A Non-linear Dimensionality Reduction Method for Improving Nearest Neighbour Classification

by

Renqiang Min

A dissertation presented to the
Faculty of Department of Computer Science
University of Toronto
In Conformity with the
Requirements for the degree of
Master of Science

Abstract

A Non-linear Dimensionality Reduction Method for Improving Nearest Neighbour Classification

Renqiang Min

Master of Science

Department of Computer Science

University of Toronto

2005

Learning in high dimensional spaces is computationally expensive because of the curse of dimensionality. Consequently, there is a critical need for methods that can produce good low dimensional representations of the raw data that preserve the significant structure in the data and suppress noise. This can be achieved by an autoencoder network consisting of a recognition network that converts high-dimensional data into low-dimensional codes and a generative network that reconstructs the high dimensional data from its low dimensional codes.

Experiments with images of digits and images of faces show that the performance of an autoencoder network can sometimes be improved by using a non-parametric dimensionality reduction method, Stochastic Neighbour Embedding, to regularize the low-dimensional codes in a way that discourages very similar data vectors from having very different codes.

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Geoff Hinton, for his guidance, inspiration and patience during my last two years' graduate study at the U of T. It is he who guided me into the area of neural networks, suggested me to work on his idea about Regularized Autoencoder Network and gave me a lot of help on the research. Many apologies to Geoff too for the several months' delay before I finally finished this thesis.

I would also like to thank members of the machine learning group at the U of T, He Xuming, Bao Kejie, Yousuf Shamim and Al-Mustansir Mukhles, and my friends at the Department of Computer Science of the U of T, Lan Hui and Zhu Xiaodan, for their helpful discussions about my research.

Last but not least, I would like to thank my parents, and my friends at Nankai University, Zhang Jian and Jiang Jie, for their support and encouragement.

Renqiang Min

December 2004

Dedications

To my parents and aunt

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Object Recognition Using Low Dimensional Codes

In the last few years, machine learning methods have been successfully applied to many application areas such as information retrieval, image processing, computational biology and computational chemistry. To analyze and interpret real datasets, we often need to do clustering or classification in a high dimensional space.

However, training of models in a high dimensional space can be very time-consuming because of the curse of dimensionality sometimes [Bishop, 1995]. Also when K-means and EM are applied to high dimensional data, they are easily trapped in bad local minima which depend on the initial configurations of the models [Ding, 2002]. To solve this problem, a lot of techniques and methods have been proposed for dimensionality reduction [Scott, 1992].

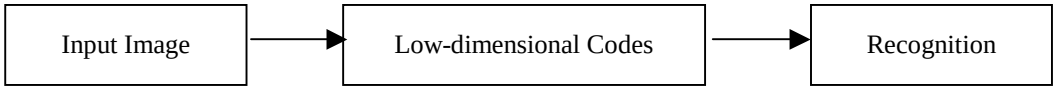
Methods like Principal Component Analysis [Jolliffe, 1986, Jackson, 1991] only give linear mappings from the high dimensional space to the low dimensional space. In my thesis, a new approach to generating low dimensional codes will be presented (chapter 2). It gives a non-linear mapping from the high dimensional data space to the low dimensional code space, and it also preserves local similarity structure within the data.

The high dimensional data used in this thesis consists of images of objects which are often complicated and have a large number of pixels. This makes the task of object

recognition difficult. Object recognition itself has many practical applications and has been explored by a lot of researchers [Lowe, 1999, Beis and Lowe, 1999, Agarwal and Roth, 2002, Fergus and Perona et al., 2003]. For example, it plays an important role in an intelligent robot system, in which a recognition system helps a robot to detect objects and to navigate. In my thesis, I will address the recognition of two kinds of objects: handwritten-digits (chapter 4) and faces (chapter 5).

An original image taken by a camera has a large number of pixels. To recognize an object contained in an image, we need to make use of the most discriminative features. Many pixels in the image do not contribute much to the recognition even in human vision. For example, in a grey level image of a digit, there may be large background regions where all the pixel values are almost equal to zero. By mapping an image to a low dimensional space, we hope to eliminate the redundancy in such regions but we also want to preserve pairwise underlying similarities between images and to account for most of the variance in the data. This should allow the low dimensional code to retain the most important information in the images. A Regularized Autoencoder Network (RAN) described in Chapter 2 can achieve this.

Generating low dimensional codes could be integrated with subsequent processing stages. But in most situations, it is used as a preprocessing step, as shown below:



In all the experiments described in this thesis, the low dimensional codes of digits or faces are generated first, and then in some low dimensional space, we do clustering or classification. In the following, some methods that can be used to generate low-dimensional codes of data will be introduced.

1.2 Dimensionality reduction Techniques

1.2.1 Principal Component Analysis

Principal Component Analysis (PCA) [Jolliffe, 1986, Jackson, 1991] also known as the Karhunen-Loeve Transform is a classical statistical method. It identifies the axes for a set of data vectors along which the correlation between components of the data vectors can be most clearly shown.

Suppose there is a data set $M=\{ X_i \mid i=1, \ldots, N \}$, where X is an n-dimensional column vector and $X = (x_1, \ldots , x_n)^T$. The mean of the data vector is $\mu = < X >$, here $< >$ stands for the average over the data set. The data set can be represented by a matrix $D = (X_1, X_2, \ldots, X_N)$. The covariance matrix of D is C with its element C_{ij} which can be calculated as shown below.

$$C_{ij} = <(x_i - \mu_i)(x_j - \mu_j)> \quad (1.1)$$

By solving the characteristic equation of the covariance matrix C, we can obtain the eigenvectors that specify the axes having the properties described above and the corresponding eigenvalues that are respectively indicative of the variance of the dataset along these axes. Therefore, just by looking at the eigenvalues, we can easily find out along which axes the dataset has little or no spread. Hence, the principal axes and the eigenvalues give a good reflection of the linear interdependence between the components of the data vectors. By choosing some eigenvectors that have the largest eigenvalues, we can form a subspace A in which the data set has the most significant amounts of variance. Thus, the dimensionality of the data can be reduced by means of this property of PCA.

Suppose B has all the eigenvectors of the covariance matrix C as its row vectors, we can transform a data vector X this way:

$$Y = B(X- \mu) \quad (1.2)$$

By applying this projection to the original dataset D, we can get an uncorrelated vector set {Y}. Since B is an orthogonal matrix, the inverse of B is equal to the transpose of B (B^T). We can use Y to obtain the original data vector X like this:

$$X = B^T Y + \mu \quad (1.3)$$

In the subspace A consisting of the eigenvectors having the largest eigenvalues, we can do the similar transformation to the above to get the low dimensional code vector Y' of

X.

$$Y' = A(X- \mu) \quad (1.4)$$

And we can reconstruct X in the way that is similar to (1.3).

$$X' = A^T Y + \mu \quad (1.5)$$

By (1.4) and (1.5), we project the original data vector to the low dimensional space spanned by A and then we use the low dimensional code to reconstruct the original data. This projection minimizes the mean-square error between the data and the reconstruction of the data.

In practice, dimensionality reduction using PCA can be done efficiently through singular value decomposition (SVD) [Golub and Loan, 1996].

1.2.2 Limitations of PCA

Every method has its intrinsic limitations, so does PCA.

Firstly, PCA can only identify linear combinations of variables; that is to say, it can only determine linear interdependencies between components of a sample of data vectors. In a word, it can only give a linear mapping from some data space to some low dimensional code space.

Secondly, since we obtain some principal axes by solving the characteristic equation of a covariance matrix, PCA can only capture the second-order correlation information between components of the data but ignores the higher-order correlation information among components of the data.

From the point of view of continuous latent variable models, we can generate data by first generating a point within a subspace and then adding noise. The coordinates of the point are the components of the latent variable. In this way, we can obtain the PCA model by assuming that (1) the subspace in which some points lie is linear; (2) the distribution of the latent variable is Gaussian; (3) the sensor noise covariance $\Psi = \frac{1}{\infty} I$, where I is the identity matrix. The PCA model is shown as below.

$$p(z) = N(z \mid 0, I) \tag{1.6}$$

$$p(y|z,\theta) = N(y \mid \mu + \Lambda z, \frac{1}{\infty} I) \tag{1.7}$$

Here y stands for observed data, z stands for latent variable, μ is the mean data vector, Λ is an N×M loading matrix, where N is the dimensions of the data vector and M is the dimensions of the low dimensional code vector. This explanation of PCA clearly shows that PCA is a Gaussian model. Therefore, it can't behave beyond its capability to capture higher-order statistics in the data.

Thirdly, when PCA is used for dimensionality reduction, we generate the low dimensional codes only in order to give a good reconstruction of the data. Thus, the low dimensional codes may be very powerful in representing the data in each class but very weak in distinguishing the data belonging to different classes.

1.2.4 Independent Component Analysis

Independent Component Analysis (ICA) [Hyvarinen et al., 2000] has been used to extract low dimensional codes of natural images, faces, texts, natural sound and so on [Olshausen and Field, 1996, Bell and Sejnowski, 1996, 1997, Hyvarinen et al., 1998]. From the view of continuous latent variable models, the standard ICA model used for dimensionality reduction can be generated under the assumption that (1) the subspace in which some latent points lie is linear; (2) Components of the latent variable have independent non-Gaussian distributions; (3) the dimension of the latent variable is the same as that of the data. Undercomplete ICA [Stone and Porrill, 1998] can be used for dimensionality reduction. The subspace in which the low dimensional codes of data lie, can be viewed as hidden causes of the data. ICA can capture higher-order statistics in the data, but the assumptions under which ICA can work confine its applications to any high dimensional data. In essence, undercomplete ICA produces Independent Components by unmixing the data, and like PCA it is a linear model.

1.2.3 Fisher's Linear Discriminant Analysis

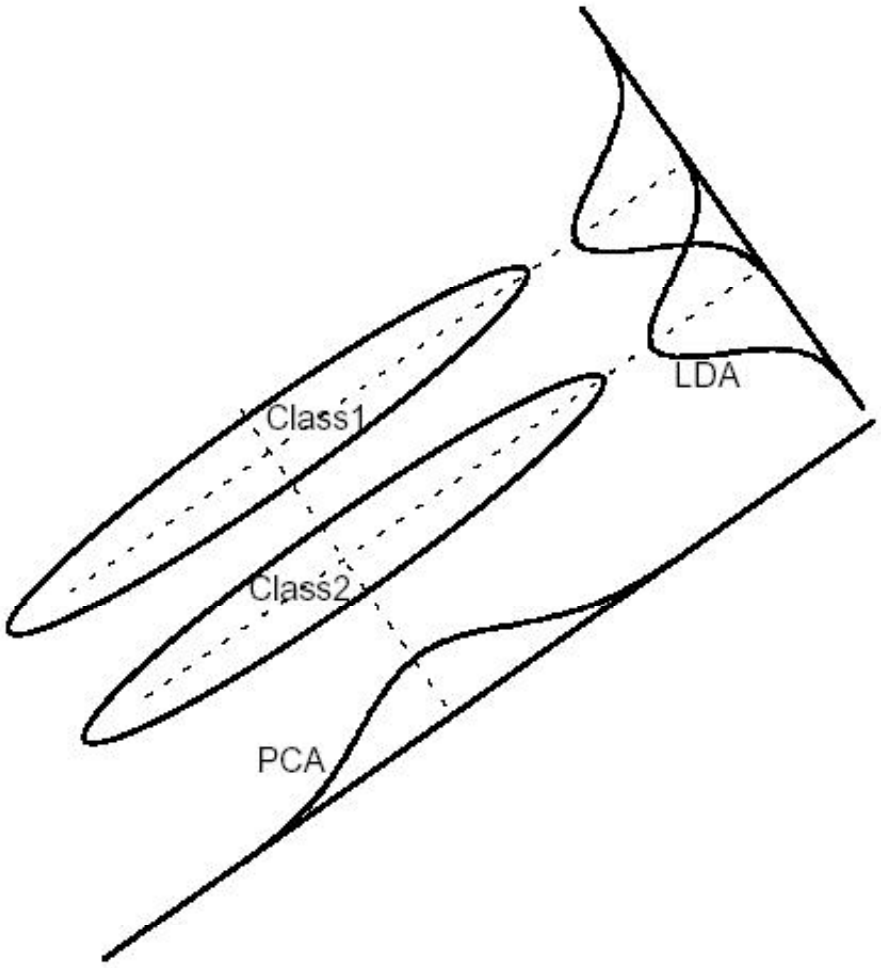


Figure 1.1: An illustration of how PCA fails in the case of two elongated Gaussian classes and LDA performs better by picking a direction that gives good discrimination [Kumar and Andreou, 1996].

Fisher's Linear Discriminant Analysis (LDA) [Kumar and Andreou, 1996] is another classic dimensionality reduction technique. It differs from PCA and ICA because it is not trying to model the density of the data. Instead, it is trying to find a low dimensional space in which classes are well separated. For a k-class N > k dimensional dataset, LDA gives k-1 orthogonal features that are chosen from all linear transformations of the

original N features. The k-l features span a k-1 dimensional subspace. In the subspace, LDA makes the between-class distance in the data as large as possible and makes the within-class scatter in the data as small as possible. Generally, the quotient of the two terms is optimized to achieve the two goals. Figure 1.1 gives an example that reducing dimensions in a discriminative model (LDA) perspective is much better than in a data-representation based model (PCA) perspective. However, LDA only depends on the means and the variances of the data for each class. If the data has a distribution far from Gaussian, LDA may not generate a good subspace in which the data can be easily separated. In addition, LDA can only give a linear mapping, and the dimensionality of the subspace is limited by the number of classes of the data.

1.2.5 Multidimensional Scaling

Multidimensional Scaling (MDS) [Borg and Groenen, 1997] is a linear Model for dimensionality reduction. Given the matrix A that stores the square pairwise distances (here I only discuss the Euclidian distance) between every two data points, MDS generates the low dimensional codes that best preserve these distances.

Suppose that $D = (X_1, X_2, \ldots, X_N)$ is an n by N data matrix with each column denoting one n-dimensional data vector, A is an N by N matrix of squared distances, and B is a matrix that stores inner products of every two data vectors, then we have

$$B = D^T D \tag{1.8}$$

$$B_{ij} = X_i^T X_j \tag{1.9}$$

$$\begin{aligned} A_{ij} &= (X_i - X_j)^T (X_i - X_j) \\ &= X_i^T X_i - 2X_i^T X_i + X_j^T X_j \end{aligned} \tag{1.10}$$

After centering the matrix A, we will get

$$B = -\frac{1}{2} A \tag{1.11}$$

Thus, we can get the matrix B from the square pairwise distance matrix A. Calculate the eigenvalues and the eigenvectors of B, and we can get

$$B = VUV^T \quad (1.12)$$

$$D = VU^{1/2} \quad (1.13)$$

where U is a diagonal matrix that contains the eigenvalues, and V is a matrix which contains eigenvectors as its columns. Choosing the eigenvectors corresponding to the first m (m < n) largest eigenvalues to constitute a low dimensional subspace V', the low dimensional codes of the data matrix D that best preserve the square pairwise distances will be

$$Y = V' U'^{1/2} \quad (1.14)$$

where U' is the diagonal matrix that contains the first m largest eigenvalues.

From the above descriptions, we can find that MDS generates low dimensional codes placing emphasis on preserving the pairwise distances between the data points. If the rows and the columns of the data matrix D both have mean zero, the projection produced by MDS will be the same as that produced by PCA. Thus, MDS is a linear Model for dimensionality reduction having the same limitations as PCA.

1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 2 describes Regularized Autoencoder Network (RAN) in detail. The Stochastic Neighbour Embedding (SNE) algorithm that is a part of RAN will be described in section 2.1.4.
- In chapter 3, some optimization methods that can be used to train RAN will be described.
- Chapter 4 presents some experimental results on digit data using low dimensional codes generated by RAN and related methods. The performance of some selected optimization methods used for the training will be compared.
- Chapter 5 first gives a review of previous techniques used for face recognition, and then shows the performance of some optimization methods used to train RAN on face data. After that, the experimental results on face recognition using low dimensional

codes generated by RAN will be given, and the result will be compared to the results produced by some previous face recognition techniques.

- Chapter 6 concludes by summarizing the content of this thesis and proposing future research directions to extend the current optimization methods and the current RAN model.

Chapter 2

Regularized Autoencoder Network

2.1 Non-linear Dimensionality reduction Techniques

Some models for dimensionality reduction have been introduced in chapter 1. However, these models can only generate linear mappings from the high dimensional space to the low dimensional space and cannot find non-linear structure in the data as discussed in chapter 1. Research on non-linear dimensionality reduction methods has been explored extensively in the last few years. In the following, a brief introduction to several non-linear dimensionality reduction techniques will be given.

2.1.1 Kernel Principle Component Analysis

Kernel PCA [Scholkopf et al., 1998, Mika and Scholkopf et al, 1999], is a kernel version of standard PCA. When used for dimensionality reduction, standard PCA generates low dimensional codes preserving most of the variance in the original data. But sometimes the data is not separable even in the original high dimensional space no matter how we redefine the axes of the data. We hope that the data can be separable in a higher dimensional space and then we can perform standard PCA in that space. Kernel PCA can help us to achieve this purpose while doing the calculations in a lower dimensional space by means of the kernel trick.

Suppose there is a centred data set $M=\{X_i \mid i=1, \ldots, N\}$, where X is an n-dimensional column vector and $X = (x_1, \ldots, x_n)^T$. We define a transformation $\phi(X)$: $R^n \rightarrow R^m$ that maps the data point X to a higher dimensional space in which the data has better separability. Then the covariance matrix of $\phi(X)$ in R^m becomes:

$$C = \frac{1}{N} \sum_i \phi(X_i) \phi(X_i)^T \tag{2.1}$$

To find the principal components, we need to solve the characteristic equation of C, $Cv = \lambda v$. Since the eigenvector v of C must lie in the span of $\phi(X_i)$, i =1, …N, there must exist a set of coefficients α such that $v = \sum_i \alpha_i \phi(X_i)$. Through constructing the symmetric Gram Matrix K and using the kernel trick,

$$\phi(X_i)^T \phi(X_j) = K(i, j) \tag{2.2}$$

the eigenvectors of C can be obtained by solving the following equation:

$$\lambda K \alpha = \frac{1}{N} K^2 \alpha \tag{2.3}$$

And the projection of a data point X_i on the k-th principal component v can be calculated as follows:

$$y_{ik} = v^T X_i = \sum_j \alpha_j K(X_i, X_j) \tag{2.4}$$

By choosing the first r principal components to form a low dimensional subspace R^r and mapping the data to R^r where r < n, we can obtain the low dimensional codes of the data.

