

# Speeding up Ozone Profile Retrieval using Machine Learning Techniques

L.M. Strijbosch

April 25, 2007



# Abstract

This thesis describes the application of machine learning techniques to the process of ozone profile retrieval, a method for retrieving a global ozone distribution from satellite measurements. The goal of this research is to investigate the possibilities of speeding up this procedure, because the lack of speed is the main drawback for real application. Neural networks and Support Vector Machines are selected to replace the forward model, which is the slowest part of ozone profile retrieval. Replacement of the forward model involves the learning of the standard mapping and the Jacobian. Alternative techniques such as Principle Component Analysis and Ensembles are evaluated to see if the results can be improved. Finally, the best performing models are integrated in the ozone profile retrieval procedure and tested on synthetic measurements. The results provide good insight in the usability of the selected techniques for application on real measurements.

**Author** L.M. Strijbosch

**Advisor** Prof. dr. J.N. Kok



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Ozone profile retrieval</b>	<b>3</b>
2.1	Satellite measurements of vertical ozone profiles . . . . .	3
2.2	The retrieval process . . . . .	4
2.3	Summary . . . . .	6
<b>3</b>	<b>Problem analysis</b>	<b>7</b>
3.1	Replacement of the forward model . . . . .	7
3.2	The training set . . . . .	7
3.3	Selected techniques . . . . .	8
3.4	Comparison to related research . . . . .	9
3.5	Conclusion . . . . .	10
<b>4</b>	<b>Neural network approach</b>	<b>11</b>
4.1	Notation and terminology . . . . .	11
4.2	Performance measurement . . . . .	12
4.3	Normalisation . . . . .	13
4.4	Network architecture . . . . .	15
4.5	Learning algorithm . . . . .	16
4.6	Conclusion . . . . .	20
<b>5</b>	<b>Support vector machine approach</b>	<b>23</b>
5.1	Support vector machines . . . . .	23
5.2	Support vector regression . . . . .	25
5.3	Test results . . . . .	27
5.4	Conclusion . . . . .	28

<b>6</b>	<b>The Jacobian</b>	<b>29</b>
6.1	Jacobian derivation . . . . .	29
6.2	Denormalisation . . . . .	30
6.3	Results . . . . .	31
6.4	Learning the Jacobian . . . . .	32
6.5	Conclusion . . . . .	33
<b>7</b>	<b>Dimensionality reduction</b>	<b>35</b>
7.1	The curse of dimensionality . . . . .	35
7.2	Principle component analysis . . . . .	35
7.3	Conclusion . . . . .	37
<b>8</b>	<b>Ensemble approach</b>	<b>39</b>
8.1	Bias / variance decomposition . . . . .	39
8.2	A crosstraining ensemble . . . . .	40
8.3	Conclusion . . . . .	42
<b>9</b>	<b>Simulation on synthetic measurements</b>	<b>43</b>
9.1	Interpretation of the results . . . . .	43
9.2	Conclusion . . . . .	46
<b>10</b>	<b>Conclusion</b>	<b>47</b>
<b>A</b>	<b>Neural network experiments</b>	<b>49</b>
A.1	Normalisation . . . . .	49
A.2	Hidden layer(s) . . . . .	49
<b>B</b>	<b>Support vector regression experiments</b>	<b>51</b>
B.1	$\epsilon$ -svr . . . . .	51
B.2	$\nu$ -svr . . . . .	53
<b>C</b>	<b>Matlab source code</b>	<b>55</b>
C.1	Jacobian derivation algorithm . . . . .	55
C.2	Ensemble meta method . . . . .	56
C.3	Export network . . . . .	57
<b>D</b>	<b>Fortran source code</b>	<b>61</b>
	<b>Bibliography</b>	<b>69</b>





# Chapter 1

## Introduction

---

The field of machine learning includes algorithms that are able to learn implicit or explicit relationships from a set of data. Clustering, classification and function approximation problems are fields of application for these algorithms. Although there are software packages available with many ready to use implementations, seldom they can be applied without work to reach their optimal configuration: each individual problem has its own characteristics and problem domain knowledge combined with a good level of machine learning expertise is required to lead to satisfying results. Such work involves investigating the effect of different pre-processing forms, parameter tuning, and interpretation of the results.

This thesis describes the application of machine learning techniques to the process of ozone profile retrieval, a method for retrieving a global ozone distribution from satellite measurements. Chapter 2 motivates the need for ozone profile retrieval together with a description of this procedure. The lack of speed is pointed out as the main drawback for real application. The replacement of the forward model, a component of ozone profile retrieval, by a machine learning technique can provide the required speed up. Besides the standard mapping of the forward model, the derivative of this function is required as well.

Machine learning algorithms require a set of training data. Chapter 3 describes the characteristics of the training data and the way it was gathered. Thereafter the choice for neural networks and support vector machines as selected techniques is motivated.

Chapters 4 and 5 focus on the learning of the standard mapping of the forward model. The fourth chapter gives a short introduction to neural networks followed by a description of the different elements that need configuration. Test results and their interpretation are accompanied by theory that provides insight in the performance. Chapter 5 starts with an introduction of support vector machines in the form of binary classifiers. This theory is extended to the two types of support vector regressors, dealing with the problem of function approximation. Test results of different parameter settings are included.

Neural networks will be shown to perform better than support vector machines on the learning of the standard mapping and therefore this technique is also selected to produce the derivative. The first half of chapter 6 describes the results of retrieving the derivative from a network that was trained on the standard mapping. The second half contains the results of an attempt to learn the derivative by a composed architecture of networks trained on a training set with derivative information.

In an attempt to further improve the achieved results two other techniques are considered in the following two chapters. Chapter 7 investigates the possibility of reducing the dimensionality of the problem by application of principle component analysis. Chapter 8 focuses on combining multiple models in an architecture known as an ensemble. Both chapters include test results of the techniques when applied to (the learning of) the standard mapping.

In chapter 9 the best performing models are integrated in the ozone profile retrieval procedure and tested on synthetic measurements. The results and behaviour of the accelerated ozone profile retrieval are explained and evaluated. The results on synthetic measurements provide good insight in the usability of this technique for real measurements.

The conclusion contains a summary of our achievements and a number of suggestions for future research.



## Chapter 2

# Ozone profile retrieval

---

This chapter introduces ozone profile retrieval. First, the reason for the retrieval of ozone profiles from satellite measurements is given. Thereafter, the need for a speed up of this procedure is pointed out. Next, we describe the different elements of the retrieval in greater detail. The iterative nature of the procedure, consisting of a forward simulation and regularised least squares fit is explained. The theory presented in this chapter is necessary for a better understanding of the benefits of our proposed solution presented in chapter 3.

### 2.1 Satellite measurements of vertical ozone profiles

Measurements of the ozone distribution on different altitudes in the atmosphere provide essential information about atmospheric processes. The amount of atmospheric ozone has an influence on the living conditions of mankind. For example, ozone at 10-18 km acts as a greenhouse gas and in turn affects the temperature distribution in the Earth atmosphere. At higher altitudes the stratospheric ozone layer protects living systems from the harmful UV-B solar radiation penetrating to the Earth surface. In the lowest 10 km of the atmosphere ozone initiates chemical reaction cycles, which results in a removal of many anthropogenic emissions. Finally, surface-near ozone is toxic and can cause damage of human health. These different roles of ozone makes it a trace gas of particular interest for atmospheric research.

The most accurate measurements of ozone profiles are performed by radiosonde balloons. These measurements provide vertical distribution of ozone with a resolution of up to 100 m and an accuracy of about 5 %. However ozone balloon measurements are sparse with most measurements in Europe and North America and thus do not provide satisfying information on the global distribution of ozone. Alternatively satellite measurements of ozone can be used. In general these measurements suffer from a lower accuracy of 10-20 % and a low vertical resolution of up to 15 km. Nevertheless owing to the *global coverage* these measurements are a very useful complement to existing networks of radiosonde measurements.

From space, satellites can measure the backscattered sunlight illuminating the Earth atmosphere. A portion of this light is absorbed by atmospheric ozone and thus from these measurements information about the present ozone quantities can be retrieved. In the spectral range of 240-340 nm the amount of reflected sunlight contains information on the *vertical distribution* of ozone because of the wavelength dependent ozone absorption [Chance and Schneider, 1997]. At shorter wavelengths the ozone absorption is very strong and thus the light has only interacted with the atmospheric constituents of the upper atmosphere. In contrast light at longer wavelengths can penetrate the full atmosphere and thus also contains information about ozone at lower altitudes. This is visualised in figure 2.1.

Already a series of space borne instruments are launched which measure the Earth reflectivity spectrum in the ultraviolet part of the solar spectrum. Here the reflectivity at a certain wavelength describes the relative amount of sunlight which is reflected by the Earth atmosphere because of atmospheric scattering and surface reflection. In 1995 the Global Ozone Monitoring Experiment (GOME) was launched on board of the ERS-2 satellite. GOME measurements provide ozone profiles with the spatial resolution of 100 x

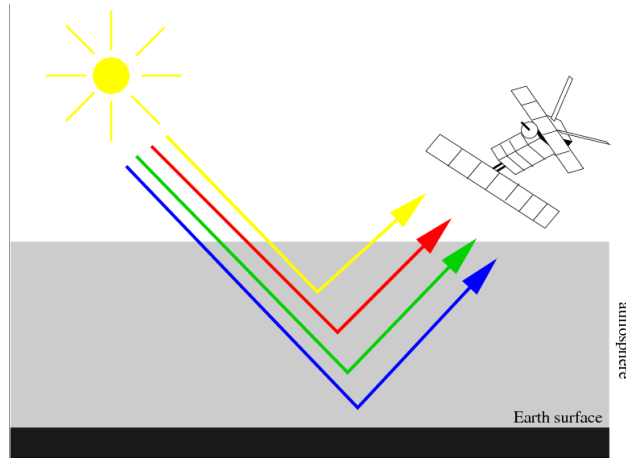


Figure 2.1: Backscattered sunlight measured by a satellite.

80 km<sup>2</sup> with a global coverage within 3 days. GOME measures the Earth's reflectivity in a wavelength range of 240-790 nm and provides information on many atmospheric trace gases such as ozone, nitrogen dioxide, sulfur dioxide, and formaldehyde. Furthermore cloud and aerosol properties can be retrieved from these measurements. The Scanning Imaging Absorption Spectrometer for Atmospheric Chartography (SCIAMACHY) on ESA's ENVISAT satellite was launched in 2002, with an improved spatial resolution of 30x27 to 30x240 km<sup>2</sup> with a global coverage with 3-6 days. The series of UV measurements is continued by the Ozone Monitoring Instrument (OMI) launched on the EOS-Aura satellite in 2004, and the GOME-2 instrument on board of a series the three METOP satellites, with the first launch at 19 October, 2006. These series of UV measurements represents an enormous amount of data to be interpreted and thus requires computational very efficient algorithms for data processing. In particular this is true for the retrieval of ozone profiles from UV measurements which includes extensive simulations of the transport of solar light through the Earth atmosphere.

## 2.2 The retrieval process

The goal of ozone profile retrieval is to retrieve a vertical ozone distribution from a satellite measurement of the reflectivity spectrum in the UV. Here we represent the reflectivity spectrum by a measurement vector  $\mathbf{r}$  with  $m$  elements  $r_1, r_2, \dots, r_m$  describing the reflectivity at different wavelengths. The ozone profile is represented in a discretized manner by a  $n$ -dimensional state vector  $\mathbf{x}$ , where the components  $x_i$ ,  $i = 1, \dots, n$  describe the mean ozone density in different altitude layers. For the interpretation of the measurements a forward model  $F(\mathbf{x})$  is needed which simulates a measurement for a given vertical distribution  $\mathbf{x}$ . In case that the model  $F(\mathbf{x})$  is exact there is at least one profile  $\mathbf{x}$  for which the model simulation  $F(\mathbf{x})$  agrees with the measurement  $\mathbf{r}$  within bounds of the measurement error  $\mathbf{e}$ :

$$\mathbf{r} = F(\mathbf{x}) + \mathbf{e} \quad (2.1)$$

The forward model represents the numerical solution of the radiative transfer equation which describes the interaction of light with the different constituents of Earth atmosphere and the Earth surface. It contains the description of atmospheric absorption and multiple elastic and inelastic scattering of light by air molecules.

To determine the most likely ozone profile which brings the forward model simulation into agreement with the measurement  $\mathbf{r}$  Eq. 2.1 has to be *inverted*. In general a least squares fit approach can be used for such a purpose. However ozone profile retrieval from GOME radiance measurements represents an ill-posed problem. This means that there is more than one profile  $\mathbf{x}$  for which the forward model simulation is in statistical agreement with the measurement. The reason for this is that, within the bounds of the measurements error  $\mathbf{e}$  the measurement is insensitive with respect to fine-scale vertical structures in the ozone profile. In turn, coarser vertical structures in the ozone profile can be retrieved well whereas for the retrieval of fine-scale vertical structures the effect of the measurement noise is amplified yielding noise dominated least squares solution. Thus to obtain a stable inversion the noise dominated contribution of a least squares fit has to be filtered out, which is called regularisation of the least squares cost function:

$$\mathbf{x}_{reg} = \min_x \{ \|\mathbf{r} - \mathbf{F}(\mathbf{x})\|^2 + R(\mathbf{x}) \} \quad (2.2)$$

It is essential that the regularisation term  $R(\mathbf{x})$  is highly sensitive to the contribution of the measurement error to the retrieved profile. For this purpose commonly one uses the norm of the profile or the norm of the derivative of the profile with respect to height. The combined minimisation of the least squares residual norm and the regularisation term provide a stable inversion of the problem. However, due to the regularisation the retrieved profile  $\mathbf{x}_{reg}$  is a smoothed version of the real profile and so any statement on vertically fine structures in the ozone distribution can not be done based on GOME measurement only.

The smoothing of the retrieval can be expressed with the averaging kernel  $\mathbf{A}$ , viz.

$$\mathbf{x}_{reg} = \mathbf{A}\mathbf{x} \quad (2.3)$$

Here the averaging kernel can be calculated for each solution  $\mathbf{x}_{reg}$  individually in a straight forward manner. The averaging kernel can also be interpreted as a projector, which filters out those structures in the vertical ozone distribution about which no information is present in the measurement. If those structures are needed for a data product  $\mathbf{x}_{data}$  one can try to add those using *a priori* knowledge on the ozone profile, viz.

$$\mathbf{x}_{data} = \mathbf{x}_{reg} + (\mathbf{1} - \mathbf{A})\mathbf{x}_a \quad (2.4)$$

where  $\mathbf{x}_a$  is e.g. an ozone profile from alternative measurements or from model simulations. It is important to realize that due to an inappropriate *a priori profile*  $\mathbf{x}_a$  errors can be introduced in the data product. Further it should be mentioned that the *trace* of matrix  $\mathbf{A}$  is called the DFS (Degrees of Freedom for Signal) and its value represents the number of independent pieces of information that can be retrieved from the measurement.

For the retrieval another difficulty is the fact that the forward model  $F$  is non-linear in the vertical ozone distribution. This requires an iterative least squares approach, which is demonstrated in Fig. 2.2. First a Taylor expansion of  $F$  around a “first guess” profile  $\mathbf{x}_0$  is calculated. It is truncated in the first order. In this way the forward simulation is linearised around  $\mathbf{x}_0$  and the simulated measurement  $\mathbf{y}$  can be written in the form of equation 2.5.

