Tommi Tuovinen

Multidimensional scaling using multilayer perceptron

Master's Thesis in Information Technology

May 29, 2013

University of Jyväskylä

Department of Mathematical Information Technology

Author: Tommi Tuovinen

Contact information: tommi.s.tuovinen@gmail.com

Supervisors: Tommi Kärkkäinen and Tuomo Rossi

Title: Multidimensional scaling using multilayer perceptron

Työn nimi: Moniulotteinen skaalaus käyttäen MLP-neuroverkkoa

Project: Master's Thesis

Study line: Software engineering

Page count: 47+18

Abstract: The objective of this thesis is to introduce the reader to the concepts of neural network and multidimensional scaling and to demonstrate how these two can be used together. The thesis introduces a construction in which a multilayer perceptron is trained by means of multidimensional scaling in order to perform dimensionality reduction. The algorithm is tested in four different test experiments.

Keywords: multidimensional scaling, neural network, dimensionality reduction, multilayer perceptron, visualization

Suomenkielinen tiivistelmä: Tämän tutkimuksen tarkoituksena on avata neuroverkon ja moniulotteisen skaalauksen käsitteitä sekä demonstroida kuinka näitä voidaan käyttää yhdessä. Tutkielmassa suoritetaan konstruktio, jossa MLP-verkko koulutetaan moniulotteisen skaalauksen keinoin suorittamaan dimension pienennystä. Algoritmia testataan neljässä eri testitapauksessa.

Avainsanat: moniulotteinen skaalaus, neuroverkko, dimension pienennys, MLP-verkko, visualisointi

Preface

For me, the writing of this master's thesis was an edifying experience. I was not familiar with the topic of the thesis beforehand and I knew that the subject would be, in any case, greatly challenging. Gladly I can once again state that with great perseverance and confidence any possible goal can be achieved. I would like to thank my supervisor, professor Tommi Kärkkäinen for the topic of the thesis, many valuable comments, and the idea of combining the concepts of multilayer perceptron and multidimensional scaling as presented in this thesis. I would also like to thank my other supervisor professor Tuomo Rossi for taking part in several review sessions. Finally I would also like to express my gratitude to my friends and family for the support.

Jyväskylä, May 29, 2013

Tommi Tuovinen

List of Figures

List of Tables

Contents

1 Introduction

We live in a time when the amount of recorded data in the world is beyond our comprehension. In science, it is common to gather huge quantities of statistical data from sensor devices. In business, the companies have a great deal of customer related information. The main question is, how can we take advantage of all this data? We somehow need to extract the valuable information from vast and multidimensional data. This is exactly the problem that *data mining* concentrates on.

The objectives of data mining can be divided into five different *tasks*. These are:

1. exploratory data analysis
2. descriptive modeling
3. predictive modeling
4. discovering patterns and rules
5. retrieval by content. (Hand, Mannila, and Smyth 2001, p. 11–15)

The first task, exploratory data analysis is heavily based on visualization. In order to analyse the data, it is often beneficial to visualize the data first and analyse the visual presentation. Visualizing data with more than three dimensions is highly problematic, and therefore it is common to use dimensionality reduction to lower the dimension of data. This thesis concentrates on a specific set of dimensionality reduction called multidimensional scaling. Based on what I have described, a specific unofficial taxonomy (Figure 1), that shows the connection between data mining and multidimensional scaling, can be formed. Dimensionality reduction is a multidisciplinary concept and therefore it is classified differently in other branches of science. In statistics, dimensionality reduction and multidimensional scaling are categorized under multivariate analysis (Seber 1984).

Let us assume that we have some numerical data in a form of a matrix. As usual, the rows represent the samples and the columns depict the variables. Number of variables is the same as the dimension of data. If we have just three variables we can visualize data, for example, with a three-dimensional scatter plot. However, usually we have more than three variables. We need to either choose the variables we want to display at the same time or

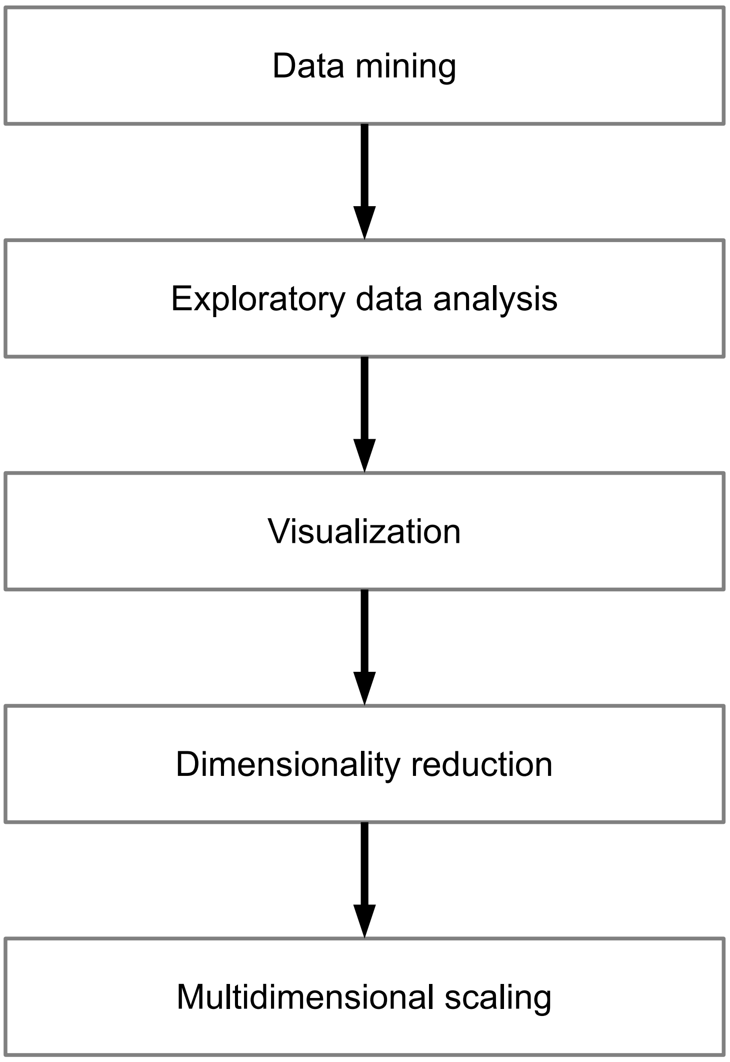

Figure 1. Multidimensional scaling in a methodological hierarchy.

then use dimensionality reduction methods to transform the information into a form with less dimensions.

This thesis concentrates on a dimensionality reduction technique called multidimensional scaling and how it can be implemented differently using neural networks. In Chapter 2 I shall briefly introduce the concept of neural networks and depict a network model called multilayer perceptron. In Chapter 3 I will describe dimensionality reduction and briefly introduce a few of well-known techniques such as multidimensional scaling. In Chapter 4 I will perform a construction that combines multidimensional scaling with multilayer perceptron. The combined technique will be tested with four experiments in Chapter 5. The last chapter includes analysis of the results of the study.

2 Neural networks

In this thesis, the term *neural network* is associated only with artificial neural networks. In general, the term may also refer to biological neural networks as the ones in the human brain (Figure 2). However, both artificial and biological neurons have many similarities (Fausett 1994, p. 5–7).

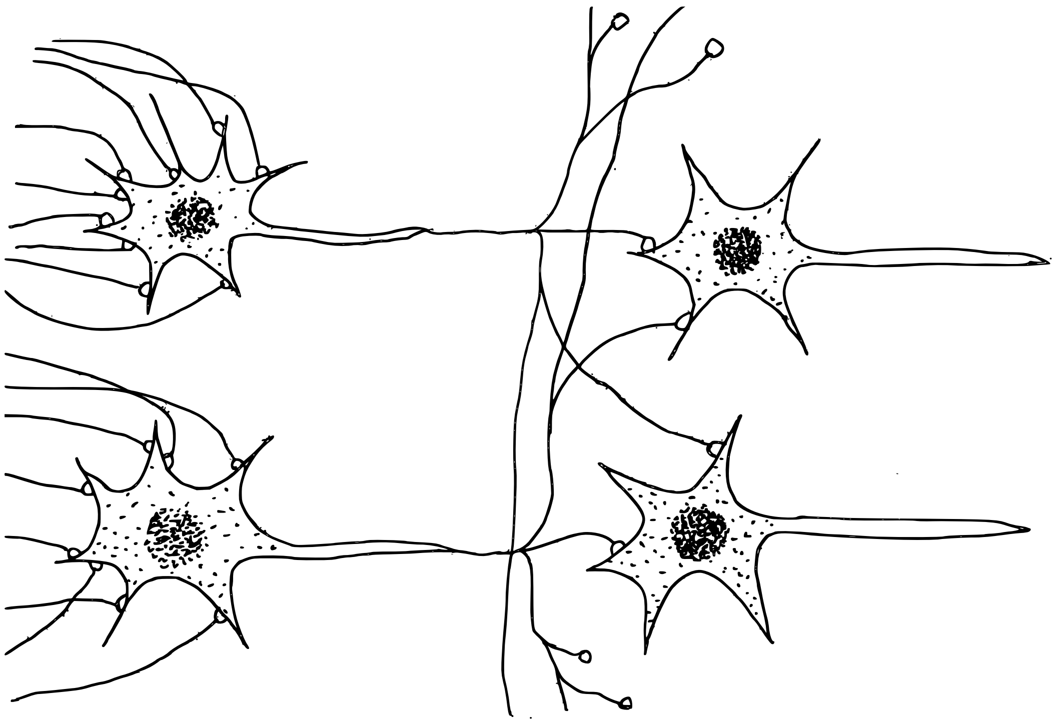

Figure 2. Biological neural network. The figure is to some extent based on the one presented in (Aleksander and Morton 1990).

Simon Haykin (Haykin 1999, p. 2) uses the following definition for neural networks:

> ”A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experimental knowledge and making it available for use. It resembles the brain in two respects:
>
> 1. Knowledge is acquired by the network from its environment through a

learning process.

2. Interneuron connection strengths, known as synaptic weights are used to store the acquired knowledge."

This definition is a great starting point. To understand better how a neural network operates, it is best to study its structure. A neural network consists of units called *neurons*. The first neuron model was invented in 1943 by Warren McCulloch and Walter Pitts in a famous article (McCulloch and Pitts 1943). This McCulloch-Pitts neuron already had many of the same properties as the neuron models used nowadays.

The structure of a neuron is described in the following section. After that I will examine activation functions, which are an important part of neurons. After introducing the essentials of the neuron, I will discuss different neural network architectures from which I have chosen a specific architecture, multilayer perceptron, under more precise observation. Finally, I will briefly list the methods and the objectives of training a neural network.

2.1 Structure of a neuron

The structure of a neuron is illustrated in Figure 3. The neurons are connected together by *synapses*. Each synapse has a (synaptic) *weight* which is a numerical value. Normally the weights are used to multiply the input signal (Fausett 1994, p. 3). A crucial part of the neuron is an *adder*. The adder sums the weighted input signals. The operation can be written into a formula:

$$u_k = \sum_{j=1}^{p} w_{kj} x_j \tag{2.1}$$

where u_k is the output of the adder. $w_{k1}, w_{k1}, \ldots, w_{kp}$ are the weights and $x_1, x_2, \ldots, x_p$ are the input signals for neuron k. (Haykin 1999, p. 10–11)

The second part of the neuron is a function called the *activation function*. The activation function compresses the output signals into some specific range. Two of the most used ranges are closed intervals [0,1] and [-1,1]. (Haykin 1999, p. 11) The activation function will be discussed in more detail in the following section.

It is common to influence the input of a neuron with a *bias*. The bias can be considered as

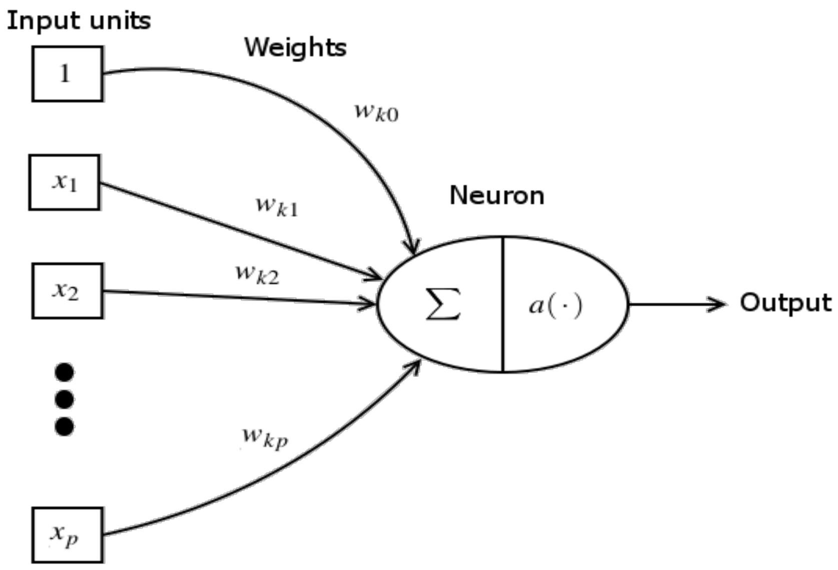

Figure 3. The structure of a neuron.

a weight applied to an input that is always equal to 1 (Fausett 1994, p. 41–42). Because of the bias, the output of adder no longer crosses the origin (Haykin 1999, p. 11). The formula determining the output of the adder stays essentially the same:

$$v_k = w_{k0}x_0 + \sum_{j=1}^{p} w_{kj}x_j = \sum_{j=0}^{p} w_{kj}x_j, \tag{2.2}$$

where w_{k0} is the bias term, x_0 is the fixed input ($x_0 = 1$) and v_k is the output of the adder. Let us define

$$\mathbf{w}_k = \begin{bmatrix} w_{k0} \\ w_{k1} \\ w_{k2} \\ \vdots \\ w_{kp} \end{bmatrix} \in \mathbb{R}^{p+1}, \qquad \tilde{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} \in \mathbb{R}^{p+1} \tag{2.3}$$

and now we can present the output of the adder in a compact form

$$v_k = \mathbf{w}_k^T \tilde{\mathbf{x}}. \quad \text{(Kärkkäinen 2000)} \tag{2.4}$$

The bias can also be replaced by an external parameter called a *threshold* that can be applied to the activation function. It does not matter whether we use a bias or a threshold because they are both substantially equivalent to each other. (Fausett 1994, p. 41–42) Later in this thesis I will use a bias instead of a threshold since it proves to be more appropriate for the purpose of this study.