2.1.2 ISOMAP

The idea of ISOMAP [Tenenbaum et al., 2000] is to map some high dimensional data to a non-linear low dimensional subspace in a way that preserves a particular kind of structure in the data. It is assumed that the data lies on, or near, a lower dimensional manifold that is embedded in the high dimensional space. The Geodesic Distance between two data points is defined as the shortest distance along the manifold and the aim is to find a low-dimensional representation that preserves the Geodesic distances as well as possible. The Geodesic distances can be estimated by finding shortest paths in a neighbourhood

graph derived from the data. The neighbourhood graph can be constructed by connecting each data point to its k nearest neighbours. Here is a sketch of the ISOMAP algorithm: (1) Construct the neighbourhood graph; (2) Calculate Geodesic Distances between every two data points using a shortest path algorithm; (3) Given these pairwise distances, use MDS (Chapter 1) to find low dimensional codes that preserve the pairwise Geodesic distances as well as possible.

ISOMAP generates low dimensional codes that preserve the non-linear geometry of the data by preserving the Geodesic Distances between every two data points. It does not try to find codes that are optimal for reconstructing the individual data points.

2.1.3 Locally Linear Embedding

Locally Linear Embedding (LLE) [Roweis and Saul, 2000, Saul and Roweis, 2003], is a non-linear dimensionality reduction technique. It generates low dimensional codes that preserve the local structure in the data and ignore the long-range structure. That is to say, "nearby points in the high dimensional space remain nearby and similarly co-located with respect to one another in the low dimensional space." The LLE algorithm [Saul and Roweis, 2000] works as follows:

1. Compute the K nearest neighbours of each data point, X_i (K is a parameter chosen by the user).
2. Compute the weights W_{ij} that best reconstruct each data point X_i from its K neighbours, minimizing the cost in Equation (2.5) under the constraints $\Sigma_j W_{ij}=1$ and $W_{ij}=0$ if X_j is not a neighbour of X_i.

$$\acute{E}(W) = \sum_i \left\| X_i - \sum_j W_{ij} X_j \right\|^2 \tag{2.5}$$

3. Compute the vectors Y_i that are best reconstructed by the weights W_{ij} by minimizing the quadratic form in Equation (2.6) by its bottom nonzero eigenvectors.

$$\begin{aligned} \Phi(Y) &= \sum_i \left\| Y_i - \sum_j W_{ij} Y_j \right\|^2 \\ &= \sum_i \left\| Y(I_i - W_i) \right\|^2 \\ &= \mathrm{trace}(Y(I_i - W_i)\,(Y(I_i - W_i))^T) \end{aligned} \tag{2.6}$$

$$= \text{trace}(YMY^T)$$

here $M = (I_i - W_i)(I_i - W_i)^T$. The constraint that Y should have the mean 0 and the variance I ensures: (1) different coordinates in the low dimensional subspace will be uncorrelated to second order; (2) the reconstruction errors for the coordinates will be measured on the same scale which is of order unity. Under the constraint, Y can be obtained by calculating the d+1 bottom eigenvectors of M, which correspond to the d smallest nonzero eigenvalues of M (d is the dimensionality of Y).

Through this algorithm, every data point is mapped into the low dimensional subspace, in which the high dimensional dot products between the edges in every data point's neighbourhood are preserved as well as possible by the low dimensional dot products. By keeping information from overlapping local neighbourhoods, the global structure of the whole data is maintained, provided that the local neighbourhoods are sufficiently connected.

As has been mentioned above, LLE focuses on generating low dimensional codes preserving local linear geometry in the data. Although both the weights W and the codes Y are obtained by minimizing reconstruction errors, the reconstructions here are different from the ones in PCA. The algorithm does not try to use the low dimensional codes to reconstruct the original data. LLE doesn't care how well each particular low dimensional code represents the original data point.

2.1.4 Stochastic Neighbour Embedding

Stochastic Neighbour Embedding (SNE) [Hinton and Roweis, 2003], which is a special case of Linear Relational Embedding (LRE) [Paccanaro and Hinton, 2000], is a probabilistic approach that maps high dimensional data points into a low dimensional subspace in a way that preserves the relative distances to near neighbours. In SNE, similar objects in the high dimensional space will be put nearby in the low dimensional space, and dissimilar objects in the high dimensional space will usually be put far apart in the low dimensional space.

The main idea behind the model SNE is to use the pairwise distances between points in the low-dimensional space to approximate a discrete probability distribution over neighbours of each data point that is generated by using distances in the high dimensional space. A Gaussian distribution centred on a point in the high dimensional space is used to define the probability distribution that the data point chooses other data points as its neighbours. In the low dimensional space, we generate another discrete distribution in the same way and the sne cost function measures how well the distribution generated in the low-dimensional space models the distribution generated in the high-dimensional space. The cost function is a sum of Kullback-Leibler divergences, one per data point [Hinton and Roweis, 2003].

In the following, the SNE algorithm will be given. For each data point, i, and each of its potential neighbours, j, we calculate the probability that i would choose j as its neighbour:

$$p_{ij} = \exp(-d_{ij}) \ / \ \sum_{k \neq i} \exp(-d_{ik}) \qquad (2.7)$$

where d_{ij} means the dissimilarity between i and j. It maybe computed by the scaled squared Euclidean distance between two data vectors, x_i, x_j

$$d_{ij} = \| x_i - x_j \|^2 \ / \ 2\sigma_i^2 \qquad (2.8)$$

where σ_i is the variance of the Gaussian centred at i. It can be set by hand to make the distribution p_i have a predefined entropy.

In the low dimensional subspace, the same operations are performed. We calculate the probability, q_{ij}, that point i picks point j as its neighbour in a similar way.

$$q_{ij} = \exp(-\| y_i - y_j \|^2) \ / \ \sum_{k \neq i} \exp(-\| y_i - y_k \|^2) \qquad (2.9)$$

The low dimensional code y here aims at preserving the relative distances of neighbours in the high dimensional space. We can achieve this by minimizing Equation (2.6) that is a sum of Kullback-Leiber divergences between the high dimensional and the low dimensional distributions over neighbours for each data point. Consequently, the two distributions are almost the same for each point i.

$$C = \sum_i \sum_j p_{ij} \log(p_{ij} \ / \ q_{ij}) = \sum_i KL(P_i \| Q_i) \qquad (2.10)$$

Due to that each point i in the low dimensional space keeps its neighbour identities in the high dimensional space, the global structure of the original dataset is preserved through integrating the information from overlapping neighbourhoods like LLE. In the paper

[Hinton and Roweis, 2003], it is claimed that SNE is superior to LLE in that SNE has a tight restriction on preserving distances between every two data points. Analyzing the term p_{ij} / q_{ij} in Equation (2.10), we will notice that "making q_{ij} large when p_{ij} is small wastes some of the probability mass in the q distribution so there is a cost for modeling a relatively big distance in the high dimensional space with a relatively small distance in the low dimensional space, though it is much less than the cost of modeling a relatively small distance with a relatively big one".

To calculate the low dimensional codes, we need to derive the derivative of the cost C with respect to the code y.

$$\partial C/\partial y_i = \partial(\sum_i \sum_j p_{ij}\log(p_{ij} / q_{ij})) / \partial y_i \quad (2.11)$$

$$= \partial(\sum_j p_{ij}\log(p_{ij} / q_{ij}) + \sum_j p_{ji}\log(p_{ji} / q_{ji})) / \partial y_i \quad (2.12)$$

Let

$$C_1 = \sum_j p_{ij}\log(p_{ij} / q_{ij}),\ C_2 = \sum_j p_{ji}\log(p_{ji} / q_{ji})) \quad (2.13)$$

So

$$\partial C/\partial y_i = \partial C_1/\partial y_i + \partial C_2/\partial y_i \quad (2.14)$$

$$\partial C_1/\partial y_i = \partial(\sum_j (p_{ij}\log p_{ij} - p_{ij}\log q_{ij})) / \partial y_i$$

$$= -\partial\sum_j (p_{ij}\log q_{ij}) / \partial y_i$$

$$= -\sum_j (p_{ij} q_{ij}^{-1}(\partial q_{ij}/\partial y_i)) \quad (2.15)$$

From Equation (2.9), we obtain that

$$q_{ij}\sum_{k\neq i} \exp(-\|y_i - y_k\|^2) = \exp(-\|y_i - y_j\|^2)$$

Calculate the derivatives with respect to y_i on both sides of the equal sign:

$$(\partial q_{ij}/\partial y_i) \sum_{k\neq i} \exp(-\|y_i - y_k\|^2) + (-2)q_{ij}\sum_{k\neq i} (y_i - y_k) \exp(-\|y_i - y_k\|^2) =$$

$$(y_i - y_j) \exp(-\|y_i - y_j\|^2) (-2)$$

Divided by $\sum_{k\neq i} \exp(-\|y_i - y_k\|^2)$ on both sides, we can get

$$\partial q_{ij}/\partial y_i + (-2)q_{ij}\sum_{k\neq i} ((y_i - y_k) q_{ik}) = (-2)q_{ij}(y_i - y_j)$$

Therefore,

$$\partial q_{ij}/\partial y_i = (-2)q_{ij}(y_i - y_j) - (-2) q_{ij}\sum_{k\neq i} (q_{ik}(y_i - y_k)) \quad (2.16)$$

Substitute the term $\partial q_{ij}/\partial y_i$ in Equation (2.15) using Equation (2.16), we get

$$\partial C_1/\partial y_i = -\sum_j (p_{ij} q_{ij}^{-1}((-2)q_{ij}(y_i - y_j) - (-2) q_{ij}\sum_{k\neq i} (q_{ik}(y_i - y_k))))$$

$$= 2\sum_j p_{ij} (y_i - y_j) - \sum_{k\neq i} (q_{ik}(y_i - y_k))$$

$$= 2\sum_j (y_i - y_j)(p_{ij} - q_{ij}) \quad (2.17)$$

In the same way, we can obtain that

$$\partial C_2/\partial y_i = 2\sum_j (y_i - y_j)(p_{ji} - q_{ji}) \quad (2.18)$$

By combining Equation (2.14), (2.17) and (2.18), we get

$$\partial C/\partial y_i = 2\sum_j (y_i - y_j)(p_{ij} - q_{ij} + p_{ji} - q_{ji}) \quad (2.19)$$

From Equation (2.19), the derivative $\partial C/\partial y_i$ can be viewed as "a sum of forces pulling y_i toward y_j or pushing it away depending on whether j is observed to be a neighbour more or less often than desired". Figure 2.1 shows how the gradient information influences the updating of the low dimensional codes.

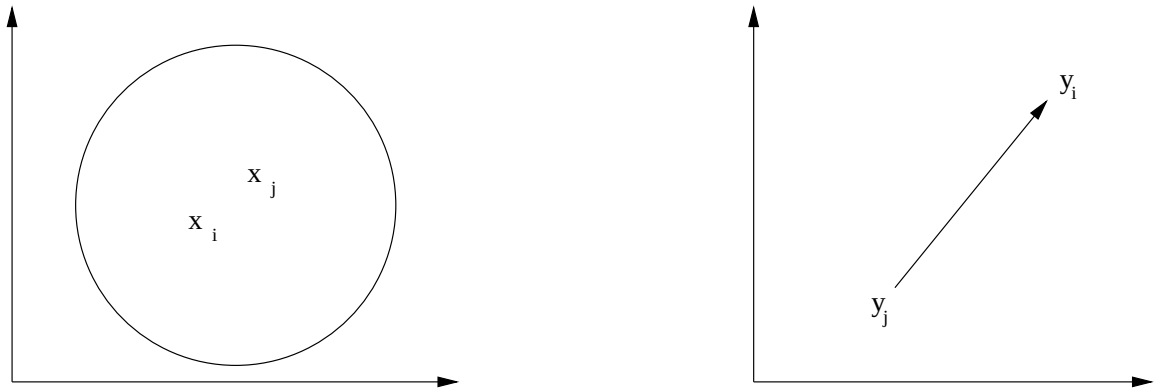


Figure 2.1: An interpretation for the gradient of the cost function C with respect to the low dimensional code vector. The left subfigure shows that in the high dimensional space, x_i chooses x_j as one of its neighbours, but in the low dimensional code space (here is a 2-d space), y_i is far from y_j and $y_j < y_i$. Here, $y_i - y_j < 0$, $p_{ij} - q_{ij} > 0$, $p_{ji} - q_{ji} > 0$. Thus, $-\partial C/\partial y_i > 0$, and y_j will be pull toward y_i.

There are many optimization methods that can be used to minimize the cost C using the gradient $\partial C/\partial y$. Steepest Gradient Descent is not efficient and can easily get stuck in bad local minima. There is a method suggested by [Hinton and Roweis, 2003] that adds random noise that decreases with time when updating the low dimensional codes of the data. The low dimensional codes are initialized by random values very close to 0, and SNE updates the codes making use of the gradient information. Thus, SNE can be viewed as a non-linear model.

Summarizing the descriptions above, we find that SNE generates low dimensional codes that preserve the global structure in the data in a probabilistic way, and it is superior to LLE in keeping the relative distances between every two data points.

2.1.4.1 SNE-encoder

If we implement a three-layer neural network with the cost function of SNE, we will get a new model called SNE-encoder. In SNE-encoder, the first layer represents input data, the second layer contains non-linear Sigmoid activation functions, and the third layer represents low dimensional codes. The structure of SNE-encoder is the same as that of the recognition part of Regularized Autoencoder Network that will be discussed in the next section.

2.2 A New Approach to Generating Low Dimensional Codes

2.2.1 Motivation

So far, several kinds of models for dimensionality reduction have been discussed: Linear and Non-linear. Table 2.1 shows the main characteristics of the different models we have described in the previous sections. To sum up, when used for dimensionality reduction: PCA, ICA, Fisher's LDA, and MDS are all linear models; Kernel PCA, ISOMAP and LLE are non-linear Models. Here, ICA means undercomplete ICA. Most non-linear models can capture both non-linear and linear structures in the data while linear models can only capture linear structures in the data. Thus, generally speaking, non-linear models are more powerful than linear models.

	Non-linear or Linear	Main Advantages or Limitations
PCA	Linear	Preserves most variance of the data; Can only capture second-order correlations between components of the data vectors
ICA	Linear	Captures higher-order statistics in the data; Works well only when the data sources are independent
LDA	Linear	Cannot handle data in which the individual classes are far from Gaussian or in which the classes have different covariances
MDS	Linear	Best preserves the pairwise distances between every two data points
Kernel PCA	Non-linear	The Gram Matrix grows with the number of data points
ISOMAP	Non-linear	Preserves the global manifold structure of the data by preserving the Geodesic distances between every two data points
LLE	Non-linear	Preserves the locally linear structure in the data
SNE	Non-linear	Preserves the relative distances to near neighbours

Table 2.1: The main characteristics of different dimensionality reduction techniques and the comparisons.

But we want to have a non-linear model that has the advantages of both SNE and PCA and will improve nearest neighbour classification.

2.2.2 Regularized Autoencoder Network

Before describing Regularized Autoencoder Network (RAN), we will first introduce the autoencoder.

The autoencoder discussed here is a five layer neural network. It consists of a recognition network and a generative network. The recognition network implemented by the first three layers will generate low dimensional codes for the input images and the generative network implemented by the last three layers will produce reconstructions of the input images. The recognition network can be viewed as an encoder converting the input into a low dimensional code and the generative network can be viewed as a decoder converting the code back into a reconstruction of the input. The units in the second layer and in the fourth layer have Sigmoid activation functions. Figure 2.2 illustrates the structure of the autoecoder.

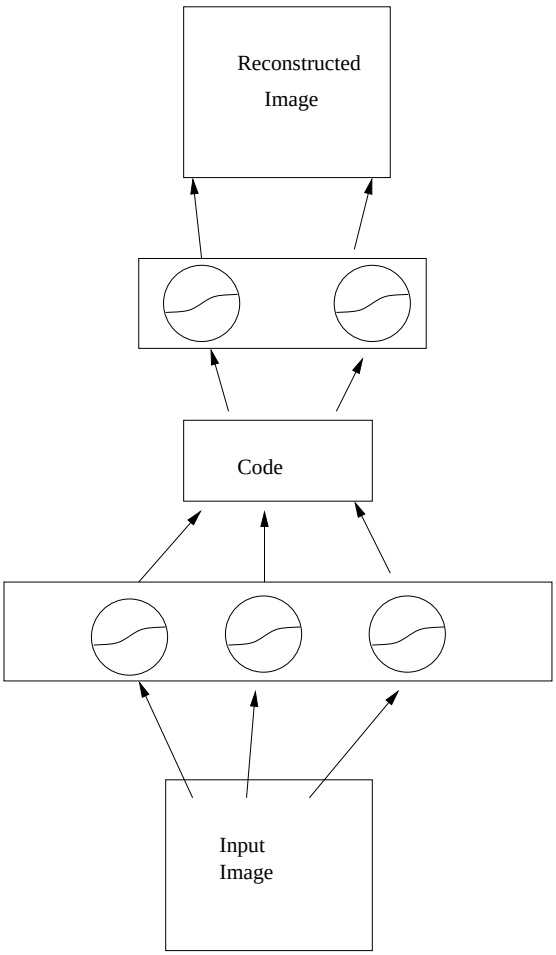


Figure 2.2: The structure of the autoencoder.

In the thesis, the objective function of the autoencoder is the sum of square reconstruction errors for face data and the cross entropy function for digit data. The autoencoder can be viewed as a non-linear generalization of PCA if the objective function is the sum of square reconstruction errors over all the training cases.

The autoencoder captures variance in the data as PCA does. Both the autoencoder and PCA focus on generating faithful reconstructions for input data except that the autoencoder gives a non-linear mapping while PCA gives a linear mapping. However, we want to have a model that can not only capture the variance in the data but can also preserve local structure within the data that is important for clustering. A natural way is to combine the autoencoder and SNE together, and we call this model the Regularized Autoencoder Network (RAN).

RAN has the same structure as the autoencoder. It consists of a recognition network and a generative network as shown in Figure 2.2. We combine the objective function of the autoencoder and the objective function of SNE in (2.10) with a multiplier τ before it to get the objective function of RAN as shown in (2.20):

$$C = C_{auto} + \tau_{sne} C_{sne} \qquad (2.20)$$

In (2.20), C_{auto} can either be the cross entropy over the training data or be sum of square reconstruction errors over the training data. In the objective function of RAN, the second part $\tau_{sne} C_{sne}$ can be viewed as a penalty term on the activities of units in the third layer of RAN. By varying the value of τ_{sne}, we can make the model suitable for learning data with different properties. If τ_{sne} is set to be very small, the behaviour of RAN will be similar to that of the autoencoder; if τ_{sne} is set to be very large, the behaviour of RAN will be similar to that of SNE. Therefore, if we hope that RAN will have both the advantages of the autoencoder and the advantages of SNE, τ_{sne} must be carefully chosen.