The inversion of this equation by use of the regularised least squares fit displayed in equation 2.2 provides an estimate  $\mathbf{x}_1$  for the next iteration. In every iteration step  $i$  the point  $\mathbf{x}_i$  is used as a new expansion point for the Taylor series. The inversion step is calculated afterwards. This procedure halts until a state of convergence is reached, which means that a profile  $\mathbf{x}_i$  is found that, when processed by the forward model, agrees with  $\mathbf{r}$  with an error less than  $e$ .

$$\mathbf{y} = \mathbf{J}\mathbf{x} + e \quad (2.5)$$

$$\text{with } \mathbf{y} = \mathbf{r} - F(\mathbf{x}_0) + \mathbf{J}\mathbf{x}_0 \text{ and Jacobian } \mathbf{J} = \frac{\partial F}{\partial \mathbf{x}_0}(\mathbf{x} - \mathbf{x}_0)$$

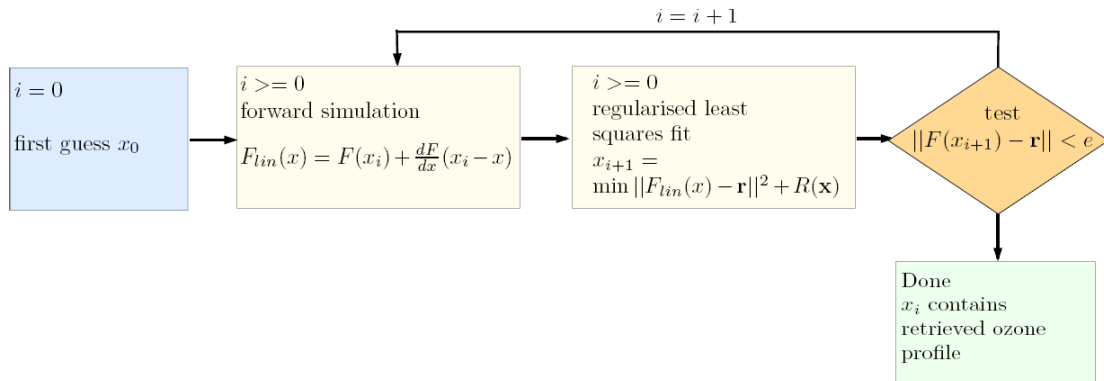


Figure 2.2: The retrieval process schematically

## 2.3 Summary

Information about the ozone distribution in the Earth atmosphere is important because it has a large influence on the living conditions of mankind. The most accurate measurements are performed by radiosonde balloons, but they are sparse and a global ozone profile cannot be obtained by these measurements only. Global coverage can be obtained by retrieving ozone profiles from satellite measurements and for this purpose multiple instruments have been sent into space. These instruments measure the reflectivity of the sunlight reflected by the Earth atmosphere. Ozone profile retrieval algorithms are able to retrieve information about the ozone distribution that best matches the satellite measurement. However, these models are computationally expensive and are not able to process the incoming data in the desired amount of time.

The retrieval process is an iterative approach consisting of two important components: a forward simulation  $F(x)$  and a regularised least squares fit. The forward simulation, which performs simulations of solar light through the atmosphere, is the computationally most expensive part of the retrieval. The regularisation in the least squares fit is needed because information about fine-scale structures in the atmosphere cannot be retrieved from satellite measurements. This results in a smoothing of the profile, displaying a lower vertical resolution than a balloon measurement. Information about the degree of smoothing is available in the averaging kernel which is calculated as part of the regularisation.

# Chapter 3

## Problem analysis

---

The motivation for the use of machine learning techniques is a desired speed up of ozone profile retrieval. These techniques will be used to replace the forward model  $F$ , which consists of two parts: the standard mapping and the Jacobian. The training data that will serve to let the machine learning algorithm learn the forward model is presented and thereafter the choice for the two selected techniques, neural networks and support vector machines, is motivated. Finally, a comparison to related research points out the differences and benefits of our approach with respect to related research.

### 3.1 Replacement of the forward model

As explained in chapter 2 the retrieval process is slow; especially the simulation of a measurement is a slow process and this has to be done in every iteration of the algorithm. The target of our research is to speed up the retrieval by replacing this simulation. In every step of the retrieval  $F(x)$  is given by the first order Taylor expansion of equation 3.1, hence both the forward model  $F$  and its derivative function  $\mathbf{J}$  need to be replaced. The training data and models that will be used for this purpose are described in the next two sections.

$$F(\mathbf{x}) = F(\mathbf{x}_{i-1}) + \mathbf{J} (\mathbf{x} - \mathbf{x}_{i-1}) + e \quad (3.1)$$

### 3.2 The training set

All machine learning techniques use training data in the learning process. In case of a supervised learning problem, as is the case with function approximation, a training set consists of pairs of inputs and outputs. In the training phase the models outputs are compared to the desired outputs and the model is adapted to fit the data better. In general, complex functions require more training samples than less complicated functions.

The forward model represents a mapping of an ozone profile  $\mathbf{x}$  to the measurement vector  $\mathbf{y}$ . Here  $\mathbf{x}$  is a vector of  $n$  elements which are averaged ozone densities in different altitude layers of the model atmosphere and  $\mathbf{y}$  represents a  $m$  dimensional vector, consisting of the reflectivity values of the model atmosphere at different wavelengths. The atmosphere is divided into layers with a thickness of 2 km up to a total height of 60 km. A training pattern consists of a pair  $(\mathbf{x}, \mathbf{y})$  with  $\mathbf{x}$  and  $\mathbf{y}$  of size  $n$  and  $m$  respectively. For the training set it is important that the ozone profiles  $\mathbf{x}$  describe the relevant variations in the atmosphere. To achieve this radiosonde measurements from the vertical ozone distribution taken by weather balloon measurements are used. Unfortunately the weather balloons are only able to measure up to around 30 km before they explode because of the decreasing air pressure. To extend the ozone profile above this altitude we adopt an ozone climatology [Fortuin and Kelder, 1998]. This climatology provides a mean profile taken from alternative observations over a number of years on a certain degree of latitude. A number of 12

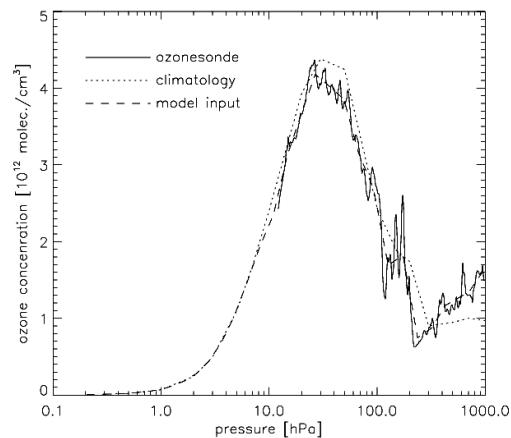


Figure 3.1: An ozone profile  $\mathbf{x}$  as function of air pressure. Above 10 hPa the climatology is adopted.

variations, each for every month, are used to take the seasonable changes into account. Figure 3.1 displays a training pattern as function of the air pressure. At (relatively) high pressure the high resolution weather balloon measurements result in more fluctuation of the profile compared to the smoother profile at higher altitudes where the source of measurement is a climatology.

The output vector  $\mathbf{y}$  of each sample is the output as calculated by the existing implementation of the forward model. In principle this makes it possible to generate as many training samples as desired if there are enough balloon sonde measurements available as input for the forward model. However in case of a shortage, model simulations of the chemical composition of the earth atmosphere can also be used to generate data.

In addition to training patterns of the form  $(\mathbf{x}, \mathbf{y})$  the derivative matrix or jacobian  $\mathbf{J}$  of size  $m \times n$  is available for each pattern. As explained in chapter 2 this matrix is an important part of the retrieval. Chapter 6 concentrates on the Jacobian and evaluates a model trained on this data as an optional solution.

### 3.3 Selected techniques

There are three desired properties a machine learning model should have in order to serve as a succesful replacement of the forward model. Firstly, to assure a reliable ozone profile retrieval the level of accuracy should be very high. Secondly, it should be possible to retrieve the Jacobian from the trained model or the model should be able to learn the Jacobian as well. Finally, the speed improvement should be large, because this is the initial reason for replacement.

#### Feed-forward neural networks

Neural networks<sup>1</sup> are a popular technique for function approximation since the 80s. In theory a neural network can learn any continuous function with arbitrary accuracy [Cybenko, 1989]. The real performance is dependent on the configuration of the parameters, its architecture and the quality of the training set. In

<sup>1</sup>In this thesis the term neural network always refers to a feed-forward neural network

the learning process a search to the global minimum of the error function is performed. However there is no guarantee of finding this optimum and ending up in a local minimum is very common.

A requirement of good performance is the availability of enough training patterns. As explained in the previous paragraph that is not going to be a problem in this case. Furthermore, a neural network can be seen as a large continuous function parameterised by the weights in the network. The derivative of a network trained on the forward model should produce the desired Jacobian. Moreover, once a neural network has been trained the output can be produced in one *forward pass*, which fulfils the speed requirement. Neural networks have, at least in theory, the desired properties and seem to be a good candidate for replacing the forward model.

### Support vector machines

Support vector machines are a technique which has become popular in the mid 90s. Especially in classification problems they often have become the technique of choice over feed-forward neural networks. In fact, feed-forward neural networks can be seen as a special case of support vector machines [Witten and Frank, 2005]. For function approximation the support vector machines variant is called support vector regression [Smola and Schoelkopf, 1998]. Important properties are the good generalisation ability and presence of control parameters which have a direct effect on the amount of (over)fitting of the training data. Disadvantage is that these control parameters are difficult to configure.

A trained support vector regressor (SVR) produces its output by a simple and fast calculation, which makes it a candidate for the problem at hand. In [Lazaro *et al.*, 2004] research to the learning and derivation of the derivative function from a support vector regressor is described. We can conclude that support vector machines also meet all requirements and are therefore selected as the second possible candidate.

Both techniques will be applied to the learning of the standard mapping  $F$  of the forward model. Dependent on the results we will decide which technique(s) to use to replace the Jacobian  $\mathbf{J}$ .

## 3.4 Comparison to related research

Machine learning algorithms are already used for interpretation of satellite measurements. For example [Muller and Kaifel, 2003], applied a neural network to retrieve ozone profiles from GOME measurements. These authors have used GOME spectral measurements and collocated measurements of ozone profiles from radiosondes and other satellites to train a neural network. Thus in their approach the complete retrieval scheme described in the previous section is replaced by a neural network. To improve the performance of the network the authors have used additional input for the neural network. For example, information on the geolocation of the ozone profile and a collocated vertical temperature profile is used. In other words the neural network makes use of the fact that the vertical ozone profile correlates with temperature and geolocation which means the use of a priori information in the retrieval process. One important feature of the approach is that it requires GOME measurements collocated with independent measurements of the vertical distribution of ozone on a global scale. This introduces two problems: First radiosonde measurements of ozone are in situ measurements which measure the ozone concentration at the position of the balloon gondola during the flight, whereas GOME measures the Earth reflection for a ground pixel of  $80 \times 960 \text{ km}^2$ . Due to internal variations of ozone within the observed scene, the ozone sonde measurement does not necessarily represent the mean ozone concentration of GOME ground pixel. Furthermore ozonesonde measurements are sparse, and are mainly performed at Europe and North America. Thus it is very difficult to represent remote areas adequately in the training data set. The advantage of the approach of Kaifel and Müller is focused on the low numerical cost of the data procession, however it only provide little insight on physical principles on how information is extracted from the measurements. Thus for the purpose of

getting a deeper understanding on the interpretation of measurements and on its limitations a retrieval approach as proposed in the previous section is needed.

The neural network approach for ozone profile retrieval can be significantly improved when one retains the inversion approach of section 2.2 but replaces the forward model with a machine learning algorithm. Because then all diagnostic tools of the inversion can be used. Since an implementation of the forward model exists, the training samples for the replacing algorithm can be generated from a set of ozone profiles representing the most relevant vertical structures of ozone. Here all ozonesonde measurements can be used without suffering from any collocation problem. Furthermore also simulated ozone distributions from large scale chemical transport models can be utilized for this purpose, which weakens the requirements on the training data set significantly. In this context the machine learning component is used only for speeding up the calculation and not learning an unknown relationship between ozone profiles and satellite measurements.

The approach can be adopted easily for the retrieval of other atmospheric constituents. For example the retrieval of aerosol properties from satellite measurements also suffers from time consuming forward simulations of the measurement. In contrast to ozone much less direct measurements of aerosol properties are available on a global scale. Here the training of the machine learning component can be at least partly based on simulated aerosol properties and due to that the operational retrieval of aerosol properties can greatly benefit from such an approach.

### 3.5 Conclusion

A solution to the problem of speeding up ozone profile retrieval has been given in the form of a replacement of the forward model by a machine learning technique. In theory neural networks and support vector machines are both able to learn a function with high accuracy and thereafter produce their output in a fraction of a second. These properties motivate the choice for further investigation of the performance of these techniques. First, the performance on the learning of the standard mapping will be evaluated. Thereafter we will decide which techniques will be used to produce the derivative.

In earlier research machine learning techniques have been applied to ozone profile retrieval. The innovative factor of our approach is that we only replace the forward model, instead of the complete retrieval process. There are multiple advantages of this approach. First of all we have an accurate model available, which can produce the training data. Secondly we are not forced to use a-priori information, because we are not trying to produce an ozone profile with information about the fine-scale structures. Further, because we keep the regularisation as part of the retrieval we still have access to the averaging kernel that provides information about the degree of required smoothing. Finally, other retrieval methods, for example the retrieval of aerosol properties, can also benefit from this approach because they contain a similar forward model.

# Chapter 4

## Neural network approach

---

This chapter introduces the neural network model used in this thesis and describes the design decisions that have been made in the process of finding an optimal network configuration. These design decisions are motivated by theory and testing results. The Matlab Neural Network Toolbox provides the required neural network functionality and is used for all experiments in this chapter. The network is trained on the training set described in chapter 3 with the purpose of learning the forward model  $F$  as accurately as possible. The first section gives a short introduction to feed-forward neural networks and the used terminology and notation. For a good understanding of the results the method of performance measurement is introduced thereafter. From this section on the different network aspects that have to be configured are described one by one. These sections start with a portion of theory and end with an interpretation of experimental results based on this theory. The process of coming to optimal configuration of the network parameters is described step by step, but was in reality more of an iterative approach. The last section summarises the results and draws conclusions about the performance and usability of a neural network as a replacement for the forward model.

### 4.1 Notation and terminology

This section introduces the terminology used throughout this thesis. The introduction is rather compact because the reader is assumed to have a basic idea of what a feed-forward neural network looks like. A good introduction can be found in [Hertz *et al.*, 1990].

A feed-forward neural network is most easily visualised as a graph where the connections are between nodes of subsequent layers. The edges connecting the nodes contain weights which are updated by a learning algorithm during the training procedure. Every node in the network calculates its output by executing an activation (or transfer) function  $\varphi$ . The output of node  $i$  in layer  $L$  is denoted by  $o_i^L$ . In this thesis a network consisting of  $L$  node layers and  $L - 1$  weight layers is called an  $L$ -layered network. A weight layer between layer  $L - 1$  and layer  $L$  is represented as a matrix  $w^L$ . In this matrix value  $w^L(j, i)$  holds the weight from node  $i$  in layer  $L - 1$  to node  $j$  in layer  $L$  and is denoted as  $w_{ji}^L$ . In a network of  $l$  layers every layer with  $L < l$  also contains a bias node  $b_L$  with accompanying bias weights  $\theta_i^L$ . This bias node has no inputs and its value is set to a (fixed) value of 1 and it is fully connected to layer  $L + 1$ . Figure 4.1 displays such a neural network. Further, when discussing a topic concerning all the weights in the network they can be referred to as a vector  $\mathbf{w}$ .

