results were obtained using the Rprop and the QuickProp.

We applied the multi-layer perceptron (MLP) to image compression, since we noticed a strong linear correlation between adjacent pixels in images. To that aim, a two layers funnel-shaped MLP has been built. Each input neuron is mapped to a pixel of a block extracted from the input image and the network is trained using the input image as its output.

A standard random initialization of the network leads to interesting results but the time required to train the neural network may be reduced using a smart initialization algorithm. Actually, it has been proved that this network perform a principal component analysis (PCA).

Thus, initializing partially the network using the result of a classical PCA algorithm may improve the required amount of time to compress an image. We tested three approaches: deflation method, the QL method and the adaptive learning algorithm for PCA method. Fastest results were obtained using the QL method but our experiments shown that the deflation can give more accurate values for most of required data.

Using a PCA-type initialization combined with the Rprop algorithm leaded to a fast image compression framework. Characteristics of grey-scale photographs appeared to be quite similar, consequently the same network can be used to compress different images. The main limitation of this framework is the use of blocks which remains visible on the decompressed image. To perform a PCA, classical analytic methods appeared to be faster and more accurate than this neural network approach.

Part II

Hopfield network

Our aim here is to recognize the digits from 0 to 9 using a Hopfield network. In a first part, we will describe the principle of the Hopfield neural network, and we will present some variations made on the learning methods to improve them. Then, in a second part, we will expose our concrete work and analyze the experiments we have made on this network.

Chapter 5

Theory

5.1 Description of the Hopfield network

Presented in 1982, the Hopfield network model belongs to the category of recurrent networks. This simple model is based on the principle of associative memories. In this report, we will study the Hopfield network only as a discrete time recurrent associative neural network.

5.1.1 Auto-associative memories

An auto-associative memory is a kind of memory that reproduces its input pattern as an output, *i.e.* a memory that associates patterns with themselves. In this dynamical system, the patterns constitute a finite number of stable states with basin of attraction around them. These stable states are called "attractors". Wherever in the domain the system is initialized, it will converge to a local attractor.
Consequently, the particularity of an associative memory is its ability to restore a piece of information using a key that may be partial or disrupted by noise. An auto-associative memory will try to categorize the pattern according to its closest prototype.

5.1.2 Architecture of Hopfield network

Analogy with physics

The Hopfield model is very close to the Ising model, which is a simplified model of the interactions between atoms. In this model, each atom is described by the direction of its magnetic momentum, which can be 1 or -1. Each atom will contribute to the magnetic field and influence its neighbors in that way.
In such a system, there might be 2^N different states, with N the number of neurons. But many of these states will hardly happen. Indeed, an atom whose spin is in opposition with the ambiant field is in a highly unstable state. Its spin will surely change to the opposite in order to reach a more stable state.
The Hopfield model reproduces the behavior of such a system.

Structure of Hopfield network

The Hopfield network is a fully intra-connected network. In other words, if we only consider the structure of the Hopfield network, then its graph is a *clique*. See 5.1 for an example.

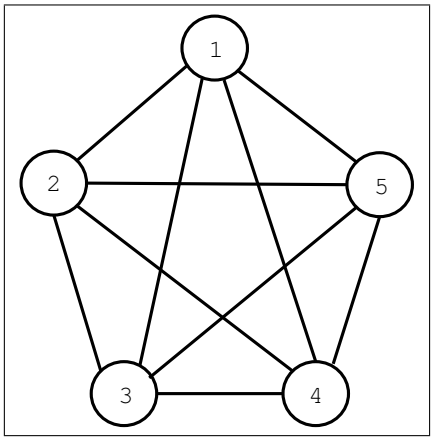


Figure 5.1: An example of Hopfield network with 5 neurons.

The Hopfield model consists of N neurons and $N \times (N-1)$ synapses. The network can store a limited number of patterns (fundamental memories) represented as vectors of binary values (1 or -1). The neurons of a Hopfield network are connected by real-valued weights. The connections between two neurons are always symmetrical.
In the following section, we will present the different learning rules that are used to store the patterns into the network.

5.2 Learning methods

The learning stage of the Hopfield network is quite simple to implement at first sight. Firstly, the learning stage yields the need only for the fundamental memories, while other networks such as the MLP need a large learning database. Secondly, the learning rules are rather simple, too. The first learning rule proposed by Hopfield was the Hebb rule, but some other rules were then proposed to best the results of the Hopfield network.
In this section, we will first present the Hebb rule. We will also present improved learning rules such as the rule proposed by Storkey and Valabregue, and the pseudo-inverse rule.

For all these rules the network is represented by a $N \times N$ synaptic weight matrix W, where N is the size of the fundamental memories from which we make this matrix. Mathematically, fundamental memories will be a collection of M vectors (fundamental memories) of size N: $m_1, ..., m_M$.

5.2.1 Hebb rule

This is the simpliest learning rule. The Hebb rule consists in strengthening the connecting units i an j whenever there is either conjoint activity or conjoint inactivity of both units. Whenever one unit is active but the other is inactive, the weight is weakened.

The rule for updating the weights come rather intuitively:

$$W_{ij} = \begin{cases} \sum_{m \in M} s_i^m \cdot s_j^m & \text{if } i \neq j \\ 0 & \text{else} \end{cases}$$

where W_{ij} is the element on i^{th} line and on j^{th} column, and s_i^m is the i^{th} element in the m^{th} fundamental memory. We can understand W_{ij} as the synaptic weight from neuron i to neuron j. As the connections between the neurons are symmetric, we also have $\forall i, j \in N \times N, W_{ij} = W_{ji}$.

This learning rule is incremental, it means that we can reuse previous matrix W if we want to add a new fundamental memorie inside the list of fundamental memories which the Hopfield network has to learn.

The figure 5.2 below shows an example of 3 by 3 fundamental memories and the corresponding weight matrix.

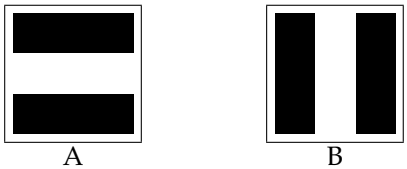


Figure 5.2: Example of fundamental memories in 3×3.

$$W_{i,j} = \begin{pmatrix} 0 & 0 & 2 & 0 & -2 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & -2 & 0 & -2 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & -2 & 0 & 2 & 0 & 2 \\ 0 & -2 & 0 & 0 & 0 & 2 & 0 & -2 & 0 \\ -2 & 0 & -2 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & -2 & 0 & 0 & 2 & 0 & 0 & -2 & 0 \\ 2 & 0 & 2 & 0 & -2 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & -2 & 0 & -2 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$

Learning capacity

Unfortunately, the Hebb rule has a low storage capacity. Using computer simulation, Hopfield first suggested that the number of patterns that may be stored in the network with N neurons is $0.15N$ if we allow a small error in associating the patterns.