2.3 Comparisons of the Autoencoder and SNE-encoder

In this section, we will describe a unifying implementation view of the autoencoder and SNE-encoder, that is, as neural networks. From the point of the view of the implementation, we will give comparisons of the autoencoder and SNE-encoder. Besides that, we will also discuss how and why the autoencoder and SNE can give guidance to each other during the training, which will give some explanations for why Regularized Autoencoder Network works well.

2.3.1 The Autoencoder Imposes Loose Constraints on the Codes

In the previous sections, we have described the fundamentals of SNE, SNE-encoder and the autoencoder. To facilitate subsequent discussions, we will describe the implementation of SNE-encoder and the autoencoder in a formal way first; And all the input data in this thesis refers to digit images or face images, so we call the high dimensional space as pixel space and the low dimensional code space as code space. As discussed earlier, the autoencoder is a five-layer neural network that implements non-linear Principal Component Analysis (PCA). As described earlier, SNE-encoder can be implemented by a three-layer neural network. The second layer contains non-linear Sigmoid activation functions as in the autoencoder. The implementation framework of SNE-encoder is the same as the recognition part of the autoencoder, except that SNE-encoder has a cost function to constrain the low dimensional codes while the autoencoder utilizes the gradient information backpropagated from the generative part to update the low dimensional codes. Now consider the five-layer autoencoder, the first three layers can be viewed as a non-linear continuous function r, which maps some input image x from a pixel space to a code space to produce a vector y.

$$y = r(x) \qquad r: R^n \rightarrow R^m \tag{2.21}$$

Here n is the dimensionality of the pixel space and m is the dimensionality of the code space. The last three layer of the autoencoder can be viewed as a non-linear continuous function g, which maps some code y to the pixel space to generate the reconstruction

image $\hat{x}$ for some input image x.

$$\hat{x} = g(y) \qquad g: R^m \rightarrow R^n \tag{2.22}$$

Combining (2.21) and (2.22), we will have

$$\hat{x} = g(r(x)) \tag{2.23}$$

For the autoencoder, we shall assume that the cost is the sum of the square reconstruction errors over all the input data[1], that is,

$$C = \Sigma_i (x_i - \hat{x}_i)^2 \tag{2.24}$$

We are seeking low dimensional codes y that account for the variance of the input images in the pixel space. Since $\hat{x}$ is an approximation of x, the function g can be viewed as an approximation of the inverse of the function r from (2.23). Because the second layer and the fourth layer are non-linear hidden units, the cost function (2.24) is not strictly concave with respect to the weights w in the autoencoder, and there may be many local minima. That is to say, there may be many continuous mapping functions g and r that can give locally optimal reconstructions of input images.

Under the cost (2.24), we just penalize the large reconstruction errors regardless of what the distance between every two input images might be. As a result, there exist some risks that the low-dimensional codes distort the geometric structure within some input data points. This will happen especially when we have not enough images for each class and the autoencoder cannot capture enough information about the mean of the images for each class or when the dimensionality of the code space is too small and the code cannot capture enough variance of the input image data.

2.3.2 SNE-encoder Imposes Rigid Constraints on the Codes

Now we turn to discussions of the implementation of SNE-encoder. Let's call the mapping function of SNE-encoder from the pixel space to the code space as r', that is,

[1] In the thesis, for face data, the cost is the sum of reconstruction errors; for digit data, the cost is cross entropy.

$$y = r'(x) \qquad r': R^n \rightarrow R^m \tag{2.25}$$

Function r' for SNE-encoder is similar to function r for the autoencoder except that r' has extra constraints on the low dimensional codes while function r has some loose constraints on the codes imposed by the information from function g.

Noticing the cost function of SNE (also the cost function of SNE-encoder) in (2.10) that constrains low dimensional codes, we write it here again:

$$C = \sum_i \sum_j p_{ij} \log(p_{ij} / q_{ij}) = \sum_i KL(P_i \| Q_i) \tag{2.10}$$

At the same time, the following constraints hold:

$$\Sigma_j p_{ij} = 1, \quad \Sigma_j q_{ij} = 1 \tag{2.26}$$

, where $p_{ij} = p_{ji}$ and $q_{ij} = q_{ji}$.Using Lagrangian multiplier, we can easily obtain that the cost C in (2.10) will be minimized when $p_{ij} = q_{ij}$. That is to say, we must find good distributions for every Q_i in the code space to approximate the corresponding P_i in the pixel space to minimize the cost of SNE-encoder. From the view of equations and referring to (2.7) and (2.9), to make $p_{ij} = q_{ij}$ is to solve the following problem:

$$p_{ij} = \exp(-\| y_i - y_j \|^2) / \sum_{k \neq i} \exp(-\| y_i - y_k \|^2) \tag{2.27}$$

Recalling that $p_{ij} = p_{ji}$, we will have an equation group composed of $\frac{N(N-1)}{2}$ equations that have the same form as (2.27). Suppose that the dimensionality of the pixel space is m, the dimensionality of the code space is n, and the number of input images is N (we will follow this assumption all the time in this thesis), there will be $n \times N$ unknown variables for the equation group. Generally, n is much smaller than $\frac{N-1}{2}$. We may think that there must exist many redundant or conflicting equations in the equation group. Therefore, we can't find an exactly accurate solution in most situations, and we must compromise to get an approximated solution.

For the autoencoder, we know that the cost in (2.24) will be minimized when we have N good reconstructions of the input images. There are $n \times N$ unknown variables and $m \times N$ equations. If $m << \frac{N-1}{2}$, we draw a conclusion that SNE has far more constraints on the low dimensional codes than the autoencoder.

The above argument may seem to be correct. However, let's consider the MDS model in which distances between every two input data points are Euclidean distances and the PCA model. We use these two models to generate the mapping from the pixel space to the code space as discussed above. Using the above argument, MDS imposes $\frac{N(N-1)}{2}$ constraints on the low dimensional codes while PCA imposes $m \times N$ constraints on the codes. By the argument, they will generate different low dimensional codes. However, these two models are equivalent and they will generate the same low dimensional codes. Therefore, the argument is definitely not correct. Actually, the number of equations cannot be indicative of degree of constraint because there are some redundant or conflicting equations in the equation group, which has been mentioned in previous discussions. The number of effective equations can be indicative of degree of constraint but not just simply the number of equations.

As a matter of fact, the autoencoder imposes loose constraints on the codes because function g must be learned and we have many choices for g. We cannot define a concrete equation group from the view of equations. It is not like PCA or MDS in which the codes can be calculated directly when minimizing some cost function. But for SNE-encoder, it is not the case. The low dimensional codes are directly influenced by the distributions Q in the code space. Constrained by (2.26), the probabilities must be divided appropriately among every code's neighbours. In another word, the poor low dimensional codes will waste probability mass and then influence the generation of some other low dimensional codes. That is to say, the codes produced by SNE are results of compromise among the codes of some neighbouring input images.

2.3.3 Geometric Intuitions of SNE and SNE-encoder

Minimizing the cost of SNE and SNE-encoder in (2.10) can produce codes that will almost preserve the distances between input images pairwise in the pixel space. When the dimensionality of the pixel space and the dimensionality of the code space are the same,

SNE and SNE-encoder can produce codes that will perfectly preserve the distances. Of course, the dimensionality of the code space is often far smaller than that of the pixel space. We can only expect to have some approximations to the $\frac{N(N-1)}{2}$ distances. Whether we can have very good approximations or not depends on the internal geometric structure of the input image data set and the dimensionality of the code space.

Let's consider the eight vertices on a cube that lies in a three-dimensional space shown in Figure 2.3, can we use SNE and SNE-encoder to map the coordinates of all the vertices to a two-dimensional space at the same time the distances between pairs of vertexes are preserved?

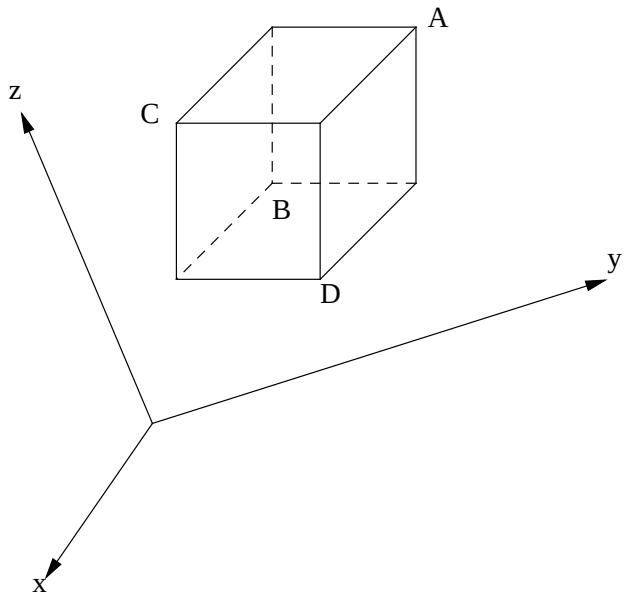


Figure 2.3: Map the three-dimensional coordinates of eight vertices of a cube to a two dimensional space.

The answer is no. Here is a simple and intuitive explanation: looking at Vertex A, B, C and D, we can find that the length of Segment AC, CD and DA are equal. In a two-dimensional space, Vertex A, C and D must form an equilateral triangle. Vertex B has the same distance to Vertex A, C and D, so B must lie at the centroid of the triangle, but we cannot make the ratio of the distance CD over the distance BC be 1.

For some points in a three-dimensional space, if they can be mapped to a two-dimensional space and the corresponding distances can be preserved, these points must lie on a plane. By analogy, if some points in an m dimensional space can be mapped to an n dimensional space (n<m) preserving the corresponding distances, there must exist a special transformation to the m dimensional coordinate system. By the transformation, we can find m new base vectors to form a new coordinate system, in which the last (m-n) components of the new coordinates of all the points are all zero or very tiny. Such kind of transformation doesn't always exist. For example, we cannot find a good transformation that succeeds in mapping coordinates of the vertexes of the cube above. The larger the dimensionality of the code space is, the more flexibility the codes will have. When the dimensionality of the code space is too small (n equals 1 or 2 for face images), many geometric structures within local neighbourhoods of input images cannot be preserved.

2.3.4 Relationship among the Autoencoder, SNE and SNE-encoder

Function r' in SNE-encoder has almost the same framework as that of function r in the autoencoder. If we put another two-layer units above the SNE-encoder framework referring to the autoencoder, we will get a new five-layer neural network. If we make the last three layers have a very good approximation to the inverse function of r' under the cost of the autoencoder in (2.24), the new five-layer neural network can be viewed as an autoencoder with some constraints on the low dimensional codes. According to Fourier Theorem that any continuous function can be approximated arbitrarily well by a set of harmonic functions, theoretically speaking, the last three-layer units can surely have a good approximation to the inverse function of function r' if there are enough hidden units in the fourth layer.

Enlightened by the above argument, we train the first three layers of units using the cost of SNE, and stop the training when the first three layers of parameters are not very far from a good local minimum, and then we train the last two layers of parameters with

the low dimensional codes and the first three layers of parameters fixed, and then we stop the training before the convergence of the last two layers of parameters, and now we throw the SNE penalty on the low dimensional codes and train the five-layer neural network as an autoencoder. What will happen? Mostly, the low dimensional codes produced by the special training process will converge to the codes produced by SNE-encoder alone, provided that the number of hidden units in the second layer and in the fourth layer is enough for the autoencoder alone and SNE-encoder alone to work well. That is to say, the last three-layer neural network gives a very good approximation to the inverse function of function r' in SNE-encoder. It also suggests that the autoencoder will behave almost the same way as SNE-encoder if we put some constraints in the code space and give the autoencoder enough guidance. And, it reflects that a five-layer autoencoder is very flexible in generating low dimensional codes.

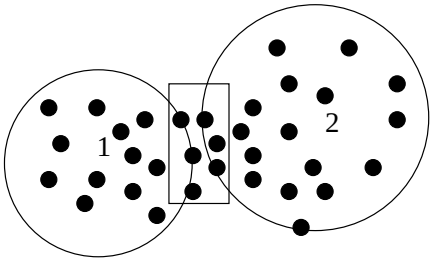


Figure 2.4: Images near the border of classes in which the images have similar shapes. Both the autoencoder and SNE have difficulty in producing good codes for the images in the rectangle so that they can be easily separated in the code space.

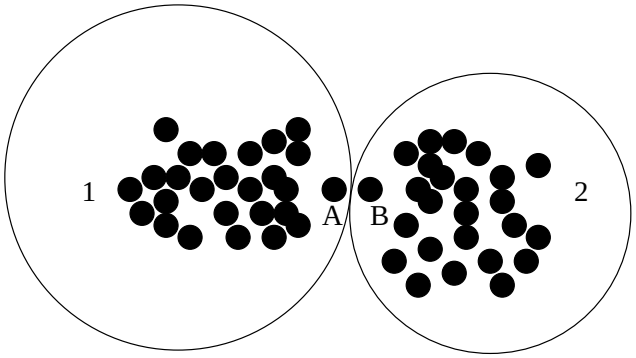


Figure 2.5: The illustration of two images near the border of classes for which SNE can often generate good

codes for KNN clustering.

We have discussed that the plasticity that the autoencoder gives to the low dimensional codes can lead to good things and can also lead to bad things. Thus, if we set the degree of constraint on the codes appropriately and combine the advantages of the autoencoder and SNE on recognizing images, we will have the model Regularized Autoencoder Network (RAN). If the input images for each class are enough for the autoencoder to capture the image information for each class, we set a small degree of constraint in the code space so that the constraint will not ruin the good things produced by the flexibility of the autoencoder; otherwise, we set a big degree of constraint in the code space. In this way, minimizing the cost of SNE will dominate the training of RAN, and the reconstruction term will function as a weak constraint on the codes. Noticing that the low dimensional codes produced by SNE are often results of compromise when $m < n$, we expect that, in general, the compromise that gives good reconstructions of input images will produce better codes than that that gives bad reconstructions.

The autoencoder focuses on giving faithful representations of input images while SNE and SNE-encoder focuses on preserving local relationships between images. Although both of them have their own characteristics on recognizing images near borders of classes using low dimensional codes, they both behave randomly when performing these tasks. Under the cost functions in (2.10) and (2.24), both models produce codes that are closely related to Euclidean distances between pairs of input images. In another words, both models cannot easily differentiate the images near the border of classes in which images have similar shapes. As is shown in Figure 2.4, images in class 1 and class 2 have similar shapes (it means that the input image vectors for the two classes have similar distribution in the pixel space), and the Euclidean distances between pairs of images in the rectangle (near the border) are not very different compared to the inter-class differences. For such cases, both the autoencoder and SNE have difficulty in producing good low dimensional codes by which images belonging to different classes can be easily separated. Figure 2.5 illustrates two images A and B near the border of classes in the pixel space for which SNE and SNE-encoder can generate good codes for KNN clustering

while the autoencoder often fails to this. During the training of SNE and SNE-encoder, the forces pulling the data point A toward class 1 are much bigger than those toward class 2, and the forces pulling the data point B toward class 2 are much bigger than those toward class 1. Therefore, it is more likely that A and B will be respectively clustered into the right classes by SNE and SNE-encoder although A and B choose each other as their nearest neighbours.

Chapter 3

Optimization Methods

We have discussed the RAN model in previous sections, and we now describe how the model can be trained. In the following sections, we describe the optimization methods that we tried: Steepest Gradient Descent (SGD), Scaled Steepest Gradient Descent method (SSGD), Conjugate Gradient Descent (CGD) and Scaled Conjugate Gradient Descent (SCGD). These methods will be compared on particular datasets in chapter 4 and 5.

3.1 Steepest Gradient Descent

Steepest Gradient Descent (SGD) is widely used for training neural networks. The algorithm works like this: the initial value of the weight vector w_0 for a neural network is set to be some value, which is often chosen randomly. Typically, small initial weights are used to avoid strongly biasing the learning. At iteration k, we compute the search direction p_k, which is the negative gradient of the objective function calculated at w_k, and then we update the weight vector along the search direction p_k as follows:

$$w_{k+1} = w_k + \Delta w_k \tag{3.1}$$

$$\Delta w_k = -\eta g_k \tag{3.2}$$

We keep updating the weight vector according to the rule in (3.1) and (3.2) until we are close to a minimum and the objective function of the neural network is approximately

locally optimized so that the gradient g is near 0. The parameter η here is called the learning rate, and in practice, its value is often set by hand.

If η is set to be too large, the update Δw_k for the weight vector w will be large each iteration, and the error may increase; if η is set to be too small, the update for the weight vector will be small each iteration, and the training will take a long time. Therefore, the value for the learning rate must be carefully chosen.

In practice, the standard SGD method is usually very inefficient. When the condition number of the Hessian matrix of an objective function is large, the curvature of the objective function will vary significantly with direction. Under such a situation, the search direction along the gradient will not point to the minimum at most points on the error surface. Therefore, it will take many updates for SGD to achieve the minimum. Figure 3.1 illustrates the behaviour of SGD method when the curvature of the objective function varies a lot with direction. Adding a momentum term to the update of the weight vector at each iteration as in (3.3) will partly solve this problem [Bishop, 1995].

$$\Delta w_k = -\eta g_k + \mu \Delta w_{k-1} \tag{3.3}$$

In the long valley of the error surface where the curvature in one direction differs a lot from that in another direction, adding the momentum term to the update along the steepest descent direction makes the algorithm achieve the minimum faster because successive updates along the direction given by the momentum tends to cancel and they actually result in bigger step size toward the minimum each iteration.

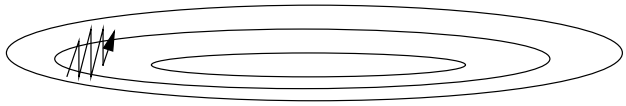


Figure 3.1: The behaviour of SGD method when the curvature of the objective function varies a lot with direction.

Although adding a momentum term improves the performance of SGD when the Hessian has large condition number, the speed of convergence is still slow and the algorithm remains inefficient.

3.2 Scaled Steepest Gradient Descent

Scaled Steepest Gradient Descent (SSGD) method is an enhanced version of SGD. It was designed for optimizing an objective function of which the Hessian has a large condition number.