The output of a network is calculated in a forward pass in which the inputs are fed to the input layer and are multiplied by the weights in the first layer. In the next layer each node takes the sum of its input values combined with the bias weight as input for its activation function. The just calculated outputs serve as inputs for the next layer and the same procedure is followed until the last layer is reached. Using the above terminology the output of a network with 3 layers can be calculated by the following formula:

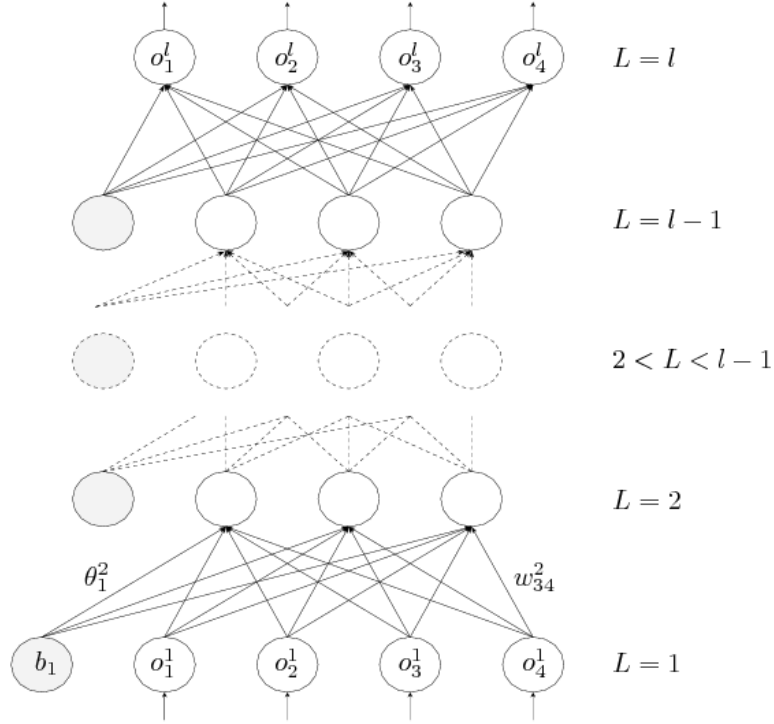


Figure 4.1: A feed-forward neural network with  $l$  layers of nodes

$$o_k^3 = \varphi_k^3 \left( \sum_j w_{kj}^3 \varphi_k^2 \left( \left( \sum_i w_{ji}^2 o_i^1 \right) + \theta_j^2 \right) + \theta_k^3 \right) \quad (4.1)$$

## 4.2 Performance measurement

The performance of a neural network can be expressed in different ways. As will be explained in section 4.5 the learning algorithm updates the weights in an attempt to minimise a certain error function. By default and also in this thesis this is the mean square error given in equation 4.2. Besides the mean square error the mean absolute error and mean relative error provide insight in the performance of the network. The absolute error is less sensitive to the presence of outliers and the relative error, holding the error as percentage of the target value, is size independent. This last property is useful especially when comparing errors of outputs which are in different scales of measurement, since the relative error is a form of normalisation. When working with machine learning techniques a certain level of error has to be accepted. It is estimated that a relative error less than 1 % is desired for the neural network to be a usable replacement for  $F(x)$  in ozone profile retrieval performed on real (unseen) measurements.

$$MSE = \frac{1}{n} \cdot \sum_{i=0}^n (o_i - t_i)^2 \quad (4.2)$$

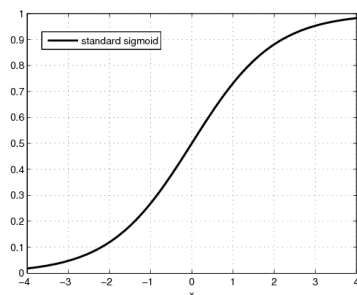


Figure 4.2: Standard sigmoid function  $\frac{1}{1+e^{-x}}$ .

During the training phase the mean square error with respect to the training set is plotted in a figure like 4.3. Besides the error on the training set the error on unseen samples is of importance. This validation error gives the best indication of how the network would perform on unseen data. There are different techniques of producing the most reliable validation error. Usually a certain percentage of the training data is left out for testing. The size of this set must be large enough to contain a diverse amount of testing samples, but on the other hand there should be enough training data left for the training set. A *10 fold cross validation scheme* is the most commonly used technique for producing the most reliable validation error. The training data is randomly split into 10 separate parts. The neural network is trained 10 times and every time another part is taken as validation set. The average of the 10 validation errors is a good overall error estimate. The reason that a number of 10 folds is used is funded mostly by the results of years of experience in different fields of machine learning [Kohavi, 1995]. An even better error estimate can be obtained by using a 10 times 10 fold cross validation scheme, but this is a more time consuming operation.

The degree of performance on unseen samples is often called the networks generalisation ability. At the moment when the training error is much lower than the test error a state of overfitting has been reached. This is undesired and the network is said to generalise poorly. For further analysis of performance on the test set the mean standard deviation of the error is interesting. It can be seen as a measure of consistency, high variation in the distribution of the error can be the cause of outliers or a bad representative training set. The results tables in the appendix contain the outcome for the mentioned measures. Here the results are presented in a compact way, often as a graph of the mean square error or the relative mean absolute error per output.

### 4.3 Normalisation

Normalisation is a form of pre-processing which scales numerical data to a certain range. There are different arguments for the use of normalisation. Output normalisation can be useful in case different outputs are in different scales of measurement. This can cause a relative small error on a certain output to have a much larger influence on the mean square error than a relative large error on another. Very small target values can also be a reason for output normalisation since some learning algorithms interpret the resulting very small errors as an indication for convergence. Output normalisation is often not necessary. Input normalisation on the other hand is almost always applied. The reason for this is that the most commonly used activation function has a high sensitivity in a small range, typically  $[-1, 1]$ . Figure 4.2 shows the sigmoidal activation function with its gradually increasing curve between  $[-1, 1]$  which flattens out towards  $-\infty$  and  $+\infty$ . The input data is therefore usually normalised into this range of higher sensitivity.

Rescaling data in a certain range is most easily accomplished by applying a linear transformation to  $[0, 1]$  using the formula in equation 4.3. A slight modification can further transform it to an arbitrary range, for example  $[-1, 1]$  (equation 4.4). Greatest disadvantage of this method is its intolerance for outliers in the input set. For example if  $x$  normally has values around 0 but has a few very large occurrences. After normalisation the most rescaled values of  $x$  lie around 0 and the few outliers will lie near to 1, undoing the desired effect of normalisation.

$$\tilde{x} = \frac{x - \min}{\max - \min} \quad (4.3)$$

$$\tilde{x} = 2 \cdot \left( \frac{x - \min}{\max - \min} \right) - 1 \quad (4.4)$$

A more robust normalisation method (known as standardisation) centers the data around 0 with a standard deviation of 1. This can be accomplished by subtraction of the mean  $\mu$  followed by division by the standard deviation  $\sigma$  (equation 4.5). This transformation brings all values, regardless of the underlying distribution or units of measurement, to comparable units. Moreover, outliers do not have such a strong effect on the results of this method.

$$\tilde{x} = \frac{x - \mu}{\sigma} \quad (4.5)$$

Besides the type of normalisation, the way this technique is applied to the data set is another design decision. On a data set consisting of multiple input variables normalisation can be applied per column or per row. Normalisation per column makes sense in the light of normalising per input variable, but when this type of normalisation is applied to unseen data it might occur that the pre calculated normalisation factors<sup>1</sup> are not representative. In case of linear normalisation this can lead to scaling to a domain larger than  $[-1, 1]$ . Row-wise normalisation does not suffer from this phenomenon because the normalisation parameters can be calculated per sample. Unfortunately this local normalisation suffers from (slightly) different normalisation parameters per sample which has a significant effect on the results.

## Test results

A number of tests have been performed. The results can be found in appendix A.1. The network that is evaluated has one hidden layer of 20 nodes. This is kept constant for better comparability. All the mentioned normalisation methods have been tested in combination with two different learning algorithms: Scaled Conjugate Gradient (SCG) and Resilient Propagation (RPROP). A more detailed description and evaluation of the learning algorithms can be found in section 4.5. Variation of the learning algorithm is interesting, because there is a good chance that a learning algorithm has a preference for a certain normalisation method. The first results table contains the different errors of networks that are trained for 1000 epochs. The best candidates are trained for 5000 epochs to see if this changes anything.

The results show that normalisation has a great impact on the network performance. Leaving out any kind of normalisation results in the worst performance. The column-wise transformations outperform the row oriented ones by a factor 10. This can be explained by the fact that a good representative training set leads to good estimates of normalisation parameters for the unseen data. The RPROP learning algorithm has a slight preference for linear normalisation in comparison to the SCG method which performs better with standardised inputs. This observation will be kept in mind in further experiments, but for now the combination of standardisation and SCG is most succesful. Another observation is the small difference

---

<sup>1</sup>For linear normalisation:  $\max, \min$ . For standardisation:  $\mu, \sigma$

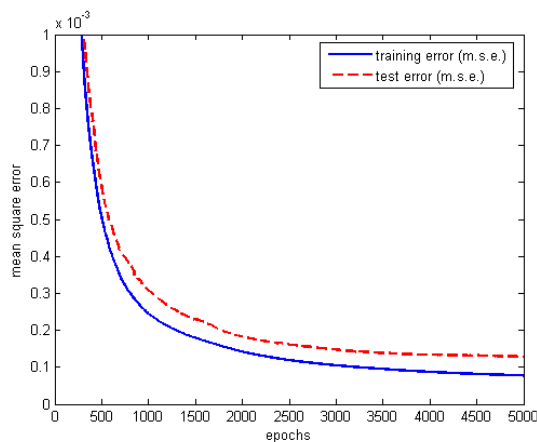


Figure 4.3: Error plot of an RPROP network trained for 5000 epochs with columnLinear normalisation.

between errors in the test and training set. Figure 4.3 shows the decreasing errors during training. Both errors keep decreasing, indicating that a state of overfitting has not been reached yet. However, convergence slowly drops to a rate in which performance improvement is neglectable. In the latest epochs the mean square error decreases with  $-5.2791e-009$ . The good performance on the test set indicates a representative training set.

## 4.4 Network architecture

The number of nodes in the different layers of a feed-forward neural network have a great influence on its behaviour. If no dimensionality reduction techniques are used the number of inputs is set to the number of input variables of the target function. Similarly the size of the output layer is determined by the number of output variables. The size and number of hidden layers is a harder design decision. Over the years research in this area has resulted in a number of rules of thumb used by practitioners. To give a network the ability to learn a non-linear function at least one hidden layer is needed, otherwise its performance reduces to that of a simple perceptron. On the other hand the maximum number of needed hidden layers is also equal to one, since [Hornik *et al.*, 1989] states that a neural network consisting of one hidden layer can approximate any given function with a finite number of discontinuities to any desired accuracy. However, in practice sometimes multiple hidden layers are used.

The size of the hidden layer(s) directly effects the number of weights in the network and therefore the complexity of the learned function. Too many nodes in the hidden layer might let the network memorise the patterns which decreases generalisation, but too few nodes limit the complexity of the function the network is able to learn. The paper [Vytautas Vysniauskas, 1993] states that the relation between the desired accuracy and the number of training patterns available can lead to an estimation of the needed number of hidden nodes. Another theory [Hinton, 1989] suggests that for every connection in the network there should be a training sample in the training set. The latter should be considered more like a rule of thumb, it is by no means a proved theory.

## Test results

It is clear that there is no straightforward approach for determining the network architecture and an empirical approach is the most practised and attractive alternative. Appendix A.2 contains experiments of networks with up to 2 hidden layers of varying sizes. The used data set consists of around 2600 patterns. In case of one hidden layer the rule of thumb<sup>2</sup> to determine the amount of network weights would suggest 10 hidden nodes. All experiments have been done for the two considered learning algorithms SCG and RPROP to see if they have a preference for a certain architecture. The following observations have been made:

- Overfitting does not occur. Even after training some of the architectures for 5000 epochs the test errors keep close to the training error.
- A hidden layer size of 20 shows big improvement over the results with 5 or 10 hidden nodes. Above 20 hidden nodes the performance increase is smaller.
- Networks trained with the RPROP learning algorithm seem to benefit from multiple hidden layers. Networks with two 2 hidden layers of size 30 and 50 (9980 connections) perform almost twice as good as networks with a single layer of 50 hidden nodes (9100) connections. This is not the case for networks trained with the SCG learning algorithm.

These results can only partially be explained by existing theories. In theory large sized networks, like networks with two hidden layers of size 50 and 50, trained on a limited training set are known to memorise the learned patterns and therefore generalise very badly. However, these results show excellent performance on the test set. Most probably this means that the test set is quite similar to the training set. With a very representative training set a good performance on the test set is to be expected.

A number of 5 or 10 hidden nodes obviously limits the performance of the network. We can conclude that a minimum of 20 hidden nodes is needed to obtain “good” performance.

The fact that the large sized SCG trained networks do not show a performance improvement over the single layer variants, but still outperform all RPROP networks is most probably caused by the more efficient way in which SCG steps through the search space. The next section contains a description of both algorithms and demonstrates the more sophisticated error minimisation procedure of SCG.

## 4.5 Learning algorithm

The learning algorithm updates the weights in a neural network in an attempt to minimise the difference between the network output and the desired output. There are many different learning algorithms available in the Matlab Neural Network Toolbox, but in this thesis we consider only resilient propagation (RPROP) and scaled conjugate gradient (SCG). Both algorithms find their origin in the steepest descent or gradient descent optimisation method which will be introduced first. A theoretical explanation of the selected algorithms is given, but the details of the weight update mechanism are omitted. More information about the backpropagation of errors can be found in [Hertz *et al.*, 1990].

### Steepest descent

Steepest descent is a method of minimising a quadratic function as in equation 4.6 where  $A$  is a matrix,  $\mathbf{x}$  and  $\mathbf{b}$  are vectors and  $c$  is a scalar. This minimisation procedure performs best if the matrix  $A$  is positive

---

<sup>2</sup>In case of 31 input nodes and 151 output nodes:  $31 * 10 + 10 * 151 = 2730$

definite and symmetric. The latter two properties ensure an error surface which has a global minimum at the bottom of a (in case of 2 dimensions) parabolic bowl (figure 4.4(a)). The minimum of the error surface can be found at the place where the *gradient*  $f'(\mathbf{x})$  is equal to 0. In an iterative manner the vector  $\mathbf{x}$  is adapted with the purpose of moving the error towards this minimum. The new  $\mathbf{x}(t+1)$  is a combination of the previous  $\mathbf{x}(t)$  and a step of size  $\eta$  in search direction  $\mathbf{r}(t)$  (equation 4.7). The search direction is defined by the negative of the gradient, because the gradient in a point  $\mathbf{x}$  always points to the direction of steepest ascent. The step size can be set to a fixed value, but can also be calculated by a *line search*. The optimal step size is found at the place where the directional derivative  $f'(\mathbf{x}(t) + \eta\mathbf{r}(t))$  is equal to 0. In figure 4.4(b) the projection of the search direction is plotted and the minimum is found at the bottom of the parabola. The gradient in that point is always orthogonal to the previous gradient, visualised in figure 4.4(c). Using this property the optimal step size can be calculated.

In case of neural networks the least squares error function  $E$  parameterised by weight vector  $\mathbf{w}$  has to be minimised. The weights are updated in a manner similar to equation 4.7. In the default learning algorithm, standard backpropagation, a fixed step size  $\eta$  is used and is called the learning rate. Although a fixed learning rate is computationally cheap the optimal value is definitely not constant. A learning rate that is set too high causes error oscillation as a result of constant jumping over the minimum. A learning rate set too low results in an endless amount of steps. Moreover the error surface of  $E$  is virtually never as smooth as in figure 4.4(a) and multiple local minima are often present. As a consequence there is a chance of getting stuck in such a local minimum. This problem will be addressed after introducing the learning algorithms.