Then it was proved that the storage capacity almost without errors of a Hopfield network is given by the formula: $M_max \simeq \frac{N}{2ln(N)}$ (see (13) for detailed demonstration). This capacity severely decreases when the patterns are correlated.

An even more strict definition of storage capacity without errors requires that all fundamental memories are recalled properly. (2) shows that under this definition, the maximum number of patterns that can be learnt properly by a Hopfield network is in fact: $M_max \simeq \frac{N}{4ln(N)}$.

Limitations

With the same patterns as presented in 5.2, the following figures simply illustrate that if lightly-noised patterns will often converge to fundamental memories (figure 5.3), severely-noised patterns may converge to undesired states that do not correspond to any of the fundamental memories that were learned (figure 5.4).

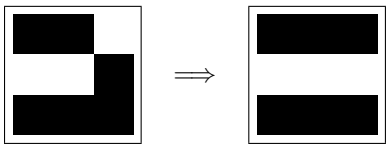


Figure 5.3: Noised pattern A and the result of the convergence.

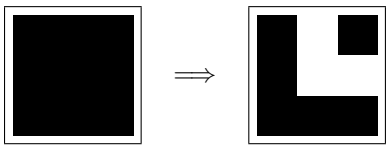


Figure 5.4: The convergence may reach an unknown stable state (spurious state).

These undesired states are called *spurious states*. They may be increased mainly when the ratio between M and N is not appropriate or when the correlation increases.

Trying to eliminate spurious states is a hard task. Hopfield first proposed the unlearning rule, which is a simple solution that tries to improve this annoying aspect of the Hopfield network.

Unlearning

The unlearning method consists to bring some corrections to the matrix W in order to avoid to have too strong (wrong) attractors. Indeed in Hopfield network we are confronted to the problem to have stable states of the Hopfield network that are (very) differents to any fundamental memories; these states are called *spurious states*.

After initializing the matrix W using the Hebb rule, we have many spurious states. Our aim is of course that the stable points of our Hopfield network are fundamentals memories and not others. The unlearning method will consist to reduce inside the matrix W influence of spurious states.

Let s be a spurious state, which is mathematically a vector of size N. From this vector we compute an error correction matrix δW that we will substract to the matrix W as follows:

$$\delta W_{ij} = \eta \times s_i \cdot s_j$$

where $\eta \ll 1$ is the unlearning rate. At last the corrected matrix is:

$$W = W - \delta W$$

We can apply this correction on the matrix W many times, but too much unlearning deforms the states field and may lead to the destruction of the network, since even stable states corresponding to fundamental memories are affected.

5.2.2 Storkey/Valabregue

(Storkey and Valabregue) proposed a new learning rule that should provide a better storage capacity even with correlated patterns.

We can compute the matrix W as following:

$$W_{ij}^0 = 0, \forall i, j$$
$$W_{ij}^v = W_{ij}^{v-1} + \frac{1}{n} s_{v,i} s_{v,j} - \frac{1}{n} s_{v,i} h_{v,ji} - \frac{1}{n} s_{v,j} h_{v,ji}$$

with:

$$h_{v,ij} = \sum_{k=1, k \neq i,j}^{n} W_{ik}^{v-1} s_{v,k}$$

This method is obviously incremental.

5.2.3 Pseudo-Inverse

The Pseudo-Inverse rule is given by :

$$w_{ij} = \frac{1}{N} \sum_{u=1}^{M} \sum_{v=1}^{M} F_{v,i} (Q^{-1})^{uv} F_{u,j}$$

with $Q = \frac{1}{N} \sum_{k=1}^{n} F_{v,k} F_{u,k}$, and N stay the number of neurons.

This learning rule is also called the projection learning rule. We can compute the pseudo-inverse thanks to the method of Greville. This method is not incremental.

5.3 The generalization stage

In the previous section, we presented various learning rules and the way they are used in the learning stage to initialize the weight matrix representing the network.

Once the learning stage over, it is possible to start the generalization stage. The patterns that are to be recognized are presented to the network through a vector representation. The network then computes the output of the neural network and inputs it back again to the network. The process is repeated iteratively until a step t where the input and the output are identical. This process is therefore classified in the category of recurrent networks.

The rule for updating the states is the following:

$$\begin{cases} if \quad \sum_{j \in N} W_{ij} \cdot s_j^t < 0 \quad then \quad s_i^t = -1 \\ if \quad \sum_{j \in N} W_{ij} \cdot s_j^t > 0 \quad then \quad s_i^t = 1 \\ if \quad \sum_{j \in N} W_{ij} \cdot s_j^t = 0 \quad then \quad s_i^t = s_i^{t-1} \end{cases}$$

5.4 Convergence

In the previous section, we presented the generalization process. Now we should check that this process does really converge to a stable state, and under which conditions.

Hopfield defined an energy function (Lyapunov energy) for the model. In the discrete version, this energy can be expressed by:

$$E = -\frac{1}{2}\sum_{i,j\in N} W_{ij}s_i \cdot s_j$$

Let's demonstrate that this energy decreases with the evolution of the system. If we consider the activation of neuron *i*, the state of this neuron will change in two cases:

$$\begin{cases} s_i^{t-1} = 1 \quad and \quad \sum_{j\in N} W_{ij} \cdot s_j^{t-1} < 0 \\ or \\ s_i^{t-1} = -1 \quad and \quad \sum_{j\in N} W_{ij} \cdot s_j^{t-1} > 0 \end{cases}$$

Let's suppose that $s_i^{t-1} = -1$. If the state of the neuron need not to be changed, the global energy remains steady. Otherwise, if the state switches to 1, the difference between the energy of the current state and the previous state is:

$$\begin{cases} \Delta E &= E^t - E^{t-1} \\ &= -\frac{1}{2}\sum_{i,j\in N} W_{ij} \cdot s_i^t \cdot s_j^t + \frac{1}{2}\sum_{i,j\in N} W_{ij} \cdot s_i^{t-1} \cdot s_j^{t-1} \\ &= -\frac{1}{2}\sum_{i\in N}(\sum_{j\in N, j\neq i} W_{ij}(s_i^t \cdot s_j^t - s_i^{t-1} \cdot s_j^{t-1})) \end{cases}$$

Neuron *i* is the only neuron that has changed between state *t-1* and state *t*, so $\forall j \in N, j \neq i, s_j t = s_j t - 1$. The previous expression then becomes:

$$\Delta E = -\frac{1}{2}(s_i^t - s_i^{t-1}) \sum_{j\in N, j\neq i} W_{ij} \cdot s_j^{t-1}$$

As we previously assumed that $s_i^{t-1} = -1$, we have the following hypothesis:

$$\begin{cases} s_i^t - s_i^{t-1} = 1 - (-1) = 2 & > 0 \\ \sum_{j\in N, j\neq i} W_{ij} \cdot s_j^{t-1} & > 0 \end{cases}$$

Consequently, we actually have $\Delta E < 0$. The energy of the Hopfield can only decrease when the state of the neuron changes with an iteration of the algorithm. This is the assurance that the network will effectively converge to stable points.