When we optimize the objective function of a neural network, at each update of the weight vector, instead of simply moving a fixed step size along the steepest descent direction as in (3.2), we have an extra learning rate for each weight in the network, which can be adapted automatically. Then the update rule becomes:

$$\Delta w_{ik} = -\eta \, \xi_{ik} \, g_{ik} \tag{3.4}$$

Here Δw_{ik} denotes the update of the i^{th} component of the weight vector that is a weight in the neural network in iteration k, ξ_{ik} denotes the gain on the learning rate for the weight in iteration k, and g_{ik} denotes the i^{th} component of the gradient vector that is the gradient of the objective function with respect to the weight in iteration k. When the derivative of the objective function with respect to a weight has the same sign in consecutive iterations, it means that the update still has a component in the old direction and it is likely to be a steady downhill direction, so we should increase gain ξ for the weight; otherwise, it means oscillation has occurred, so we should decrease the gain for the weight. Noticing that the sign of Δw_{ik} is the converse of the sign of g_{ik}, therefore, we can adapt the gains for different weights in the neural network according to the rule in (3.5).

$$\xi_{ik} = \begin{cases} \xi_{ik-1} + \omega_1 & \text{if } \Delta w_{ik-1} \, g_{ik} \le 0 \\ (1-\omega_2) \, \xi_{ik-1} & \text{if } \Delta w_{ik-1} \, g_{ik} > 0 \end{cases} \tag{3.5}$$

Here ω_1, ω_2 are small positive numbers set by hand. To avoid the gains becoming extremely small, we also have the rule in (3.6) to update them after they are adapted by the rule in (3.5).

$$\xi_{ik} = \max(\omega_3, \xi_{ik}) \tag{3.6}$$

Here ω_3 is a small positive number. An upper bound can also be imposed on the gains ξ_{ik} to avoid the gains becoming extremely large, but it is not used in the thesis.

During the training of RAN, ω_1 is set to be 0.01, ω_2 is set to be 0.02 and ω_3 is set to be 0.05. And momentum term is also used. In fact, the update for each weight in RAN is as in (3.7).

$$\Delta w_{ik} = -\eta \xi_{ik} g_{ik} + \mu \Delta w_{ik-1} \tag{3.7}$$

In the long valley of the weight space where the curvature of the objective function changes significantly with direction, different learning rates in different directions adapted by SSGD method help to find the minimum in a fast and stable way.

3.3 Conjugate Gradient Descent

The linear Conjugate Gradient Descent (CGD) method [Nocedal and Wright, 1999] was first proposed for optimizing quadratic functions. A set of vectors $p_1, p_2, \ldots, p_n$ are said to constitute a conjugate set with respect to a non-singular symmetric matrix A if they satisfy (3.8).

$$p_i^T A p_j = 0 \qquad \forall ij: i \neq j, i=1,\ldots, n \tag{3.8}$$

Suppose that we have a quadratic objective function of a simple linear neural network, $E(w): R^n \rightarrow R$, CGD can generate n exact search directions composed of a conjugate set with respect to the Hessian of E, which can be calculated recursively as in (3.9).

$$p_{k+1} = -r_k + \beta_{k+1} p_k \tag{3.9}$$

where

$$r_k = g_k \tag{3.10}$$

$$\beta_{k+1} = r_{k+1}^T r_k / \; p_k^T r_k \tag{3.11}$$

$$r_{k+1} = r_k + \alpha_k E''(w_k) p_k \tag{3.12}$$

$$\alpha_k = p_k^T r_k / p_k^T E''(w_k) p_k \tag{3.13}$$

Here g_k is the gradient of E calculated at w_k and $E''(w_k)$ means the Hessian of the quadratic function E.When k = 0, r_0 is the gradient of the objective function E at the starting point w_0, and the value of w_0 is chosen at random. Along these search directions,

if we update the vector w as in (3.14), the quadratic function E can be optimized within n steps [Nocedal and Wright, 1999].

$$w_{k+1} = w_k - \alpha_k p_k \tag{3.14}$$

From the above descriptions, we can find that we only need the information obtained in the previous iteration to calculate the weight vector w_{k+1} at iteration.

However, the above algorithm only works well for minimizing convex quadratic objective functions. For a general non-quadratic objective function of a general neural network, E(w): $R^n \rightarrow R$, Fletcher and Reeves showed that, by approximating the conjugate search directions described above and by using line search to calculate the step size that satisfies the Wolf Conditions, we can obtain a good non-linear version of CGD [Nocedal and Wright, 1999]. Here the Wolf conditions can ensure that sufficient decrease in each step will be made and the updated vector g_{k+1} will not be too far away from stationary points of the objective function E. There are many variants of the non-linear CGD method, and most of them only differ in the choice of the parameter β_{k+1} in (3.11), which will result in different approximations to the conjugate search directions. One of the most important invariants was proposed by Fletcher and Reeves, the β_{k+1} is defined as in (3.15).

$$\beta_{k+1} = g_{k+1}^T (g_{k+1} - g_k) / g_k^T g_k \tag{3.15}$$

Here g_{k+1} is the gradient of the non-linear objective function E calculated at w_{k+1}.

To make line search fast, we often utilize the information about the function values and the gradients at the previous tried points to fit a quadratic or cubic function, then we find the minimum of the interpolation function and use it as a new trial point. During the evaluation of each trial point, we must calculate the function value and the gradient of the objective function at the point, which is expensive. Therefore, we often set a limit to the number of function evaluations allowed in each line search.

Despite that, using CGD with step sizes calculated by line searches to train neural networks that have a huge number of weights or objective functions that are computationally expensive to evaluate is still slow and often takes a long time. And the performance of the method is sensitive to the parameters used in the line search procedure.

3.4 Scaled Conjugate Gradient Descent

Scaled Conjugate Gradient Descent (SCGD) was introduced to avoid line searches along the conventional conjugate search directions [Moller, 1993].

SCGD works in a similar way to the linear version of CGD except that it approximates the step size term containing the Hessian in (3.13) and it introduces a strategy to make the Hessian of the non-linear objective function always positive definite during each update of the weight vector.

Within the neighbourhood of each updated weight vector, SCGD generates quadratic approximations to the objective function as in (3.16).

$$E_{qw}(y) = E(w+y) = E(w) + E'(w)^T y + \frac{1}{2} y^T E''(w) y \qquad (3.16)$$

And SCGD gives step size like linear CGD based on the quadratic approximations by approximating the term that is the product of the Hessian $E''(w_k)$ and the search direction vector p_k as in (3.17).

$$s_k = E''(w_k)p_k = [E'(w_k + \sigma_k p_k) - E'(w_k)]/ \sigma_k \qquad (3.17)$$

Here $E'(w_k)$ is the gradient of the objective function E and $0 < \sigma_k << 1$. By adding some multiple of the unit matrix to the Hessian $E''(w_k)$, we can ensure that the Hessian will be positive definite. Then we will have

$$s_k = [E'(w_k + \sigma_k p_k) - E'(w_k)]/\sigma_k + \lambda \; p_k \qquad (3.18)$$

We denote the denominator in (3.13) by δ_k, and we will have

$$\delta_k = p_k^T E''(w) p_k \qquad (3.19)$$

If the Hessian is not positive definite, we will have $\delta_k < 0$, then we raise λ to make the Hessian positive definite, then the updated δ_k will become

$$\delta_{k2} = \delta_k + (\lambda_{k2} - \lambda_k) p_k^T p_k > 0 \qquad (3.20)$$

From (3.20), we can easily get

$$\lambda_{k2} > \lambda_k - \delta_k / p_k^T p_k \qquad (3.21)$$

It is not known that how to set λ_{k2} to get an optimal step size. In [Moller, 1993], λ_{k2} is

set to be $2(\lambda_k - \delta_k / p_k^T p_k)$. SCGD also measures how good the quadratic approximation at each update is using the formula in (3.22) to increase or reduce the parameter λ.

$$\Delta_k = [E(w_k) - E(w_k + \alpha_k p_k)] / E(w_k) - E_{qw}(\alpha_k p_k) \quad (3.22)$$

Where $E_{qw}(\alpha_k p_k)$ is the quadratic approximation to the objective function at $w_k + \alpha_k p_k$. If $\Delta_k < 0.25$, $\lambda_k = 4\lambda_k$; $\Delta_k > 0.75$, $\lambda_k = 0.5\lambda_k$. The smaller the λ, the bigger the step size.

SCGD can minimize non-linear objective functions of neural networks along some approximated conjugate search directions, but the mechanism used to increase and reduce the parameter λ is not accurate, and it doesn't guarantee that the update of the weight vector always stays in the trust region.

Chapter 4

Experiments on Digit Recognition

4.1 Digit Data and the Configurations of Models

The digit data used here contains 9000 hand-written digit images with the greyscale from 0 to 1. There are 900 images for each of the 10 digits. In the experiments, we divided the digit data into three disjoint datasets with each containing 3000 digit images and 300 images for each digit. We label the three digit datasets as: set 1, set 2 and set 3. The size of each image is 16 by 16. Figure 4.1 shows 100 images for 10 digits.

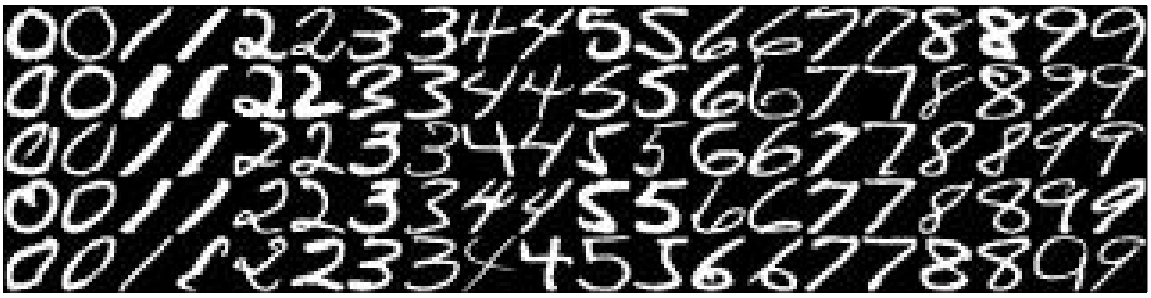


Figure 4.1: Some images of hand-written digits in the digit dataset.

In the pixel space, the hold-one-out clustering error by 4NN is 180 out of 3000 on set 1, 200 out of 3000 on set 2, and 196 out of 3000 on set 3.

Since most of the pixel values of a digit image are either near 0 or near 1, we used logistic outputs for the units in the fifth layer of an autoencoder and we used

cross-entropy instead of the sum of square reconstruction errors as the objective function of the autoencoder because it makes the calculations of derivatives easier. Figure 4.2 shows the histogram of the pixel values of a handwritten digit image. Figure 4.3

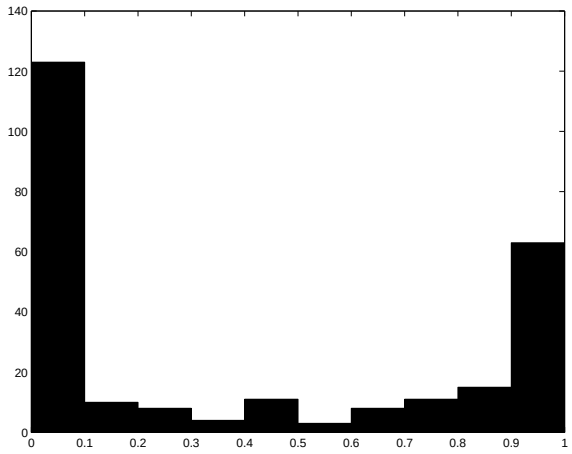


Figure 4.2: The histogram of the pixel values of a hand-written digit image.

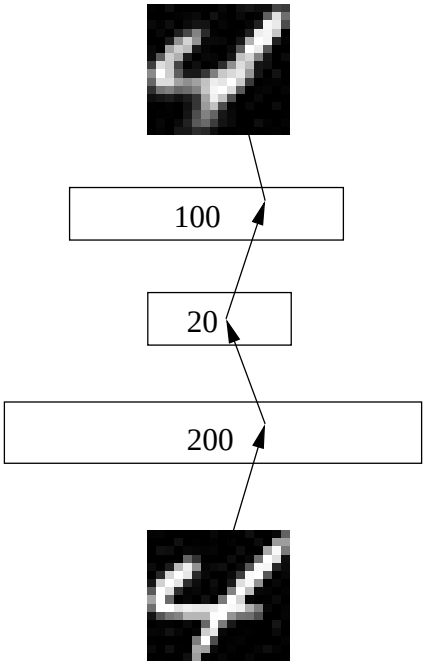


Figure 4.3: The configuration of the autoencoder and RAN for digit data.

illustrates the configuration of an autoencoder. Through a lot of experiments, we find that the following configuration of the autoencoder works very well on digit recognition: the

number of hidden units in the second layer is 200, the number of hidden units in the third layer (i.e., the dimensionality of the code space) is 20, and the number of hidden units in the fourth layer is 100. The structure of RAN is the same as that of the autoencoder except that we use the SNE cost to regularize the activities of the units in the third layer.

4.2 The Training of RAN by Different Optimization methods on Digit Data

4.2.1 Single batch Training

When 3000 training cases are used as one batch to train RAN: even if the learning rates are carefully chosen, the training by SSGD is slow, and the training by CGD and SCGD is intolerably slow because choosing step sizes to optimize the objective function of RAN on the 3000 training cases is very difficult. SSGD works better than the other two methods do under this situation.

However, the objective function of a SNE-encoder is easy to minimize using the 3000 training cases as one batch and the training is fast.

4.2.2 Mini-batch Training

We used mini-batch data to train RAN and we found that the training was much faster than that using all the data as one batch.

4.2.2.1 Mini-batch Data

Many experiments on training RAN have been done on different data sets with mini-batch data in which one of the training cases is assigned to only one mini-batch. The results show that: (1) SSGD has the best performance. (2) CGD doesn't work on

mini-batch data with big mini-batch size (the number of cases in one mini-batch), because it is very expensive to get an allowable step length along the search direction to satisfy the Strong Wolf Conditions or because the training oscillates on different mini-batches and no progress can be made. When CGD is applied on the whole data set with 3000 cases as mentioned in section 4.1 (the extreme situation), it takes almost one minute to evaluate the objective function value of RAN on our machine, and the method is often trapped into a bad region and the value of the objective function cannot be reduced. (3) With the right mini-batch size, the performance of SCGD is comparable to that of SSGD. Training by SCGD is faster than that by SSGD.

4.2.2.2 Redundant Mini-batch Data

In those experiments on mini-batches described in section 4.2.2.1, one training case is only assigned to one mini-batch, and we will lose some similarity information between some pairs of training cases and only a small amount of the pairwise similarity information is used during the training. Thus, we did the following to get a new kind of mini-batch data called redundant mini-batch data: permute the 3000 training cases randomly, divide the 3000 cases into several mini-batches with the same size, and repeat doing the above process until we get the desired number of mini-batches. When redundant mini-batch data is used during the training of RAN, much more pairwise similarity information will be utilized when minimizing the KL divergence part over the mini-batches.

In the following, we will show the experimental results on set 1 using 15 times redundant mini-batch data in which the mini-batch size is 200 and there are 225 mini-batches altogether. Table 3.1 gives the summary of the experimental results.

Table 4.1 shows that SSGD finds the best parameters for RAN by which we can generate the best low dimensional codes for KNN clustering. We set τ_{sne} (the coefficient in front of the cost of SNE) to be 150. Here, in SSGD, we set the learning rates to be 5.0e-5; in SCG, we used the following strategy to control the line searches on each mini-batch: set the maximum number called L of line searches allowed on every

mini-batch each time to be not very small (we set it to be 10 here). At the same time, make sure that not much time is wasted on some batches without finding a successful line search. That is to say, L cannot be set to be a large number; in SCGD, we set the parameter λ, used for adjusting the Hessian of the objective function and the step size along the P-R Conjugate Gradient search direction, to be 5.0e-3. We allowed a big step size at first and let the method automatically reduce λ by itself when necessary. And the maximum number of successful searches on each mini-batch each time is 10.

	4NN Errs	Time Cost (hour)
SSGD	90	2.2
CGD	120	11.3 (Training oscillates on different mini-batches)
SCGD	98	1.2

Table 4.1: The performance of different optimization methods on optimizing RAN using redundant mini-batch digit data with the mini-batch size 200.

We also used the three digit datasets mentioned in section 4.1 to give more detailed comparisons of the three optimization methods. We use one of the three sets as a validation set to determine when to stop training and use the other two corresponding sets as test sets to test different methods. Each dataset is used to generate 225 mini-batches of data with the mini-batch size 200. On the validation set, we stop the training when the best KNN error is found[2], and we use the sum of the reconstruction errors of the 3000 training cases in the validation set at that point as criteria to stop the training on the other two test sets. We set τ_{sne} to be 150. Table 4.2, table 4.3 and table 4.4 give the detailed clustering results of RAN corresponding to SSGD, CGD and SCGD. The rows of these tables represent the clustering errors out of 3000 corresponding to different validation sets indicated respectively by the first entries of the rows. The columns of these tables

[2] We found in preliminary experiments that 4NN nearly always give the lowest clustering errors on the digit data.

represent the clustering errors out of 3000 corresponding to different test sets indicated respectively by the top entries of the columns. The entries in the diagonal of these tables represent the clustering errors out of 3000 on the validation sets. From these tables, we find that SSGD has the best performance on training RAN among the three methods.

In the experiments as shown in table 4.1 and table 4.3, we cannot make CGD work well on redundant mini-batch data with the mini-batch size 200, so we try to use redundant mini-batch data with smaller mini-batch size. We do the permutation of the 3000 training cases first, and then we divide them into 30 mini-batches with the mini-batch size 100. Repeat the above process 20 times, and we will get 600 mini-batches with each batch containing 100 training cases.

SSGD	Set 1	Set 2	Set 3
Set 1	90	114	129
Set 2	96	111	131
Set 3	96	114	122

Table 4.2: The clustering results of RAN trained by SSGD using redundant mini-batch digit data.

CGD	Set 1	Set 2	Set 3
Set 1	120	175	180
Set 2	130	171	175
Set 3	125	180	165

Table 4.3: The clustering results of RAN trained by CGD using redundant mini-batch digit data.

SCGD	Set 1	Set 2	Set 3
Set 1	98	132	155
Set 2	113	114	143
Set 3	123	136	133

Table 4.4: The clustering results of RAN trained by SCGD using redundant mini-batch digit data.

In the experiments on the redundant mini-batch data with the mini-batch size 100, the settings for SSGD, CGD and SCGD are the same as those described above. We set τ_{sne} to be 150. Table 4.5 shows the results produced by the three methods when training RAN on the redundant mini-batch data with mini-batch size 100.

Table 4.5 again shows that RAN trained by SSGD generates better low dimensional codes for KNN clustering than RAN trained by the other two methods (in the table, K equals to 4). We also find that SCGD is the fastest one among the three and CGD is the slowest one among the three.

	4NN Errs	Computational Cost (hour)
SSGD	102	0.7
CGD	104	1.4
SCGD	104	0.3

Table 4.5: The performance of different optimization methods on optimizing RAN using 600 mini-batches of digit data with the mini-batch size 100.

4.2.3 Combining Different Optimization Methods

One kind of combination of optimization methods is: during the training of RAN on one of the three datasets, run CGD first until it's impossible for it to make any progress, and then run SSGD. We used 600 redundant mini-batches with mini-batch size 100 so each case occurs in two mini-batches, and run CGD for 25 epochs. In the last 5 epochs, CGD does not make any progress. Then, we continue to run SSGD on the same redundant mini-batch data set for 10 epochs. The best KNN clustering error is not as good as that produced by SSGD alone. The combination of first running SSGD and then running CGD is not good either.