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T \mathbf{x} + c \quad (4.6)$$

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \eta \cdot \mathbf{r}(t) \quad (4.7)$$

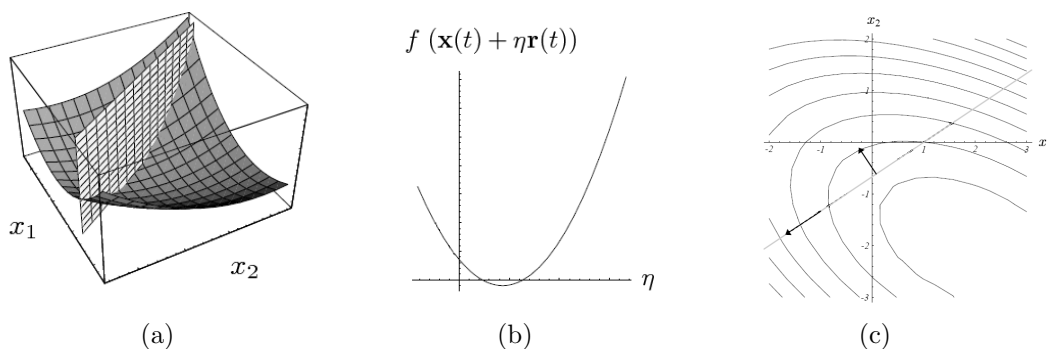


Figure 4.4: (a) The search space and a search direction. (b) The search direction as a parabolic function of which the minimum defines the optimal step size  $\eta$ . (c) The direction of the gradient in this position is orthogonal to the previous gradient.

Alternatively, the problem of minimising  $E$  by finding a weight update vector  $\mathbf{y}$  that is added to the current weight vector  $\mathbf{w}$  can be posed as the minimisation of the Taylor expansion of  $E$  around the point  $\mathbf{w}$ . Differentiation results in equation 4.9. Steepest descent only uses *first order* derivative information leading to the familiar problem of solving  $E'(\mathbf{w}) = \mathbf{0}$ .

$$E(\mathbf{w} + \mathbf{y}) = E(\mathbf{w}) + E'(\mathbf{w}) \cdot (\mathbf{y} - \mathbf{w}) + E''(\mathbf{w}) \cdot \frac{1}{2}(\mathbf{y} - \mathbf{w})^2 + \dots \quad (4.8)$$

$$E'(\mathbf{w} + \mathbf{y}) = E'(\mathbf{w}) + E''(\mathbf{w}) \cdot (\mathbf{y} - \mathbf{w}) + \dots \quad (4.9)$$

## Resilient propagation

The difference between standard backpropagation (steepest descent with a fixed  $\eta$ ) and resilient propagation [Riedmiller M., 1992] is that the latter ignores the size of the derivative in the weight update rule. The weight update rule of standard backpropagation (equation 4.10) makes direct use of the partial derivative of  $E$  with respect to the current weight. The size of this derivative can vary and is sometimes very large, which can lead to the earlier mentioned oscillation effect, similar to a learning rate that is set to high. The resilient propagation algorithm only looks at the sign of the derivative. More precisely, it looks at the change of the sign. The sign of the previous derivative is saved (for each weight in the network) and multiplied with the current one. When negative the previous weight update was too large and caused to jump over a local (or global) minimum. This weight update is corrected by undoing the previous weight update (which was saved) and replacing it by a fraction of it. This fraction is determined by the the first learning rate  $\eta^-$ . In case the current partial derivative has the same sign as the previous one the previous weight update is increased by a certain factor dependent on the second learning rate  $\eta^+$ . The rule is defined in equation 4.11. The previous weight update of weight  $w_{ij}$  is denoted by  $\Delta_{ij}(t-1)$ . Weight  $w_{ij}$  is updated with  $\Delta_{ij}$  and computed as in equation 4.11. The advanced weight update rule of resilient propagation makes the algorithm converge much faster than standard backpropagation. Moreover it is a “cheap” improvement which makes it suitable for large sized networks.

$$w_{ij}(t+1) = w_{ij}(t) + \eta * \frac{\partial E}{\partial w_{ij}}(t) \quad (4.10)$$

$$\Delta_{ij} = \begin{cases} \Delta_{ij}(t-1) * \eta^+, & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \Delta_{ij}(t-1) * \eta^-, & \text{if } \frac{\partial E}{\partial w_{ij}}(t-1) * \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ \Delta_{ij}(t-1), & \text{else} \end{cases} \quad (4.11)$$

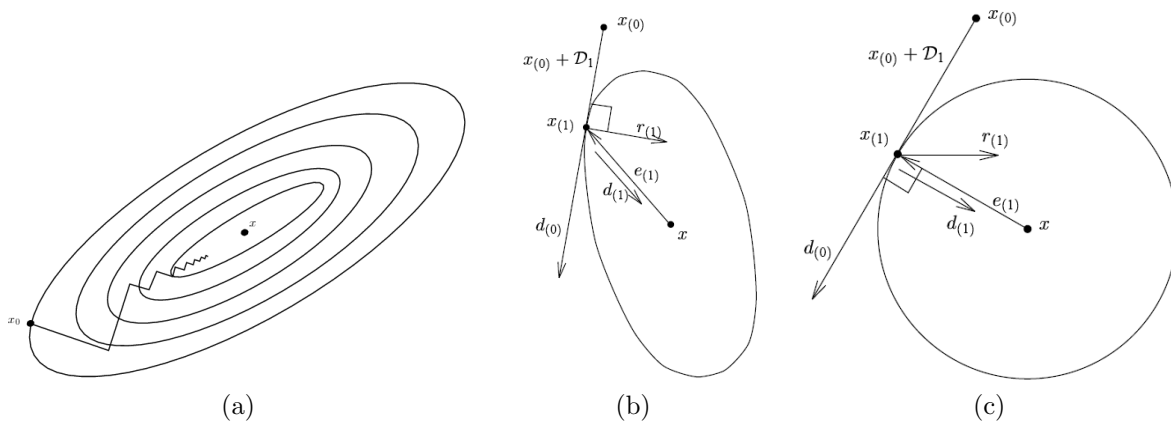
with  $0 < \eta^- < 1 < \eta^+$

## Scaled Conjugate Gradient

The disadvantage of a fixed step size can be overcome by a line search in the search direction. Unfortunately, even with use of a line search method steepest descent still has slow convergence. This is due to the orthogonality constraint of subsequent search directions which results in a zig zag path through the search space. Especially the combination of an error surfaces with a long *valley* and unlucky starting position requires many iteration steps (figure 4.5(a)).

$$d_i^T \cdot A \cdot d_j = 0 \quad (4.12)$$

The scaled conjugate gradient learning algorithm is one of the learning algorithms based on the conjugate gradient methods. These methods provide a way of minimising an  $n$  dimensional quadratic function with a positive definite and symmetric  $A$  in  $n$  steps. Conjugate gradient methods use more information about the error surface than just the negative of the gradient to determine the next search direction. The subsequent



search directions are required to be  $A$ -orthogonal or *conjugate* to each other. Two search directions  $d_i$  and  $d_j$  are  $A$ -orthogonal if equation 4.12 holds. Conjugation can be looked upon as a generalisation of orthogonality for which  $A$  is the unity matrix. This can be visualised by looking at the error surface as a *stretched* search space. Figure 4.5(b) displays a contour line of such a stretched space and any pair of vectors that are perpendicular in this space are orthogonal. Figure 4.5(c) shows the same space stretched along the eigenvector axes of  $A$  such that the elliptical contours become circular. Any pair of vectors that appear perpendicular in this space are called  $A$ -orthogonal or conjugate. In figure 4.5(b) and (c) the search is started from  $x_0$  into direction  $d_0$ , at  $x_1$  steepest descent would choose direction  $r_1$ , but the conjugate gradient method would directly step towards the middle of the stretched “circle”. More details and information about the procedure to calculate these conjugate directions can be found in [Shewchuk, 1994]. Although theoretically sound this method has a few disadvantages in practise. Conjugacy of search direction easily gets distorted by inaccurate line searches due to numerical accuracy and limited computing time. This problem is commonly solved by resetting the search direction to the negative of the gradient after a predefined number of iterations. Despite these problems, which result in an increase of iteration steps, conjugate gradient methods usually converge faster than steepest descent.

The scaled conjugate gradient learning algorithm applies this technique to the neural network error function. Unlike steepest descent it uses second order derivative information in the optimisation process. Differentiation of equation 4.8 leads to the problem of solving the linear system:  $E'(\mathbf{w}) + E''(\mathbf{w}) \cdot (\mathbf{y} - \mathbf{w}) = 0$ . In theory this can be done in  $n$  steps using the conjugate gradient method. This will only succeed in case  $E''(\mathbf{w})$  is positive definite and symmetric, otherwise it has to be transformed so it can be used as it would have these properties. This transformation is necessary but results in decreasing performance. Further this algorithm does not explicitly calculate the Hessian matrix  $E''(w)$  and utilises an approximation technique for the line search to reduce complexity. These last two properties make this algorithm well suited for use in neural networks with a large number of weights. More details can be found in [Moller, 1993].

Both algorithms are designed to find a minimum in a search space with a unique and therefore global minimum. However for difficult problems error surfaces are usually rough and contain a number of local minima. Luckily, a sufficiently large and preferably randomised training set keeps the network error moving across the error surface without getting stuck into a local minimum. Adding a small amount of noise to the training data for a few cycles of the learning algorithm can also help the network jump out of a local minimum and continue its search for the global minimum.

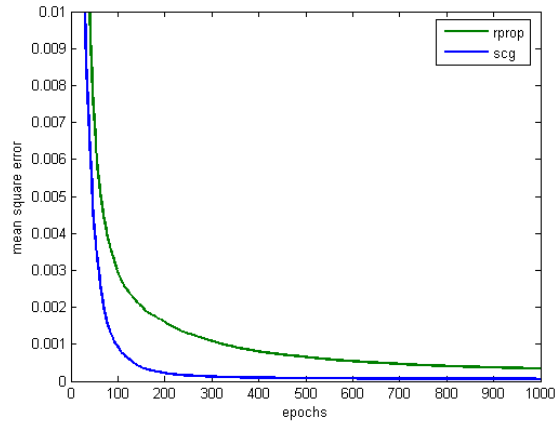


Figure 4.5: Comparison of the training error of the RPROP and SCG learning algorithm.

### Test results

There is no separate appendix for learning algorithm experiments since the needed experiments are included in the appendices containing the normalisation and hidden layer experiments. It is clear that SCG has a preference for standardisation and RPROP prefers linear normalisation, but the reason for this is unknown. Overall, SCG performs better than RRPOP which lead to the conclusion that the more sophisticated search for the local minimum is effective. This is clearly visible in figure 4.5 which compares the RPROP and SCG training errors. The training error of the SCG network descends much faster, compared to the gradual descendance of RPROP.

## 4.6 Conclusion

In this chapter we have tried to find an optimal configuration for the different parameters of a feed-forward neural network. Normalisation of the inputs has shown to be essential for good performance. The column wise variants showed better results than the row oriented ones. The choice for linear normalisation or standardisation depends on the selected learning algorithm. Because the scaled conjugate gradient (SCG) learning algorithm has proved to outperform resilient propagation (RPROP) the choice for standardisation is well motivated. The property that standardisation is more robust also speaks in its favor.

Our experiments with different hidden layer configurations have lead to the conclusion that a number of 20 hidden nodes is enough for good performance. More hidden nodes or the addition of multiple hidden layers only results in an increase of performance for the RPROP trained networks. The estimated requirement of a relative error under 1 % is easily met by the best performing single layer SCG as can be seen in figure 4.6. The neural network with the configuration of table 4.1 is selected to replace  $F$  in case the support vector machines fail to perform better.

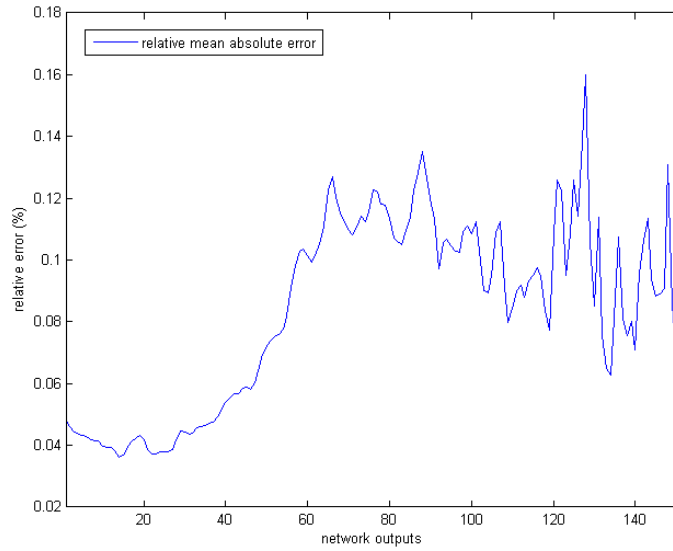


Figure 4.6: Relative mean absolute error optimal single layer neural network.

property	value
normalisation method	Column-wise standardisation
number of training epochs	5000
number of hidden layers	1 of size 20
learning algorithm	Scaled conjugate gradient (SCG)
m.a.e. test set	0.003702
relative m.a.e. test set	0.083915 %

Table 4.1: Optimal configuration single hidden layer network.



# Chapter 5

## Support vector machine approach

---

This chapter introduces support vector machines and contains experimental results of this technique applied to the learning of the forward model  $F$ . A linear separable binary classification problem is used to explain a separating hyperplane and how it can be found. This theory is extended to the classification of non linear separable data. A transformation maps the input space to a much higher dimension in which the data is linear separable again, and the problem can be solved in the same manner as before. Thereafter, the two support vector regression variants and their parameters are described. The outcome of experiments of both variants trained on the standard mapping of  $F$  can be found in the last section.

### 5.1 Support vector machines

Support vector machines were originally designed to solve classification problems [Cortes and Vapnik, 1995]. A binary classification problem is easily solved when the input data is linear separable, meaning that a linear decision function can be found that separates the two classes. Figure 5.1 illustrates such a problem. Every data point  $\mathbf{x}$  either belongs to class  $y_i = 1$  or  $y_i = -1$ . Equation 5.1 shows how the linear decision function  $\mathbf{w} \cdot \mathbf{x} + b = 0$  separates the two classes. Every configuration of  $\mathbf{w}$  and  $b$  that satisfies this constraints solves the problem, but the optimal solution is found when the decision function separates the data by the largest possible margin.