2.2 Activation function

The activation function is used to convert the transformation from linear to nonlinear. In addition, the use of the activation function limits output of the neuron to a certain range. Normally the same activation function is applied to all neurons in the neural network (Fausett 1994, p. 7). In this section I will describe two commonly used activation functions: threshold function and logistic function.

The threshold function (Figure 4) is a simple step function:

$$a(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{if } x < 0, \end{cases} \tag{2.5}$$

where x is the output of the adder and a is the activation function (Haykin 1999, p. 12).

The logistic function (Figure 4) is defined as

$$a_k(x) = \frac{1}{1+e^{-kx}} \tag{2.6}$$

and again a_k is the activation function and x is the output of the adder (Haykin 1999, p. 12). k is a parameter that determines the steepness of the function. A greater value inflicts greater steepness. An important notice is that logistic function is both continuous and differentiable which is, as we will later discover, very useful. In addition, the logistic function can be derived using Bayes' theorem and thus the outputs of the function can be interpreted as posterior probabilities (Bishop 1995, p. 82, 83).

Both activation functions introduced compress output into the closed interval [0,1]. As mentioned before, sometimes the interval [-1,1] is preferred. In order to achieve this range, the threshold function can be formulated differently:

$$a(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{if } x = 0, \\ -1, & \text{if } x = 0. \end{cases} \tag{2.7}$$

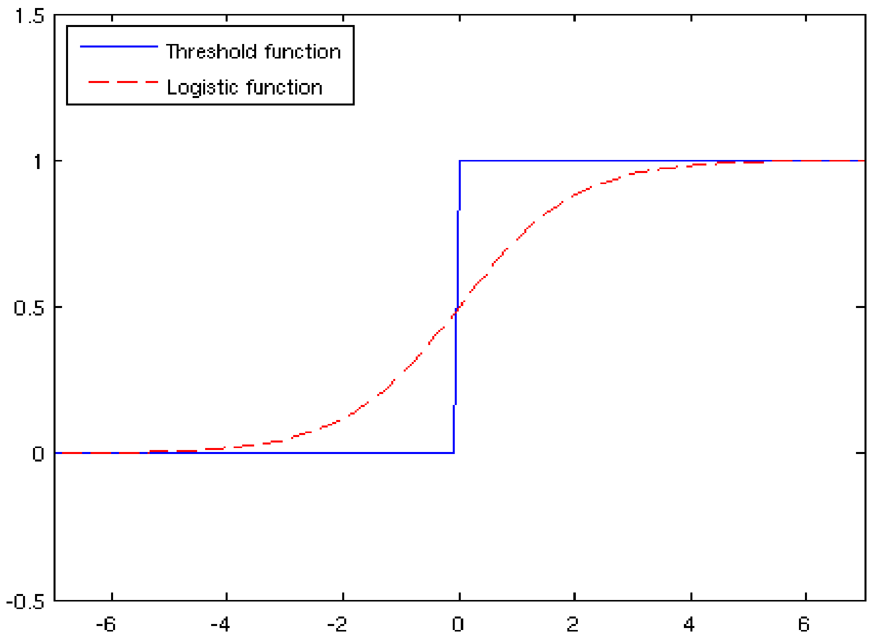

Figure 4. Threshold and logistic function

This function is called the *signum function*. Respectively, instead of the logistic function we can use the *hyperbolic tangent function*:

$$a(x) = tanh(x). \quad \text{(Haykin 1999, p. 14–15)} \tag{2.8}$$

The hyperbolic tangent function appears to be similar to the logistic function except that it has the range from [-1,1]. As a result of this, the function passes through the origin.

Now that we have defined the activation function we can mathematically present the whole functionality of one neuron as

$$S_k = a(\mathbf{W}_k^T \tilde{\mathbf{x}}), \tag{2.9}$$

where S_k is the output of neuron k. $\mathbf{W}_k^T \tilde{\mathbf{x}}$ is the output of the adder as we defined in Section 2.1. This definition will prove to be useful when formulating a notation for a whole neural network.

2.3 Neural network architectures

In this section I will go through two basic neural network architectures: single-layer feedforward networks and multilayer feedforward networks. I will start with the single-layer architecture for the reason that it is the simplest of the architectures. There are also other architectures, such as recurrent networks, but it is not essential for the thesis to examine them.

2.3.1 Single-layer feedforward networks

The single-layer feedforward network (Figure 5) or simply the single-layer network has two layers: an input and an output layer. The input layer consists of *input units* that relay the original input of the neural network. Since the input layer does not contain neurons, it does not count as a real layer and therefore the name single-layer network. The output layer consists of output units, which are normal neurons acting the way I have described earlier. The input and output layers are connected together by weighted synapses. The signals always go from only the input layer to the output layer as the term feedforward implies.(Haykin 1999, p. 21) To be clear, the input units of feedforward networks are never connected to other input units by synapses and respectively output units are never connected to each other.

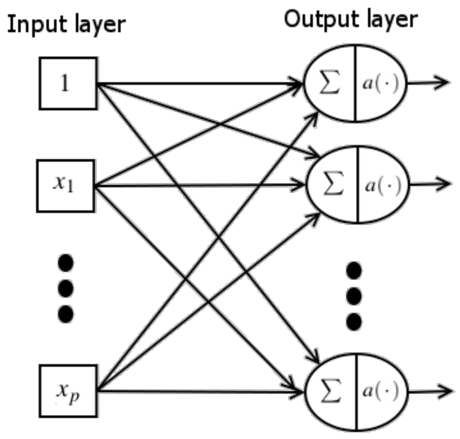

Figure 5. The single-layer feedforward network.

2.3.2 Multilayer feedforward networks

A multilayer feedforward network or simply a multilayer network is mainly the same as a simple-layer network except that it has more layers. These additional layers are called *hidden layers* (Haykin 1999, p. 21). As the output layer, the hidden layer consists of ordinary neurons. The only difference between these layers is the position. The output layer is always the last layer considering the direction of the signal course. The hidden layers are positioned between the input layer and the output layer. The next section presents a specific kind of multilayer network called a *multilayer perceptron*.

2.4 Multilayer perceptron

A multilayer perceptron (Figure 6) or an MLP-network is one of the multilayer network architectures. According to Simon Haykin (Haykin 1999, p. 156–157), the MLP-network is a neural network that has three specific properties:

1. The activation function used in neurons is both nonlinear and differentiable everywhere.
2. The neural network contains at least one hidden layer.
3. The neural network has a high level of connectivity meaning that there are plenty of synapses.

A multilayer perceptron has the ability of *universal approximation* since it satisfies the assumption of universal approximation theorem (Haykin 1999, p. 208–209). As George Cybenko (Cybenko 1989) proves, a multilayer perceptron with only one hidden layer is satisfactory, with support in an n-dimensional unit hypercube, to approximate any continuous function.

Next I will introduce mathematical representation for an MLP-network. This will be done similarly as in (Kärkkäinen 2000) with only slightly different symbols for variables. I have already formulated the representation of one neuron in Section 2.2 as

$$S_k = a(\mathbf{W}_k^T \tilde{\mathbf{x}}). \tag{2.10}$$

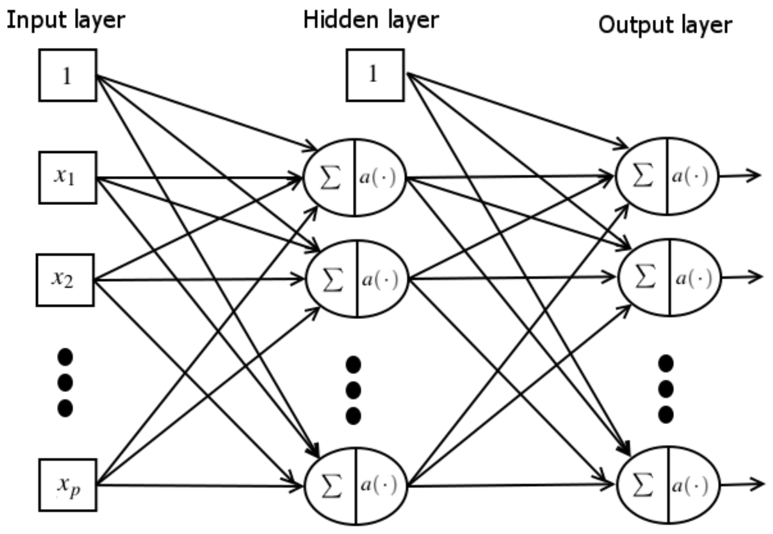

Figure 6. Multilayer perceptron with one hidden layer.

Now I will start by constructing one layer. $\mathbf{w}_k$ contains weights for neuron k or its synapses to be correct. But in this case we need all the weights of the layer. Let us assume that our layer has m neurons. We define the weights as $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m$. Next we gather these weights into a matrix W as

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_m^T \end{bmatrix} \quad \text{(Kärkkäinen 2000).} \tag{2.11}$$

Now linear transformation for input $\tilde{x}$ is simply

$$\mathbf{v} = \mathbf{W}\tilde{\mathbf{x}} \quad \text{(Kärkkäinen 2000).} \tag{2.12}$$

Notice the difference between this form and (2.4). Here we have output of multiple adders as a vector whereas in (2.4) we have an output of a single adder as a real.

The next step is to insert the activation functions to the previous formula. In order to do this, we can write the linear transformation V open as

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} \tag{2.13}$$

and then apply the activation functions as

$$\mathbf{s} = \begin{bmatrix} a_1(v_1) \\ a_2(v_2) \\ \vdots \\ a_m(v_m) \end{bmatrix} \tag{2.14}$$

However, we can accomplish the same result with a smarter notation. Let us define a *diagonal function-matrix* containing the activation functions

$$\mathscr{A} = \begin{bmatrix} a_1(\cdot) & 0 & \cdots & 0 \\ 0 & a_2(\cdot) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_m(\cdot) \end{bmatrix} \quad \text{(Kärkkäinen 2000).} \tag{2.15}$$

Now we have a definition for the functionality of one layer:

$$\mathbf{s} = \mathscr{A}(\mathbf{v}) = \mathscr{A}(\mathbf{W}\tilde{\mathbf{x}}). \quad \text{(Kärkkäinen 2000)} \tag{2.16}$$

In consequence of previous notation we can now easily define a general algebraic form for an MLP-network. We know how to represent a layer and we know that the output of one layer is the input of the next layer. The recursive formula of an MLP-network with $L-1$ hidden layers is

$$\mathbf{o} = \mathbf{o}^L = \mathscr{N}(x) = \mathscr{A}^L(\mathbf{W}^L\tilde{\mathbf{o}}^{(L-1)}), \tag{2.17}$$

where

$$\begin{cases} \mathbf{o}^0 = \tilde{\mathbf{x}} \\ \mathbf{o}^l = \mathscr{A}^l(\mathbf{W}^l\tilde{\mathbf{o}}^{(l-1)}), \qquad \text{for } l = 1,\ldots,L. \end{cases} \quad \text{(Kärkkäinen 2000)} \tag{2.18}$$

However, simpler but still nonlinear transformation is gained if the activation term of the final layer is excluded, thus making the final layer linear (Kärkkäinen 2002). As a result, the formula for a multilayer perceptron with one hidden layer would be defined as

$$\mathbf{o} = \mathscr{N}(x) = \mathbf{W}^2\tilde{\mathscr{A}}^1(\mathbf{W}^1\tilde{\mathbf{x}}) \tag{2.19}$$

and a perceptron with two hidden layers would be defined as

$$\mathbf{o} = \mathscr{N}(x) = \mathbf{W}^3\tilde{\mathscr{A}}^2(\mathbf{W}^2\tilde{\mathscr{A}}^1(\mathbf{W}^1\tilde{\mathbf{x}})). \tag{2.20}$$

2.5 Training

As I earlier mentioned, neural networks have the ability to learn things. Naturally, in order to learn, a neural network must be trained. Training in general is performed by adjusting the weights by optimizing a loss function. One of the most well-known training algorithms is named as *backpropagation algorithm* and one of its main basis is the article (Rumelhart, Hinton, and Williams 1986).

Neural networks can be regarded as a branch of machine learning. Machine learning algorithms can be divided into classes of which the two most important are *supervised learning* and *unsupervised learning*. Supervised learning is usually done by using training sets. The training set contains the preferred results. Generally, the training set is given in the form $\{(x_i, y_i)\}$ and the objective of the learning is to find a function f so that $f(x_i) = y_i$ for all i (Shavlik and Dietterich 1990, p. 1). When training a neural network, the training set contains the desired output for certain inputs. The weights are adjusted so that all the network's output results are as close as possible to the ones recorded in the training set.

Instead, unsupervised learning searches for regularities in the data which are mainly in the form of clusters. Unsupervised learning includes clustering methods and self-organizing maps. (Shavlik and Dietterich 1990, p. 263–265) Self-organizing maps were introduced by Teuvo Kohonen in (Kohonen 1982).

2.6 Applications

One of the main applications for neural networks is classification and its subcategory pattern recognition. One classical example of pattern recognition is analysing handwritten text by machine. If we have a well trained neural network and the text to analyse is to some extend similar with the training material, the network should be able to recognize every character in the text. (Fausett 1994, p. 8–9, 71–76)

Feedforward neural networks are commonly used for prediction and function approximation. Especially the MLP-network is excellent for approximation because as I earlier mentioned it is an universal approximator. In addition, neural networks are used as *autoencoders*, which will be discussed further in the next section with the other dimensionality reduction techniques.