We also tried the combination of SSGD and SCGD to train RAN, but the performance of the combination is not superior to that of SSGD alone.

4.3 Experimental Results of Different Models on Digit Data

The autoencoder (single batch training)	Set 1	Set 2	Set 3
Set 1	102	117	127
Set 2	116	105	140
Set 3	109	117	124

Table 4.6: The clustering results of the autoencoder using single batch training on digit data.

The autoencoder (mini-batch training)	Set 1	Set 2	Set 3
Set 1	97	125	126
Set 2	107	111	135
Set 3	100	126	123

Table 4.7: The clustering results of the autoencoder using mini-batch training on digit data.

SNE (single batch training)	Set 1	Set 2	Set 3
Set 1	132	148	148
Set 2	140	143	149
Set 3	139	149	144

Table 4.8: The clustering results of SNE on digit data.

In this section, we will compare the performance of the RAN model to the performance of the autoencoder model, the SNE model, the SNE-encoder model, the PCA model and the LLE model on the three digit datasets. We'll do the experiments of validation and test

as described in section 4.2.2.2 for each model. Since SSGD is the best method that has been found, we use SSGD to train the autoencoder, RAN, SNE and SNE-encoder in all the experiments described in this section.

SNE-encoder (single batch training)	Set 1	Set 2	Set 3
Set 1	130	140	135
Set 2	140	136	138
Set 3	133	140	132

Table 4.9: The clustering results of SNE-encoder on digit data.

Table 4.6 shows the clustering results for the autoencoder. During the training on each dataset, the 3000 training cases are used as one batch. On the validation set, we stop the training when the best KNN clustering error is found, and then we use the sum of the reconstruction errors at that point as criteria to stop the training on the test sets.
Mini-batch training is also tried on the autoencoder. The 225 redundant mini-batches of data with mini-batch size 200 used to train RAN are used to train the autoencoder. Table 4.7 shows the detailed results.

Table 4.8 and table 4.9 show the clustering results for SNE and SNE-encoder respectively. The dimensionality of the code space here in the two models is the same as that in RAN. The structure of the SNE-coder model is the same as that of the recognition part of RAN discussed in section 4.1. We use all the 3000 training cases as one batch in each dataset to train these two models and we use the sum of the KL-Divergence over the 3000 training cases in the validation set at the point that the best KNN clustering error is found as criteria to stop the training on the test sets.

Table 4.10 and 4.11 show the KNN clustering results generated by PCA and LLE. On the validation set, we find the best dimensionality of the code space through which the best KNN clustering error can be obtained, and we perform PCA or LLE and KNN on the test sets using the dimensionality of the code space found. We find that the clustering performance of LLE is better when the number of neighbors K in the LLE model is just

slightly larger than the dimensionality of the code space than that when K is much larger than the dimensionality of the code space. That is to say, only when the local structure within the neighborhoods in the high-dimensional space is almost perfectly preserved in the low-dimensional space, KNN works best using the codes generated by LLE. Despite that, LLE has the worst performance on clustering among the models discussed.

PCA	Set 1	Set 2	Set 3	dimensionality of code space
Set 1	120	133	145	32
Set 2	148	116	144	27
Set 3	128	139	130	27

Table 4.10: The clustering results of PCA on digit data.

LLE	Set 1	Set 2	Set 3	dimensionality of code space
Set 1	362	494	388	16
Set 2	399	420	376	15
Set 3	425	448	339	14

Table 4.11: The clustering results of LLE on digit data.

Now we turn to the discussion of the clustering performance of RAN. Table 4.2 shows the clustering results of RAN using mini-batch training by SSGD. In the RAN model here, the cost of SNE is used as a regularization term. Comparing table 4.2 to the tables in this section, we find that RAN has the best performance on clustering digit data. It slightly improves the clustering performance of the autoencoder.

We also tried to train the RAN model as described above using single batch training on one dataset, that is, we use 3000 training cases as one batch to train RAN. The training is very slow because the calculation of the probability distributions in the code space is very expensive. We tried different coefficients in front of the cost of SNE, but the

clustering performance is not so good as that of the autoencoder trained using single batch training.

RAN (single batch training)	Set 1	Set 2	Set 3
Set 1	98	104	116
Set 2	105	93	115
Set 3	104	104	107

Table 4.12: The clustering results of RAN using single batch training on digit data.

However, we can train RAN using a large single batch of digit data in another way: set the coefficient τ_{sne} in front of the SNE cost to be an appropriate value, train RAN as usual for some epochs, and then reduce the coefficient τ_{sne} slowly to 0, and then train the whole network as an autoencoder. Here, SNE is used to help initialize the autoencoder. Table 4.12 gives the clustering results of RAN trained this way. In the experiment, τ_{sne} is initially set to be 3000. When the number of epoch is greater than 50, we set $\tau_{\text{sne}} = \tau_{\text{sne}} \times 0.99$ on each epoch. We find that RAN trained this way often produces better clustering performance than that of the autoencoder using single batch training.

There is another method to train RAN using a large single batch of digit data: run the autoencoder to completion first and then add the SNE regularization term with slowly increasing τ_{sne}. We find that RAN trained by this method works almost as well as RAN trained by the above method. When set 3 is used as the validation set, the best clustering error produced by RAN trained by this method is 107 out of 3000. But this method is more difficult to control than the above method because we must decide when to stop the training of the autoencoder.

Comparing the results in table 4.12 and table 4.2 to the clustering results generated by some other models, we find that RAN has the best clustering performance on digit data. From these tables, we also find that PCA has the second best clustering performance, SNE-encoder produces better codes for KNN clustering than SNE alone, and the codes produced by LLE are not suitable for KNN clustering.

Chapter 5

Experiments on Face Recognition

5.1 Some Previous Face Recognition methods

Face recognition has been explored by a lot of researchers and many face recognition methods have been presented. The Correlation method [Brunelli and Poggio, 1993] gives the similarity score as the angle between two images represented as vectors of pixel intensities. The Eigenface method [Turk and Pentland, 1991] projects face images onto a subspace spanned by principal components capturing most of the variance in the face data, and then computes similarity scores of projected images. The Fisherface method [Belmumeur et al., 1996] projects face images to a subspace which maximizes the inter-class variances and at the same time minimizes the intra-class variances. Each individual can be viewed as one class. Then it returns the similarity score of projected images. The δ ppca [Moghaddam et al., 1998] method models image differences including intra-individual differences and extra-individual differences using an eigenface density estimation technique. It gives similarity score of two images based on the a posteriori probability of membership in the intra-individual class. The RBM [Teh and Hinton, 2001] method returns similarity score of two images based on the negative free energy when the RBM is fed with the two images.

And there are many other types of methods such as template-based methods, deformable models, feature-based methods etc [Yuille, 1991, Pentland et al., 1994, Samaria, 1994, Wiskott et al., 1997, Lawrence et al., 1997]. There are lots of methods

proposed for finding features of complicated images, and one of them that we should mention is the method introduced by Lowe that uses the SIFT keys to find local scale invariant features [Lowe, 1999]. In section 5.7.2, we will compare the performance of RAN on face recognition to those of some methods mentioned here.

5.2 The Olivetti Faces

The Olivetti face dataset was constructed by AT&T Laboratory Cambridge. The Matlab file for the face dataset used in this thesis can be found in [Roweis webpage]. In the image set, there are 10 face images for each of 40 individuals. All the images were taken against a dark homogeneous background with the individuals in upright frontal positions. For each individual, the images in the face set are different from each other in lighting conditions, orientation (a small degree of rotations or different poses), expression (with eyes or mouth open/closed), with glasses/without glasses, or a combination of these variations. Figure 5.1 shows some face images in the face image set. By using the Olivetti faces in addition to the FERET faces, we can have some confidence that our results are not due to the peculiarities of one dataset.

For the face images contained in the Matlab file, the hairstyle and part of the contour for each face is cut off. The size of the processed face images is 64 by 64 pixels. Besides, we used Nearest Neighbour Interpolation, which is a sampling method that determines the pixel value of a point in the sampled image from the closest pixel to the centroid of a block in the original image, to sample the processed images. The size of the sampled images is 41 by 41 pixels, and all the consequent experiments are based on the sampled faces. That is to say, the dimensionality of the pixel space for the experiments conducted on the Olivetti faces in the thesis is 1681. The best hold-one-out clustering error by 1NN in the pixel space is 22 out of 400.

Figure 5.1: 100 face images for 10 individuals in the Olivetti face dataset.

5.3 Performance of Different Optimization Methods on the Olivetti Faces

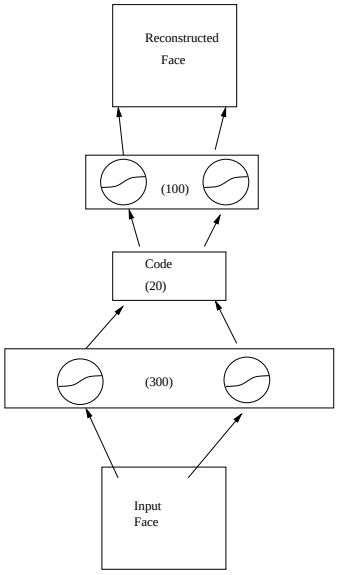


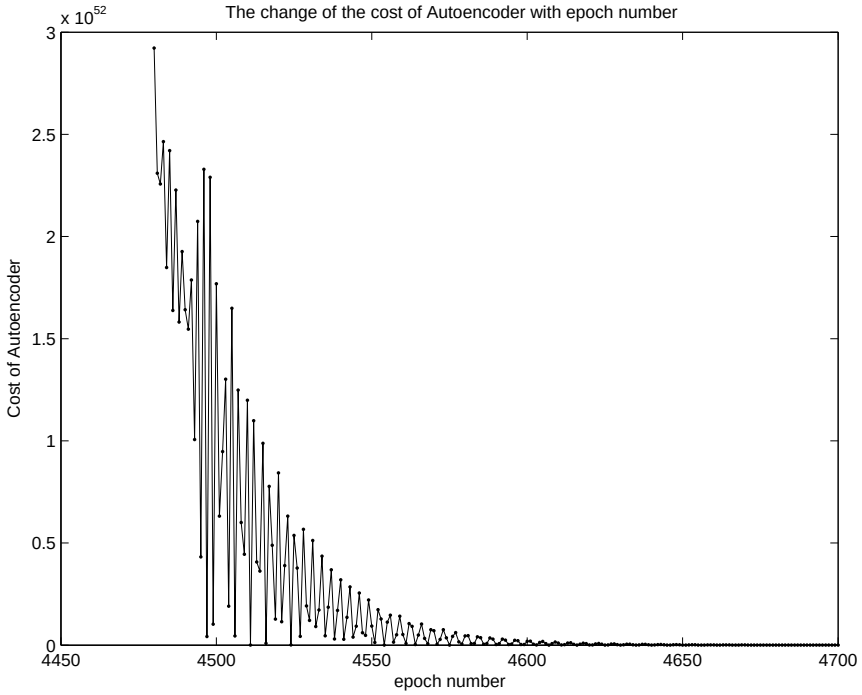
Figure 5.2: The structure of the autoencoder for clustering the Olivetti faces.

We have compared the performance of Scaled Steepest Gradient Descent (SSGD), Conjugate Gradient Descent (CGD) and Scaled Conjugate Gradient Descent (SCGD) for clustering digits in chapter 4. In this section, we will use the above methods to optimize an autoencoder on the Olivetti face data to check which optimization method is the most robust one in face recognition. We compare different methods by comparing the

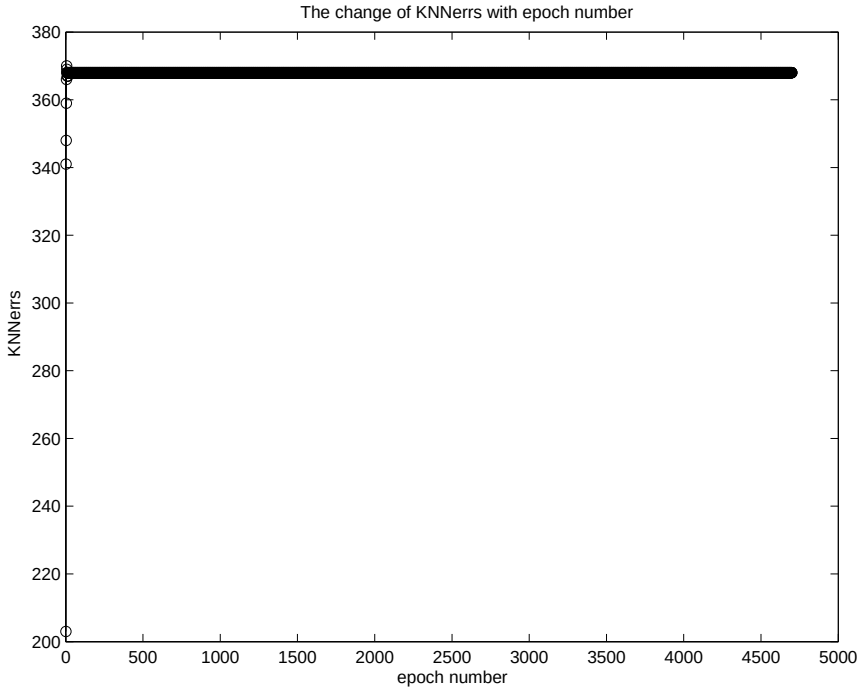
hold-one-out clustering errors using the codes calculated at different local minima that are found by different optimization methods, and we also compare the computational cost of the different methods. Through lots of experiments, we found that the best configuration on the autoencoder for clustering the Olivetti faces is as follows numrhid = 300, numghid = 100, and numydims = 20. Here numrhid refers to the number of hidden units in the second layer, numghid refers to the number of hidden units in the fourth layer, and numydims refers to the dimensionality of the code space (i.e. the number of hidden units in the third layer). Figure 5.2 shows the structure of the autoencoder mentioned above. In all the consequent experiments on face data, the cost of the autoencoder is the sum of reconstruction errors. And here, the 400 face images are used as one batch to train the autoencoder.

5.3.1 SSGD on the Olivetti Faces

When Scaled Steepest Gradient Descent (SSGD) is used to train the autoencoder, the training is very sensitive to the learning rates. When the learning rates are too large, the update steps obtained from the partial second-order information of the derivatives will not help a lot and the update will oscillate in some regions of the weight space; we will have many cost spikes during the training; it is very easy to get stuck at some poor local minima. Under the above situation, Figure 5.3 (a) shows how the cost of the autoencoder changes with the epoch number and Figure 5.3 (b) shows what the KNNerr (the hold-one-out clustering error in the code space) will look like. From the two figures, we can find that the cost fluctuates many times during the training and the clustering error remains very large. The cost is 1.19e+41 at epoch 4700 while it is 1.17e+04 at epoch 3000. Between epoch 3000 and epoch 4700, there is a huge cost spike. In all, large learning rates make it very hard for the autoencoder to achieve good local minima once the training is trapped into some region.

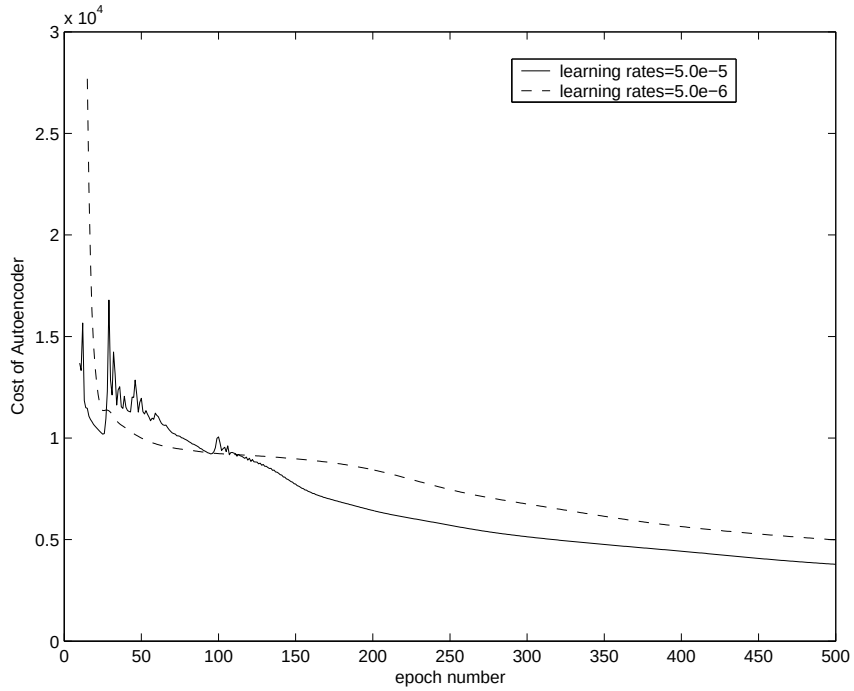


(a)

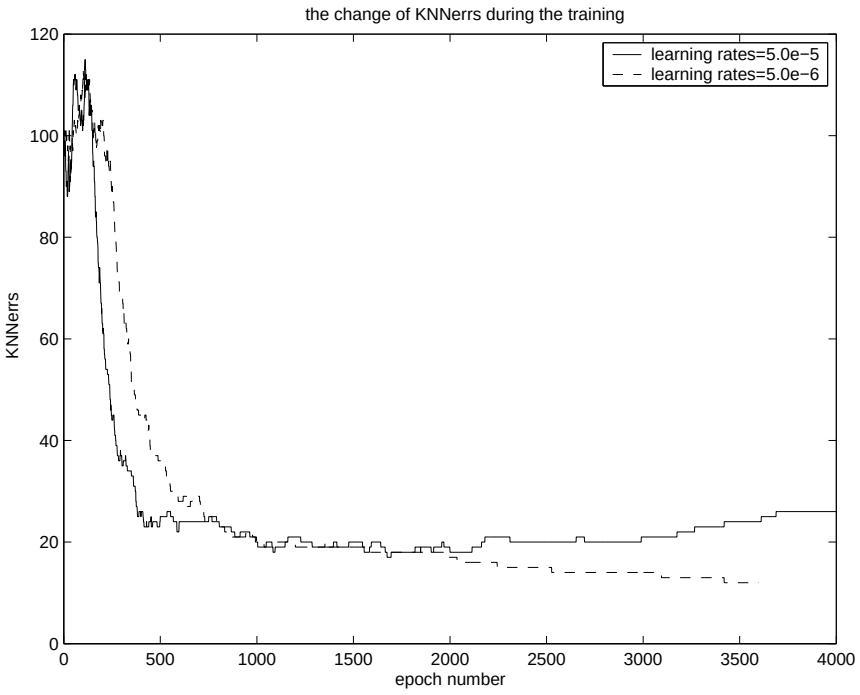


(b)

Figure 5.3: The performance of SSGD with large learning rates when optimizing Autoencoder on Olivetti face dataset: a) the change of the cost of Autoencoder with epoch number; b) the change of KNNerrs by 1NN with epoch number.