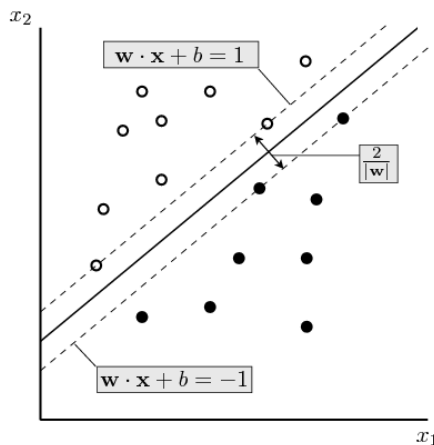


Figure 5.1: Binary classification problem with optimal separating hyperplane.

$$\begin{cases} \mathbf{w} \cdot \mathbf{x} + b > 0 & \text{for } y_i = 1, \\ \mathbf{w} \cdot \mathbf{x} + b < 0 & \text{for } y_i = -1 \end{cases} \quad (5.1)$$

The desired linear decision function is often called the optimal separating hyperplane. This hyperplane is depicted in figure 5.1 by the dashed lines and its width is equal to  $2/|\mathbf{w}|$ . This can be seen as follows. First normalise equation 5.1 to the form of equation 5.2, which in turn can be reformulated to equation 5.3. Consider a data point  $\mathbf{x}$  of class  $y_i = 1$  that lies on the left border of the hyperplane. This point satisfies  $\mathbf{w} \cdot \mathbf{x} + b = 1$  and thus can be written as  $\mathbf{x} = (1 - b)/\mathbf{w}$ . The vector from an arbitrary point  $\mathbf{x}_p$  on line  $\mathbf{w} \cdot \mathbf{x} + b = 0$  to  $\mathbf{x}$  is given by  $\mathbf{x} - \mathbf{x}_p = (1 - b/\mathbf{w}) - (b/\mathbf{w}) = 1/\mathbf{w}$ . The distance to the line can be found by projecting this vector on the hyperplanes unit normal vector  $\mathbf{w}/|\mathbf{w}|$  which results in  $1/|\mathbf{w}|$ . The same holds for a point of the other class on the otherside of the hyperplane, which adds up to a total width of  $2/|\mathbf{w}|$ . Finding the optimal separating hyperplane is a matter of maximising its width. This optimisation problem can be formulated as the minimisation problem of equation 5.4. There exist efficient methods to solve such a quadratic minimisation problem given a not too high dimension of the input.

$$\begin{cases} \mathbf{w} \cdot \mathbf{x} + b \geq 1 & \text{for } y_i = 1, \\ \mathbf{w} \cdot \mathbf{x} + b \leq -1 & \text{for } y_i = -1 \end{cases} \quad (5.2)$$

$$y_i((\mathbf{w} \cdot \mathbf{x}) + b) \geq 1 \quad (5.3)$$

$$\begin{aligned} \min \quad & \frac{1}{2}(\mathbf{w} \cdot \mathbf{w}) \\ \text{subject to:} \quad & y_i((\mathbf{w} \cdot \mathbf{x}) + b) \geq 1 \\ & i = 1, \dots, l \end{aligned} \quad (5.4)$$

Sometimes a problem is not completely linear separable, then we look for a hyperplane which correctly separates most data points. This can be accomplished by use of a soft margin hyperplane. Two slack variables  $\xi_i, \xi_i^*$  weaken the constraints of the optimisation problem. Points that lie inside the margin, or on the wrong side of the hyperplane are tolerated but result in a positive value of the penalty term (linearly correlated to the size of margin violation). Equation 5.5 shows the new problem formulation. The parameter  $C$  that is configured by the user regulates the trade off between the margin and training error. Higher values of  $C$  lead to “harder” margin constraints.

$$\begin{aligned} \min \quad & \frac{1}{2}(\mathbf{w} \cdot \mathbf{w}) + C \cdot \sum_{i=1}^l \xi_i \\ \text{subject to:} \quad & y_i((\mathbf{w} \cdot \mathbf{x}) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, i = 1, \dots, l \end{aligned} \quad (5.5)$$

Now we come to the application of this technique to classification problems which cannot be sufficiently well separated by a linear decision function. As mentioned before the data points will be transformed by a non-linear transformation  $\phi(x_i)$  to a high dimensional feature space. In this space we will search for a (linear) optimal separating hyperplane. A difficulty with a high dimensional search space is that the quadratic optimisation problem is too time consuming to solve. This problem is avoided by rewriting the quadratic optimisation problem and the resulting linear decision function to their dual formulation, given in equation 5.6 and 5.7 respectively. The most important property of the dual formulation is that all occurrences of  $\phi(x_i)$  occur in the form of dot-products. With the additional requirement of equation 5.8 on the transformation function, the new quadratic programming problem can be solved in the high dimensional space without explicitly doing the mapping to it. Two common types of transformation functions, also known as *kernel functions*, are the polynomial kernel and the radial basis function (RBF) kernel. They are given in equation 5.9 and 5.10 respectively. Finding the optimal kernel for the problem at hand is a matter of experimentation.

$$\begin{aligned}
\mathbf{max} \quad & \sum_i \alpha_i - \frac{1}{2} \cdot \sum_i \sum_j \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i) \phi(\mathbf{x}_j) \\
\text{subject to:} \quad & 0 \leq \alpha_i \leq C, i = 1, \dots, l \\
& \sum_{i=1}^l y_i \alpha_i = 0
\end{aligned} \tag{5.6}$$

$$\sum_{i=1}^l y_i \alpha_i^* \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}) + b \tag{5.7}$$

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \phi(\mathbf{x}_j) \tag{5.8}$$

$$K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle^d \tag{5.9}$$

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right) \tag{5.10}$$

The details of the derivation of the dual presentation are beyond the scope of this thesis. It suffices to say that the problem is defined as a Lagrangian with coefficients  $\alpha_i$ . Software packages exist that can solve such a quadratic programming problem and find the solution  $\alpha^*$ . The elements of  $\alpha^*$  are only greater than zero for the points that lie on the border of the separating hyperplane. These points are called *support vectors*, denoted as  $\phi(\mathbf{x}_i)$  in equation 5.7, and define the separating hyperplane. This technique is named after these points because they are such an important part of the solution.

## 5.2 Support vector regression

It is possible to apply the principles of the support vector machine technique to regression problems [Smola and Schoelkopf, 1998]. Where support vector machines find a linear decision function that optimally separates the data, a support vector regressor finds a linear function that optimally describes the relation between the input and output pairs in the training data. An  $\epsilon$ -insensitive loss function is introduced to obtain a problem similar to the classification example of the previous section. This can be visualised as a tube of width  $2\epsilon$ , where  $\epsilon$  is the amount of tolerated deviation from the regression function. The function that captures most points into its tube is what we are looking for. Figure 5.2 displays an optimal linear approximation function for the given input set. As before, this function is defined by the support vectors lying on the edge of the tube.

The minimisation problem of support vector machines can be rewritten as in equation 5.11. The constraints are only violated by patterns that result in an error  $> \epsilon$ . The slack variables  $\xi_i, \xi_i^*$  are again present to allow some points to lie outside the tube. The minimisation of  $|w|$  should be interpreted as finding a regression function with maximal *flatness*.

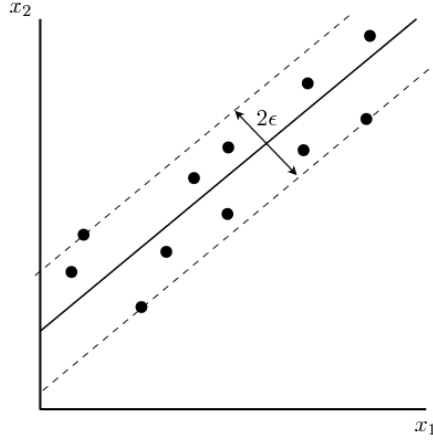


Figure 5.2:  $\epsilon$ -insensitive tube.

$$\begin{aligned}
 \min \quad & \frac{1}{2}(\mathbf{w} \cdot \mathbf{w}) + C \cdot \sum_{i=1}^l (\xi_i + \xi_i^*) \\
 \text{subject to:} \quad & (\mathbf{w} \cdot \mathbf{x}) + b - y_i \leq \epsilon + \xi_i \\
 & y_i - (\mathbf{w} \cdot \mathbf{x}) + b \leq \epsilon + \xi_i^* \\
 & \xi_i \geq 0, i = 1, \dots, l
 \end{aligned} \tag{5.11}$$

The value of  $\epsilon$  is of great influence on the performance of a support vector regressor. If it is set too high the tube is too wide and easily captures all data, resulting in large errors. In the extreme case the function outputs just the mean of the training data. If  $\epsilon$  is too small it is impossible to find a tube containing enough points to result in a good approximation function. Further it is important to see that the value of  $\epsilon$  is related to the value of  $C$  in the sense that a small  $\epsilon$  will require higher error tolerance, which can be introduced by lower values of  $C$ . This form of support vector regression is known as  $\epsilon$ -support vector regression ( $\epsilon$ -svr).

$$\begin{aligned}
 \min \quad & \frac{1}{2}(\mathbf{w} \cdot \mathbf{w}) + C \cdot \left( \nu \epsilon + \sum_{i=1}^l (\xi_i + \xi_i^*) \right) \\
 \text{subject to:} \quad & (\mathbf{w} \cdot \mathbf{x}) + b - y_i \leq \epsilon + \xi_i \\
 & y_i - (\mathbf{w} \cdot \mathbf{x}) + b \leq \epsilon + \xi_i^* \\
 & \xi_i \geq 0, i = 1, \dots, l
 \end{aligned} \tag{5.12}$$

Finding the optimal value for  $\epsilon$  is a difficult problem. Another variant, known as  $\nu$ -support vector regression ( $\nu$ -svr), automatically minimises  $\epsilon$  by adding the constraint  $\nu \epsilon$  to the optimisation problem. The result is given in equation 5.12. When rewritten to its dual formulation the  $\epsilon$  disappears as parameter and the new parameter  $\nu$  needs to be configured instead. The details of this derivation and the relationship between  $\epsilon$  and  $\nu$  are not part of this thesis, but can be read in [Chang and Lin, 2002]. Important practical properties of  $\nu$  are the following:

- $\nu$  has a direct influence on the amount of support vectors of the solution.
- $\nu$  is a value in the interval  $[0,1]$ .
- There always exists a  $\nu$  that achieves the same result as the optimal value of  $\epsilon$  in  $\epsilon$ -svr.

These properties make  $\nu$  easier to configure than  $\epsilon$ . Often  $\nu$ -svr yields better results than  $\epsilon$ -svr, which will be investigated for our problem in the next section.

## 5.3 Test results

Our regression problem has multiple inputs and outputs. The publication [Vazquez and Walter, 2003] proposes a form of multi-output svr but there is no software available that implements this method. The software package Spider provides a meta method through which a separate model can be trained for each output. We decided to use this package to get a first indication of the capabilities of svr.

### $\epsilon$ -support vector regression

Appendix B.1 contains the results of experiments with  $\epsilon$ -svr. First the effect of the size of the training set is tested. In theory more samples would result in better accuracy, because more points are considered in the process of positioning the  $\epsilon$ -insensitive tube. The results support this theory. This is useful to know, because the experiments from now on are carried out with a training set of 1000 samples, in order to reduce computation time. Once the optimal models have been found they are trained on the complete set, which will result in a moderate performance improvement.

Thereafter the different forms of normalisation have been tested. Support vector regression has a strong preference for column wise linear normalisation. This also was noticeable in the amount of training time. The standardised normalisation variants required more than twice as much time to train. Column wise linear normalisation will be used in all future experiments.

Thusfar all experiments were carried out with a polynomial kernel of degree 3, C set to 0.1 and  $\epsilon$  set to 0.01. The third table shows the results for variations of C and  $\epsilon$ . As explained in the previous section these two values are related. Higher values of  $\epsilon$  logically require higher values for C, and for lower values of  $\epsilon$  it is the other way around. We varied  $\epsilon$  between 0.0001 and 0.1 and for each of these configurations multiple values for C were tested. The relationship between C and  $\epsilon$  is visible in the results. For example an increase of C in combination with  $\epsilon=0.1$  is leading to better results, but for  $\epsilon=0.0001$  an decrease of C is beneficial. We achieved the best results with  $\epsilon$  and C set to 0.001 and 0.001 respectively.

The last two tables show experiments with rbf kernel and polynomial kernels. The best results are produced by a rbf kernel with  $\sigma$  equal to 2 and a polynomial kernel of degree 3. These two variants (in combination with good values for C and  $\epsilon$ ) are trained on the complete training set and indeed result in the best accuracy achieved thusfar. However this accuracy is still a lot lower than that of the neural networks trained on the same set. For example the absolute relative error of the best performing network was equal to 0.083915 %, which is a lot lower than the best relative error of svr: 0.455350%. Moreover, the improvements between the different configurations are small. It does not seem likely that  $\epsilon$ -svr will outperform the neural networks in any configuration.

### $\nu$ -support vector regression

As explained in the previous section the configuration of  $\nu$ -svr is easier, because it finds the optimal width of the  $\epsilon$ -insensitive tube by itself. Appendix B.2 contains two tables with experiments of this svr variant. The value for  $\nu$  is known to be between 0 and 1. Therefore we kept C constant and varied  $\nu$  over this interval. The performance improves for higher values of  $\nu$ .

Then, we have tested the effect of different values for C and we found an optimal configuration with  $\nu$  and C set to 1 and 0.1 respectively. Clearly the performance of  $\nu$ -svr is better than that of  $\epsilon$ -svr, but it

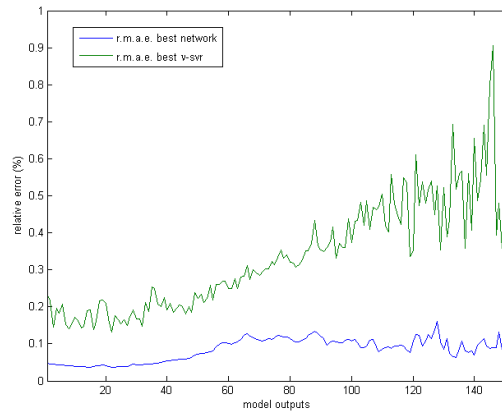


Figure 5.3: Comparison of relative mean absolute error of best performing network and  $\nu$ -svr.

fails to come close to the neural network performance of the previous chapter. The difference between the different configurations are present, but again it does not seem likely that the the level of accuracy of the neural networks will be reached. However, the relative mean absolute errors are below 1 % and therefore they satisfy the requirement to function as a replacement for the forward model.

## 5.4 Conclusion

Support vector regression performs not as good as neural networks looking at the tested configurations. The dissapointing results can partially be explained by the fact that we used a combined single output model instead of a true multiple output regression model. The benefit of the latter is that the optimisation problem would be bound by constraints based on the errors of all outputs, which is a (much) better representation of the approximated function. Nevertheless, in chapter 8 will be shown that a combined single output model of neural networks is able to reach a mean relative error of  $\approx 0.08$  compared to a mean relative error of the best  $\nu$ -svr of  $\approx 0.33$ . Figure 5.3 shows a plot of these two errors.

A more theoretically based approach to finding the optimal parameter configuration might lead to better results than the empirical approach followed in this thesis. The paper [Cherkassky and Ma, 2003] gives some hints about the relation of the noise distribution over the training set and the optimal value for  $\epsilon$ . The presented experimental results should be interpreted as an initial assessment more than a definitive conclusion. However, for the rest of this thesis neural networks become the machine learning technique of choice, because of their better performance.

# Chapter 6

## The Jacobian

---

This chapter contains a description of the research to the most effective and accurate way of reproducing the jacobian matrix  $\mathbf{J}$ . It is divided in two parts. Sections 6.1 - 6.3 focus on the retrieval of the Jacobian from a neural network trained on the forward model. The remaining sections describe and evaluate an attempt to learn the derivative with a composed neural network architecture from a training set with derivative information.