3 Dimensionality reduction

Nowadays it is common that one has to examine and analyze multidimensional data. In order to understand and visualize multidimensional data, dimensionality reduction is needed. The less dimensions there are, the easier the information is to interpret. Visualizing data that has more than three dimension is problematic and thus the desirable dimension of the data is usually two or three. Generally, a dimensionality reduction technique takes the data points for input and transforms them into a space with fewer dimensions than the starting space.

There are a great deal of different techniques for dimensionality reduction such as principal component analysis, Sammon mapping, isomap, classical multidimensional scaling and diffusion maps. Laurens van der Maaten has created a MATLAB toolbox (Maaten) containing 34 well-known dimensionality reduction techniques. Dimensionality reduction techniques can be divided into linear and nonlinear techniques. The nonlinear techniques usually outperform the linear techiques for the reason that the nonlinear techniques can manage complicated nonlinear data, whereas linear techniques cannot.

In this chapter, I will describe four dimensionality reduction methods: principal component analysis, autoencoders, neuroscale and multidimensional scaling. The principal component analysis is the most used dimensionality reduction method. Autoencoders and Neuroscale are neural networks based methods and therefore interesting to be observed. One technique, multidimensional scaling will be discussed more accurately since it has a crucial role in this thesis.

3.1 Principal component analysis

The principal component analysis (PCA) is the most used linear dimensionality reduction method. It is based on the papers done by Karl Pearson (Pearson 1901) and Harold Hotelling (Hotelling 1933a, 1933b). PCA is a linear dimensionality reduction technique so therefore it cannot manage complex nonlinear data. However, there have been many improvements made to the PCA thus creating methods such as kernel PCA (Schölkopf, Smola, and Müller 1998), which is in fact a nonlinear method.

Let us define PC_1, PC_2, ..., PC_p as *principal components*. The principal components are orthogonal linear combinations of the original variables:

$$\begin{aligned} PC_1 &= a_{11}x_1 + a_{21}x_2 + \cdots + a_{p1}x_p \\ PC_2 &= a_{12}x_1 + a_{22}x_2 + \cdots + a_{p2}x_p \\ &\vdots \\ PC_p &= a_{1p}x_1 + a_{2p}x_2 + \cdots + a_{pp}x_p \end{aligned} \tag{3.1}$$

where x_1, x_2, ..., x_p are the original variables and $a_j^T = [a_{1j}, \ldots, a_{pj}], (j \in \{1, \ldots, p\})$ is a vector of constants. Vectors a_1, a_2, ..., a_p are chosen so that the first principal component PC_1 has the highest possible variance. The rest of the principal components are determined consequently so that the second principal component has the second highest variance and the last principal component has the lowest variance. Generally, only the first few principal components are needed and the others are discarded. (Chatfield and Collins 1980)

3.2 Autoencoders

An autoencoder is a special kind of a neural network used for dimensionality reduction. The autoencoder consists of an encoder and a decoder. The encoder transforms the data into a form of lower dimensions. The decoder takes the output of the encoder and increases the number of dimensions back to the original. (Maaten, Postma, and Herik 2009)

3.3 Neuroscale

Neuroscale is a nonlinear dimensionality reduction method, in which a neural network named *radial basis function (RBF) network* is applied to the dimensionality reduction method called Sammon mapping. Sammon mapping will be discussed more in the following section as one of the multidimensional scaling techniques.

The neuroscale can be extended by including a concept of *subjective dissimilarities*. These dissimilarities are additional information assigned to each data point pair. Normally the subjective dissimilarities are some kind of classifying knowledge. In the simplest case, the data points of the same class have zero dissimilarity whereas the data points from different classes have a constant dissimilarity. The subjective dissimilarities and the spatial dissimilarities can be combined into a single formula:

$$\delta_{ij} = (1-\alpha) * d_{ij} + \alpha * s_{ij}, \quad 0 \le \alpha \le 1, \tag{3.2}$$

where d_{ij} is the spatial dissimilarity between data points i and j, s is respectively the subjective dissimilarity. α is a coefficient that indicates the weight of the dissimilarity type. If α is one, only the subjective dissimilarity has influence. If α is zero then spatial dissimilarity is the only one to have effect. The proper value for α is discovered through trial and error. (Lowe and Tipping 1996)

3.4 Multidimensional scaling

Multidimensional scaling (MDS) can be considered as a group of techniques for generating a *spatial representation* out of the *proximities* of objects (Kruskal and M. 1978, p. 7). The spatial representation is normally a set of points in a two or three dimensional Euclidean space. Proximity is a synonym for similarity meaning how similar two objects are with each other.

A much used example to demonstrate multidimensional scaling is the construction of a map of cities using either road distances or travel times between the cities. Joseph Kruskal and Myron Wish (Kruskal and M. 1978, p. 7–9) construct a map of ten cities in the United States with the knowledge of the airline distance between the cities. Mardia et al. (Mardia, Kent, and Bibby 1980) use road distances of 12 British cities and Trevor and Michael Cox (Cox and Cox 2001) present a similar example using journey times by road between some British cities. To honour this tradition, I will later introduce a similar example.

The main principle of multidimensional scaling is that distances of datapoints in the representation that the method creates should be as close the real distances as possible. It is intuitively understandable that this rule does not define rotation, translation, and reflection and therefore we may freely perform any of these operations. It is as if one had a map in one's hands: one can rotate it, move it or even turn it upside down and yet the distances on the map stay the same.

Multidimensional scaling can be divided into metric and non-metric multidimensional scaling. Metric multidimensional scaling is based on the assumption that the dissimilarities of observed objects have the attributes of Euclidean distance. In following sections I will introduce two common metric methods: classical scaling and Sammon mapping. I will also briefly discuss non-metric multidimensional scaling. There are also plenty of other multidimensional scaling techniques but they are outside the scope of this thesis.

3.4.1 Classical scaling

Classical multidimensional scaling was presented by Warren Torgerson (Torgerson 1952) with the help of theoretical knowledge introduced by Young and Householder (Young and Householder 1938). In classical scaling, dissimilarities are considered as Euclidean distances (Cox and Cox 2001, p. 6). An interesting characteristic of the classical multidimensional scaling is that the solution it gives is identical with the one of the PCA when the dissimilarities are Euclidean (Maaten, Postma, and Herik 2009).

A crucial part of the solution is determining eigenvalues and -vectors. Eigenvalue determines the significance of the corresponding eigenvector. The number of eigenvectors is equivalent with dimensions of the representation. (Cox and Cox 2001)

Classical scaling is rather simple compared to other multidimensional scaling techniques and there are plenty of implementations of it available. The numerical computing environment MATLAB has a function called *cmdscale* (Mathworks 2013a) for executing classical multidimensional scaling. However, since classical scaling assumes dissimilarities to be Euclidean distances it cannot be applied into non-metric data. Non-metric multidimensional scaling can be computed in MATLAB with the function *mdscale* (Mathworks 2013b).

3.4.2 Non-metric multidimensional scaling

In non-metric multidimensional scaling, we are dealing with non-metric data which means that the dissimilarities cannot be interpreted as distances with metric attributes. For example, we might be dealing with data that exists on an ordinal scale. In non-metric scaling, the dissimilarities can not be used as they are. Instead, we use disparities (denoted as $\hat{d}$) that are a result of a transformation performed to dissimilarities. The only constraint is that these disparities must have the same rank ordering as the dissimilarities. This gives us the monotonicity constraint:

$$\delta_{ij} < \delta_{i'j'} \Rightarrow \hat{d}_{ij} \leq \hat{d}_{i'j'} \quad \text{for all} \quad 1 \leq i, j, i', j' \leq n, \tag{3.3}$$

where δ_{ij} is the dissimilarity between the objects i and j, $\hat{d}_{ij}$ is the disparity acquired from the dissimilarity δ_{ij} with a transformation, and n is the number of the objects. (Cox and Cox 2001)

Non-metric multidimensional scaling is commonly performed by minimizing a *loss function* (Cox and Cox 2001). A well-known loss function called the STRESS was introduced by Joseph Kruskal (Kruskal 1964) based on the earlier studies of Roger Shepard (Shepard 1962a, 1962b). Kruskal uses the term *goodness of fit* to describe how similar the distances of configuration are to the original dissimilarities. To measure the goodness Kruskal defines so called *raw stress* function

$$S_{raw} = \sum_{i<j} (d_{ij} - \hat{d}_{ij})^2 \tag{3.4}$$

where d_{ij} is the distance between the points i and j in the new configuration. The smaller the returned stress value is, the better fit the configuration has. As Kruskal points out, when uniform scaling is conducted to the configuration of points, the goodness of fit changes its value. As the ratio of distances stays the same, it would be logical that the goodness of fit would stay the same. Therefore Kruskal introduces a scaling factor $\sum_{i<j} d_{ij}^2$ to solve this problem. He also adds a square root comparing this operation to one choosing standard deviation over variance. The final result for the STRESS function is

$$S = \sqrt{\frac{\sum_{i<j} (d_{ij} - \hat{d}_{ij})^2}{\sum_{i<j} d_{ij}^2}} \quad \text{(Kruskal 1964).} \tag{3.5}$$

One notable loss function was presented by John Sammon Jr (Sammon Jr 1969) as a part of nonlinear mapping algorithm. The function is defined as

$$S = \frac{1}{\sum_{i<j} \delta_{ij}} \sum_{i<j}^{N} \frac{(d_{ij} - \delta_{ij})^2}{\delta_{ij}}. \tag{3.6}$$

Sammon uses the method of deepest steepest descent in order to minimize this function. In literature, Sammon's nonlinear mapping algorithm is often referred as *Sammon's mapping*.

Takane et al. (Takane, Young, and De Leeuw 1977) present a multidimensional scaling algorithm called alternating least squares algorithm for individual differences scaling (ALSCAL). The main idea of the algorithm is to minimize a loss function called *SSTRESS* with a *alternating least squares* (ALS) method. The SSTRESS is highly similar to the earlier described raw STRESS function. In a simplified form the formula of the SSTRESS is

$$S = \sum_{i<j} (d_{ij}^2 - \hat{d}_{ij}^2)^2. \tag{3.7}$$

so contrary to the STRESS the distances and disparities here are squared.

4 MDS using neural networks

So far I have shortly introduced the concepts of MDS and neural networks. In this chapter I will combine these two subjects. Although neural networks have been used in dimensionality reduction (methods such as autoencoders and selforganizing maps), MDS and neural networks are seldom used together. Neural networks have been used together with principal component analysis with promising results (Baldi and Hornik 1989; Abbas and Fahmy 1992; Oja 1992). Andrew Webb uses multidimensional scaling and radial basis function together through iterative majorization (Webb 1995). Using neural networks in non-metric multidimensional scaling has been studied by Wezel et al. in (Wezel et al. 2001) where they use neural networks for transforming dissimilarities of original data into disparities so that the the monotonicity constraint represented in Equation 3.3 fulfils. Otherwise the method resembles traditional non-metric MDS, so the neural network is used only to improve the non-metric phase.

Although use of neural networks in dimensionality reduction has been studied in several cases, I am quite confident that identical method to what I am about to present has not been introduced yet. In first section, I will describe the main idea of the new method. In the following section, I will reveal the essential parts of the code implementation.

4.1 MLPMDS

I will now present a method that combines the concepts of MDS and MLP. The method is referred here as MLPMDS. I have chosen the MLP model for the purposes of construction for the reason that it can be expressed in a very compact algebraic form and it has the abilities of universal approximator and unbiased estimate. The main idea is to apply MLP-network into the SSTRESS function. As I have earlier described, the SSTRESS function is defined as

$$S = \sum_{i<j} (d_{ij}^2 - \hat{d}_{ij}^2)^2. \tag{4.1}$$

When using metric data, $\hat{d}_{ij}$ is merely the distance between objects i and j in the original data space. d_{ij} is the distance of transformed datapoints $\mathbf{y_i}$ and $\mathbf{y_j}$ in the feature space. Thus

$$d_{ij}^2 = (\mathbf{y_i} - \mathbf{y_j})^T (\mathbf{y_i} - \mathbf{y_j}). \tag{4.2}$$

Now we will replace the $\mathbf{y_i}$ and $\mathbf{y_j}$ terms in the equation with MLP-networks $\mathscr{N}(x_i)$ and $\mathscr{N}(x_j)$. The MLP-network $\mathscr{N}(x_i)$ is given the data point x_i from data space as an input. The network transforms the data point into the feature space and returns the transformed data point y_i as output. Let us name the new loss function as *SMLP* and define it as

$$SMLP = \sum_{i<j} (\|\mathscr{N}(x_i) - \mathscr{N}(x_j)\|^2 - \|x_i - x_j\|^2)^2. \tag{4.3}$$

Now that the error function SMLP is defined we need to minimize it in order to obtain the weights. The presented problem is an unconstrained minimization problem and there are plenty of suitable algorithms for solving the problem.

4.2 Implementation

The main parts of the MATLAB code implementation for MLPMDS can be found in Appendix A. Before using the MLPMDS algorithm one should be aware of what kind of structure the used MLP should have, in other words what is the size of each layer. The size of the input and the output layer is determined by the problem. The size of the input layer is the same as the dimension of the original data. The size of the output layer is the same as the target dimension. For example, if we are reducing the dimension of 10-dimensional data into two, the size of the input layer should be ten and size of the output layer should be two. The size of the hidden layer is not determined but instead should be chosen by the user or some heuristic. If the hidden layer is too small the distances are impossible to reserve and the results will be poor. However, if the chosen size of the hidden layer is too big the training will last unnecessarily long and sometimes too accurate results might be undesired. For example, when approximating a function a smoother approximation might be more useful than a highly oscillating function that actually goes through all the given points.

Let us examine how the multilayer perceptron itself is implemented in the code. I have chosen to use the logistic activation function

$$a_k(x) = \frac{1}{1+e^{-kx}} \tag{4.4}$$

and selected the constant k to equal one. Thus written in MATLAB the activation function can be expressed as

```
afun = 1./(1+exp(-x));
```

Using the algebraic definition in Equation 2.19, the multilayer perceptron is simply defined as

```
N(x)=W2*[1;afun(W1*[1;x])];
```

where $W1$ and $W2$ are the weight matrices, $afun$ is the previously described activation function and x is the network input. Notice how the biases are added on the run.