Chapter 6

Hopfield experimental results

Several aspects of the Hopfield network may flow on its recognition and generalization capacities. In order to study these behaviors, we developed an application featuring:

- Different learning algorithms
- Learning and testing databases selections
- Pattern deterioration through noise
- Evaluation methods
- Statistical measures

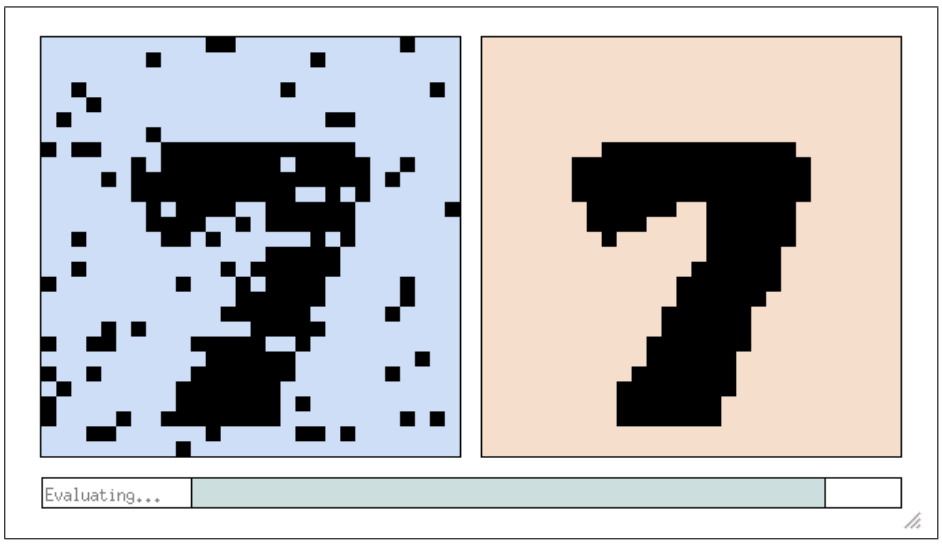


Figure 6.1: Screenshot of the Hopfield network application during an evaluation. The left panel shows the network input; the right one shows the current state of the network.

The Hopfield network was initially introduced with the Hebb learning rule (see 5.2). A first section is dedicated to this rule. We study the influence of the size of the input data and the correlation between the fundamental memories that are learned. These parameters flow on the

learning and generalization capacities. In certain conditions, spurious states invade the state-space, dramatically decreasing the capacities of the network.

The Hebb learning rule is not the only existing one. Other learning techniques have been studied in the last twenty years. Another section compares four learning techniques:

- Hebb learning
- Hebb learning with spurious states unlearning
- Storkey/Valabregue learning rule
- Pseudo-inverse learning rule

6.1 Network performance using Hebb learning rule

In this section we describe several experiments that we did with the Hopfield network. We first describe the experimental framework on which are based our results. The problem of spurious states — which appears very quickly — is then discussed. Spurious states seriously limit the learning capacity of the network. These limitations depend on the neuron count and the correlation between the memories. The influence of these parameters is studied in the two next parts of section. Finally we conclude about the learning capacity of an Hopfield network using the Hebb learning rule.

6.1.1 Experimental framework

Our objective is to study the learning capacity of the network. Therefore all experiments are done in the following way:

1. Learn the fundamental memories using Hebb rule.
2. For each memory,
   (a) Launch the evaluation.
   (b) Compute the hamming distance between the input and the result of the network.
   (c) If this distance equals zero, the memory has been successfully learned.

The learning capacity of the network is the maximum count of different memories that can be remembered without errors. This section is focused on the Hebb learning rule. Others rules are compared in the following section.

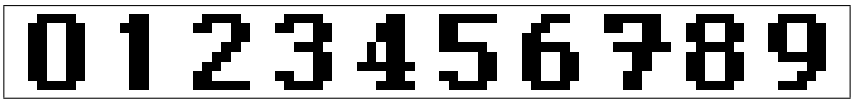


Figure 6.2: 8x8 learning database

Our first learning database corresponds to the 'ideal' models of the ten digits. Each pattern is a binary input image of 8x8 pixels. A memory can be seen as an element of $\{-1, 1\}^{64}$. Firstly we learn only the five first digits $(0 - 4)$. These five fundamental memories have been learned and tested following our experimental framework.

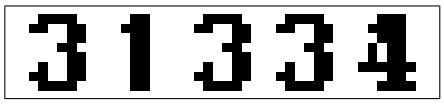


Figure 6.3: Results of the evaluation of digits 0, 1, 2, 3, and 4

0	→	3
1	→	1
2	→	3
3	→	3
4	→	4

Figure 6.4: Bad recognition of the five first digits

Figure 6.3 shows the convergence results and figure 6.4 summaries the digits recognition. With five digits only one is correctly recalled ('1'); two are nearly recalled ('3' and '4'); and the two last are *attracted* by the fundamental memory '3'. If we apply the same test to the ten digits, they *all* converge to the single fundamental memory '8'.

These results are a bit disappointing. This could be explained by the lack of neurons, we thus tried with bigger fundamental memories.

6.1.2 Neuron count

In an Hopfield network, each neuron correspond to a *bit* of the fundamental memories. In the previous case we add fundamental memories in $\{-1, 1\}^{64}$ thus the number of neurons was 64.

The network has secondly been tested with fundamental memories in $\{-1, 1\}^{784}$. This second database contains the ten digits, where each digit has a size of 28x28 pixels. These digits are learned and tested the same way as previously: they all systematically converge to the unknown pattern given in figure 6.5.

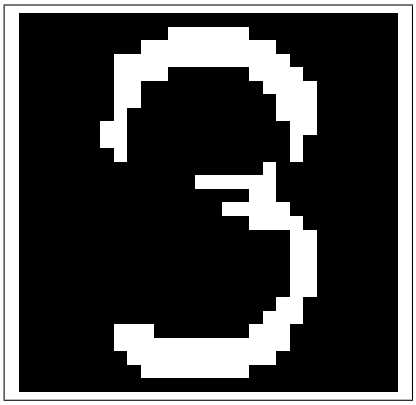


Figure 6.5: Attractor of the ten 28x28 digits

Better results could first be expected due to the neuron count increase. According to **??** the maximal number of fundamental memories that can be learned without error is:

$M_{max} = N/2log_e(N)$, where N is the network size, *i.e.* the number of neurons.