### 6.1 Jacobian derivation

In the process of ozone profile retrieval the forward model  $F$  is expanded in a first order Taylor series around an ozone profile  $x_0$  (equation 6.1). The expansion point  $x_0$  is adapted in every step of the retrieval process in an iterative manner. The Jacobian  $\mathbf{J}$  is a matrix of size  $m \times n$  given a function of  $n$  inputs and  $m$  outputs. An element  $J_{ij}$  contains the partial derivative  $\partial F_i / \partial \mathbf{x}_j$ , in which  $F_i$  is the  $i$ 'th output and  $\mathbf{x}_j$  the  $j$ 'th input, with  $i = 1 \dots m$  and  $j = 1 \dots n$ .

$$F(x) = F(\mathbf{x}_0) + \mathbf{J}(\mathbf{x} - \mathbf{x}_0) + e \quad (6.1)$$

Intuitively the easiest way of calculating the Jacobian of a learned function  $F(x)$  is by differentiating its approximator. In this case the neural network. Using the terminology introduced in chapter 4 the network function of a network consisting of 3 layers can be described by equation 6.2. In theory the derivative of this function provides the derivative of the function on which the network is trained.

$$o_k^3 = \varphi_k^3 \left( \sum_j w_{kj}^3 \varphi_j^2 \left( \left( \sum_i w_{ji}^2 o_i^1 \right) + \theta_j^2 \right) + \theta_k^3 \right) \quad (6.2)$$

$$\frac{\partial o_k^3}{\partial o_i^1} = \underbrace{\varphi_k^3}_{1} \cdot \left( \sum_j \overbrace{w_{kj}^3 \cdot \varphi_j^2}_{2} \cdot \underbrace{w_{ji}^2}_{3} \right) \quad (6.3)$$

Equation 6.3 contains the partial derivative of an arbitrary output with respect to an arbitrary input of a network with 3 layers. As described by [Lee and Oh, 1997] the following procedure extracts these derivatives from a trained network. This algorithm calculates the partial derivative in a *backwards pass* similar to the backpropagation of errors in standard backpropagation [Hertz *et al.*, 1990]. Instead of backpropagating the error through  $\delta$ 's, here derivative information is backpropagated through jacobian deltas  $\delta^{jacob}$ .

The braces in equation 6.3 accompany the equally numbered steps in the enumeration below.

1. Calculate the jacobian deltas  $\delta^{jacob} = \varphi_k^L$  of the output layer<sup>1</sup>. This term reduces to one in case of

---

<sup>1</sup>In standard backpropagation the  $\delta$  is calculated by multiplying this value with the mapping error.

a linear activation function on the output nodes.

2. Calculate the deltas from the hidden layer(s) as usual with the difference that they are based on the jacobian output deltas.
3. The summation  $\sum_j w_{ji}^2 \cdot \delta_j^2$  of the deltas in the lowest hidden layer with the input weights coming from input node  $i$  results in the partial derivative  $\partial o_k^3 / \partial o_i^1$ .

For testing purposes a network consisting of 2 input, 10 hidden and 2 output nodes has been trained on 100 patterns of the function given in equation 6.4. The inputs were randomly taken from domain  $[0, 1]$  and the outputs calculated by the target function. Figure 6.1 graphs the 4 partial derivatives of the network function for each of the training patterns. Appendix C.1 contains the Matlab source code that was written to retrieve the Jacobian from the neural network.

$$\begin{cases} y_1 = 6x_1 + 2x_2 \\ y_2 = 4x_1 - 6x_2 \end{cases} \quad (6.4)$$

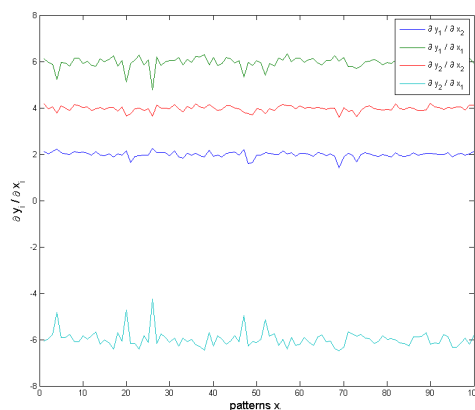


Figure 6.1: Partial derivatives derived from network trained on 100 samples of equation 6.4

## 6.2 Denormalisation

In Chapter 4 the various normalisation methods of the inputs have been described and evaluated. The type of normalisation has its influence on the function that is learned by the neural network: instead of learning a function  $f(x)$  a transformation of  $x$  by a function  $n(x)$  leads to the training of function  $f(n(x))$ . Obviously the derivative  $f'(n(x))$  is not equal to  $f'(x)$ . By application of the chain rule the derivative can be *denormalised* to the original. Equation 6.5 shows the derivation.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n} \frac{\partial n}{\partial x} \quad (6.5)$$

The first part of the right side of equation 6.5 is calculated by the network. The second part is the derivative of the applied normalisation function. Equations 6.6 and 6.7 show the denormalisation factors for linear normalisation and standardisation.

$$\frac{\partial n}{\partial x} = \frac{1}{\max - \min} \quad \text{for } n(x) = \frac{x - \min}{\max - \min} \quad (6.6)$$

$$\frac{\partial n}{\partial x} = \frac{1}{\sigma} \quad \text{for } n(x) = \frac{x - \mu}{\sigma} \quad (6.7)$$

### 6.3 Results

The derivation method was applied on a neural network trained on the forward model. The results are reasonably good on one half of the Jacobian, but very poor on the other. Figure 6.2 shows the target and network partial derivatives on three wavelengths as function of the altitude. These wavelengths are evenly divided over the total spectrum. Large oscillations occur only on the higher altitudes.

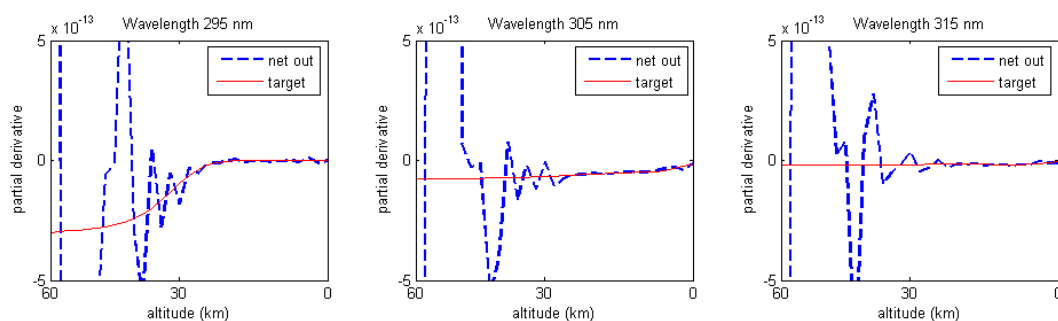


Figure 6.2: Partial derivatives on three wavelengths, large oscillation for altitudes higher than 30 km.

The explanation of this large oscillations can be found in the construction of the training set. The ozone densities in the first 30 km of the atmosphere are measured by weather balloon sondes. These highly detailed measurements provide enough information to let the network interpolate to a degree where derivation of the Jacobian, containing information about the behaviour of a function around a certain point, is possible. The higher located layers are constructed from a *climatology* with 12 variations for every month of the year. The level of information in these layers seems insufficient for the network to learn the function well enough for Jacobian derivation.

#### Generating extra samples

To test the mentioned hypotheses extra training samples containing information about the derivative have been generated. For each training pattern  $(\mathbf{x}, \mathbf{y})$  there are  $n$  new samples generated with the method of figure 6.3. A small value  $\Delta x$  is added to element  $j$  of vector  $\mathbf{x}$  to generate  $\mathbf{x}^{new}$ . The new output pattern is given by  $\mathbf{y}^{new} = \mathbf{y} + ((\partial F_i / \partial \mathbf{x}_j) \cdot \Delta \mathbf{x}, \dots, (\partial F_m / \partial \mathbf{x}_j) \cdot \Delta \mathbf{x})$ . Before the sample generation procedure is applied the Jacobian data needs to be normalised to the derivative of the function the network is learning. Division by the appropriate normalisation factor given in equations 6.6 and 6.7 does accomplish this.

Figure 6.4 shows the derivative (and the target) on the same three wavelengths for different values of  $\Delta x$ . Clearly, the value of  $\Delta x$  has a large influence on the effectiveness of this method. With  $\Delta x = 0.1$  the best approximation was accomplished. Larger or smaller values for  $\Delta x$  show increasing forms of oscillation.

The generated samples help the neural network learning (derivative) information about the top half of the atmospheric layers. Unfortunately this method generates new samples under the assumption the function

for each sample  $(\mathbf{x}, \mathbf{y})$

for  $1 \leq i \leq n$  where  $n = |\mathbf{x}|$ , generate a sample of the form  $(\mathbf{x}^{new}, \mathbf{y}^{new})$

for  $1 \leq j \leq n$

$\mathbf{x}_j^{new} = \mathbf{x}_j + \delta_{ij} \Delta x$ , with  $\delta_{ij}$  the Kronecker  $\delta$

end

for  $1 \leq k \leq m$ , where  $m = |\mathbf{y}|$

$\mathbf{y}_k^{new} = \mathbf{y}_k + \mathbf{J}_{ik} \cdot \Delta x$

end

end

end

Figure 6.3: Sample generation with use of derivative information

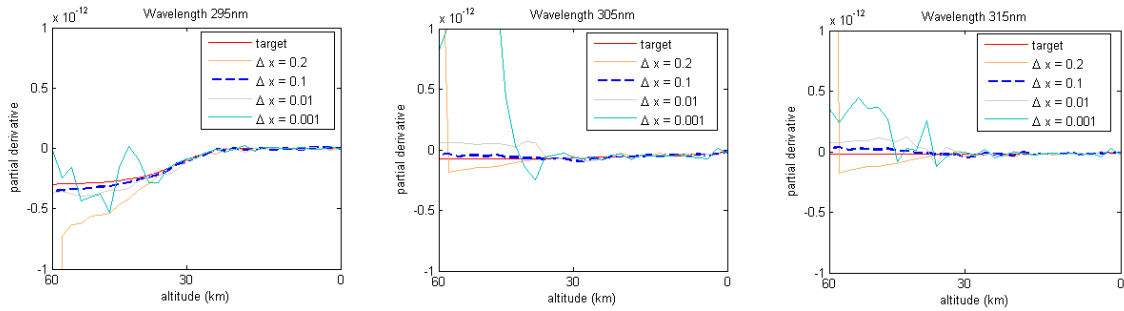


Figure 6.4: Partial derivatives for different values of  $\Delta x$  on three wavelengths

behaviour is linear in a neighbourhood of size  $\Delta x$  around a point  $x$ . Because of the non-linear nature of the forward model a significant level of noise is introduced by the added samples. An explanation for the fact that a rather large value for  $\Delta x$  gives the best performance could be that for smaller values of  $\Delta x$  the natural noise tolerating properties of neural networks make the network “ignore” the new samples. On the other hand too large values for  $\Delta x$  result in a too high assumption of linearity of  $F(x)$  which is not correct.

This method has provided more insight in the performance of the network but does not seem to be a satisfying way of calculating the Jacobian.

## 6.4 Learning the Jacobian

An alternative to deriving the Jacobian from a network trained on  $F(x)$  is to let another network learn it. This would require a network with  $n \times m$  outputs. In case of 31 inputs and 151 outputs this would result in 4681 outputs. Because a network of this size requires a lot of resources and training time the decision has been made to divided the problem up into 31 sub problems. This problem can be solved by a composed neural network architecture of 31 networks, each of them trained on 1 column of the Jacobian. All networks will have the usual  $n = 31$  inputs. Matlab scripts have been written to provide an easy interface to the training and simulation functionality of the composed network architecture.

wavelength	m.a.e. derived J	m.a.e. derived J + generated samples	m.a.e. learned J
295 nm	1.26522047E-13	4.03467964E-14	4.81863512E-15
305 nm	5.93273870E-14	3.57948826E-14	3.47122637E-15
315 nm	1.68692209E-14	8.60611937E-15	7.15887473E-16

Table 6.1: Mean absolute error (m.a.e) and mean relative error (m.r.e.) of a derived and learned Jacobian.

A number of experiments have lead to a chosen network configuration of 15 hidden nodes per network and scaled conjugate gradient as learning algorithm. Further the inputs are column-wise normalised using standardisation. The outputs have been linearly scaled to an order of magnitude around  $1 \times e^0$ . The latter is done because learning of a function with very small target values leads quickly to very small mean square error values. To avoid endless searching on very flat error surfaces the scaled conjugate gradient learning algorithm has an extra stopping condition, besides reaching an error of 0, in the form of a minimum value of the gradient. This minimum value is by default equal to  $1 \times e^{-6}$  and without rescaling of the output values this stopping condition is triggered prematurely. Consequently the network outputs have to be scaled back afterwards.

### Test results

Table 6.1 shows a comparison of the performance of a derived Jacobian from a network trained on the standard training set, a derived Jacobian from a network trained on the standard set with addition of generated samples and of the composed architecture trained with Jacobian training data. Concerning the derived Jacobian it is clear that the generated samples result in better performance. As visualised in the previous section the generated samples improve the performance especially on the higher altitudes. However these regions are still more problematic resulting in the highest absolute errors in these parts of the derivative matrix. Another disadvantage of the addition of extra generated samples is that the performance on the standard mapping decreases.

The composed architecture produces a significantly lower error and shows a higher level of consistency over the different altitudes. Comparison of the absolute errors of a derived and learned Jacobian in the most troublesome regions of the Jacobian showed that the learned Jacobian outperformed the derived Jacobian by a factor 10. The relative errors of the learned Jacobian in these regions are around 5 %. This makes it clear that, from these alternatives, Jacobian learning is the best option for producing the derivative of the forward model. Chapter 9 shows experimental results of ozone profile retrieval with this technique integrated. Then, it will become clear if the accuracy of the learned  $\mathbf{J}$  is high enough.

## 6.5 Conclusion

Reproduction of the Jacobian has shown to be a more difficult problem than learning the standard mapping. Beforehand, deriving the Jacobian from a trained network seemed an easy solution to this problem, but this proved to be infeasible in practise. We have developed a sample generation algorithm to add derivative information to the training set. This method proved that the large errors in one half of the Jacobian were created by a lack of detailed information about the contents of the upper half of the atmosphere. Although not resulting in good enough results for real application this method did uncover the source of the problem leading to the conclusion we had to search for another solution. It was found in the learning of the Jacobian by a composed architecture. Because of the better results this method has been selected to replace  $\mathbf{J}$ .



## Chapter 7

# Dimensionality reduction

---

The dimension of a data set is defined by the number of variables. In general learning algorithms need more time and have greater difficulties learning high dimensional problems. For this reason every dimensionality reduction of the input variables is very welcome. This chapter starts with the explanation of the *curse of dimensionality*, which gives a theoretical motivation for the disadvantages of high dimensional data. Principal component analysis is explained afterwards followed by a section containing the performance of a neural network trained on different reduced training sets transformed by principle component analysis. Finally references to other, more advanced dimensionality reduction algorithms are given as a suggestion for future research.

### 7.1 The curse of dimensionality

The curse of dimensionality is a term that was introduced by [Bellman, 1961] and it is a term often used in the field of data mining to emphasise that adding more input variables increases the dimensionality of the space in which the algorithm is searching for the global optimum. Unless these new variables add truly useful information they should be left out. Bellman illustrates this with the example of a cartesian grid of spacing  $1/10$  on the unit cube. If this cube represents our search space it would contain  $10^{10}$  points in case of 10 dimensions and  $10^d$  in case of  $d$  dimensions. Bellman translates this to a function approximation problem by looking at it as a search in a search space defined by crude discretisation of the variables. If the approximated function has  $d$  dimensions with a certain defined smoothness<sup>1</sup> a number of evaluations in the order of  $(1/\epsilon)^d$  is needed to obtain an approximation scheme with uniform approximation error  $\epsilon$ . This should be interpreted as an exponential increase in computation time, for every added variable.

Although Bellmans problem definition of function approximation is a bit simple, more advanced learning algorithms have a significant influence on the complexity, the curse of dimensionality is undeniably existant. For example the number of needed weights in a neural network grows in case of adding more inputs. Intuitively every input should add a portion of information not covered by any other input and therefore highly correlated inputs are considered useless. An ideal dimensionality reduction algorithm removes all unneeded correlation without losing any information from the processed data.