The SMLP function is minimized by using the fminunc optimization algorithm for the optimization toolbox of MATLAB. The fminunc finds the minimum of unconstrained multivariate function. The optimization algorithm could be improved by giving it the gradient of the SMLP function but unfortunately determining the gradient of the SMLP had to be left out because of the scope of the thesis.

The starting point is generated randomly with MATLAB's build-in function randn() which generates normally distributed pseudorandom numbers. It should be noted that the starting point represents the initial weights. The size of starting point is determined by the number of elements in the weight matrices. As we know the structure of the MLP, we can calculate the size of the starting point as

```
bias_size = 1; % the size of the bias is naturally one.
w1_column_size = input_layer_size+bias_size;
w1_row_size = hidden_layer_size;
w1_size = w1_column_size*w1_row_size;
w2_column_size = hidden_layer_size+bias_size;
w2_row_size = output_layer_size;
w2_size = w2_column_size*w2_row_size;
```

```
starting_point_size = w1_size + w2_size;
```

In order to carry out the optimization, a conversion from a vector to weight matrices is needed, since the answer the minimization calculates is in vector form. The problem of reshaping one vector into two weight matrices is fairly simple. First we need to split the vector into two from the right point so that we have the elements of the first weight matrix in the first part of the vector and the elements of the second weight matrix in the second part of the vector. Then we need to reshape the two new vectors into right kind of matrices based on the knowledge of the size of columns and rows in the matrices. All the needed information to perform these two steps was already calculated earlier while defining the size of the starting point.

5 Experiments

The first step of MLPMDS is to train the MLP. The requirement for training is sufficient amount of data points for input and the corresponding distances of the data points in the original space. On demand, the distances can be calculated from the data points of the original space. After the training is completed, the MLP can be given data points from the original space as an input and the perceptron transforms these points into the feature space. If the training is successful the transformation should preserve the distances as well as possible.

A randomly selected starting point always causes slightly different results, and therefore slight differences occur also in stress values. During the experiments, the training was repeated ten times for each network with the hidden layer of different size. From these ten different stress values, mean, standard deviation, and minimum value were derived and recorded into tables. The lower these values are, the better the networks performs the transformation. From the networks with the same size hidden layer, the one with the lowest stress value was selected to represent the networks with the hidden layer size in question. The fminunc optimization algorithm was applied with default options except for two stopping criteria options: maximum amount of iterations was defined as 1,000 and maximum amount of function evaluations was defined as 100,000. These values are larger than the default values to ensure better accuracy.

This chapter consists of four experiments. The first one is a simple linear transformation from three dimensions into two dimensions. The second experiment is similar to the classical city example used commonly to demonstrate MDS. The third experiment is a classification problem on the basis of the well-known Iris flower data set introduced by R.A. Fisher. The last experiment is also a classification problem but using a data set with a significantly larger amount of variables. The data set in question contains attributes of wines.

5.1 Simple linear example

In this example I simulated a transformation of the diagonal of a unit cube into two dimensional plane. The MLP was trained by taking 25 points from the diagonal of the unit cube

(Figure 7). The objective is to search for a mapping that transforms these points into the diagonal of a unit square. The training set was defined as

$$X = \begin{bmatrix} 0.04 & 0.08 & \cdots & 1.00 \\ 0.04 & 0.08 & \cdots & 1.00 \\ 0.04 & 0.08 & \cdots & 1.00 \end{bmatrix}. \tag{5.1}$$

A $25 * 25$ distance matrix is derived from X by calculating pairwise Euclidean distances:

$$D = \begin{bmatrix} 0 & 0.0693 & 0.1386 & \cdots & 1.5935 & 1.6628 \\ 0.0693 & 0 & 0.0693 & \cdots & 1.5242 & 1.5935 \\ 0.1386 & 0.0693 & 0 & \cdots & 1.4549 & 1.5242 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1.5935 & 1.5242 & 1.4549 & \cdots & 0 & 0.0693 \\ 1.6628 & 1.5935 & 1.5242 & \cdots & 0.0693 & 0 \end{bmatrix}. \tag{5.2}$$

Naturally only upper or lower triangular is needed for training.

The training was successful and the results concerning the stress are shown in Table 1. From the table it can be read that using three neurons on the hidden layer gives perhaps the best results, since the mean and standard deviation are the lowest. However the values are quite low already with hidden layer of only one neuron.

Hidden layer size	Mean of stress	STD of stress	Min value
1	6.0602e-05	5.6867e-05	1.7135e-05
2	6.7479e-06	5.2879e-06	1.1097e-07
3	3.0519e-06	3.2542e-06	1.9053e-07
4	3.9105e-06	3.3914e-06	5.845e-07
5	9.9468e-06	1.7744e-05	2.3417e-07

Table 1. Stress values for the diagonal experiment.

After training the weights the original inputs were entered to the trained network. As anticipated, the network output is not the same as the diagonal of a unit square located in origin since the transformation only preserves the distances. As the visualized results later show the rotation and location of points varies notably. MLPMDS generates the initial weights

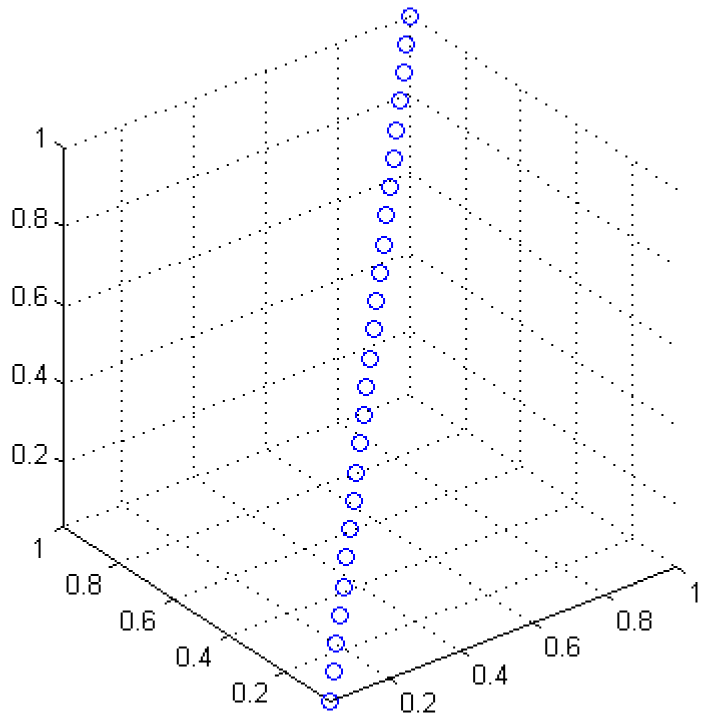

Figure 7. The original data points.

randomly so if we train the network twice with the same training data we will get different weights both times. Therefore, when looking the visualization of the results one must not concentrate on the values of location or become confused by the different rotation. Instead one should concentrate on what is substantial here: the shape that is defined by the pairwise distances between the objects.

Let us now define a new input matrix for the trained network. Let $X2$ be defined as

$$X2 = \begin{bmatrix} 0.02 & 0.06 & \cdots & 0.98 \\ 0.02 & 0.06 & \cdots & 0.98 \\ 0.02 & 0.06 & \cdots & 0.98 \end{bmatrix}. \tag{5.3}$$

In other words, $X2$ is the same as X with all its elements subtracted by 0.02. When $X2$ is given to the trained network the corresponding 2D points are obtained as an output.

The results of the dimensionality reduction can be found in Appendix C. As can be seen, the output points form a straight line and the latter points are located between the output of the training data as expected. Even when the size of the hidden layer is only one, the results seem correct, which is generally rare but understandable considering the linear nature of the problem. The impact of the stress can be seen best on the slightly curved line of Figure 13 but when taken the small intervals of the X-axis into account the error is actually insignificant. On the whole, the results seem promising so it is reasonable to carry on into the more meaningful experiments.

5.2 The map of the cities

I earlier mentioned the famous example of creating a map of cities from the distances by using MDS. The MLPMDS is not suitable for solving these kind of problems since the MLP takes data points as an input instead of the distances. In other words, the requirement already includes the answer, since the inputs for the MLP are the location points of the cities. So let us examine a situation where we are given the locations of the cities, and we conduct a transformation without actual dimensionality reduction by using the MLPMDS. The objective of this example is simply to test that the MLPMDS does not spoil the mapping in a situation where the dimension stays the same. As a matter fact, the perfect function for this situation would be the identity function.

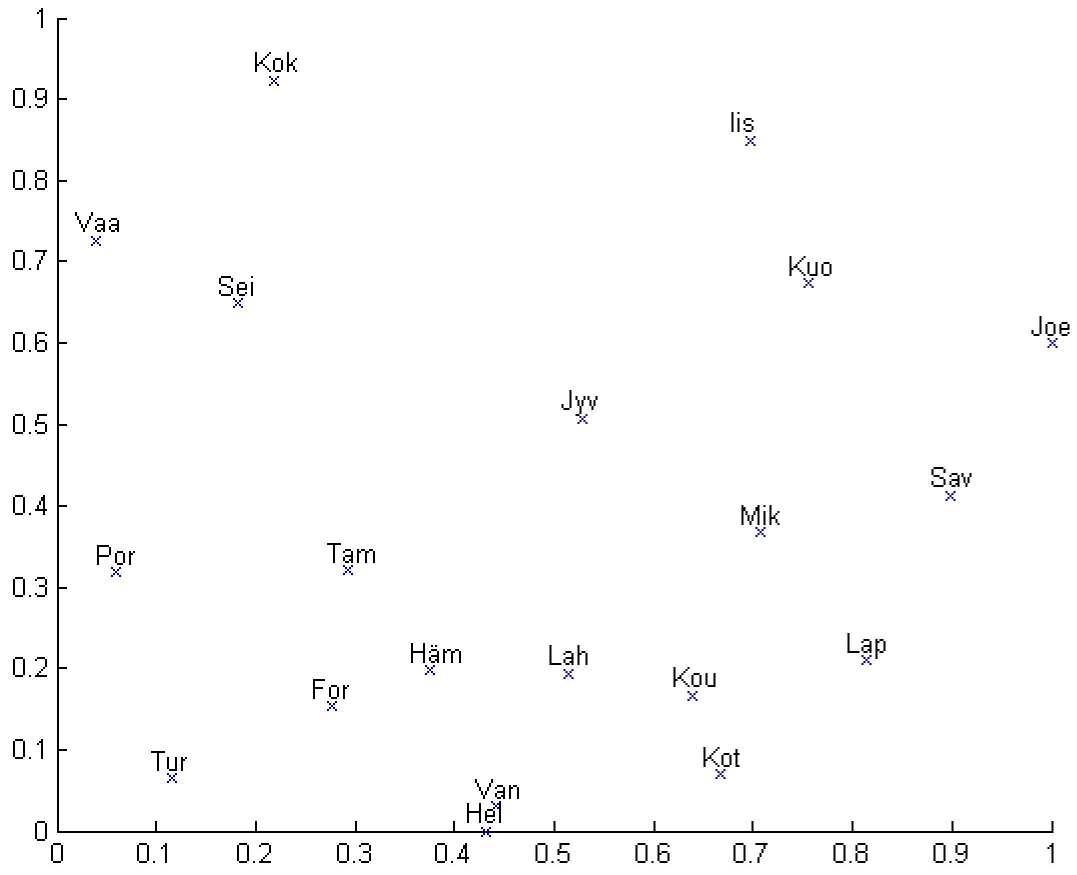

Figure 8. The original map of the cities.

The data for this example contains twenty cities from southern Finland. The data has been normalized into the interval [0,1] and can be seen in Figure 8. The full names of the cities can be found in Appendix 7. 75 percent of the data is used for the training, meaning that the training data includes locations of 15 cities. The stress values from the training can be found in Table 2. In this case, it seems that satisfactory size for the hidden layer would be two. However, if more accurate results are desired then five would be a good choice for the size of hidden layer since the mean of stress does not decrease notably with larger sizes.

After the network was trained, the training data was entered to the MLP and after that the remaining five cities were also transformed using the network. The results are illustrated

Hidden layer size	Mean of stress	STD of stress	Min value
1	5.3802	8.7582e-03	5.37
2	1.4227e-05	1.1114e-05	9.2964e-07
3	6.5883e-06	3.5766e-06	2.0881e-06
4	3.541e-06	2.2258e-06	1.3159e-08
5	1.9329e-06	1.694e-06	1.93e-07
6	1.9828e-06	1.4073e-06	4.9335e-07
7	1.3902e-06	1.4767e-06	2.5569e-07
8	2.298e-06	3.6611e-06	1.9929e-07
9	1.8806e-06	1.9753e-06	4.4519e-07
10	5.0191e-06	1.0089e-05	1.5341e-07

Table 2. The stress statistics for the city map experiment.

in Figure 9. In order to make the map more readable I have rotated and reflected it. The similarity between this and the original map is notable and the five cities not in the training data are located in their rightful places.