So — theoretically — if $N = 64$, near to 20 fundamental memories can be learned. With $N = 784$ this learning capacity is approximatively 132. In the two cases, it is more than we need with our 10 digits!

Which reason could explain such a difference between the theoretical results and the experimental ones? A particularity of our ten digits is that they are strongly inter-correlated. For example, a '8' is very close to a '3' or a '0'. This strong correlation could be implied in our bad results. Section **??** shows some experiments with totally uncorrelated fundamental memories.

Before studying the influence of correlation between fundamental memories on the learning capacity, let us wonder about the attractor given in figure 6.5. This state is an attractor but does not correspond to a fundamental memory. Such states are called *spurious states*.

6.1.3 Spurious states

Fundamental memories are not the only attractors in the state-space of an Hopfield network (FIXME: ref). In the previous case, all digits converged to a common spurious state. This can be intuitively explained: the mean of the ten digits is very close to this states, see figure 6.6.

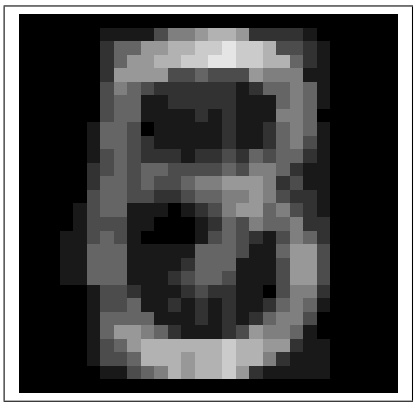


Figure 6.6: Average of the ten 28x28 digits

This spurious state is a strong attractor which catch all evaluations. A first import characteristic of learning rules is the quantity and the attractiveness of spurious states that are generated. In our case, the Hebb rule give very bad results: each fundamental memory is masked by a spurious state.

Some spurious states can be collected with the following algorithm:

1. Choose a random initial state.
2. Let the network converge to an attractor.
3. If this attractor is not a fundamental memory then it is a spurious state.

When running this multiple times with the Hebb rule, we obtain two distinct spurious states: figure 6.6 and its opposite (white becomes black, and black becomes white). These spurious states are very attractiveness: they split the state-space in two. The network just decide if an input pattern is closer to the first spurious state or closer to its opposite.

More spurious states can be obtained with other fundamental memories or other learning methods. See Figure **??**, which represents some spurious states obtained with the Pseudo-inverse rule (see FIXME) applied to the 8x8 database.

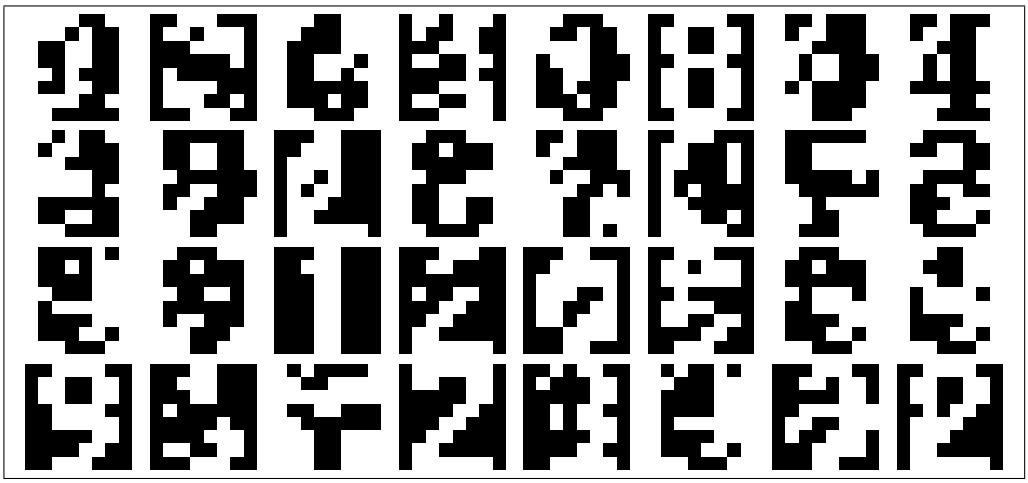


Figure 6.7: Some spurious states with 8x8 digits

FIXME: lien avec la partie theorique.

Let us come back to our ten 28x28 digits. They all converge to the same spurious state. This state corresponds to the average of these fundamental memories. The huge correlation between these digits could be an explanation of this result.

6.1.4 Correlation between fundamental memories

The learning capacity seems to be very reduced when memories are highly correlated. In order to verify this assumption, we built another database presented in figure 6.8. These fundamental memories are members of $\{-1, 1\}^{100}$ (10x10 pixels).

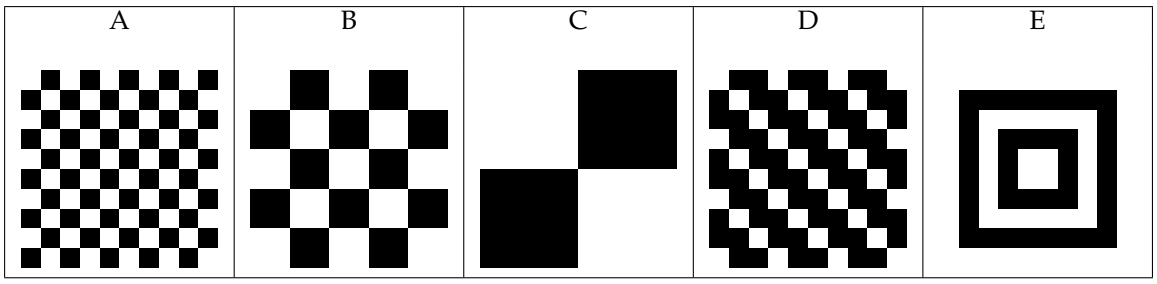


Figure 6.8: Uncorrelated fundamental memories

The Hamming distance between each pair of memories is given in figure 6.9. These distances are more or less always the same. Notice that for vectors of size 100, 50 is an important distance.

These fundamental memories are tested following our experimental framework: they are all perfectly learned. This experience corroborates the idea that our troubles come from the correlation between fundamental memories.