(a)



(b)

Figure 5.4: The performance of SSGD with two different learning rates when optimizing Autoencoder on Olivetti faces: (a) the change of the cost of Autoencoder during the training; (b) the change of KNNerrs (1NN clustering errors in the code space) during the training.

But when the learning rates are too tiny, the training will be unbearably slow. Only if the learning rates are appropriately set, the update steps calculated based on the partial second-order information of the derivatives will lead the weights to achieve good local minima and the cost of the autoencoder will decrease smoothly.

In Figure 5.4, the comparison of the training with two different learning rates is shown. Figure 5.4 (a) shows how the cost of the autoencoder changes with the epoch number when SSGD with two different learning rates is used. Figure 5.4 (b) presents the change of KNNerrs with epoch number under SSGD with different learning rates.

In these experiments, momentum is also used to make the updates of the weights more stable. We used small momentum (0.5) in the early phase of the training and big momentum (0.95) in the final phase of the training, which is the same as what we did in the experiments on digit data.

When the learning rates for both the recognition part and the generative part of the autoencoder are set to be 5.0e-6 as shown in Figure 5.4 by dashed lines, the best 1NN clustering error is 12 out of 400 at epoch 3500. That is to say, the error rate is 3.0%. At this point, the cost of the autoencoder is 2.27e+03. If the learning rates are set to be bigger as shown in Figure 5.4 by solid lines, the cost of the autoencoder will fluctuate and will not change smoothly although the cost will mainly decrease. Under SSGD, it takes the autoencoder about 1.6 hours to achieve the best clustering result on our machine. In a word, when the learning rates and the momentum are set appropriately, the cost of the autoencoder will change very smoothly as the training proceeds, and we will get good clustering results on the face data.

5.3.2 CGD on the Olivetti Faces

In the experiments testing the performance of CGD on the Olivetti face data, we used the code developed by C.E. Rasmussen. The code implements the CGD with cubic and quadratic interpolation presented in chapter 2. We modified some constants for line searches in the code to make it suitable to minimize the specific objective function under

discussion. Because the dimensionality of the weight vector in the autoencoder is huge, the total time cost for CGD's minimizing the objective function on each dimension once will be unbearably long. Thus, We just run CGD on the whole training face data, and if the total number of successful line searches is achieved or the value of the objective function is below some threshold, we will stop the training. The best result is presented by dashed line in Figure 5.3 by which we will compare the performance of SSGD, CGD and SCGD later. Under CGD, the best clustering error is 17 out of 400, and it takes about 2.9 hours to achieve the best clustering result on our cluster machine.

We also tried mini-batch training using CGD on the Olivetti faces, but the result is not good. Since the size of the training data is not large and the objective function on the whole training set is not very hard to minimize, it is easy to understand why the stochastic update given by mini-batches will not be very helpful in searching for better local minima.

5.3.3 SCGD on the Olivetti Faces

During the training of the autoencoder by SCGD, we restart the method at the updated weight vector several times. We restart the method at iteration 200, iteration 400, iteration 800 and iteration 1600. During the first 200 iterations, we set the parameter λ to be 5.0e+4, and we set it to be 5.0e+5 in the consequent iterations. Here, λ is used for adjusting the Hessian of the objective function and the step size along the P-R Conjugate Gradient search direction, which we have discussed in chapter 2. We raise or lower λ at every updated point in the weight space by seeing if we have a good quadratic approximation to the objective function at that point, but the objective function of the autoencoder is far from quadratic in many regions of the weight space and λ is reduced very fast. Avoiding that λ is set to be too small, we restart the method several times. Because SCGD always reduces the objective function as the training proceeds, we set bigger λ to have bigger step size in order to accelerate the training.

Under the setting above, SCGD decreases the cost of the autoencoder very fast. After

2100 iterations, we will get the best clustering result as shown by dotted line in Figure 5.3. The best KNNerrs is 14 out of 400, and it takes about 1.0 hour to achieve this result on our machine. When using SCGD to train the autoencoder, we needn't do line searches, which is why it is much faster than CGD.

5.3.4 Comparisons of the optimization methods

We give a short summary of the hold-one-out clustering results produced by SSGD, CGD and SCGD on the Olivetti faces in Table 5.1. From the table, we can find that SSGD produces the best clustering result while SCGD is the fastest optimization method. CGD is the slowest optimization method and it has the worst performance among the three.

We also give the change of the cost of the autoencoder and the change of the KNNerrs during the training by the three methods in Figure 5.5.

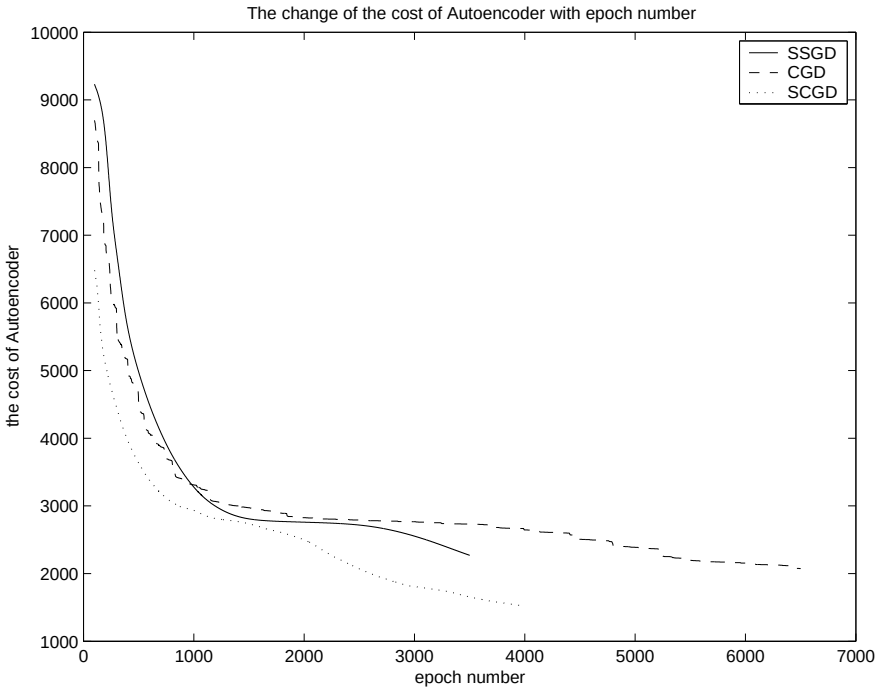
Optimization method	KNNerrs (out of 400)	Computational Cost (hour)
SSGD	12	1.6
CGD	17	2.9
SCGD	14	1.0

Table 5.1: The summary of the final results produced by three optimization methods on the Olivetti faces It's "cheating" to stop at the best KNNerr by looking at the answer, but it's the same "cheat" for all the methods here.

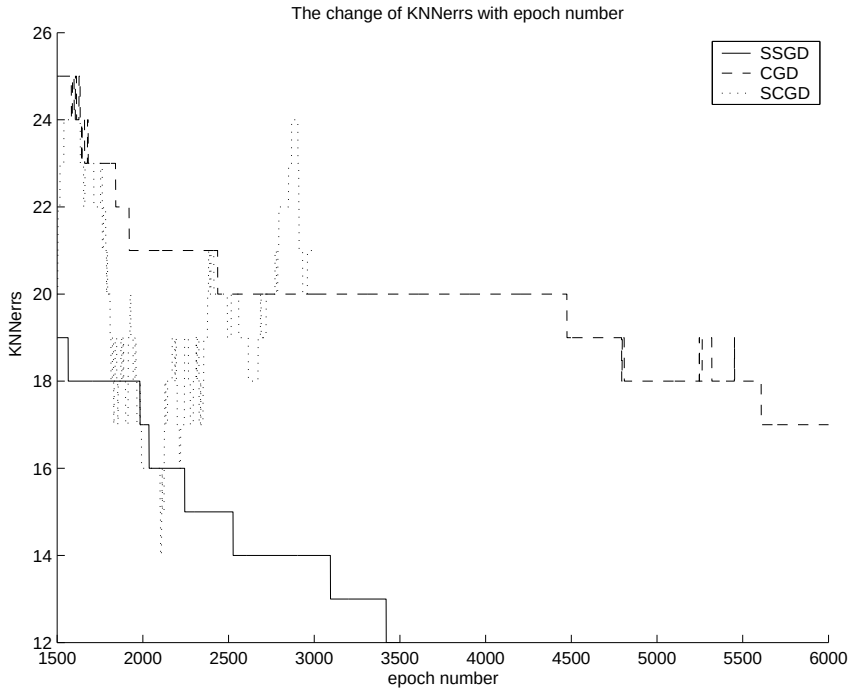
In Figure 5.5 (a)[3], it clearly shows again that SCGD is the fastest method. We can find that CGD decreases the cost of the autoencoder faster than SSGD in the early phase

[3] In figure 5.5 (a), the costs of Autoencoder in the first 20 epochs (iterations) are not shown because they are much larger than those in later epochs (iterations) and showing them at the same time makes it difficult to show the costs in later epochs (iterations) in an appropriate scale.

of the training. It's normal because CGD has several line searches each iteration that is



(a)



(b)

Figure 5.5: The performance of SSGD, CGD and SCGD on the Olivetti faces during the training: (a) the cost of the autoencoder versus epoch number; (b) the KNNerrs versus epoch number (or iteration number

for CGD and SCGD).

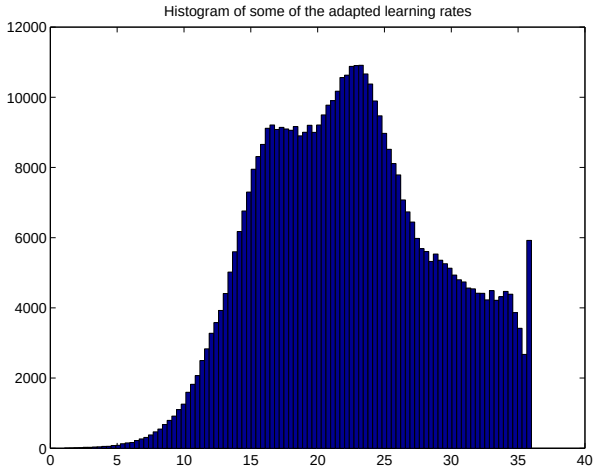


Figure 5.6: Histogram of adapted learning rates for the weights from the input image to the hidden layer in the recognition part of the autoencoder.

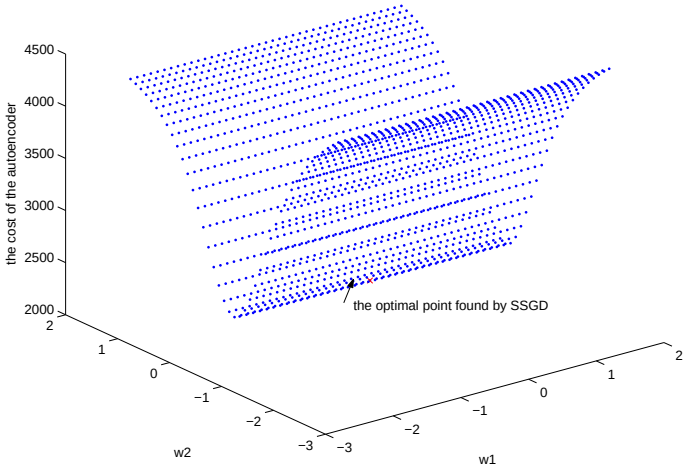


Figure 5.7: The contour of the objective function of the autoencoder on the Olivetti faces in the region where the optimal point found by SSGD lies. Here, w1 is the direction along which the objective function of the autoencoder has the smallest gradient and w2 is the direction along which the objective function of the autoencoder has the largest gradient at the optimal point found by SSGD.

equivalent to several epochs in SSGD. However, SSGD becomes faster than CGD in the late phase of the training although CGD has line searches. SSGD becomes faster and faster because the learning rates are adapted to have different updating amount on different components of the weight vector. Figure 5.6 shows the adapted learning rates for the weights from the input image to the hidden layer in the recognition part of the autoencoder.

Again from Figure 5.5, we can find that there are no large cost spikes during the training and all the three methods decrease the cost of the autoencoder very smoothly to achieve the best clustering results.

Although SCGD is the fastest method among the three, it doesn't give the best clustering result. Besides that, we simply use the quadratic approximation to measure how and when to raise or lower the parameter λ in method. There is no mechanism to make the step size always in the trust region at each update of the weight vector. It will make the performance of the method very unstable. The local minima of the objective function of the autoencoder must lie in a region in which the curvature of the objective function in one direction varies a lot as we move in another direction as shown in Figure 5.7.

Table 5.2 shows the cost of the autoencoder, the maximum curvature, the minimum curvature and the ratio of the two curvatures at the points found by SSGD, CGD and SCGD.

Figure 5.7 shows the contour of the objective function of the autoencoder trained on the Olivetti faces in the region where the optimal point found by SSGD lies. Here, we just chose two representative directions, of which w1 is the direction along which the objective function of the autoencoder has the smallest gradient and w2 is the direction along which the objective function of the autoencoder has the largest gradient at the optimal point found by SSGD. In Figure 5.7, the curvature of the objective function along the direction w2 changes significantly as we move along the direction w1. It is very difficult for SCGD to find the local minima in such regions because we will obviously have poor quadratic approximations to the objective function in these regions and it is

very likely that λ is lowered a lot and the method will jump off the local minima because it does not do a line search.

	Height	Max Curvature	Min Curvature	MaxCur/MinCur
SSGD	2.27e+03	2.88e+03	1.32e-01	2.17e+04
CGD	2.13e+03	1.22e+03	1.78e-01	6.86e+03
SCGD	2.41e+03	1.02e+04	9.51e-06	1.07e+09

Table 5.2: The information of the objective function of the autoencoder at the optimal points found by SSGD, CGD and SCGD.

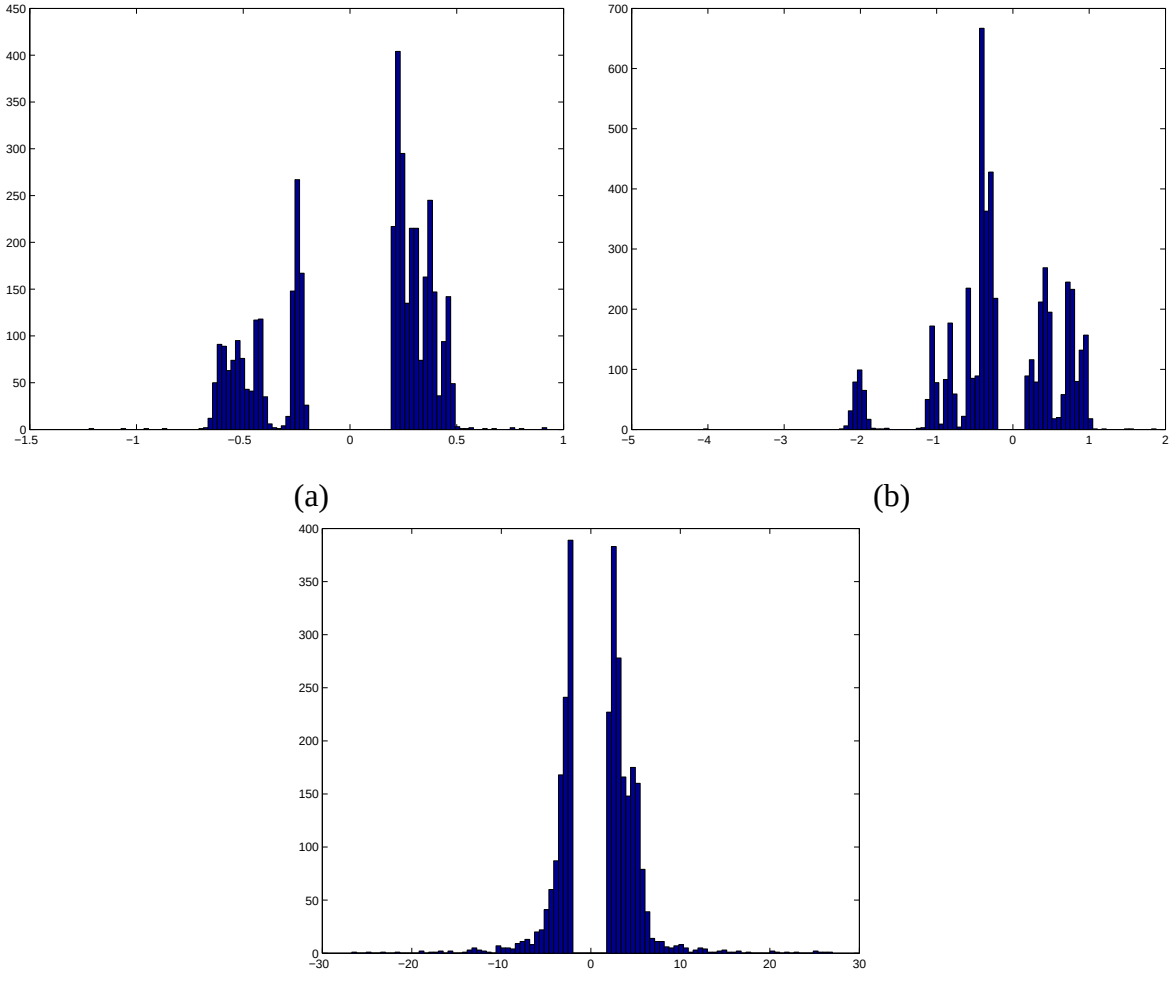


Figure 5.8: Histograms of the components of the gradients at the points where the best clustering results

found by: (a) SSGD, (b) CGD and (c) SCGD are achieved (To facilitate plotting: in (a), the bins between –0.2 and 0.2 are removed; in (b), the bins between –0.2 and 0.2 are removed; in (c), the bins between –2 and 2 are removed).

We can also compare the performance of the three methods by investigating the gradient of the objective function of the autoencoder with respect to the weight vector at the points where the best clustering results found by different methods are achieved. Figure 5.6 shows the gradient information corresponding to the three methods.

Figure 5.8 (a) shows that the components of the gradient at the point found by SSGD are distributed in a small neighbourhood centred at 0, and the maximum absolute value of the components is below 0.2; Figure 5.8 (b) shows that the components of the gradient at the point found by CGD are distributed in a larger region centred at 0 and the maximum absolute value of the components is near 4; Figure 5.8 (c) shows that lots of components of the gradient at the point found by SCGD are very far from 0 and the maximum absolute value of the components is near 30. We also used SCGD to train the autoencoder until the cost is below 1000 and it cannot be reduced a lot, we investigate the gradient information and find that the components are also distributed in a large region and the maximum absolute value of the components is near 8. These results show that it is difficult for SCGD to reach the small neighbourhood centred at a good local minimum of the objective function of the autoencoder although it can reduce the objective function very fast. By contrast, adapted learning rates make it easier for SSGD to reach a good local minimum.