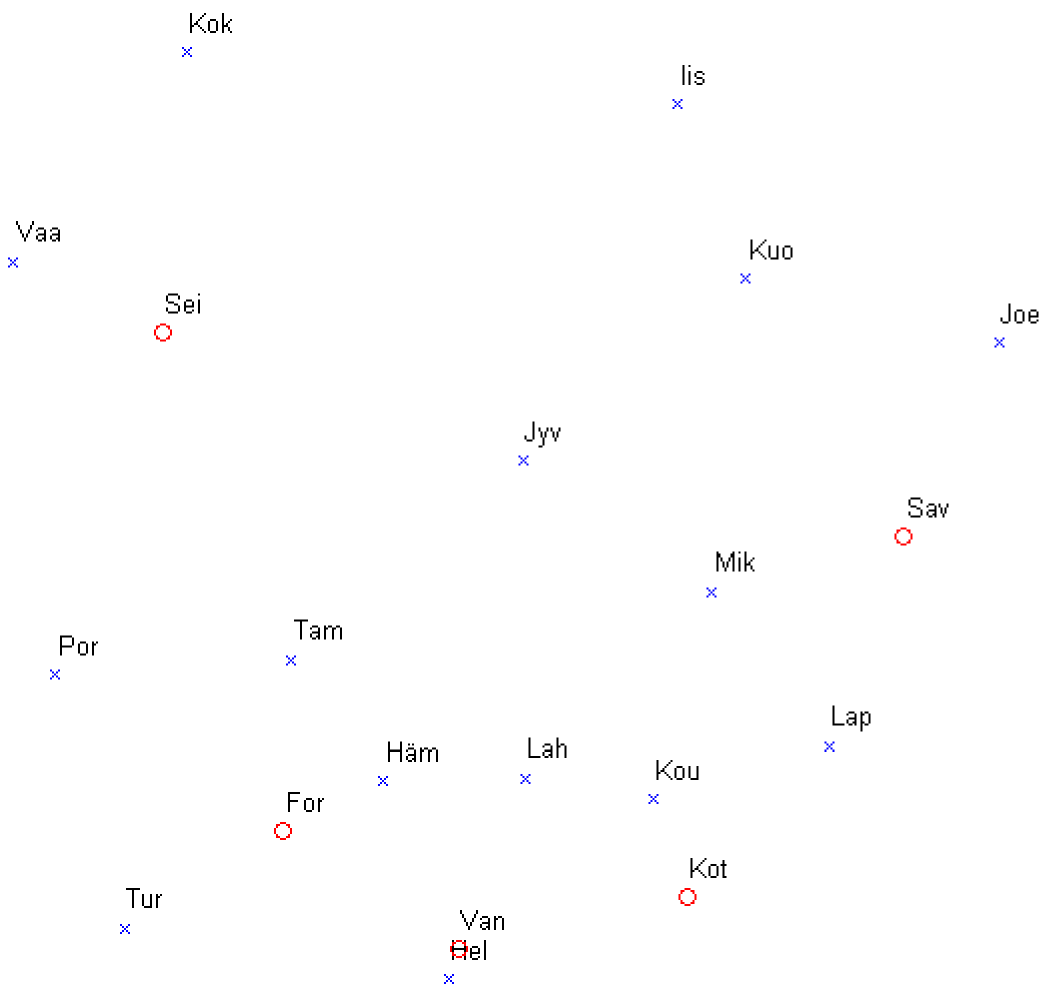

Figure 9. The map given by the MLP (the size of the hidden layer equals five). The blue crosses indicate the training data and the red circles represent the normal data.

5.3 Iris data set

The Iris data set (Fisher 1936) has four numerical variables, which are sepal length, sepal width, petal length and petal width. The data set contains 150 instances, 50 from each three classes: Iris Setosa, Iris Versicolour, and Iris Virginica, and for this example all data was normalized into the interval [0,1]. The experiment was conducted by selecting 40 first instances from each class for training purposes. The rest 30 instances were given to the trained network. The dimensionality reduction was performed separately to both 2D and 3D. The visualized results can be found in Appendix D. The stress/error values are recorded on the Tables 3 and 4. For both cases the proper hidden layer size could be four based on the low mean and standard deviation values. Note how the stress means are considerably lower in 3D case than in 2D. However, this advantage is lost in visualization since 3D data is more problematic to present and interpret.

Hidden layer size	Mean of stress	STD of stress	Min value
1	1.6693	9.7395e-10	1.6693
2	0.13757	1.4707e-03	0.13498
3	0.12577	3.7646e-03	0.12306
4	0.12135	1.7353e-03	0.11924
5	0.12198	3.27e-03	0.11944
6	0.12001	9.8907e-04	0.11886
7	0.12061	1.6251e-03	0.11771
8	0.12038	1.3797e-03	0.11856
9	0.12056	2.2095e-03	0.11767
10	0.11999	1.1805e-03	0.11782

Table 3. The stress statistics for the IRIS experiment with the target dimension of two.

When examining the visualizations in 2D case, Iris Setosa can be seen clearly apart from the other two. Versicolour and Virginica are similar to each other but they can still be rather easily distinguished from each other. The results of 3D case are very similar to ones in 2D. When using a very small hidden layer some rather interesting consequences occur. When using size one hidden layer for reducing dimension into two the network output forms a completely straight line. Also when reducing the dimension to three similar effects can also

Hidden layer size	Mean of stress	STD of stress	Min value
1	1.6693	4.9063e-10	1.6693
2	0.13626	1.5057e-03	0.13495
3	0.0073665	1.1481e-04	0.0072318
4	0.0072614	7.1095e-05	0.0071581
5	0.00716	2.109e-04	0.0068015
6	0.0071191	1.5111e-04	0.0067903
7	0.0072128	4.5503e-04	0.006496
8	0.0072896	7.6908e-04	0.006451
9	0.0070507	4.9086e-04	0.0066521
10	0.0069247	3.4617e-04	0.0065182

Table 4. The stress statistics for the IRIS experiment with the target dimension of three.

be seen. With the hidden layer size of one, the result is a straight line in 3D. With the hidden layer size of two the result is a plane. This is explained by the structure of the used MLP. When there is only one hidden layer and the final layer is linear, the hidden layer has only one neuron and all the output values $(y_1, y_2, \ldots, y_n)$ are form $xw_i + w_0$, where x is the output of the neuron in hidden layer, $w_i, 1 \leq i \leq n$ are the weights and w_0 is the weight of the bias term. The expression clearly depicts a straight line.

5.4 Wine data set

The wine data set (Forina) has thirteen numerical variables: alcohol, malic acid, ash, alcalinity of ash, magnesium, total phenols, flavanoids, nonflavanoid phenols, proanthocyanins, color intensity, hue, OD280/OD315 of diluted wines, and proline. The data set contains 178 instances from three different classes. The classification is made based on the type of wine. All wines were produced in the same region but by using different varieties.

The experiment was conducted by selecting 80 percent of instances from each class for training purposes. The rest of the instances were given to the trained network. The dimensionality reduction was performed separately to both 2D and 3D. The visualized results can be found in Appendix E. The stress/error values are recorded on Tables 5 and 6.

Hidden layer size	Mean of stress	STD of stress	Min value
1	8.7584e-04	4.9193e-05	0.00081625
2	6.3268e-04	3.0801e-04	5.9816e-05
3	1.8327e-04	3.1107e-04	6.5592e-06
4	3.5029e-04	3.952e-04	2.5869e-05
5	3.3623e-04	3.4119e-04	3.3134e-05
6	1.37e-04	2.2334e-04	4.2842e-05
7	8.4189e-05	6.7509e-05	3.7204e-05
8	7.9659e-05	9.1561e-05	1.139e-05
9	8.4805e-05	4.2011e-05	3.8311e-05
10	7.8743e-05	6.9795e-05	1.6333e-05

Table 5. The stress statistics for the wine experiment with the target dimension of two.

Hidden layer size	Mean of stress	STD of stress	Min value
1	8.887e-04	8.4645e-05	7.5544e-04
2	6.8823e-04	1.9453e-04	1.4067e-04
3	1.3258e-04	2.4124e-04	2.0969e-05
4	7.6814e-05	3.9921e-05	1.7305e-05
5	6.4468e-05	4.3982e-05	1.4523e-05
6	7.4839e-05	4.0398e-05	4.1283e-05
7	5.0048e-04	7.7793e-04	5.0193e-05
8	7.1945e-05	2.6e-05	3.8054e-05
9	1.9981e-04	2.2652e-04	2.5415e-05
10	2.0486e-04	1.7884e-04	1.6445e-05

Table 6. The stress statistics for the wine experiment with the target dimension of three.

When inspecting the stress values it seems that in 2D case the hidden layer size of seven is sufficient, as there is no significant change in the stress values after that. In 3D case the sufficient size for the hidden layer could be four. With hidden layer size of seven the mean of stress seems notably high but also the standard deviation is great and it is therefore likely that there are one or more exceptional cases which cause abnormality to the mean. In addition, the minimum value seems reasonable.

The wine data set is usually used by choosing only few variables at a time for visualization so that the differences between the classes might be clear to see. However, in this case where we try to visualize thirteen variables in 2D it is only anticipated that results do not distinguish the classes clearly. Fortunately the results show some kind of concentrations instead of total chaos. Similarly to the iris data set, the first class stands out from the other two.

6 Conclusions

The experiments give some proof that the MLPMDS does indeed achieve its goals. Nevertheless, the algorithm needs to be further examined and verified. The four experiments in the thesis do not give accurate results on the trustworthiness of the algorithm. In addition, extensive comparison between MLPMDS and other dimensionality reduction techniques would be crucial.

When using the MLPMDS, the most important option is probably the selection of hidden layer size, since it is one of those attributes that the method does not automatically determine. According to the experiments the size of input (original dimension) and output (target dimension) layers have the greatest influence on the selection. It would seem that the size of the hidden layer does not need to be any greater than the number of variables in the data. In the wine experiment, where the variable count was the highest, it was highly appropriate to choose a hidden layer size much lower than the original dimension. Another thing that the user must decide is the size of training data. The sufficient size for the training data depends on the problem and the data. Presumably the same instructions about the size of the training data, that apply generally to the training of MLP, hold in this case too. The MLPMDS uses the logistic function as an activation function. The parameter k was simply chosen as one but to be accurate a more proper value could be chosen. In addition, in some cases a different function, such as the hyperbolic tangent function, might give better results.

One of the disadvantages for using neural network is that a trained network cannot be trained with new data without starting the whole process from the beginning. If new instances are inserted to training data after the training, the whole training has to be done all over again in order to utilize all of the training data. The training of the MLP in MLPMDS is very time-consuming because the optimization problem is highly complicated. The optimization would be greatly improved if the gradient of the loss function was given. For further studies determining the gradient would be important and fortunately it can be calculated analytically. The accuracy of the mean of stress and other stress related values could be improved by doing repeating training more than ten times per hidden layer size.

The promising characteristics of the MLPMDS are that it is a fairly simple and logical method, and once the MLP is trained the dimensionality reduction is done very fast and effectively. Nevertheless, a great deal of further research is needed to better evaluate the value of the MLPMDS and to improve its faults. I would be curious to see some more research on the subject in the future.

Bibliography

Abbas, Harzem M, and Moustafa M Fahmy. 1992. “A neural model for adaptive Karhunen Loeve transformation (KLT)”. In *Neural Networks, 1992. IJCNN., International Joint Conference on,* 2:975–980. IEEE. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=226861.

Aleksander, I., and H. Morton. 1990. *An introduction to neural computing.* Chapman / Hall London.

Baldi, Pierre, and Kurt Hornik. 1989. “Neural networks and principal component analysis: Learning from examples without local minima”. *Neural networks* 2 (1): 53–58.

Bishop, C.M. 1995. *Neural networks for pattern recognition.* Oxford University Press.

Chatfield, C., and A. Collins. 1980. *Introduction to multivariate analysis.* Chapman & Hall/CRC.

Cox, T.F., and M.A.A. Cox. 2001. *Multidimensional scaling.* Chapman & Hall/CRC.

Cybenko, G. 1989. “Approximation by superpositions of a sigmoidal function”. *Mathematics of Control, Signals, and Systems (MCSS)* 2 (4): 303–314. http://www.ise.ncsu.edu/fangroup/ie789.dir/Cybenko.pdf.

Fausett, L. 1994. *Fundamentals of neural networks: Architectures, algorithms, and applications.* Prentice Hall.

Fisher, R.A. 1936. *Iris data set, UCI Machine Learning Repository.* http://archive.ics.uci.edu/ml/datasets/Iris.

Forina, M. et al. *Wine data set, UCI Machine Learning Repository.* http://archive.ics.uci.edu/ml/datasets/Wine.

Hand, D.J., H. Mannila, and P. Smyth. 2001. *Principles of data mining.* MIT Press.

Haykin, S. 1999. *Neural networks: A comprehensive foundation.* 2nd edition. Upper Saddle River, NJ: Prentice Hall.

Hotelling, H. 1933a. “Analysis of a complex of statistical variables into principal components”. *Journal of educational psychology* 24 (6): 417.

———. 1933b. “Analysis of a complex of statistical variables into principal components”. *Journal of Educational Psychology* 24 (7): 498–520.

Kärkkäinen, T. 2000. “MLP-network in a Layer-Wise Form: Derivations, Consequences and Applications to Weight Decay”.

———. 2002. “MLP in Layer-Wise Form with Applications to Weight Decay”. *Neural computation* 14 (6): 1451–1480. http://www.mitpressjournals.org/doi/abs/10.1162/089976602753713016.

Kohonen, Teuvo. 1982. “Self-organized formation of topologically correct feature maps”. *Biological cybernetics* 43 (1): 59–69.

Kruskal, J.B. 1964. “Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis”. *Psychometrika* 29 (1): 1–27. http://link.springer.com/article/10.1007/BF02289565.

Kruskal, JB, and Wish. M. 1978. *Multidimensional Scaling.* Sage Publications.

Lowe, D., and M. Tipping. 1996. “Feed-forward neural networks and topographic mappings for exploratory data analysis”. *Neural Computing & Applications* 4 (2): 83–95. http://eprints.aston.ac.uk/666/1/misc96-008.pdf.

Maaten, L. van der, E. Postma, and J. van den Herik. 2009. “Dimensionality Reduction: A Comparative Review”. *Journal of Machine Learning Research* 10:1–41. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.125.6716&rep=rep1&type=pdf.

Maaten, L.J.P. van der. *Matlab Toolbox for Dimensionality Reduction (v0.8 - April 2012).* http://homepage.tudelft.nl/19j49/Matlab_Toolbox_for_Dimensionality_Reduction.html.

Mardia, K.V., J.T. Kent, and J.M. Bibby. 1980. *Multivariate analysis.* London: Academic press.