### 7.2 Principle component analysis

Principle component analysis is an unsupervised dimensionality reduction technique, it reduces the dimension of the inputs without considering the accompanying outputs. This method belongs in the category of *feature extraction* techniques meaning that the new set of inputs is a transformation of the original variables, instead of a selection of the most informative inputs.

---

<sup>1</sup>The smoothness of a function is defined by a constant value  $K$  which describes the maximum slope between two points of this function. Thus,  $|f(x) - f(y)| = K|x - y|$

Data in a vector space is defined with respect to a certain basis, normally the standard basis. The amount of *variance* measured along the axes of the basis provides information about the amount of information that can be preserved when projecting the data on these axes. Projecting data on a subset of the basis vectors is a form of dimensionality reduction. This is illustrated in figure 7.1 (a) where it is clear that the projection on the  $x$  axis preserves more of the original variance compared to the projection on the  $y$  axis. Figure 7.1 (b) shows even a better variance preservation in case of projection on line  $l$  defined by the linear relationship between variable  $x$  and  $y$ . If the data set is transformed to a basis consisting of  $l$  and another vector  $m$  orthogonal to  $l$ , reduction to the  $l$  axis would preserve most of the variance (figure 7.1(c)). This is the idea of principle component analysis, it transforms the data set to a basis of which the axes preserve the greatest amount of variance. These orthogonal basis vectors are the *principle components* and are ordered by the amount of explained variance. Data reduction takes place by eliminating the dimensions in which the least amount of variance is explained.

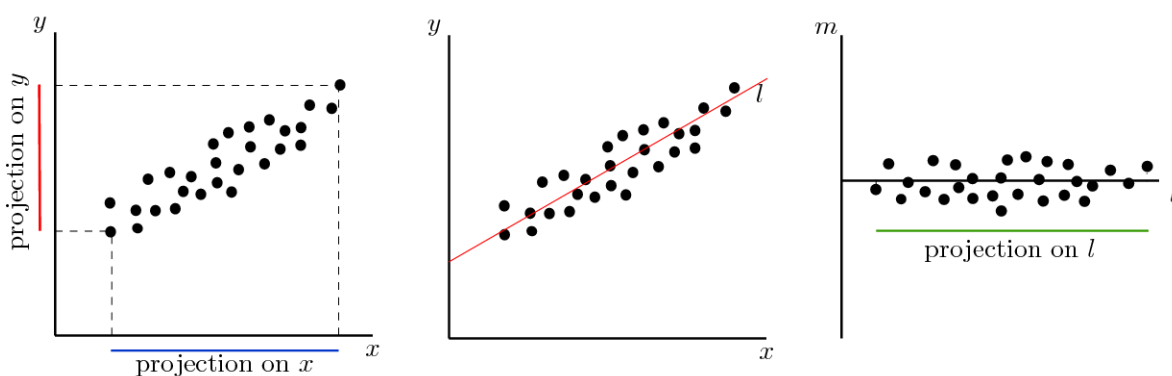


Figure 7.1: (a) Projection on the  $x$  and  $y$  axis. (b) Optimal variance preserving projection line. (c) Data set transformed to new basis.

A more precise description of this technique is the following. The data set  $X$  is a matrix of size  $m \times n$  containing the  $n$  dimensional patterns  $\mathbf{x}$  in its  $m$  rows.

1.  $X$  is standardised per column to be able to make a fair comparison between variables of different measures.
2. Calculate the co-variance matrix  $C$ . The co-variance matrix contains information about the linear relation between each pair of variables in  $X$ .
3. Find the eigenvectors and eigen values of the covariance matrix. These eigenvectors are the principle components, which will form the basis of the transformed data.
4. Sort the eigenvectors in order of decreasing eigen value. A high eigenvalue equals a large amount of explained variance.
5. Take the sum of the  $n$  eigenvalues and select a set of  $p < n$  eigenvalues which sum up to more than, for example, 95 % of the total sum. The  $p$  eigenvectors belonging to the selected eigenvalues form the *reduced* basis of the new data.
6. Transform the old data to the new basis. If  $P$  contains the selected principle components, apply the transformation:  $X_{new} = P^T \cdot X_{old}$ .

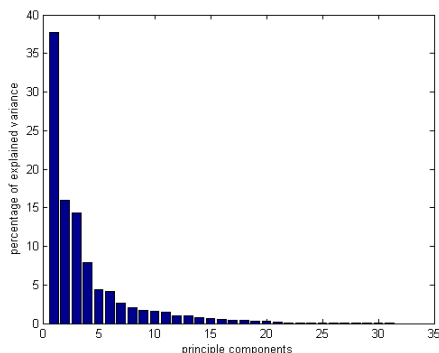


Figure 7.2: Percentage of explained variance by principle components

# principle components	m.a.e. over test set
10	0.012396
15	0.0061281
20	0.0062217
31	0.0056252

Table 7.1: Mean absolute error of neural networks trained on reduced training sets.

Once the principle component matrix  $P$  has been calculated it can be similarly applied to the test set. Principle component analysis is an extra pre-processing step, performed after normalisation.

### Test results

The training set used for learning of the standard mapping consists of input vectors of dimension 31. Principle component analysis has been applied on this set. Figure 7.2 shows a diagram of the explained variance by the 31 extracted principle components. The first 10 principle components explain more than 90% of the variance, the first 15 more than 95% and with 20 principle components 99% of the variance is preserved. Table 7.1 shows the mean absolute error produced by a neural network trained on a reduced training set created from respectively 10, 15, 20 and 31 principle components. The network had 20 hidden nodes and was trained using the scaled conjugate gradient algorithm for 1000 epochs. A network with the same configuration trained on this set without dimensionality reduction gives a mean absolute error of 0.005218. We can conclude that principle component analysis can reduce the number of inputs by a factor two, without resulting in serious performance loss. Further reduction to 10 components quickly leads to a large performance decrease.

## 7.3 Conclusion

Principle component analysis has shown to be an effective way of reducing the training set dimension. It did not result in a performance increase, but the amount of training time reduces significantly. This speed up is welcome especially in the case of learning the jacobian, which can take over 24 hours, depending on the size of the training set and number of epochs.

The addition of more input variables, which could possibly improve performance is now less of a drawback since this technique evaluates their “value” beforehand. An example is the solution of the simple scattering solution of the forward model. This is an intermediate result in the simulation of the measurement of which the calculation takes little time compared to the ultimate multiple scattering solution. This single scattering solution is a simulated measurement of 151 values. Obviously a dimensionality reduction procedure is useful in this case.

The linear nature of principle component analysis makes it one of the simpler techniques used for dimensionality reduction. It only uses linear relationships between the input variables, but in reality often there exist non-linear correlations which could also be used for dimensionality reduction. Techniques like kernel principle component analysis apply a non-linear transformation to a high dimensional feature space in which the non-linear relationships can be efficiently found by use of kernel functions. The idea is similar to that of support vector machines. In case there is need for better performance or further dimensionality reduction this would be an interesting alternative.

## Chapter 8

# Ensemble approach

---

It is well known that a combination of models can lead to an improvement of performance. Ensembles are combined models, for example a number of different neural networks. All participating models have a certain degree of influence on the result. In contrast to the modular architecture used for learning the Jacobian (chapter 6) where each participating model calculates a part of the output, here each model computes all outputs. In case of regression the ensemble output is a weighted average of the outputs of its members. This chapter starts with a theoretical explanation for the possible performance improvement. This involves the bias / variance decomposition and its extension to ensemble theory. Further, the two most popular ensemble methods known as bagging and boosting are introduced. This chapter concludes with the results of an ensemble, which is composed by bagging, that is trained on the forward model  $F$ .

### 8.1 Bias / variance decomposition

An intuitive explanation that speaks for the use of multiple (different) machine learning models is that a solution to a difficult problem is often found by consulting multiple experts with a different point of view on the problem. Machine learning algorithms with different configurations trained on different training sets can be seen as the different experts. A weighted average of the outputs can be seen as a mixing of opinions.

A theoretical foundation for the ensemble methodology is found by analysis of the error of a single model compared to that of the ensemble. This analysis has the same structure as the bias / variance decomposition of the error of a single model. This decomposition describes the *expected* mean square error  $E\{MSE\}$  given an unlimited supply of training data. It is described for a model with a single input and output, trained with samples of the form  $(x, y)$ , but the same holds for multiple in and outputs. An important insight is that there is a difference between the target function  $g(x)$  and the target values  $y$  in the training set. This difference is caused by noise and is (almost) always present. The models output given a sample  $x$  is denoted as  $f(x)$ .

$$\begin{aligned} E\{(f(x) - g(x))^2\} &= E\{(f(x) - E\{f(x)\} + E\{f(x)\} - g(x))^2\} \\ &= E\{[(f(x) - E\{f(x)\}) + (E\{f(x)\} - g(x))]^2\} \\ &= E\{(f(x) - E\{f(x)\})^2\} + (E\{f(x)\} - g(x))^2 + \end{aligned} \tag{8.1}$$

$$\begin{aligned} &2E\{(f(x) - E\{f(x)\})(E\{f(x)\} - g(x))\} \\ &= \text{Variance} + \text{Bias}^2 \end{aligned} \tag{8.2}$$

In the first line the expected model output  $E\{f(x)\}$  is added and subtracted within the squared term. In the third line the cross terms cancel to 0, because  $f(x) \cdot E\{f(x)\} = E\{(f(x))^2\}$  and  $g(x) \cdot E\{f(x)\} = E\{f(x) \cdot g(x)\}$ . This leaves only the variance and squared bias term. A more detailed decomposition can be found in [Chandra and Yao, 2006].

The *bias* term signifies how much the models output differs from the target function  $g(x)$ . This term will minimise in the training process, but never vanish completely because of the difference between the training data targets  $y$  and  $g(x)$ . The *variance* term can be interpreted as the models sensitivity to the training data. A very small variance indicates a behaviour strongly affected by the training data which might indicate a state of overfitting. The optimal bias / variance combination is the state in which the highest level of accuracy is achieved without a decrease of generalisation.

This problem known as the *bias / variance trade off* can be generalised to the *accuracy / ambiguity trade off* in ensemble theory, but first follows a short intuitive motivation for the use of ensembles. Each trained model has a certain dependence on the training set defined by the variance term in its error. This will result in good performance on samples in the test set which are similar to the training set, but performance on unseen samples not very similar to the training set is probably worse. By averaging the outputs of multiple models the dependence on the training set is weakened which results in a reduction of the errors variance. This only holds provided a (high) level of diversity of the ensemble members.

$$f_{ens} = \sum_i w_i f_i \quad (8.3)$$

The output  $f_{ens}$  of an ensemble is calculated as in equation 8.3. The error of the ensemble is given by the weighted mean square error of its members  $f_i$  (equation 8.4). This error can be decomposed in the same manner as before, by adding and subtracting  $f_{ens}$  inside the quadratic term. The elimination of the cross terms is omitted, but can be found in [Chandra and Yao, 2006]. Equation 8.5 holds because  $\sum_i w_i = 1$  and all  $w_i \geq 0$ . The reformulation to equation 8.7 shows that an increase of  $\sum_i w_i (f_i - f_{ens})^2$ , known as the *ambiguity term*, reduces the ensemble error with respect to the target function  $g(x)$ . A high diversity of the ensemble members results in an increase of this term. Please note that a too high level of diversity results in less accuracy of the individual models, increasing the first term on the right handside of equation 8.7 and undoing the effect of a bigger ambiguity term. This is what is known as the accuracy / ambiguity trade off.

$$\sum_i w_i (f_i - g(x))^2 = \sum_i w_i (f_i - f_{ens} + f_{ens} - g(x))^2 \quad (8.4)$$

$$= \sum_i w_i [(f_i - f_{ens})^2 + (f_{ens} - g(x))^2 + 2(f_i - f_{ens})(f_{ens} - g(x))]$$

$$\text{and, } \sum_i w_i (f_{ens} - g(x))^2 = (f_{ens} - g(x))^2 \quad (8.5)$$

$$\text{and, } \sum_i w_i 2(f_i - f_{ens})(f_{ens} - g(x)) = 0$$

$$= \sum_i w_i (f_i - f_{ens})^2 + (f_{ens} - g(x))^2 \quad (8.6)$$

hence,

$$(f_{ens} - g(x))^2 = \sum_i w_i (f_i - g(x))^2 - \sum_i w_i (f_i - f_{ens})^2 \quad (8.7)$$

## 8.2 A crosstraining ensemble

As pointed out in the previous section a high level of diversity on the models in your ensemble is desired. Diversity can be created by combining different types of models and by varying their configuration. Fur-

Ensemble members	m.a.e. ensemble	std. ensemble	m.a.e. NN	m.a.e. SVR
1	0.007355746	0.003193093	0.008077	0.013912
3	0.006002006	0.002487828		
6	0.005759845	0.002370543		
12	0.005585769	0.002285309		

Table 8.1: Mean absolute error (MAE) neural network only ensemble compared to best performing single models.

Model types	Ensemble members	m.a.e. ensemble	std. ensemble
NN, RBF, KNN	6	0.006062831	0.002661869
NN, KNN	6	0.006168657	0.002627079
NN, RBF	6	0.005974382	0.002591127
RBF, KNN	6	0.013705554	0.008064928

Table 8.2: Mean absolute error (MAE) varied models ensemble compared to best performing single models.

thermore models can be trained on different, preferably independent data sets. The two most popular methods for creating these training sets are called *bagging* and *boosting*. A practical introduction to both methods can be found in [Witten and Frank, 2005].

In this chapter an initial assessment on the usability of ensembles is made using the software package ENTOOL [C. Merkwirth, 2003] written for Matlab. This package implements a training scheme called *crosstraining* which is closely related to bagging. The training scheme consists of the following steps:

1. The data is divided in two sets, a training set and a test set.
2. Several different models are trained on the training set.
3. The models with the lowest error on the test set become ensemble members.
4. The data set is divided again in a way that the new set has minimal overlap with the former ones.
5. The procedure stops if the ensemble has the desired size.

This training scheme introduces diversity in the ensemble by training the models on different (independent) training sets. Unfortunately the ENTOOL package does not provide functionality for multi-output regression, therefore a *meta method* is implemented which trains a separate ensemble for each of the outputs. The source code can be found in Appendix C.2. The number of models of which an ensemble consists is determined by the number of folds of the crosstraining scheme in combination with a parameter which controls the amount of models selected in each of these folds. Table 8.1 shows the results for different ensemble sizes. A test set excluded from the crosstraining scheme was used to produce the mean absolute errors in the table. Results from the best performing single neural network<sup>1</sup> and support vector regressor are added for comparison. The first row describes a model consisting of ensembles with just one member, which equals a composed architecture of neural networks, each trained on a single output. Adding more models clearly improves performance, although the performance gain between 6 and 12 ensemble members is much lower compared to the improvement with respect to a single member ensemble. Figure 8.1 shows a comparison of the best performing ensemble with a (single) neural network trained with the RPROP algorithm for 1000 epochs<sup>2</sup>. An improvement over all outputs is visible.

<sup>1</sup>The network was trained with RPROP, because in ENTOOL the networks can only be trained with RPROP.

<sup>2</sup>The neural networks in the ensembles were also trained for 1000 epochs.

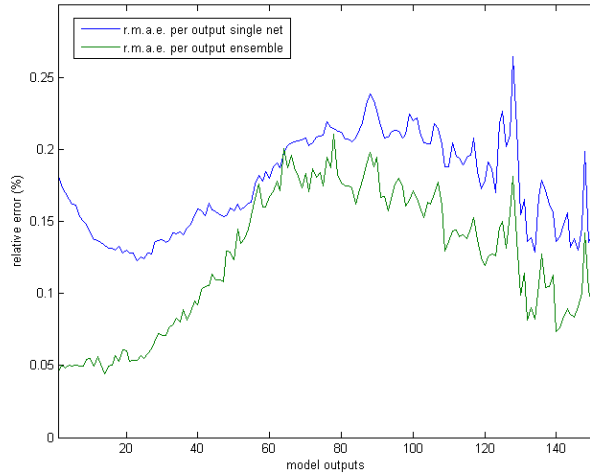


Figure 8.1: Relative mean absolute error per output of best performing ensemble and neural network.