Now that the Hopfield network remembers all its fundamental memories, it is possible to experiment its generalization capacities. A Hopfield network is an associative memory: it should

-	B	C	D	E
A	50	48	48	50
B	-	50	50	40
C	-	-	48	50
D	-	-	-	54

Figure 6.9: Hamming distances between fundamental memories

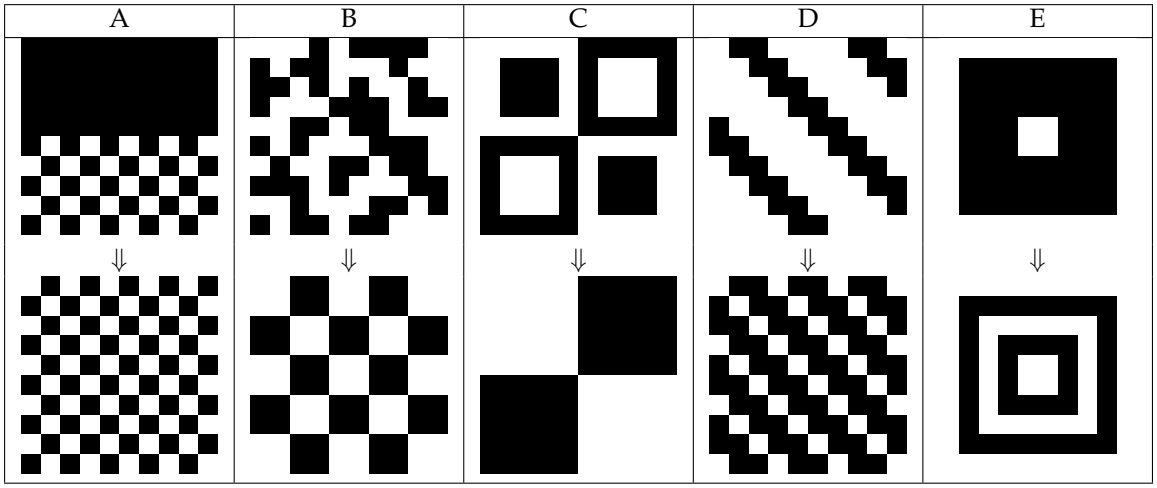


Figure 6.10: Five successful evaluations with uncorrelated patterns

be able to converge to a fundamental memory with partial information as input. Figure 6.10 shows five successful evaluations of the network.

Even with a big quantity of noise, fundamental memories are correctly recalled. This illustrates the second important property of learning rules: a good learning rule must remember its fundamental memories, even if they are strongly correlated. Obviously this is not the case of the Hebb rule.

A learning technique can thus be characterized with two aspects:

- Learning capacity *vs* fundamental memories correlation.
- Amount and attractiveness of spurious states.

6.2 Learning rules

Initially the Hopfield network was introduced with the Hebb learning rule. Since its apparition, some other rules and techniques have been studied. In this section we compare four learning techniques following the characteristics developed previously. These characteristics concern spurious states and correlation between fundamental memories.

A first part describes our experimental framework. This is followed by the study of the Hebb rule (with and without unlearning), the Storkey/Valabreque rule and finally the Pseudo-inverse rule. A last part summaries these results and gives a conclusion about these learning techniques.

6.2.1 Experimental framework

Two aspects of learning techniques can be studied. Our first interest is the influence of correlation on the learning capacity. The learning capacity corresponds to the maximum number of fundamental memories that can be learned and perfectly recalled. The correlation is computed with the following formula:

$$C = \frac{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \| \sum_{k=1}^{M} m_i^k * m_j^k \|}{\frac{M*(N-1)*(N-2)}{2}} \tag{6.1}$$

where N is the size of the network, M is the count of fundamental memories and m^k is the k-th fundamental memory. This value is always in $[0, 1]$. Figure 6.11 shows the correlation of our databases, computed with formula 6.1.

28x28 database	$\rightarrow$	0.63
8x8 database	$\rightarrow$	0.43
uncorrelated database	$\rightarrow$	0.17

Figure 6.11: Total correlation of the 8x8, the 28x28 and the uncorrelated databases

A (correlation, learning capacity) point can be obtained with the following algorithm:

1. Choose a random set of fundamental memories. Their count must be bigger than the maximal possible learning capacity.
2. Choose a desired correlation in $[0, 1]$.
3. While correlation is different from desired correlation do:
   (a) Choose a random fundamental memory.
   (b) Make a random change into this memory.
   (c) If the new correlation is closer to the desired one keep this change, else return in the previous state.
4. Now that we have a set of fundamental memories with the given correlation, search the maximal learning capacity. Let N be 1 and do:
   (a) Try to learn N memories.
   (b) If the N memories are successfully recalled continue with (N + 1) else the maximum learning capacity has been reached.
5. The correlation is the desired correlation in $[0, 1]$; the learning capacity is (N - 1).

When applying this many times, we obtain a cloud of points that illustrates the influence of correlation on learning capacity. For each learning rule this algorithm is applied with a network of 14x14 = 196 neurons.

The second interesting aspect of learning rules is related to spurious states. Our experiments about spurious states were very primitive but gives some simple results. For each learning rule we just estimated the percent of spurious states into the set of attractors of the state-space. The following algorithm has been applied:

1. For i = 0 to 1000 do

(a) Choose a random initial state.

(b) Let the network converge.

(c) If the result is a spurious state, increment spurious state count.

2. The percent of spurious states is given by spurious states count / 1000.

For each rule, this was applied with the 8x8 database, the 28x28 database and the uncorrelated database.

Let us now see the results of these experiments.

6.2.2 Hebb with or without unlearning

The simplest and most known learning rule for Hopfield network is the Hebb rule. As seen in the previous section, this rule is highly dependant of the correlation between the fundamental memories.

A simple variant of the Hebb rule consists in unlearning spurious states. Each time the network is attracted to a spurious states, this state is unlearned a bit (it is the opposite of learning).