Mathworks. 2013a. *Classical multidimensional scaling - MATLAB - MathWorks Nordic.* http://www.mathworks.se/help/stats/cmdscale.html.

———. 2013b. *Nonclassical multidimensional scaling - MATLAB - MathWorks Nordic.* http://www.mathworks.se/help/stats/mdscale.html.

McCulloch, W.S., and W. Pitts. 1943. "A logical calculus of the ideas immanent in nervous activity". *Bulletin of mathematical biology* 5 (4): 115–133. http://cns-classes.bu.edu/cn550/Readings/mcculloch-pitts-43.pdf.

Oja, Erkki. 1992. "Principal components, minor components, and linear neural networks". *Neural Networks* 5 (6): 927–935. http://users.ics.aalto.fi/oja/Oja92.pdf.

Pearson, K. 1901. "On lines and planes of closest fit to systems of points in space". *Philosophical Magazine and Journal of Science* 2 (11): 559–572.

Rumelhart, D.E., G.E. Hinton, and R.J. Williams. 1986. "Learning representations by back-propagating errors". *Nature* 323 (6088): 533–536. http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf.

Sammon Jr, J.W. 1969. "A nonlinear mapping for data structure analysis". *Computers, IEEE Transactions on* 100 (5): 401–409. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1671271&tag=1.

Schölkopf, B., A. Smola, and K.R. Müller. 1998. "Nonlinear component analysis as a kernel eigenvalue problem". *Neural computation* 10 (5): 1299–1319. http://web.ebscohost.com/ehost/pdfviewer/pdfviewer?sid=2bab5e15-6a03-411f-a6bb-4b7d6db49d21@sessionmgr115&vid=2&hid=110.

Seber, G.A.F. 1984. *Multivariate Observations.* John Wiley & Sons.

Shavlik, Jude W, and Thomas Glen Dietterich. 1990. *Readings in machine learning.* Morgan Kaufmann.

Shepard, R.N. 1962a. "The analysis of proximities: Multidimensional scaling with an unknown distance function. I." *Psychometrika* 27 (2): 125–140. http://link.springer.com/article/10.1007/BF02289630?LI=true.

Shepard, R.N. 1962b. “The analysis of proximities: Multidimensional scaling with an unknown distance function. II”. *Psychometrika* 27 (3): 219–246. http://link.springer.com/article/10.1007/BF02289621?LI=true.

Takane, Y., F.W. Young, and J. De Leeuw. 1977. “Nonmetric individual differences multidimensional scaling: an alternating least squares method with optimal scaling features”. *Psychometrika* 42 (1): 7–67. http://takane.brinkster.net/yoshio/p007.pdf.

Torgerson, W.S. 1952. “Multidimensional scaling: I. Theory and method”. *Psychometrika* 17 (4): 401–419.

Webb, Andrew R. 1995. “Multidimensional scaling by iterative majorization using radial basis functions”. *Pattern Recognition* 28 (5): 753–759. http://www.sciencedirect.com/science/article/pii/0031320394001359.

Wezel, M. van, W. Kosters, P. Van Der Putten, and J. Kok. 2001. “Nonmetric multidimensional scaling with neural networks”. *Advances in Intelligent Data Analysis* 2189:145–155. http://link.springer.com/chapter/10.1007/3-540-44816-0_15?LI=true.

Young, G., and A.S. Householder. 1938. “Discussion of a set of points in terms of their mutual distances”. *Psychometrika* 3 (1): 19–22. http://www.galileoco.com/literature/youngHouseholder38.pdf.

Appendices

A Code implementation

```
% Trains MLP by defining the weights. Returns the
% weights in a vector.
% x - original data points
% layers - sizes of MLP layers (1X3 vector)
% min_options - options for optimization
% a0 - starting point for optimization
function v = train_mlp(x,layers, min_options, a0)
  if ~exist('a0', 'var') || isempty(a0)
a0=generate_startingpoint(layers);
  end
  orig_dist = pdist(x','euclidean');
  fun = @(weights)(lossfun(x,weights,layers,orig_dist));
  v = fminunc(fun,a0,min_options);
end

% Generates a starting vector of pseudorandom numbers
% (from the normal distribution) based on the network
% size information.
% layers - sizes of MLP layers (1X3 vector)
function y = generate_startingpoint(layers)

  input_layer = layers(1);
  hidden_layer = layers(2);
  output_layer = layers(3);
  bias = 1;

  size1 = (input_layer+bias)*hidden_layer;
  size2 = (hidden_layer+bias)*output_layer;
  y=randn(1, size1+size2);
end

% The loss function (SMLP). Returns the total stress.
% X_mat - network inputs (matrix)
% w - weights
% layers - sizes of MLP layers (1X3 vector)
% orig_dist - original distances
function stress = lossfun(X_mat,w,layers,orig_dist)
  % Defines distance vectors
  nn_output_dist = nn_dist( X_mat, w, layers);
  % Validates distance vectors
  if (length(nn_output_dist)~=length(orig_dist))
   error('Validate:dist_vectors', ...
       'Matrix sizes do not match.');
  end

  stress = 0;
  for i=1:length(nn_output_dist);
   stress = stress + (nn_output_dist(i)^2-orig_dist(i)^2)^2;
  end
end

% Calculates the distances of mlp outputs
% X_mat - network inputs (matrix)
% w - weights
% layers - sizes of MLP layers (1X3 vector)
function dist = nn_dist( X_mat, w, layers)
  n = length(X_mat);
  A = zeros(layers(3),n);
  for i=1:n
A(:,i) = mlp(X_mat(:,i),w,layers);
  end
  dist = pdist(A','euclidean');
end

% Multilayer perceptron
% x - network input
% w - weights reshaped as a vector
% layers - sizes of MLP layers (1X3 vector)
function y = mlp( x, w, layers)
 validate_nn_parameters(layers);
 [W1 W2] = vector2weights(w, layers);
 y=W2*[1;afun(W1*[1;x])];
end

% The logistic activation function
function y = afun(x)
  y = 1./(1+exp(-x)); % k=1
end

% Generates weightmatrices from given vector.
%   v - vector to be reshaped
% layers - sizes of layers (1x3 vector)
function [w1 w2] = vector2weights(v, layers)
  validate_nn_parameters(layers);

  % Initialize layer size variables
  input_layer = layers(1);
  hidden_layer = layers(2);
  output_layer = layers(3);
  bias = 1;

  % Validate parameters
  nn_tot_size = (input_layer+bias)*hidden_layer+...
    (hidden_layer+bias)*output_layer;
  if (nn_tot_size ~= length(v))
   error('Vector size does not match with the neural network.')
  end

  % Split vector into two
  v1_end=(input_layer+bias)*hidden_layer;
  v1=v(1:v1_end);
  v2=v(v1_end+1:length(v));

  % Reshape vectors into matrices
  w1 = reshape(v1, hidden_layer, input_layer+bias);
  w2 = reshape(v2, output_layer, hidden_layer+bias);
end

% Runs multiple inputs through MLP.
% Returns outputs as a matrix.
% x_mat - network inputs (matrix)
% w - weights
% layers - sizes of MLP layers (1X3 vector)
function Y = multi_mlp( x_mat, w, neurons)
  n = length(x_mat(1,:));
  Y = zeros(neurons(3),n);
  for i = 1:n;
Y(:,i)= mlp(x_mat(:,i),w,neurons);
  end
end
```

B Abbreviations of cities

For	Forssa
Hel	Helsinki
Häm	Hämeenlinna
Iis	Iisalmi
Joe	Joensuu
Jyv	Jyväskylä
Kok	Kokkola
Kot	Kotka
Kou	Kouvola
Kuo	Kuopio
Lah	Lahti
Lap	Lappeenranta
Mik	Mikkeli
Por	Pori
Sav	Savonlinna
Sei	Seinäjoki
Tam	Tampere
Tur	Turku
Vaa	Vaasa
Van	Vantaa

Table 7. Abbreviations of cities

C Simple linear example results

Here are the visualized results of the experiment conducted on the diagonal of the unit cube. ’x’-marks indicate the network output with the data used for training and ’o’-marks indicate the network output with the data that was outside the training set.