We also tried the combination of SSGD and CGD, as well as the combination of SSGD and SCGD, but the results are not good.

From the above discussions, we conclude that SSGD is the most effective method in minimizing the objective function of the autoencoder among the three methods.

5.4 The Autoencoder and SNE-encoder on the Olivetti faces

Through the experiments on optimization, we find that SSGD method is the most robust method among the three methods mentioned. In the following experiments, SSGD method is used to optimize Autoencoder, SNE-ENCODER and Regularized Autoencoder Network (RAN).

5.4.1 The Autoencoder on the Olivetti faces

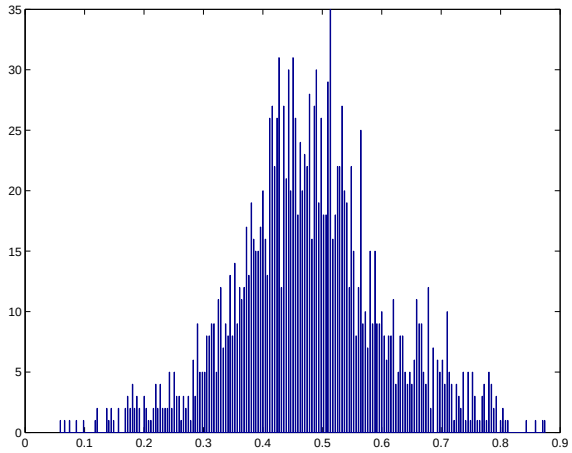


Figure 5.9: Histogram of pixel values of a face image before normalizing.

As is shown in Figure 5.9, the pixel values of face images are much more Gaussian than those of the digit images, so we use linear outputs in the last layer of an autoencoder. Before training the autoencoder, we normalize the images so that the pixel value for each position on the face image is centred at 0 over all the training images.

In section 5.3.1, we have mentioned that the best clustering error rate using the codes generated by the autoencoder can achieve 3.0% for Olivetti faces and it takes about 1.6 hours to reach the best result using SSGD. The configuration of the autoencoder for achieving the best clustering result is: there are 300 hidden units in the second layer, 20 hidden units in the third layer (i.e., the dimensionality of the code space), and 100 hidden units in the fourth layer. We also mentioned that the learning rates must be carefully chosen to make the cost of the autoencoder mainly decrease very smoothly and then the

good clustering results can be obtained. It suggests that the Hessian of the objective function of the autoencoder are very badly conditioned. Thus, it makes the training very difficult and sensitive to the learning rates.

Figure 5.10: The original faces and their respective reconstructions by the autoencoder.

At the best local minimum achieved, the sum of square reconstruction errors over the 400 face images is 2.27e+03. Therefore, the average sum of square reconstruction errors for each image is 5.68 and the average squared error for each pixel value is 0.0034.

Figure 5.10 shows some face images and their corresponding reconstructions by the autoencoder. Each original image is shown in the top row and its reconstructed image is shown at the corresponding location in the bottom row below it. The left four columns show the images with the smallest sums of square reconstruction errors, and the right four columns show the images with the largest sums of square reconstruction errors. If we look at the face images and their respective reconstructed images in the right four columns by eyes, we can see that the images and their reconstructions are very similar. An interesting thing here is that the reconstructed images do not have obvious glasses even if its original face image is with glasses. This happens because most of the face images are not with glasses. The autoencoder captures enough information from the training data and it will generate the reconstructions as shown in Figure 5.6. When there are enough face images for each individual, the autoencoder can capture enough information from each face image. Consequently, the autoencoder can generate very good low dimensional codes for most of the input images.

5.4.2 SNE-encoder on the Olivetti faces

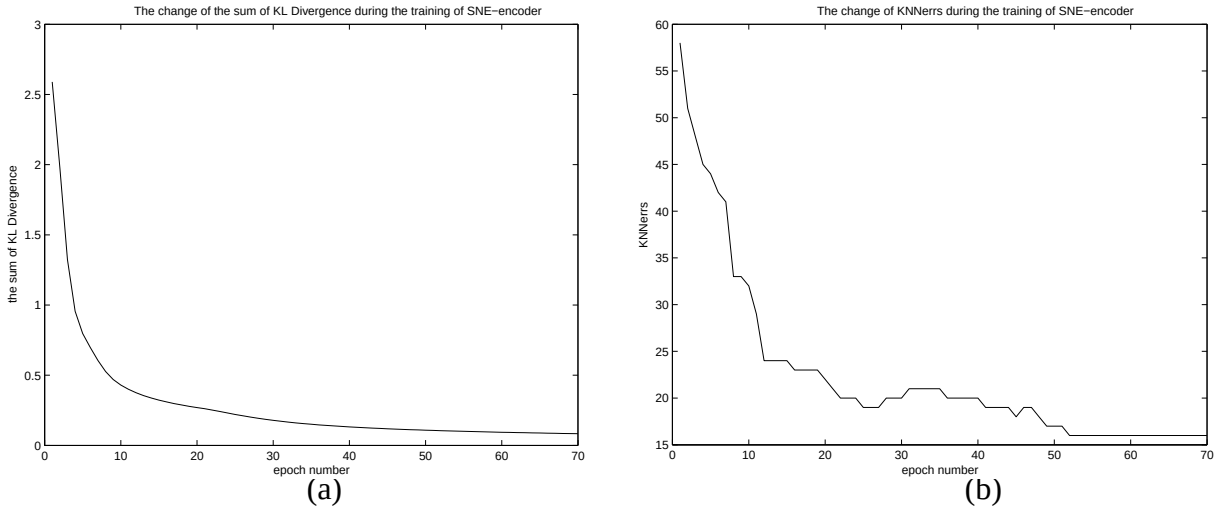


Figure 5.11: The training of SNE-encoder on the Olivetti faces: (a) the sum of KL Divergence versus epoch number; (b) KNNerrs versus epoch number.

We used a three-layer SNE-encoder here as discussed in chapter 2, in which numrhid = 300 (the number of hidden units) and numydims = 20 (the dimensionality of the code space). On the Olivetti face dataset, the training of SNE-encoder is not sensitive to the learning rates. We set the learning rates to be 1.5 for all the weights, and it takes less than 3 minutes to achieve the best clustering error on our machine on which we have tested the performance of the autoencoder and of three optimization methods. If we set the learning rates to be small, we can get the same good results as we get using large learning rates, but it will take more time.

Figure 5.11 illustrates the training of the SNE-encoder on the Olivetti face dataset. It shows that the cost of the SNE-encoder reduces very fast with the epoch number and there is no cost spike although the learning rates are large. When the sum of KL Divergence is below 0.10, we will get the best hold-one-out clustering error (1NN), 16 out of 400. That is, the clustering error rate is 4.0%.

5.4.3 Comparisons of the autoencoder and SNE-encoder

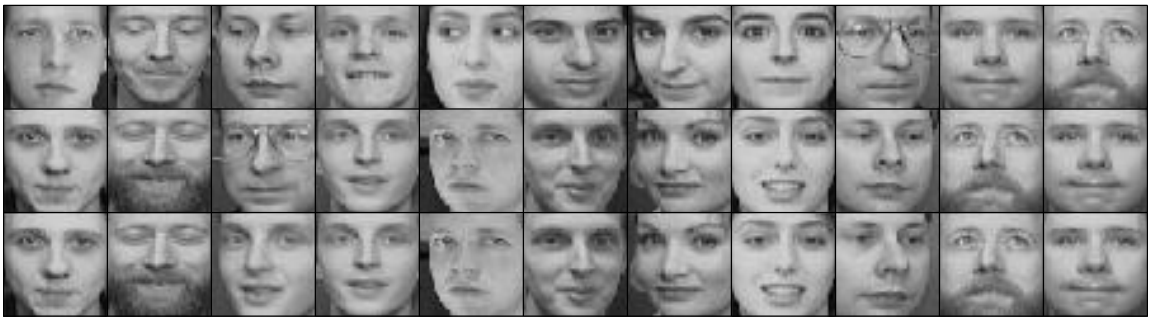


Figure 5.12: Some face images in the Olivetti faces on which both the autoencoder and the SNE-encoder got wrong in the 1NN clustering. The first row represents the given images, the second row and the third row respectively represent the nearest neighbours for the given images found by 1NN using the low dimensional codes generated by the autoencoder and by the SNE-encoder.

Figure 5.13: Some face images in the Olivetti faces on which the autoencoder got right but the SNE-encoder got wrong in the 1NN clustering. The first row represents the given images, the second row and the third row respectively represent the nearest neighbours for the given images found by 1NN using the low dimensional codes generated by the autoencoder and by the SNE-encoder.

Figure 5.14: One face image in the Olivetti faces on which the SNE-encoder got right but the autoencoder

got wrong in the 1NN clustering. The first row represents the given images, the second row and the third row respectively represent the nearest neighbours for the given images found by 1NN using the low dimensional codes generated by the autoencoder and by the SNE-encoder.

From the discussions in section 5.4.1 and section 5.4.2, we will see that the autoencoder can generate better low dimensional codes for KNN clustering than SNE-encoder while the training of SNE-encoder is much faster than that of the autoencoder. We know that the training of the autoencoder is slow and sensitive to learning rates because the Hessian of the objective function of the autoencoder is ill-conditioned. Besides that, the size of the Olivetti face set is not large and the calculations of probability distributions in the code space is not expensive, so it is understandable that the training of SNE-encoder is very fast. The results support our argument in chapter 2 that the codes generated by SNE-encoder are in a more constrained region than the codes generated by the autoencoder.

When we check the clustering results on the Olivetti faces using the codes generated by the autoencoder and SNE-encoder further, we find that: there are 11 face images for which both the autoencoder and SNE-encoder generate bad codes; there are 5 face images for which the autoencoder generates good codes but SNE-encoder generates bad codes; and there is 1 face image for which SNE-encoder generates a good code but the autoencoder generates a bad code. Here, good codes mean the codes that favour accurate KNN clustering, and bad codes mean the codes that are not suitable for KNN clustering. Among the input data set, the subset for which the autoencoder fails to generate good codes overlaps a lot with that for which SNE-encoder fails to generate good codes. We have tried to give some intuitive explanations to this in chapter 2.

Figure 5.12, figure 5.13 and figure 5.14 illustrate some clustering results generated by the autoencoder and SNE-encoder. In each of the three figures, the first row represents the given images, the second row represents the nearest neighbours for the given images above them found by 1NN using the low dimensional codes generated by the autoencoder, and the third row represents the nearest neighbours for the given images in the corresponding columns found by 1NN using the low dimensional codes generated by the

SNE-encoder. Figure 5.12 shows some face images in the Olivetti faces on which both the autoencoder and the SNE-encoder got wrong in the clustering. Looking at the images in the second row and in the third row, we can find that they have very similar expressions and poses to the corresponding given images in the first row. Figure 5.13 shows some face images in the Olivetti faces on which the autoencoder got right but the SNE-encoder got wrong in the clustering. Figure 5.14 shows one face image in the Olivetti faces on which the SNE-encoder got right but the autoencoder got wrong in the clustering.

In the previous experiments, the dimensionality of the code space is big enough for the autoencoder to capture the main variance of the data and for SNE-encoder to give very good approximations in the code space to the probability distributions that each face image chooses other face images as its neighbours in the pixel space. But what if the dimensionality of the code space doesn't satisfy the above constraint? Table 5.2 gives some results produced by the autoencoder, SNE-encoder when the dimensionality of the code space is small.

When the dimensionality of the code space is small, we also tried to set numrhid (the number of hidden units in the second layer) = 600, numghid (the number of hidden units in the fourth layer) =100 and numrhid =300, numghid = 100 to train the autoencoder, but the results we obtained are not as good as those shown in Table 5.2. There are only 400 images in Olivetti faces, thus, if we set the number of hidden units in the second layer of the autoencoder to be large, it is very easy for the autoencoder to be overfitted. Suppose that we will map the 400 faces to a two dimensional space and the number of hidden units in the second layer and in the fourth layer is large enough for the autoencoder to give perfect reconstructions, it is very likely that the low dimensional codes are distributed in a small densely populated region. Obviously, the performance will be very bad if we use these codes for KNN clustering.

Table 5.3 clearly shows that SNE-encoder generates much better codes for KNN clustering than the autoencoder does when the dimensionality of the code space is small. From the table, we can see that we can get very good low dimensional codes by SNE-encoder even if we map the images to a three dimensional or five dimensional space

noticing that the clustering error in the pixel space is 22 out of 400.This is because the SNE-encoder is optimizing a function which preserves the information used by KNN in the original space.

Autoencoder (KNNerrs)	SNE-encoder (KNNerrs)	numydims	numrhid	Numghid (only for the autoencoder)
104	79	2	300	300
75	23	3	300	300
45	18	5	300	300

Table 5.3: The clustering results when the dimensionality of the code space is small.

5.5 Regularized Autoencoder Network on the Olivetti Faces

In section 5.4, we discussed the experimental results generated by the autoencoder and SNE-encoder. We find that the autoencoder and SNE-encoder have their respective characteristics and they can be complementary to each other.

Since SNE-encoder guides the low dimensional codes to the desired region faster than the autoencoder while the autoencoder can generate better codes, we will make SNE-encoder lead the training of RAN in the early phase and make the autoencoder lead the training in the late phase.

There are two methods to realize this idea. The first method is: Set τ_{sne} in Equation (2.20) to be big at first and reduce it fast as the training proceeds so that τ_{sne} is near 0 after several hundred epochs' training; the second method is: train the recognition network of RAN using the cost of SNE first; fix the weights of the recognition network, and feed the best low dimensional codes produced by the recognition network so far to the generative network, and train the generative network; finally, unfreeze the weights of

the recognition network, and train the recognition network and the generative network together.

In the experiment using the first method, we set τ_{sne} to be 1.0e+4 at first. After epoch 300, we set $\tau_{sne} = \tau_{sne} \times 0.99$. After 2000 epochs' training, we will get the best clustering result. The best hold-one-out clustering error is 13 out of 400, and it takes about 1.3 hours on our cluster machine.

In the experiment using the second method, we set the learning rates to be 5.0e-7 and τ_{sne} to be 8.0e+5 (during the training of SNE-encoder, this coefficient can be viewed as a time to the learning rates). We train the recognition network using the cost of SNE-encoder 50 epochs and then fix the weights we have obtained so far. Then we train the generative network 1250 epochs. During this phase, we set the learning rates to be 5.0e-6 to accelerate the training. After that, we unfreeze the weights of the recognition network and train the recognition network and the generative network together as Autoencoder ($\tau_{sne} = 0$ and this training starts from epoch 1300). At epoch 3500, we get the best clustering result with the error 12 out of 400.

Table 5.4 presents the clustering results using the codes generated by Regularized Autoencoder It also gives comparisons to the performance of the autoencoder and SNE-encoder. It shows that the training of Regularized Autoencoder is slightly faster than that of the autoencoder, and it generates better codes than SNE-encoder.

numydims = 20	KNNerrs (out of 400)	Computational Cost (hour)
Autoencoder	12	1.61
SNE-encoder	16	0.047
RAN (method 1)	13	1.44
RAN (method 2)	12	1.30

Table 5.4: The comparisons of the clustering performance using the codes generated by Autoencoder, SNE and RAN. We cheat by using the lowest error obtained during training, but it is the same cheat for all methods.

Method 2 makes the training faster and it seems to be much better than method 1. However, it is difficult to find the best point at which we should stop training the recognition network and start training the generative network when we use this method 2. Besides that, when we should stop training the generative network is also a problem. In the early phase of the training, if we train the recognition network using the cost of SNE to achieve very good clustering results and then we repeat the remaining steps in method 2, we will get the same clustering results as those produced by SNE-encoder alone. This happens perhaps because we are trapped into the region in which the minima of the objective function of SNE lie. Method 1 doesn't give the results so well as method 2, but it makes the training much more stable. Using this method, the training of RAN is not as sensitive to the learning rates as that of the autoencoder.

When the cost of SNE is adjusted by setting τ_{sne} not very large and high perplexity is applied when calculating the probabilities in the pixel space, the clustering performance is not good.

Now we turn to the experiments on small dimensionality of the code space. When numydims = 2, the clustering error using the codes generated by RAN (method 1) is 73 out of 400; when numydims = 5, the clustering error is 17 out of 400. Referring to Table 5.2, we can find that the results are better than those corresponding to the autoencoder and SNE-encoder alone.

RAN cannot give better low dimensional codes for KNN clustering on Olivetti faces than the autoencoder when the dimensionality of the code space is 20 because there are enough images for each individual in the dataset and the autoencoder alone can capture enough information about the variance of the data and so that the codes generated by the autoencoder alone are good enough for KNN clustering. But RAN helps to make the training more stable and make it easier to find the minima.

5.6 The FERET Faces

The FERET database was constructed for defining a standard procedure for measuring the performance of face recognition algorithms [Phillips et al., 1999]. The FERET face

dataset used in this thesis is the same dataset used for testing the performance of rate-coded Restricted Boltzmann machines [Teh and Hinton, 2001]. The FERET face set contains 1002 frontal face images taken over a period of several years under different lighting conditions. 818 of them are used as the training set and the other 184 of them are divided into four disjoint test sets.

The Δexpression test set contains 110 face images of different individuals. They all have another image in the training set that was taken under the same conditions at the same time but with a different expression.

The Δdays test set contains 40 images from 20 individuals with each having two face images. Each individual has two images taken from the same session in the training set and has two images taken from another session 4 days earlier or later in the Δdays test set.

The Δmonths test set is similar to the Δdays test set except that the time between the sessions was at least three months and the lighting conditions vary. There are 20 images of 10 individuals in this set.

The Δglasses test set contains 14 images of 7 different individuals. Each individual has a pair of face image in the training set and another pair of face images in the Δglasses test set. The difference between the two pairs of face images for each individual is that one pair has glasses but the other does not.

Since the original face images have irrelevant information such as parts of shoulder and neck etc, the images were normalized by Y. W. Teh. The detailed normalization procedures can be found in the paper [Teh and Hinton, 2001]. Figure 5.15 shows some processed face images. After normalization, the dimensionality of the pixel space is 1768.