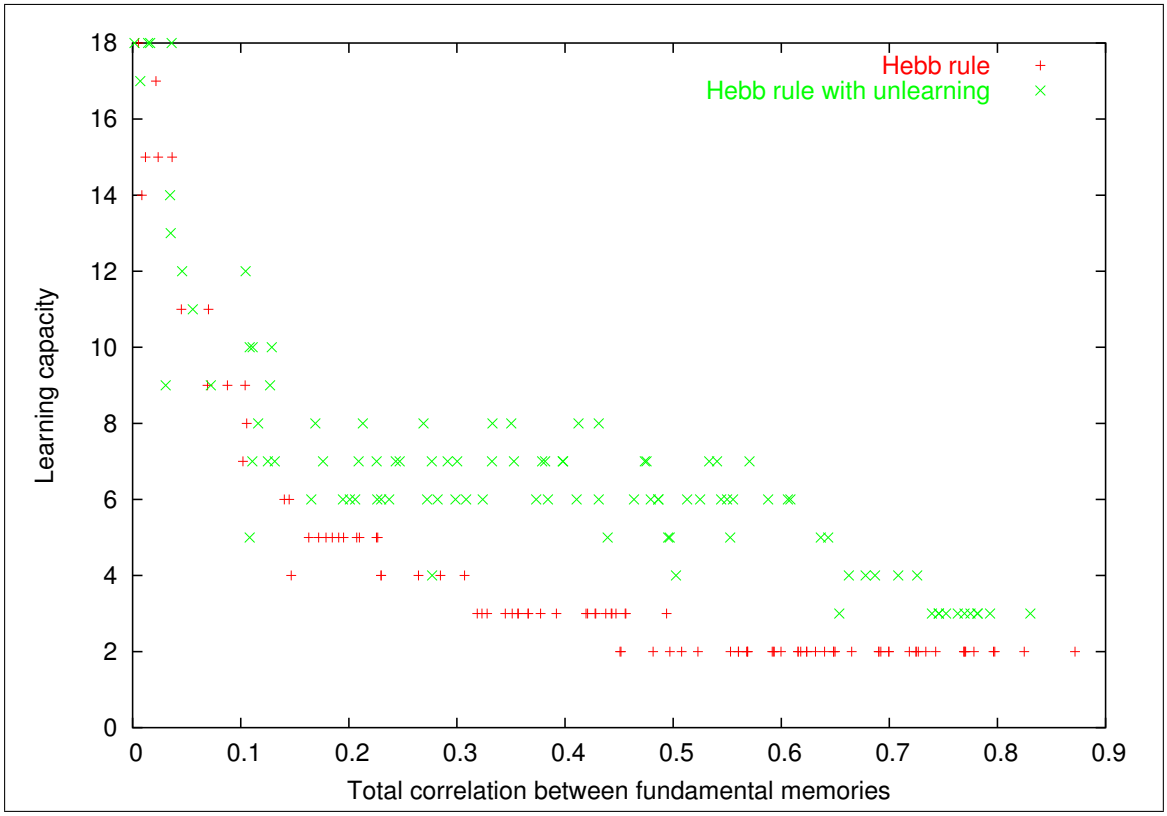


Figure 6.12: Learning capacity of the Hebb rule with or without unlearning with 196 neurons

Figure 6.12 shows the learning capacity of the Hebb rule and the Hebb with unlearning technique (see: FIXME). Theses curves corresponds to the results given in previous section: with totally uncorrelated patterns, the learning capacity is near to 18. This value corresponds to the theoretical maximal learning capacity without errors:

$$M_{max} = N/2log_e(N) = 196/2log_e(196) \approx 18,5.$$

When correlation grows, this capacity decreases quickly: only two patterns can be learned with a correlation of 0.5.

Unlearning slightly helps: with a correlation bigger than 0.1, unlearning allows to learn — in average — 4 memories more. Even with that the learning capacity is most of time less than 8 which is even not enough for our ten digits. Definitively, the Hebb rule does not permit a good use of the Hopfield network in real conditions, *i.e.* with correlated patterns.

Concerning spurious states, two cases appear:

- The patterns have not been learned, each fundamental memory is attracted to a spurious state: 100% of the attractors are spurious states.
- The patterns have been learned (uncorrelated database), we observe 69% of spurious states.

Unlearning often slightly deteriorates the state-space: we can observe a average of 10% more spurious states when the patterns are successfully learned.

6.2.3 Storkey/Valabregue learning rule

The third learning technique we tried is the Storkey/Valabregue rule. The learning capacity of this technique is given in figure 6.14.

28x28 database (10 digits)	→	1 perfectly recalled, 1 nearly recalled
8x8 database (10 digits)	→	8 perfectly recalled
uncorrelated database (5 patterns)	→	5 perfectly recalled

Figure 6.13: Learning results of Storkey/Valabregue rule with 196 neurons

As in the Hebb rule, the correlation plays a major role. But the results are much better: the maximum observed learning capacity is 85, which is near to five times more than for the Hebb rule. With a realistic correlation (0.5), about 5 patterns can be learned. We thus have a much bigger maximum learning capacity, but it remains poor for a real use. Figure 6.13 gives the learning results on our databases with the Storkey/Valabregue rule.

This learning method also generates a lot of spurious states. Following the databases, the percent of attractors that are spurious states evolves between 66% and 74%.

6.2.4 Pseudo-inverse learning rule

The last learning method that we tried is the Pseudo-inverse learning rule. When applying the experimental framework to determine the influence of correlation on the learning capacity, a new phenomenon appears. With a huge quantity of memories the weights matrix obtained with this rule converge to the zero matrix. With such a matrix, the network act as a *identity* function: any memory is output without any changes. In the state-space, each state is a minimum local. When applying the experimental framework without changes, the result is the following: the learning capacity is a constant, which equals 2^N where N is the size of the network. With an identity function, any point of $\{-1,1\}^N$ is successfully returned!