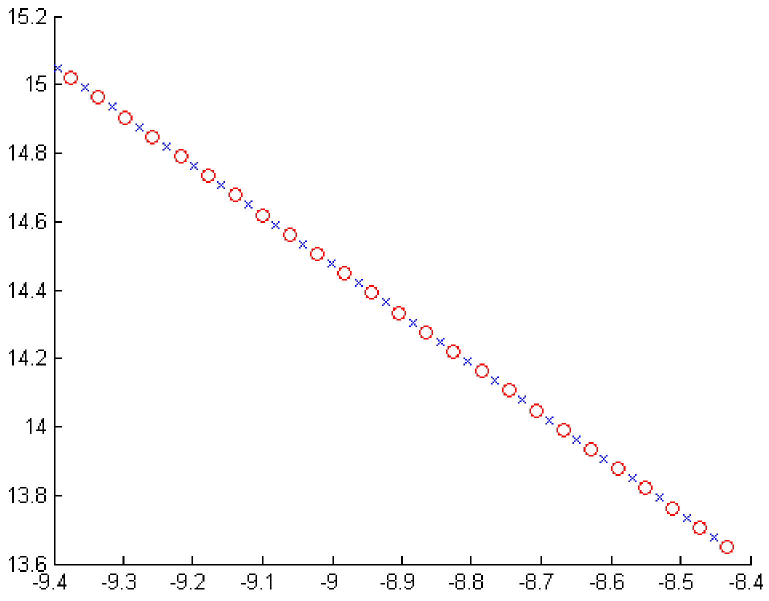

Figure 10. The output of the MLP, hidden layer size equals one

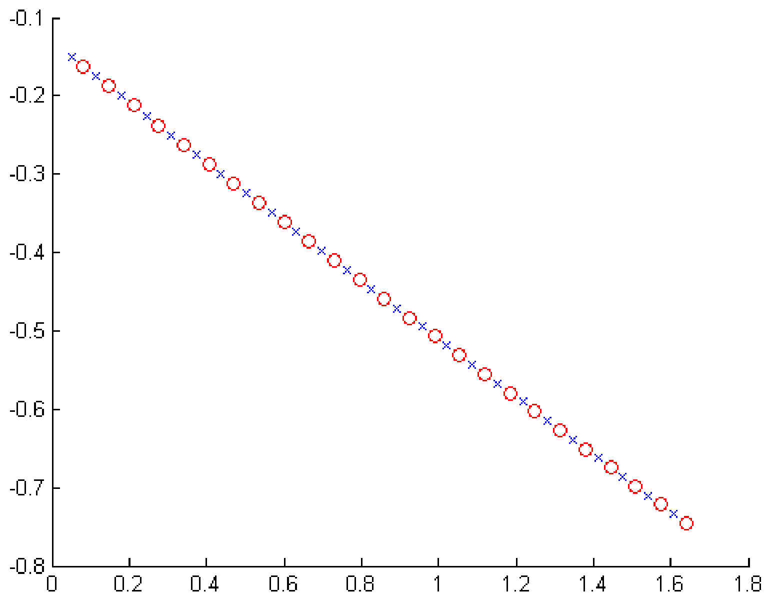

Figure 11. The output of the MLP, hidden layer size equals two

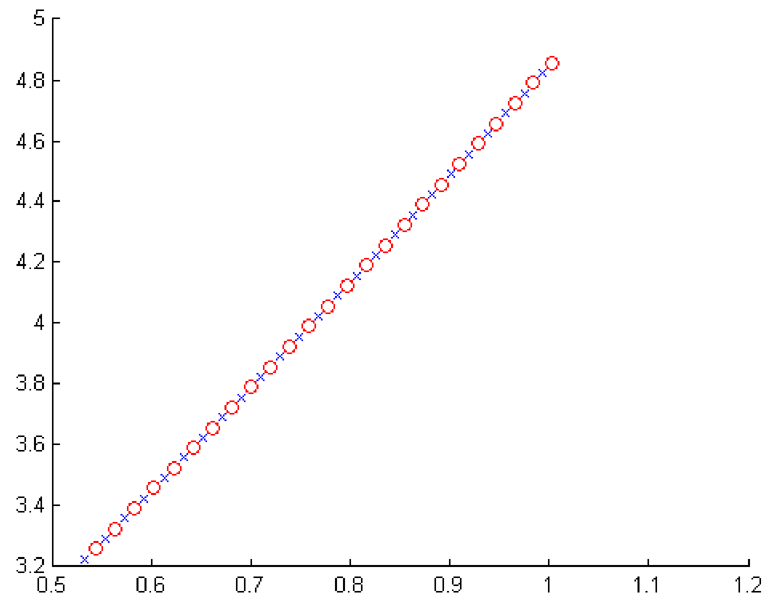

Figure 12. The output of the MLP, hidden layer size equals three

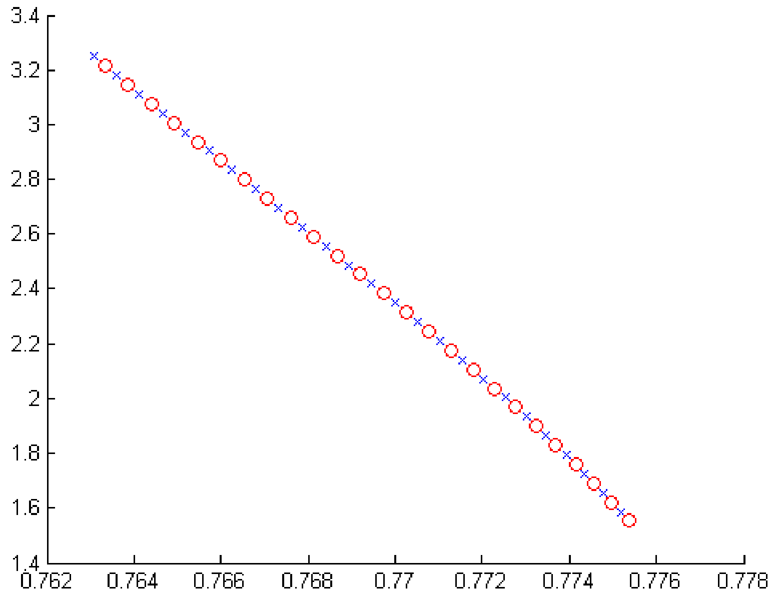

Figure 13. The output of the MLP, hidden layer size equals four

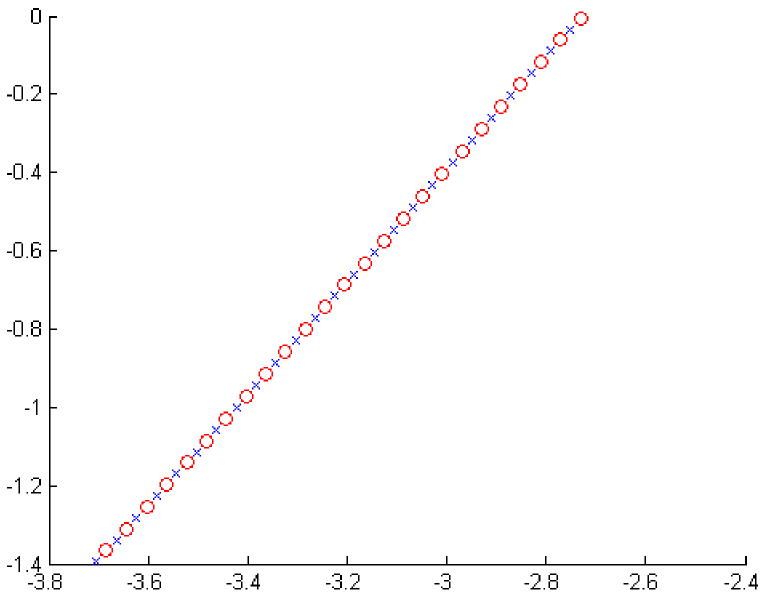

Figure 14. The output of the MLP, hidden layer size equals five

D Iris data set results

Here are the visualized results of the experiment conducted on the Iris data set. The experiment was repeated with different size hidden layer and with both 2D and 3D target space. ’x’-marks indicate the network output with the data used for training and ’o’-marks indicate the network output with the data that was left out from the training. The color indicates the class of the flower (red = Iris Setosa, blue = Iris Versicolour, green = Iris Virginica).

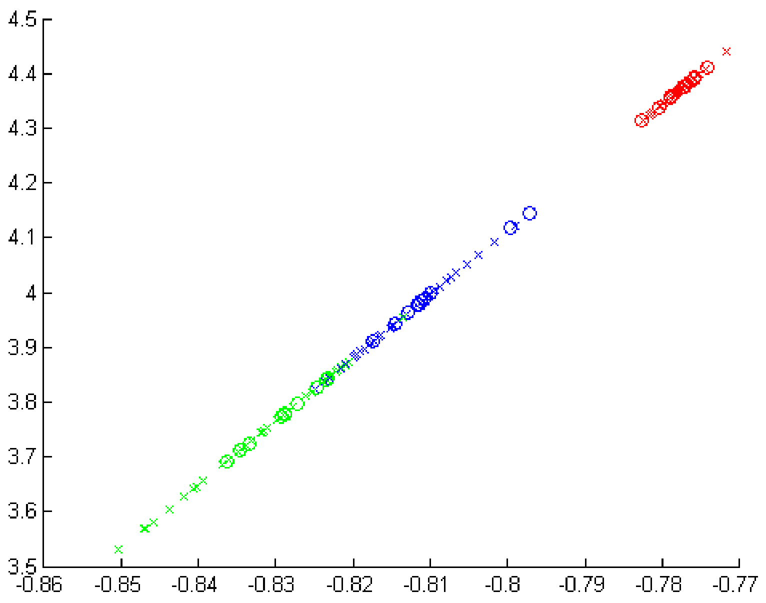

Figure 15. The output of the MLP, hidden layer size equals one (reduction into 2D).

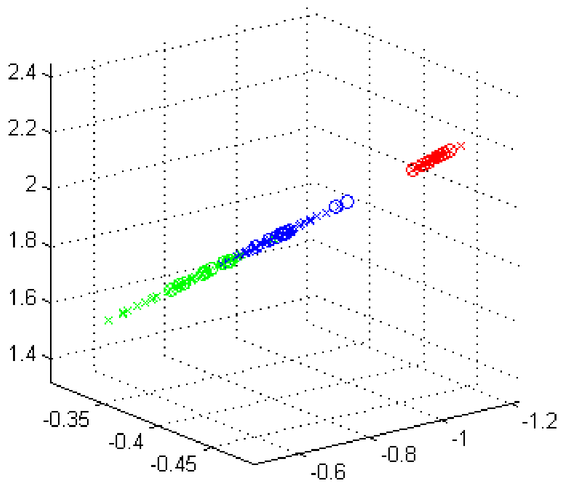

Figure 16. The output of the MLP, hidden layer size equals one (reduction into 3D).

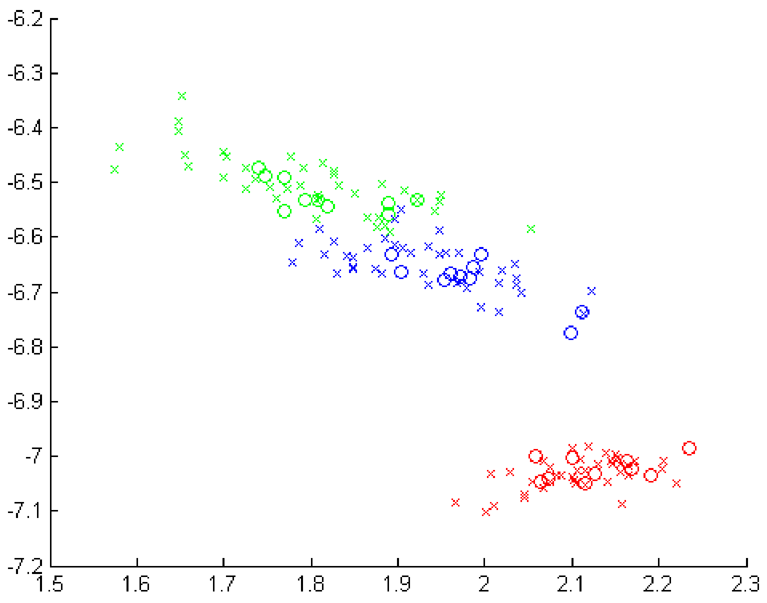

Figure 17. The output of the MLP, hidden layer size equals two (2D).

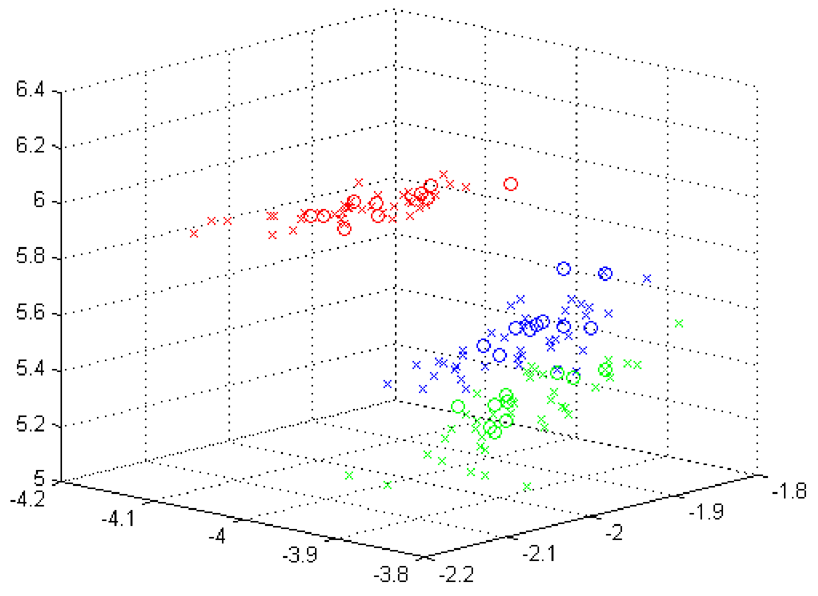

Figure 18. The output of the MLP, hidden layer size equals two (3D).

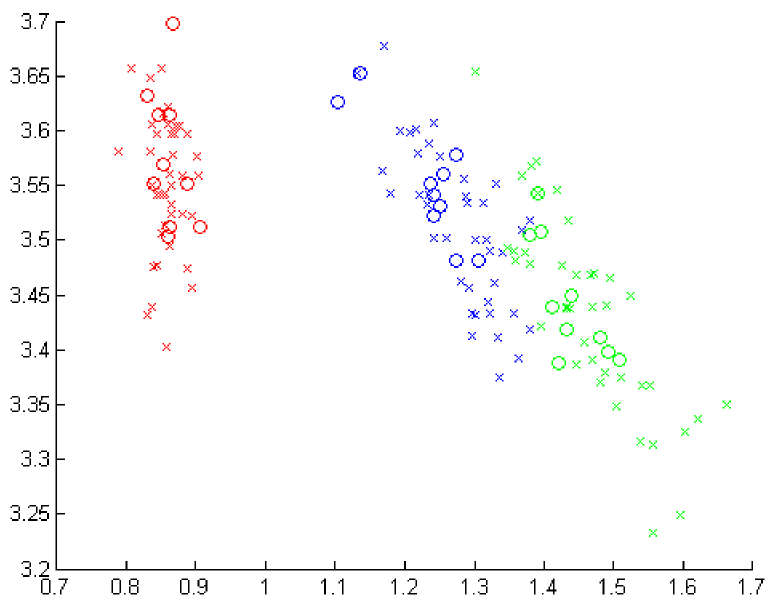

Figure 19. The output of the MLP, hidden layer size equals five (2D).

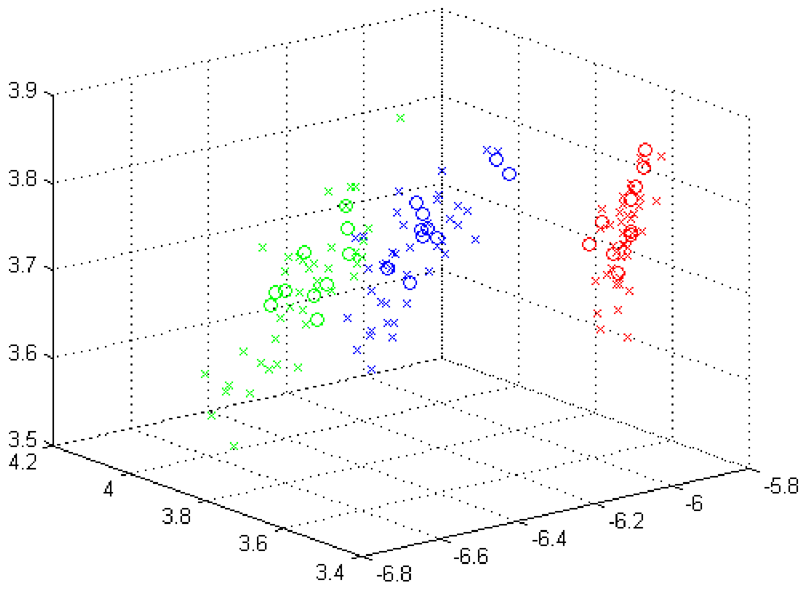

Figure 20. The output of the MLP, hidden layer size equals five (3D).

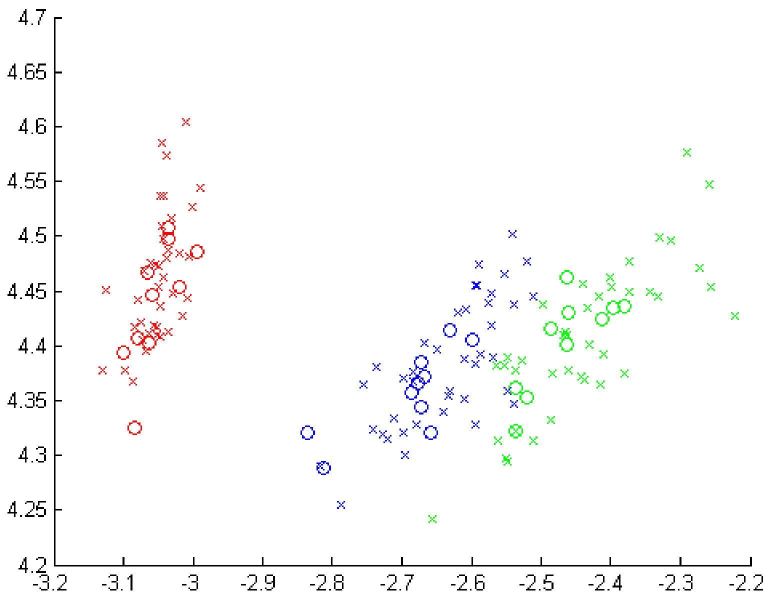

Figure 21. The output of the MLP, hidden layer size equals ten (2D).

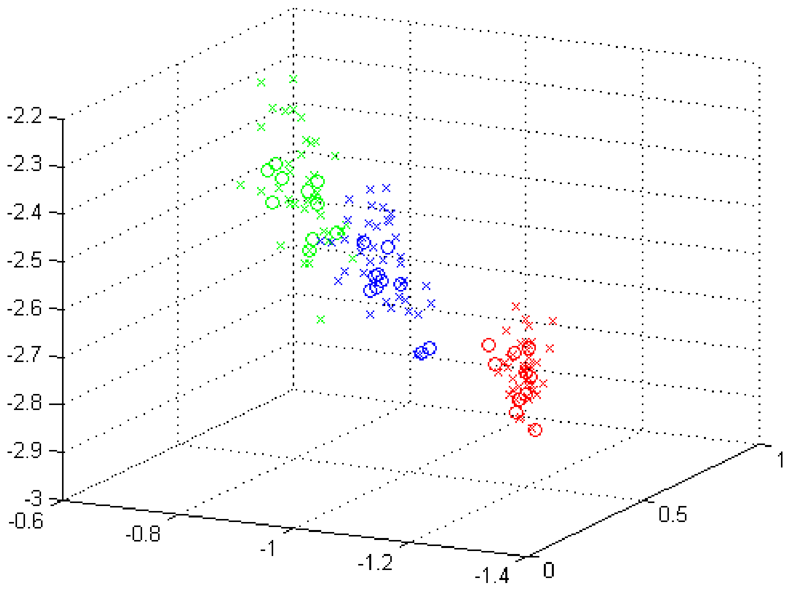

Figure 22. The output of the MLP, hidden layer size equals ten (3D).