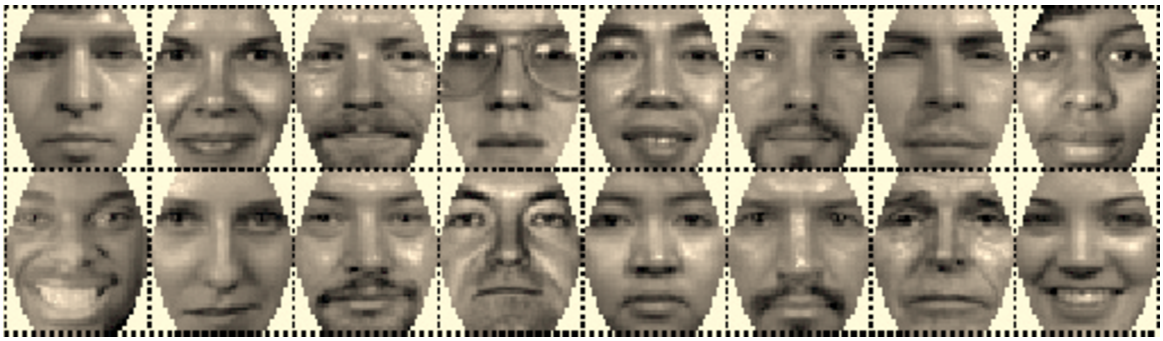


Figure 5.15: Some processed FERET face images [Teh and Hinton, 2001].

5.7 Experiments on the FERET Faces

5.7.1 Unsupervised Clustering on the FERET Faces

We train Autoencoder, SNE and Regularized Autoencoder Network (RAN) on the 1002 face images, and then we use the good low dimensional codes to calculate the hold-one-out clustering errors on the dataset.

The configuration of the networks achieving the best clustering result is as follows: numrhid = 600, numydims = 180 and numrhid = 200.

There are not enough images for each class to allow the autoencoder to capture enough information about the mean of the faces for each individual. The codes produced by the autoencoder are therefore not good for KNN clustering on this dataset. During the training of RAN, if we adjust τ_{sne} appropriately, it can produce good results. The training of RAN is as follows: in the first 100 epochs, we set τ_{sne} to be 0; after epoch 100, we set τ_{sne} to be 1500 and make the minimization of the cost of SNE dominate the training. The best clustering error using the codes generated by RAN is 95 out of 1002. Table 5.5 shows the clustering results using the codes generated by the autoencoder, SNE-encoder and RAN on the FERET faces.

	In Pixel Space	Autoencoder	SNE-encoder	RAN
Clustering Err	174	151	110	95
Error Rate (%)	17.37	15.07	10.98	9.48

Table 5.5: Clustering results on FERET faces using: pixel values, the codes generated by Autoencoder, the codes generated by SNE and the codes generated by RAN.

5.7.2 Face Recognition on FERET faces

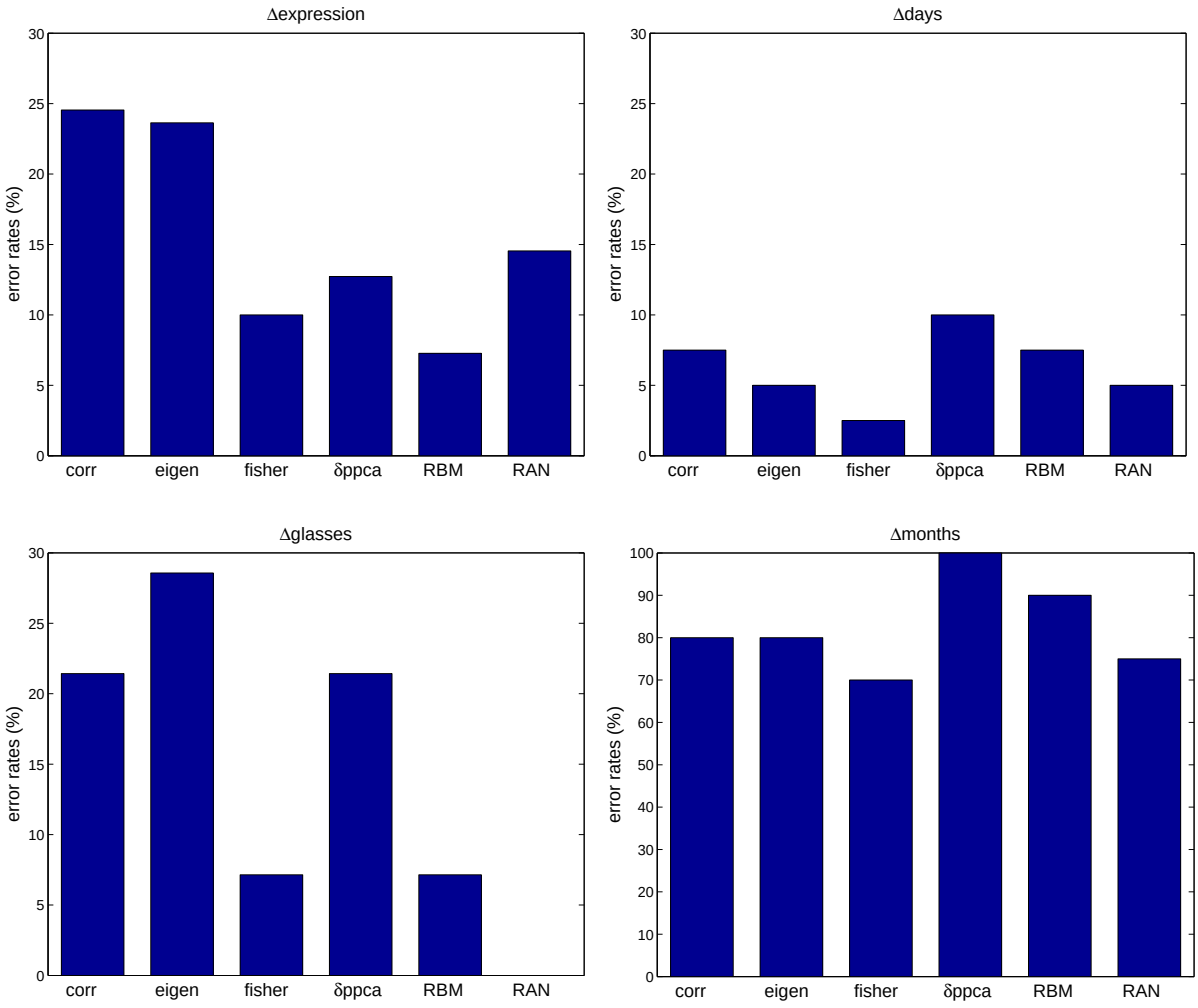


Figure 5.16: Error rates of different models on the four testing sets of the FERET faces.

From section 5.7.1, we find that RAN can generate the best low dimensional codes for KNN clustering for the FERET faces among the three models. Therefore, we will use RAN to train the face images in the training set described in section 5.6 to find good parameters of RAN. Then, we use the trained RAN to calculate the low dimensional codes of the face images in the training set and in the test set. For each image in the test set, after we find its nearest neighbour in the training set using the low dimensional codes we have obtained, we assign the testing image to the individual that its nearest neighbour belongs to.

There are 818 face images in the training set, and there are 112 individuals that only have one image each in the training set. In order to make SNE work better, we preprocessed the training set: we add some Gaussian noise to the 112 images and put the 112 “noisy” images into the training set. Before the preprocessing, the clustering error on the training set using the codes generated by SNE is 79 out of 706 (we only take into account the individuals that have two or more images in the training set). After the preprocessing, the clustering error is 74 out of 706.

Through many experiments, we find that the best configuration of RAN that produces the best clustering result on the training set is the same as that described in section 5.7.1: numrhid = 600, numydims = 180 and numrhid = 200. We set τ_{sne} to be 0 in the first 3800 epochs’ training. After epoch 3800, we set τ_{sne} to be 1500. At epoch 4000, we will get the best clustering error 66 out of 930 (including the 112 “noisy” images) on the training set. Then we stop the training of RAN and use the parameters of RAN to do recognition on the test sets. The recognition error rates on the four testing sets by RAN are shown in Figure 5.16.

By Figure 5.16, we compared the performance of RAN to that of five other face recognition methods described in the paper [Teh and Hinton, 2001].

From Figure 5.16, we can find that RAN has comparable performance to the other face recognition methods. Although it gives perfect results on the Δglasses set, it fails to recognize face images with different expressions accurately. Pair training is used in the training of RBM. It lets the model know which parts of face images are unimportant and might have variable values. Therefore, RBM does best on the Δexpression set among all the methods. RAN does better than most of the other methods on the Δdays set and on the Δmonths set.

Chapter 6

Conclusions

Analysis of high dimensional data is often encountered in object recognition. But learning in the original high dimensional space is often very expensive and computationally intolerable sometimes. So object recognition using low dimensional codes becomes important.

We proposed a method called Regularized Autoencoder Network that consists of a recognition network producing low dimensional codes, a generative network giving reconstructions of the input data from the codes, and a regularizer called SNE that encourages similar input vectors to have similar codes. It can generate low dimensional codes that have a better reflection of the pairwise underlying similarities in the input data than the raw data themselves.

We tried learning the parameters of Regularized Autoencoder Network using Scaled Steepest Gradient Descent, Conjugate Gradient Descent and Scaled Conjugate Gradient Descent. Among the three methods, Scaled Steepest Gradient Descent has the best performance on optimizing the network.

6.1 Discussion

SSGD has the best performance on optimizing the objective function of an autoencoder. The adapted learning rates help to stably find a local minimum of the objective function.

Although SCGD is fast in optimization, it has no well-defined mechanism to control the parameter λ in the method, which will influence the step size a lot during the training. In another words, the mechanism that increases and reduces the step size in the method doesn't guarantee that the update is always in the trust region during the training.

We used SSGD to optimize the objective function of the autoencoder, SNE and RAN. RAN has the best performance on generating low dimensional codes of digits and faces for recognition. When there are enough images for each class, RAN results in more stable training although it generates almost the same good codes as the autoencoder does. When the size of the training set is large, the objective function of the autoencoder is difficult to minimize and we often cannot find a good local minimum; but for RAN, we can find a good local minimum of it very easily under such a situation; when there are not enough images for each class, generally, RAN generates better codes than the autoencoder does because the autoencoder cannot capture enough information about the data for each class.

The autoencoder fails to generate good codes for KNN clustering for some images although the codes produced allow accurate reconstructions. RAN partly solves this problem and improves the clustering performance a little on these images.

RAN is an unsupervised learning algorithm. During the training, we needn't know the label information for each training case. The codes generated are very suitable for clustering. In addition, the probability matrix, which gives the probabilities that each image chooses other images as its neighbours, is highly dependent on the training data. When doing classification, the RAN trained doesn't have good generalization on the images that are not very similar to the corresponding training images, for example, on face images having exaggerated expressions. Since both the autoencoder and SNE fail to generate good codes for some images such as face images with different expressions and hand-written digit images with peculiarities, it's understandable that the combination of them, RAN, cannot generate good codes for these images either. Also, the computation of the probability matrix is expensive when the training set is very large.

6.2 Future Work

On optimization, one direction of future work is to extend SCGD to make it work as well as SSGD. Instead of using quadratic approximation to measure when to increase or lower the parameter λ, we could introduce the mechanism of adapted learning rates into SCGD method when optimizing non-linear objective functions. Another direction of future work on optimization is to extend the Stochastic Conjugate Gradient Descent method proposed by [Schraudolph and Graepel, 2003]. The drawback of that method is that we need to construct an m-dimensional (m is smaller than the dimensionality of the input data space) Krylov subspace at each update of the weight vector, which is expensive. If we can have some cheap and good approximations to the Krylov subspaces, the Stochastic Conjugate Gradient Descent method may be very effective in optimizing the objective function of the autoencoder and RAN.

On Regularized Autoencoder Network, one direction of future research is related to the probability matrix calculated on the training set. The probability matrix mentioned so far contains the probabilities that each data point chooses all other data points as its neighbours. We can simplify the matrix to make it contain the probabilities that each data point chooses its k nearest neighbours as neighbours.

We notice that RBM has very good performance on recognizing face images with varying expressions because it trains on pairs of images belonging to the same individual. Therefore, another direction of future work on RAN can extend the current RAN model to make it suitable for training on pairs of training cases belonging to the same class.

Bibliography

[Agarwal and Roth, 2002] S. Agarwal and D. Roth. Learning a Sparse Representation for Object Detection. *Proceedings of the European Conference on Computer Vision (ECCV),* 2002.

[Beis and Lowe, 1999] J. Beis and D. Lowe. Indexing without invariants in 3D object recogniton. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1000-1015, 1999.

[Bell and Sejnowski, 1996] A. Bell and T. Sejnowski. Learning higher-order structure of a natural sound. *Network: Computation in Neural Systems*, 7:261-266, 1996.

[Bell and Sejnowski, 1997] A. Bell and T. Sejnowski. The independent components of natural scenes are edge filters. *Vision Research*, 37:3327-3338, 1997.

[Belmumeur et al., 1996] P. Belmumeur, J. Hesanha and D. Kriegman. Eigenfaces versus fisherfaces: recognition using class specific linear projection. *European Conference on Computer Vision*, 1996.

[Bishop, 1995] Neural Networks for Pattern Recognition. Oxford University Press, 1995.

[Borg and Groenen, 1997] I. Borg and P. Groenen. Modern Multidimensional Scaling - Theory and Applications. Springer, 1997.

[Brunelli and Poggio, 1993] R. Brunelli and T. Poggio. Face recognition: features versus templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(10):1042-1052, 1993.

[Ding, 2002] C. Ding, X. He, H. Zha and H.Simon. Adaptive dimension reduction for clustering high dimensional data. *Proc. 2nd IEEE Int'l Conf. Data Mining*, pp.147-154, Maebashi, Japan, 2002.

[Fergus and Perona et al., 2003] A. Fergus, P. Perona, and A. Zisserman. Object class recognition by unsupervised scale-invariant learning. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2003.

[Golub and Loan, 1996] G. Golub and C. Loan. Matrix Computations, 3rd edition. Johns Hopkins, Baltimore, 1996.

[Hyvarinen et al., 1998] A. Hyvarinen, E. Oja, P. Hoyer, and J. Hurri. Image feature extraction by sparse coding and independent component analysis. In *Proceeding International Conference on Pattern Recognition (ICPR'98)*, pages 1268-1273, Brisbane, Australia, 1998.

[Hyvarinen et al., 2000] A. Hyvarinen, J. Karhunen and O. Oja. Independent Component Analysis, New York: Wiley-Interscience, 2000.

[Hinton and Roweis, 2003] G. Hinton and S. Roweis. Stochastic Neighbour Embedding. *Advances in Neural Information Processing Systems 15*, pp. 833-840. MIT Press, Cambridge, MA, 2003.

[Jackson, 1991] J. Jackson, A User's Guide to Principal Component Analysis. New York: John Wiley and Sons, 1991.

[Jolliffe, 1986] I. Jolliffe. Principle Component Analysis. Springer Verlag, 1986.

[Kumar and Andreou, 1996] N. Kumar, A. Andreou. A Generalization of Linear Discriminant Analysis in Maximum Likelihood Framework. *Proceedings of the Joint Statistical Meeting, Statistical Computing section*, Chicago, Aug 4-8, 1996.

[Lawrence et al., 1997] S. Lawrence, C.Giles, A. Tsoi, and A. Back. Face recognition: A convolutional neural network approach. *IEEE Transactions on Neural Networks*, vol. 8, no. 1, pp. 98-113, 1997.

[Lowe, 1999] D. Lowe. Object recognition from local scale-invariant features. *Proceeding of the International Conference on Computer Vision*, Corfu, 1999.

[Mika et al., 1999] S. Mika, B. Scholkopf, A. Smola, K. Muller, M. Scholz, and G. Ratsch. Kernel PCA and de-noising in feature spaces. *Advances in Neural Information Processing Systems*, volume 11, pages 536-542. MIT Press, Cambridge, MA, 1999.

[Moghaddam et al., 1998] B. Moghaddam, W. Wahid, and A. Pentland. Beyond eigenfaces: probabilistic matching for face recognition. *IEEE International Conference on Automatic Face and Gesture Recognition*, 1998.

[Moller, 1993] A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks* 6(4), pp. 525-533.

[Nocedal and Wright, 1999] J. Nocedal and S. Wright. Numerical Optimization. Springer Verlag, 1999.

[Olshausen and Field, 1996] B. Olshausen and D. Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. Nature, 381:607-609, 1996.

[Paccanaro and Hinton, 2000] A. Paccanaro and G. Hinton. Extracting Distributed Representations of Concepts and Relations from Positive and Negative Propositions. *Proceedings of the International Joint Conference on Neural Networks*, IJCNN 2000.

[Pentland et al., 1994] A. Pentland, B. Moghaddam, and T. Starner. View-based and modular eigenspaces for face recogniton. *IEEE Conference on Computer Vision and Pattern Recognition*, 1994.

[Phillips et al., 1999] P. Phillips, H. Moon, S.Rizvi, and P. Rauss. The FERET Evaluation Methodology for Face-Recognition Algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1999.

[Roweis and Saul, 2000] S. Roweis and L. Saul. Nonliear dimensionality reduction by locally linear embedding. Science, 290, 2000.

[Roweis webpage] http://www.cs.toronto.edu/~roweis/data.html

[Samaria, 1994] F. Samaria. Face Recognition Using Hidden Markov Models. PhD thesis, University of Cambridge, 1994.

[Saul and Roweis, 2003] L. Saul and S. Roweis. Think Globally, Fit Locally: Unsupervised Learning of Low Dimensional Manifolds. *Journal of Machine Learning Research*, v4, pp. 119-155, 2003.

[Scholkopf et al., 1998] B. Scholkopf, A. Smola, and K. Muller. Nonlinear Component Analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299-1319, 1998.

[Schraudolph and Graepel, 2003] N. Schraudolph and T. Graepel. Combining Conjugate Direction Methods with Stochastic Approximation of Gradients. *Proc. 9th Intl. Workshop*

Artificial Intelligence and Statistics, Key West, 2003

[Scott, 1992] D. Scott. Mutivariate Density Estimation: Theory, Practice, and Visualization. John Wiley, 1992.

[Stone and Porrill, 1998] J. Stone and J. Porrill. Undercomplete Independent Component Analysis for signal separation and dimension reduction. Technical report, Department of Psychology, Sheffield University, 1998.

[Teh and Hinton, 2001] Y. Teh, G. Hinton. Rate-coded Restricted Boltzmann Machines for Face Recognition. *Advances in Neural Information Processing Systems 13*, MIT Press, Cambridge, MA, 2001

[Tenenbaum et al., 2000] J. Tenenbaum, V. Silva and J. Langford. A global geometric framework for nonlinear dimensionality reduction. Science, pp. 2319-2323, 2000.

[Turk and Pentland, 1991] M. Turk and A. Pentland. Eigenfaces for recogniton. *Journal of Cognitive Neuroscience*, 3(1):71-86, 1991.

[Wiskott, 1997] L. Wiskott,and M. Fellous, N. Kruger, and C. Malsburg. Face Recognition by Elastic Bunch Graph Matching. *IEEE Transactions of PAMI*, 19(7):775--779, 1997.

[Yuille, 1991] A. Yuille. Deformable Templates for Face Recognition. *Journal of Cognitive Neuroscience*, vol. 3, no. 1, pp. 59-79, 1991.