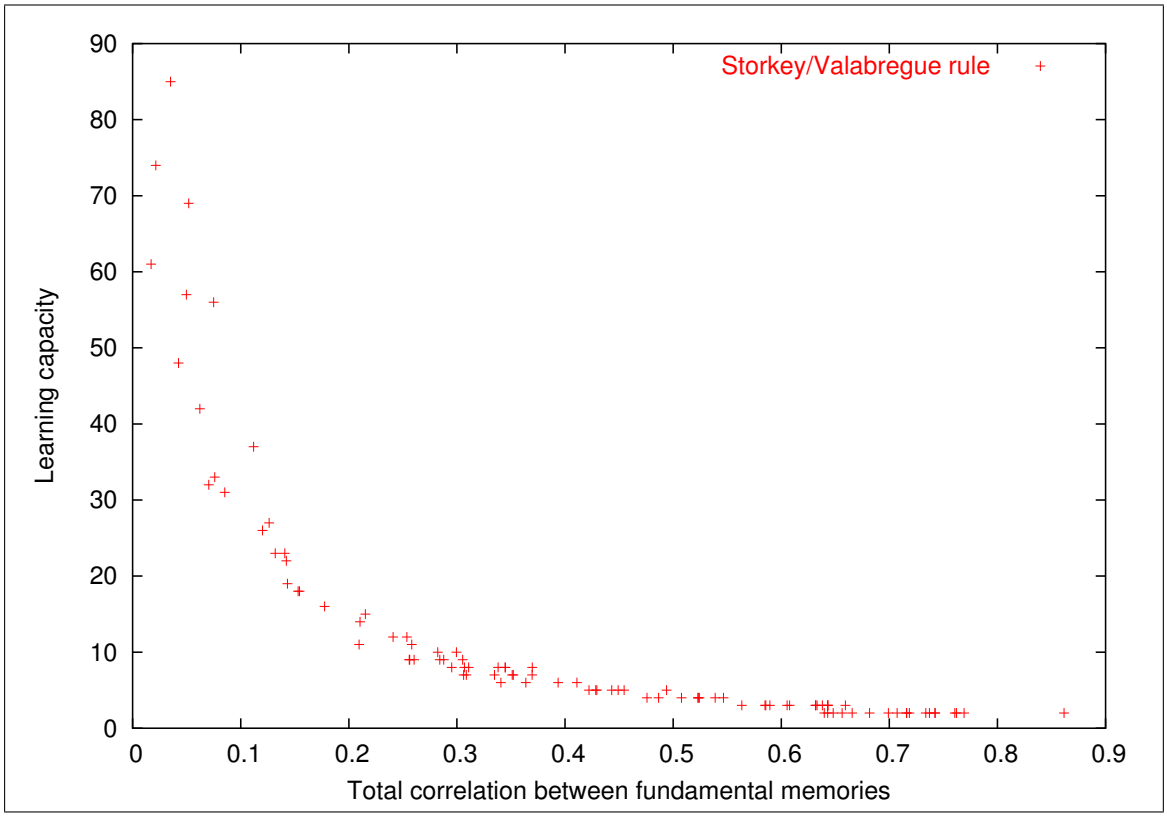


Figure 6.14: Learning capacity of the Storkey/Valabregue rule with 196 neurons

Thus we had to add a new condition: any learned state must be exactly recalled, and any non-learned and non-spurious state must be attracted by a state different from itself.

The last part of the experimental framework has been modified the following way:

1. Try to learn N memories.
2. Evaluate the N memories.
3. Evaluate M states that are not learned.
4. If the N memories are successfully recalled and none of the M states converge to their self, continue with (N + 1) else the maximum learning capacity has been reached.

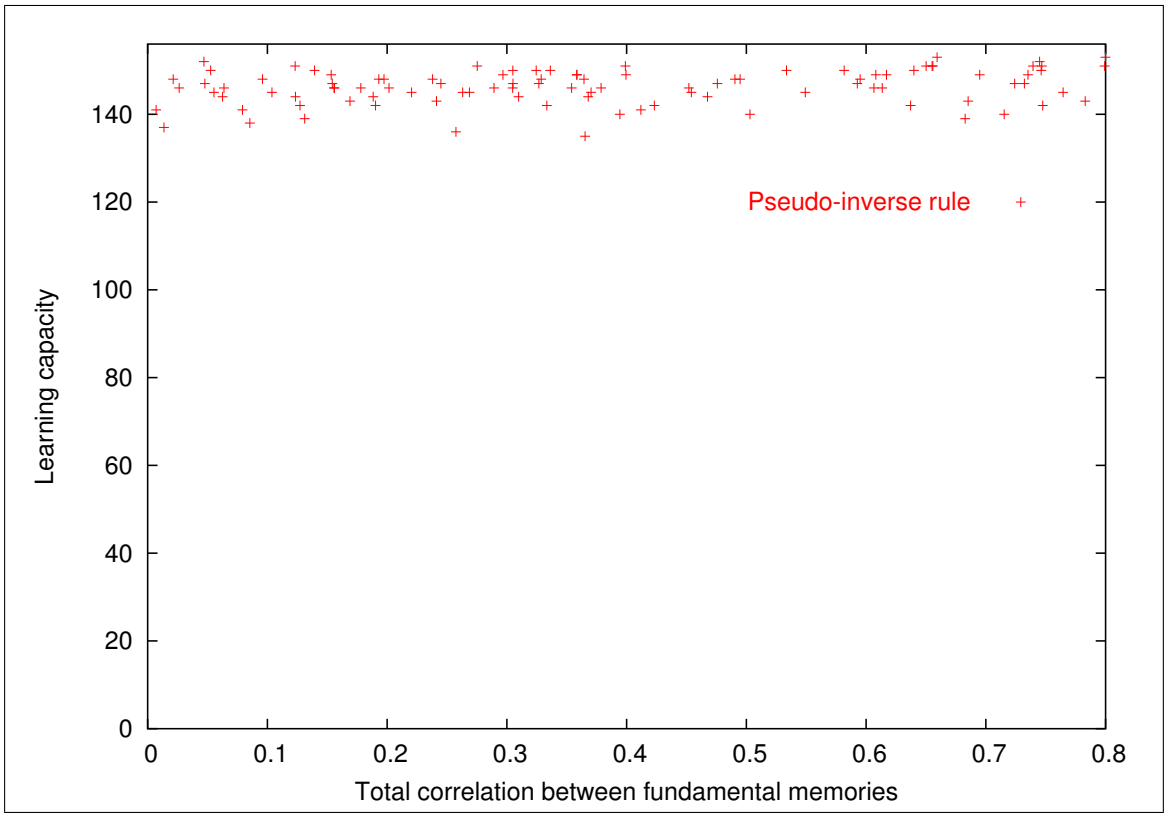


Figure 6.15: Learning capacity of the Pseudo-inverse rule with 196 neurons

The results, given in figure 6.15, are a bit surprising. Indeed the learning capacity seems to be independent of the correlation. This capacity is systematically limited by a phenomena of *over-learning*: when learning more than 140 patterns, all states of state-space progressively becomes spurious states. This rule clearly remains the one which gives the best results. Obviously, all our databases are perfectly learned thanks to this amazing learning capacity.

The main drawback of the Pseudo-inverse learning rule is the quantity of spurious states that are generated. The percent of attractors that are spurious states evolves in the range [66%-95%]. More the patterns are correlated, more this learning rule generates spurious states.

6.2.5 Summary

The learning capacities of the learning rules that we tested are compared in figure 6.16.