E Wine data set results

Here are the visualized results of the experiment conducted on the wine data set. The experiment was repeated with different size hidden layer and with both 2D and 3D target space. ’x’-marks indicate the network output with the data used for training and ’o’-marks indicate the network output with the data that was left out from the training. The color indicates the class of the wine (red = type 1, blue = type 2, green = type 3).

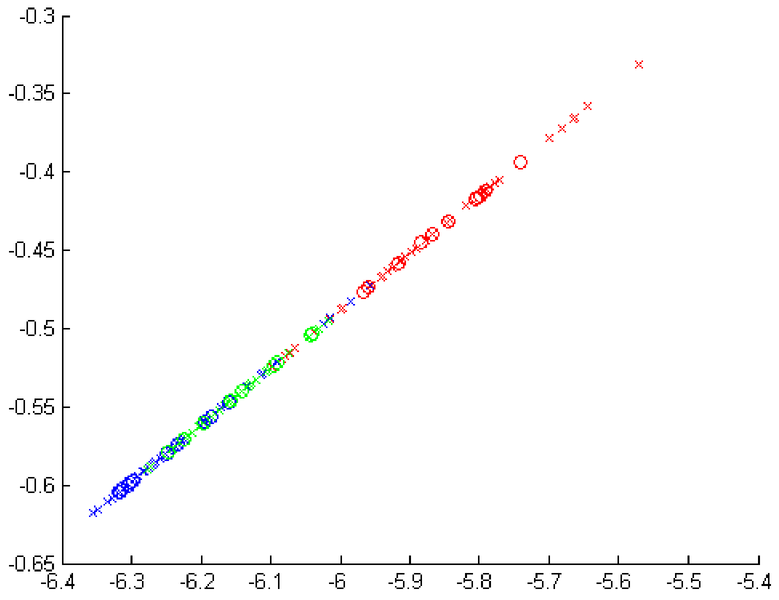

Figure 23. The output of the MLP, hidden layer size equals one (reduction into 2D).

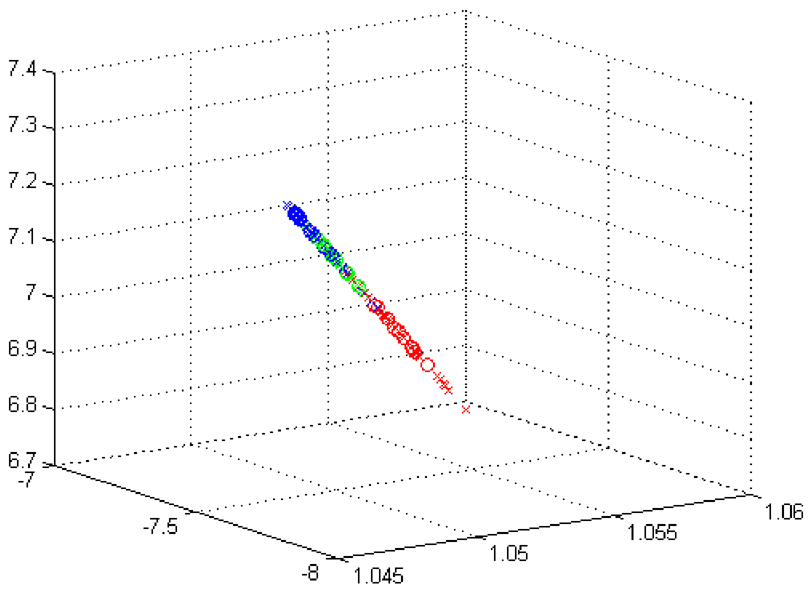

Figure 24. The output of the MLP, hidden layer size equals one (reduction into 3D).

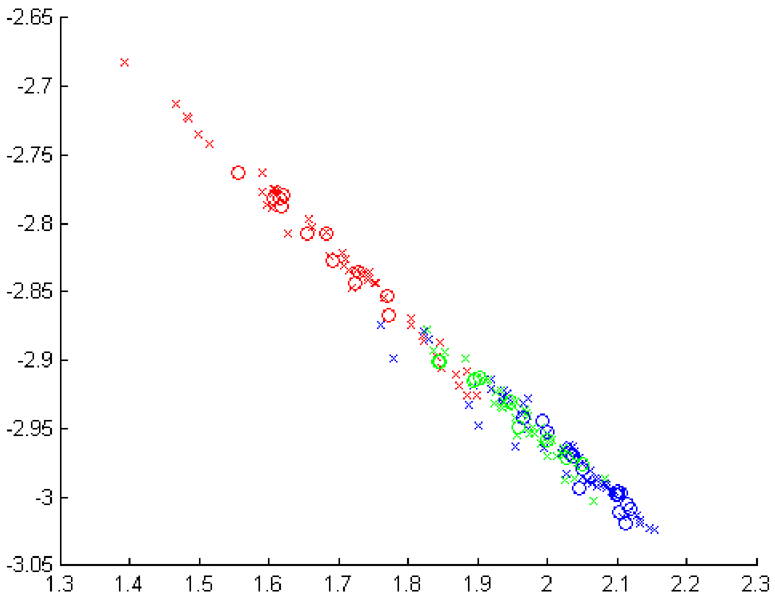

Figure 25. The output of the MLP, hidden layer size equals two (2D).

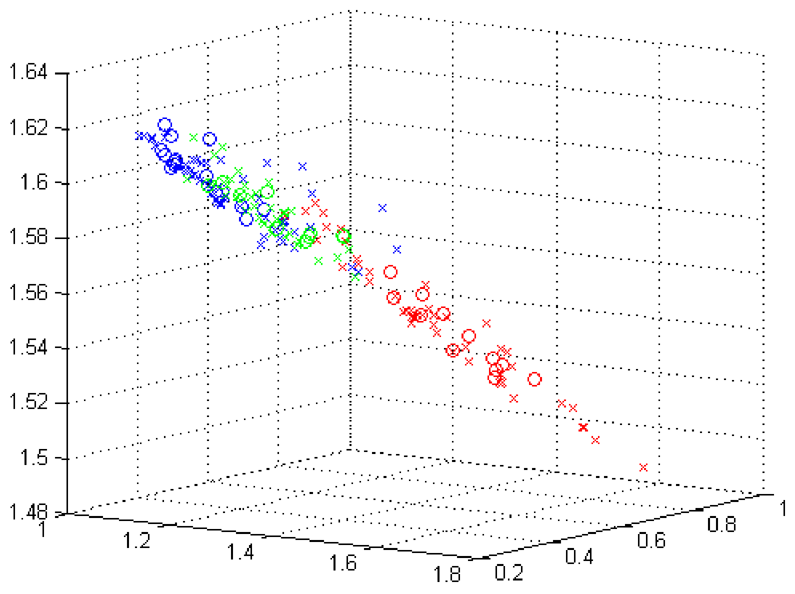

Figure 26. The output of the MLP, hidden layer size equals two (3D).

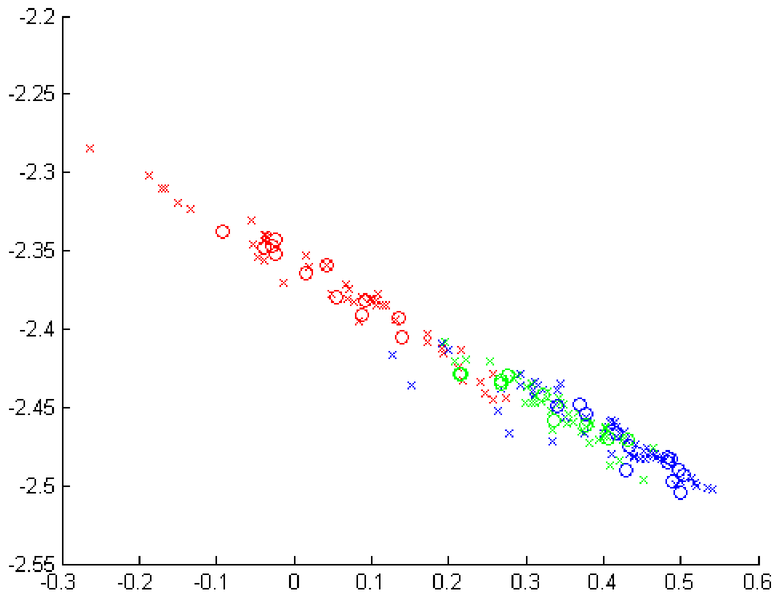

Figure 27. The output of the MLP, hidden layer size equals five (2D).

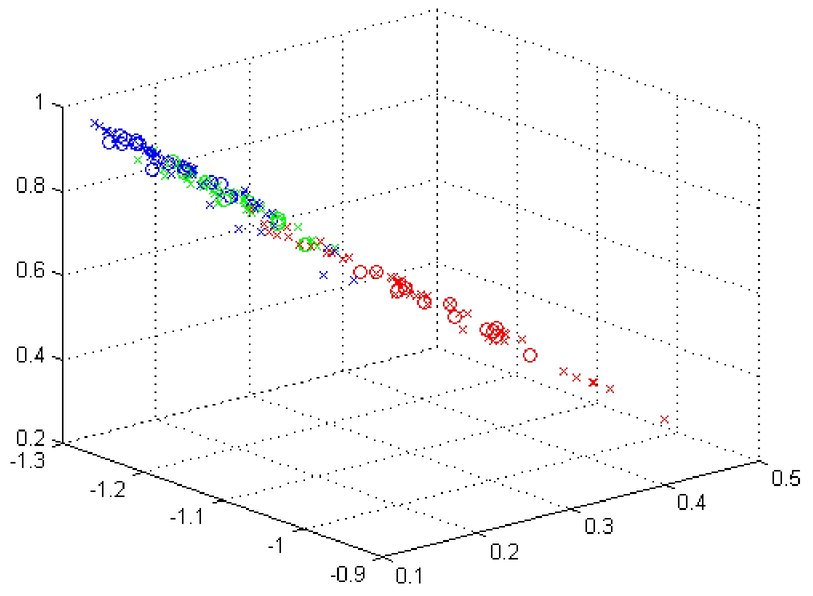

Figure 28. The output of the MLP, hidden layer size equals five (3D).

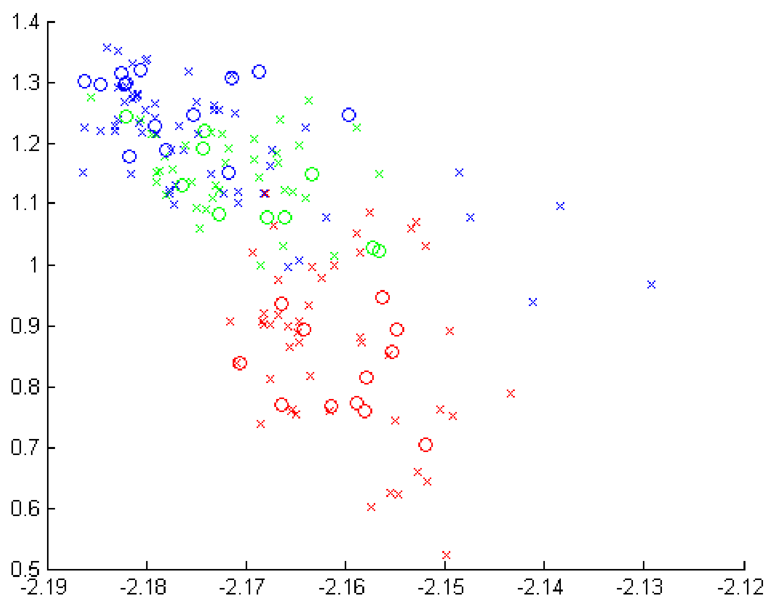

Figure 29. The output of the MLP, hidden layer size equals ten (2D).

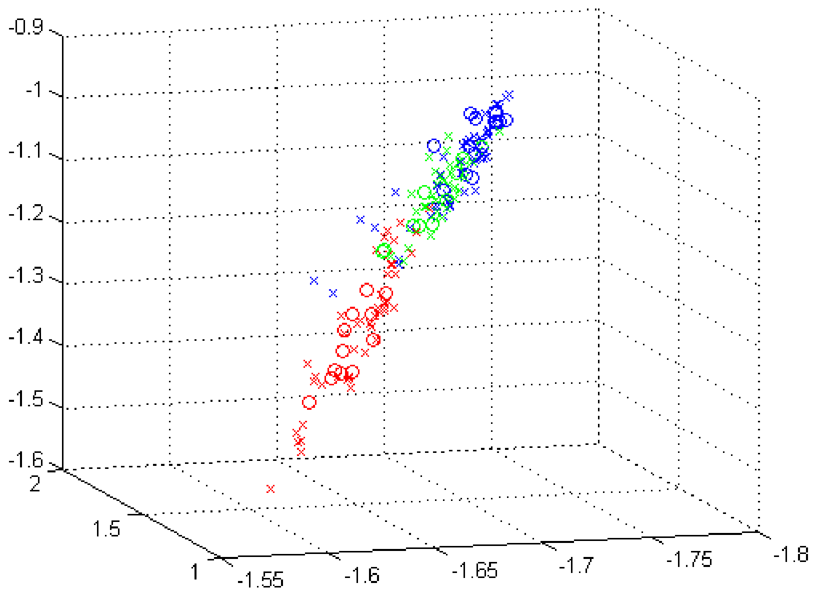

Figure 30. The output of the MLP, hidden layer size equals ten (3D).