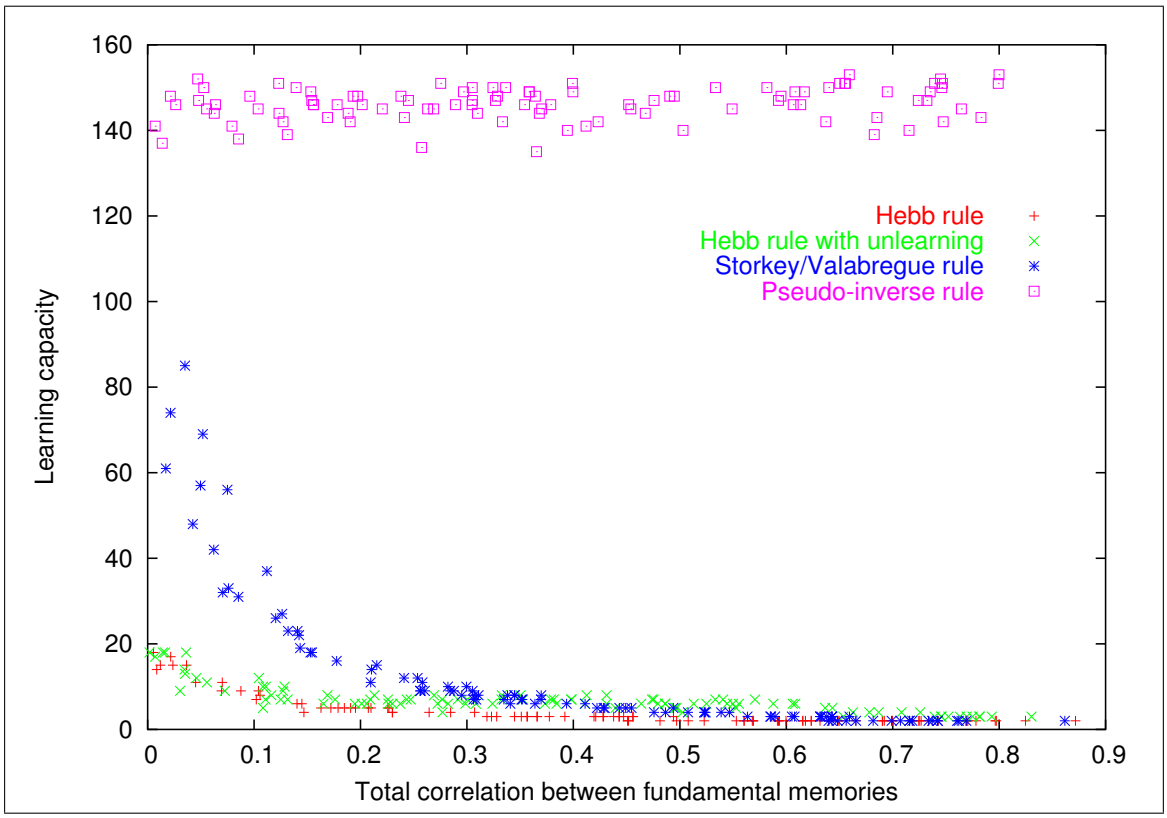


Figure 6.16: Summary of learning capacities with 196 neurons

The main advantage of the Hebb rule is its simplicity. Weights are directly set to correlation between pairs of pixels, and theoretically it works. Its main drawback is the bad support of correlated patterns. When recognizing digits, the fundamental memories are very correlated: Few pixels differs between a '0', a '8' or a '3'. In such conditions the Hebb rule do not work successfully.

Pseudo-inverse works very well even with highly correlated patterns. The learning capacity of this rule is much better than the one of the Hebb rule. Unfortunately this rule is neither local neither incremental: computing weights with this learning rule requires to have all the fundamental memories at a given time. Moreover this calculations cannot be distributed since the require the inversion of a matrix.

Intermediates rules have been introduced such as the Storkey/Valabregue one. This rule is an approximation of the pseudo-inverse, so that its calculation is local and incremental. A simple relationship between this different training rules can be established. When using the iterative definition of the inverse of a matrix (see 1.2), our rules can be obtained from the Pseudo-inverse one:

- Hebb: zero-order expansion of the inverse.
- Storkey/Valabregue: equivalent to the first-order expansion of the inverse.
- Pseudo-inverse: infinite expansion.

Bibliography

[1] A.Jacobs, R. (1988). Increased rates of convergence through learning rate adaption. *Neural Networks*, 1:295–307.

[2] Amit, D. (1989). *Modeling brain function: the world of attractor neural networks*. Cambridge University Press.

[3] Anguita, D., Parodi, G., and Zunino, R. (1993). Speed improvement of the back-propagation on current generation workstations. In *World Congress on Neural Networking*, volume 1, pages 165–168. Lawrence Erlbaum.

[4] Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

[5] Blake, C. and Merz, C. (1998). UCI repository of machine learning databases.

[6] Breton, S. (1999). *Une approche neuronale du contrôle robotique utilisant la vision binoculaire par reconstruction tridimensionelle*. PhD thesis, Université de Haute Alsace.

[7] Chen, L.-H. and Chang, S. (1995). An adaptive learning algorithm for principal component analysis. *IEEE transactions on neural networks*, 6(5):1255–1263.

[8] Dreyfus, G., Martinez, J.-M., Samuelides, M., Gordon, M. B., Badran, F., Thiria, S., and Hérault, L. (2002). *Réseaux de neurones, Méthodologie et applications*. Eyrolles.

[9] Fahlman, S. (1988). Faster-learning variations on back-propagation: An empirical study. In *Connectionist Models Summer School*, pages 38–51.

[10] Fahlman, S. E. and Lebiere, C. (1990). The cascade-correlation learning architecture. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, Denver 1989. Morgan Kaufmann, San Mateo.

[11] Finnoff, W., Hergert, F., and Zimmermann, H. G. (1993). Improving model selection by nonconvergent methods. *Neural Networks*, 6(6):771–783.

[12] Fiori, S. and Piazza, F. (1999). A comparison of three pca neural techniques. In *European Symposium on Artificial Neural Networks*, pages 275–280.

[13] Haykin, S. (1999). *Neural Networks, a comprehensive foundation*. Prentice Hall International Editions.

[14] Igel, C. and Hüsken, M. (2000). Improving the Rprop learning algorithm.

[15] Izui, Y. and Pentland, A. (1990). Speeding up back-propagation. *IJCNN-90-WASH-DC*, 1:639–642.

[16] MacKay, D. J. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

[17] Oja, E. (1989). Neural networks, principal components, and subspaces. *International Journal Neural Systems*, 1:61–18.

[18] Prechelt, L. (1994). Proben1: A set of neural network benchmark problems and benchmarking rules. Technical Report 21/94.

[19] Prechelt, L. (1997). Connection pruning with static and adaptive pruning schedules.

[20] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1985). *Numerical Recipes in C —The Art of Scientific Computing —Second edition*. Cambridge University Press.

[21] Riedmiller, M. and Braun, H. (1992). Rprop- a fast adaptive learning algorithm.

[22] Sanger, T. D. (1989). Optimal unsupervised learning in a single layer linear feedforward neural network. *Neural Networks*, 2:459–473.

[23] Saporta, G. (1990). *Probabilités, Analyse des données et statistique*.

[24] Schiffmann, W., Joost, M., and Werner, R. (1992). Optimization of the backpropagation algorithm for training multilayer perceptrons. Technical report, Koblenz, Germany.

[25] Silva, F. and Almeida, L. (1990). Speeding up backpropagation. In *Advanced Neural Computers, North Holland*, pages 151–158. R. Eckmiller (Ed.).

[Storkey and Valabregue] Storkey, A. and Valabregue, R. A hopfield learning rule with high capacity storage of time-correlated patterns.

[26] Tollenaere, T. (1990). SuperSAB: fast adaptive back-propagation with good scaling properties. *Neural Networks*, 3:561–573.

[27] Utans, J. and Moody, J. (1991). Selecting neural network architectures via the prediction risk: application to corporate bond rating prediction. In *Proc. of the First Int. Conf on AI Applications on Wall Street*, Los Alamos,CA. IEEE Computer Society.