The results in table 8.1 are taken from neural-network only ensembles. More diversity in an ensemble can be introduced by adopting different model types. Table 8.2 contains some experiments of ensembles consisting of neural networks (NN), k-nearest neighbour (KNN) regressors and radial basis networks (RBF). None of these combinations outperform the neural-net-only ensemble. It is obvious from the very poor results of the RBF, KNN combination that neural networks are best in learning this training set. Following the theory of the previous section the too low level of accuracy of the other two models brings a too high level of ambiguity in the ensemble resulting in a decreasing performance of the whole.

### 8.3 Conclusion

The results at the end of the previous section show that on average ensembles perform better than single models. Application of this technique to Jacobian learning, where improvement of the accuracy might be necessary, should seriously be considered in future research. This would result in a composed architecture of ensembles. More advanced training schemes, like boosting, which force new models to be trained on data on which other models perform below average seem an interesting alternative to the crosstraining scheme used thus far.

## Chapter 9

# Simulation on synthetic measurements

---

The techniques that are selected as the optimal replacement for the forward model and its derivative have been integrated into the existing ozone profile retrieval procedure. The source code that has been written for this purpose can be found in appendix D. The source code for exporting the networks out of matlab is listed in appendix C.3. Before showing an analysis of the effects of the replacement a brief explanation of the outcome of the retrieval without replacement is given. This includes the desired form of the averaging kernel and the tolerable values for chi-square. For a better interpretation we start with the replacement of  $F$ , leaving the derivative calculation untouched. Thereafter the effect of replacement of the Jacobian is investigated. This chapter ends with concluding remarks regarding the results and some suggestions for further research.

### 9.1 Interpretation of the results

A typical result of a retrieval using the forward model  $F$  and its Jacobian  $J$  is displayed in figure 9.1(a). The dotted line represents an ozone profile adopted from a radio sonde measurement at de Bilt, Netherlands (1 March 1996). This profile is used to simulated a measurement spectrum between 290 and 320 nm. The inversion of this measurement simulation provides the solid line, which shows much less vertical structures than the radiosonde profile. The corresponding smoothing of the retrieval results directly from the regularisation of the inversion, which is described by the averaging kernel. Applying the averaging kernel to the radiosonde profile yields the smoothed sonde profile (dashed line), which can be directly compared with the retrieval result.

As described in chapter 2 the averaging kernel  $A$  contains information about the sensitivity of the measurement on different altitudes. The degree of sensitivity is directly correlated to the amount of smoothing required to obtain a meaningful retrieval. Figure 9.1(b) shows row vectors of  $A$  which contain information about how the retrieved value at a certain altitude is correlated to the *real* ozone distribution at *all* other altitudes. For example the blue solid line contains information about the retrieved ozone distribution at a height of 1 km and this line show a negative sweep around 20 km. This means that there is a negative correlation between the retrieved value at 1 km and the real ozone distribution at 20 km. This makes the data interpretation more difficult but is a direct result from the regularisation. The inversion gives a DFS value of 4.80, which means that about 5 independent pieces of information can be retrieved from the measurement. A chi-square value of 0.9 is achieved, which indicates virtually no bias in the forward model, which is expected with a synthetic measurement retrieval. In case of real measurements often a chi-square of 4 and larger (up to 30) is achieved.

For integration in the ozone profile retrieval model one extra input variable had to be added to the training set. This *albedo* variable contains information about the reflectivity of the Earth surface. For each sample in the training data five different Earth reflectivity values are taken into account resulting in a factor 5 increase of the data. This variable introduces an extra complexity noticeable as a small decrease in the performance of the neural networks. This variable was not introduced before, because we did not expect

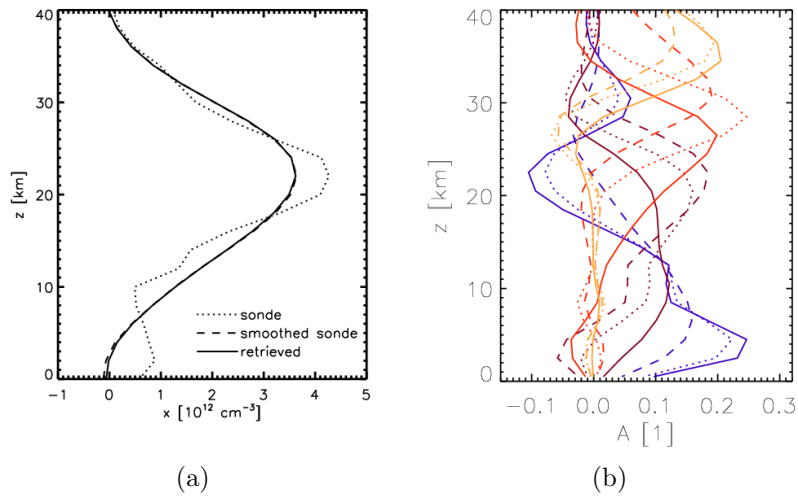


Figure 9.1: (a) A typical retrieved ozone profile. (b). Row vectors of averaging kernel  $A$

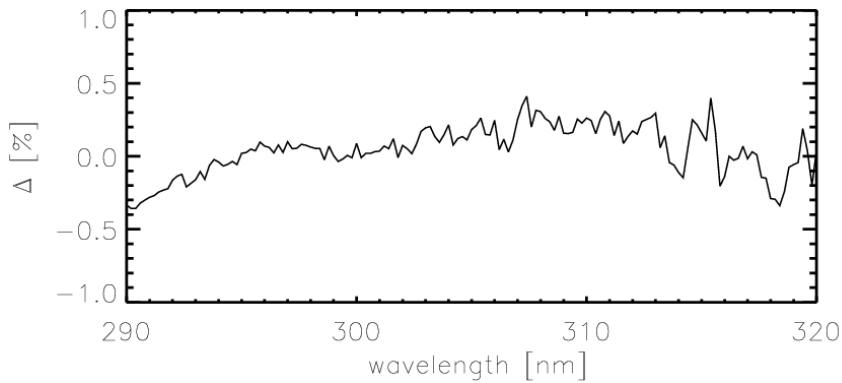


Figure 9.2: The relative absolute error on the produced spectrum. Everywhere under 0.5 %.

it to have a large effect on the performance and a smaller training data set is practical during testing. Our assumption has proven to be right, since the decrease in performance is small.

### Replacement of the standard mapping

If we replace only the spectrum  $F$  with the neural network we see a small effect on the result. The chi-square changes from 0.9 to 2.1. As mentioned the chi-square achieved in real measurements is often much higher and therefore this increase is not of relevance. Figure 9.2 shows the difference between the forward models spectrum and the spectrum from the neural network. The relative absolute error stays under 0.5 as expected from the results of chapter 4. The averaging kernel changes only very little, which will not have any significant effect for a retrieval from real measurements.

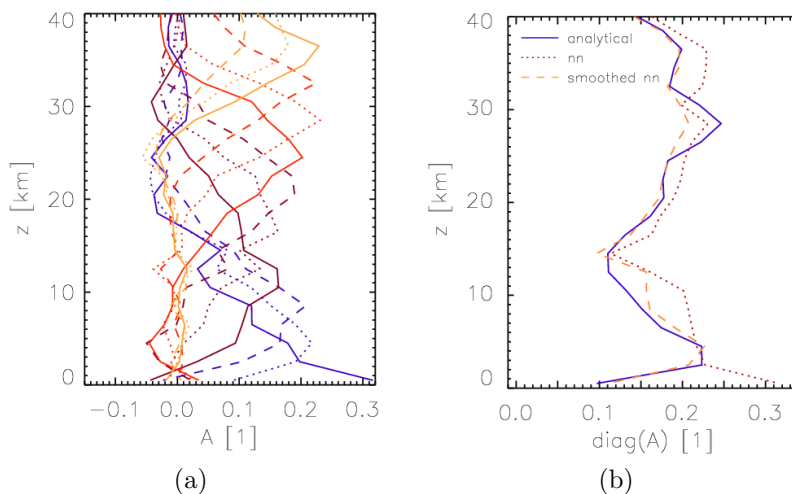


Figure 9.3: (a) Row vectors of the averaging kernel  $A$ . (b) The diagonal of  $A$ .

### Replacement of the Jacobian

The situation differs if in addition to  $F$  the Jacobian is replaced by its neural network correspondence. Here we see a clear effect on the regularisation as can be seen by the averaging kernel in figure 9.3 (a). The retrieval indicates in this case a DFS of 5.8, which is one DFS more than for the reference case. A close look shows that this apparent extra information comes from that part of the averaging kernel which describes the smoothing in the lowest few km. In figure 9.3 (a) the averaging kernel displays wrongly a high sensitivity of the retrieval with respect to surface near ozone (blue solid line). Also the strong negative side lobe is not present. For the retrieval at other altitudes, visualised by the other lines of the figure, differences are much smaller. Moreover, the averaging kernel seems to be improved, meaning that the retrieval tries to extract information that is not present in the measurements, and so the effect of the measurement noise on the corresponding parts of the profile is increased. The differences in the averaging kernels can nicely be seen when one looks at the diagonal of the averaging kernel. The elements of the diagonal describe the sensitivity of the retrieval at a particular altitude with respect to the ozone concentration of the real profile at the same altitude. Figure 9.3 (b) shows the diagonals of the reference retrieval and the neural network based retrieval. We see that at nearly all altitudes the neural network retrieval (red-dotted line) overestimates the diagonal of the reference, but largest difference occur at the surface near layer (1 km).

This error is caused by a noise-like contribution of the neural network on the Jacobian. The problem with these type of errors is that they are wrongly interpreted by the inversion as ozone related structures. It seems to the inversion that a high level of information is available, which in fact is measurement noise. Figure 9.4 shows the derivative  $\partial F/\partial x$  of the measurement simulation as function of wavelength with respect to ozone in the lowest model layer (1 km altitude). To reduce the noise-like contribution of the network we have smoothed the Jacobian with a Gaussian function with a full width at half-maximum of 0.9 nm. Figure 9.5 shows the absolute errors of the smoothed (dashed) and original (red) network output. The fine-scale spectral (noise like) features are reduced with this approach, but logically the error on broad spectral features is still present. In figure 9.3 (b) the dashed orange line displays the result on the diagonal of the averaging kernel. A consequence of the additional smoothing is a decreased sensitivity in the lowest few km leading to a reduction of the DFS to an acceptable value of 4.81.

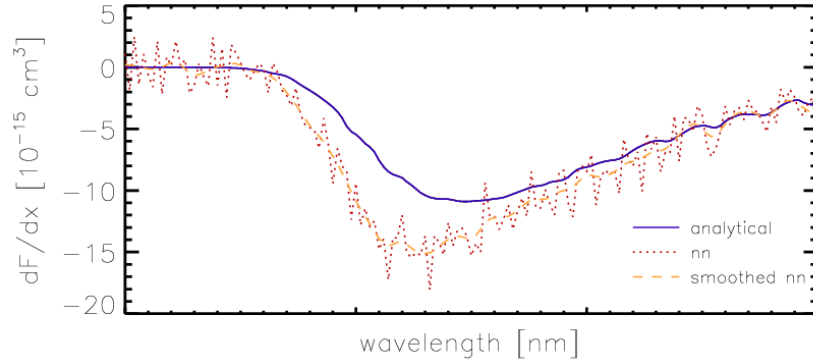


Figure 9.4: The target and neural network produced Jacobian on the lowest altitude.

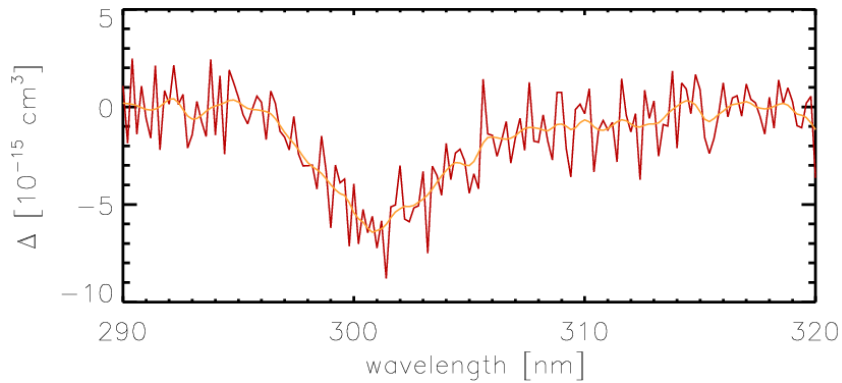


Figure 9.5: Absolute errors of the smoothed and original network output.

## 9.2 Conclusion

Overall, with these accuracies the neural network can be used for the interpretation of real measurement. However, there is room for improvement especially concerning the Jacobian. The smoothing of the network output has fixed the problem with wrongly interpreted noise, but logically also removes any detailed measurement information which was not caused by the network. Further research should point out if it is possible to raise the accuracy of the produced Jacobian which would automatically result in a smoother network output. The use of ensembles should be considered. Alternatively an approach like described in [Schwander *et al.*, 2001] could be used. This would mean that a small number of the output values are calculated in the analytical way by the forward model. If an execution of the forward model for all  $n$  outputs would take  $t$  sec, a calculation of  $m < n$  outputs requires proportionally less time. These outputs are fed to a trained neural network together with the ozone profile. These new inputs contain important reference points of the desired output, hopefully resulting in a boost in accuracy. Application of principle component analysis on the training data could be useful to avoid the curse of dimensionality. A consequence of this method would be an increase of calculation time dependent on the number of pre-calculated outputs. However a considerable increase of accuracy, of the Jacobian in particular, would be worth this sacrifice.

# Chapter 10

## Conclusion

---

In this chapter we describe the achievements of this research together with a number of recommendations for the future. The different elements of our research were the selection of a suitable machine learning replacement for both the standard mapping and the derivative of the forward model, exploring techniques to further improve the results and an analysis of experiments of ozone profile retrieval on synthetic measurements after the selected models had been integrated.

Ozone profile retrieval derives a vertical ozone distribution from a satellite measurement of backscattered sunlight from the Earth atmosphere. A forward model exists that is able to simulate a measurement for a given ozone distribution. A regularised least squares fit uses the outcome of the forward model to find the ozone profile best matching the satellite measurement. A speed up of ozone profile retrieval is best obtained by replacement of the forward model because it is both the slowest component and all diagnostic tools giving information about the inversion can still be used.

The training data consists of pairs of ozone profiles and satellite measurements. The ozone profiles are taken from weather balloon measurements extended by information from a climatology. The outputs are calculated by the existing implementation of the forward model. Support vector machines and feed-forward neural networks have been selected as candidates for replacement because they are both able to learn a function like the forward model and they both can produce their output very fast after training.

Experiments with the configurable properties of feed-forward networks has resulted in an optimal neural network configuration that has a mean relative error of  $\approx 0.08$  % on the test data. Important insights we have come to are that the type of normalisation is dependent on the choice of learning algorithm and that the resilient propagation learning algorithm has more benefit of multiple hidden layers and hidden nodes than the scaled conjugate gradient learning algorithm. The best performing neural network had one hidden layer of 20 hidden nodes and was trained with the scaled conjugate gradient learning algorithm. The normalisation method of choice was column wise standardisation.

Different parameter tunings of  $\epsilon$ -svr clearly showed the correlation between the  $\epsilon$  and C parameter. Multiple experiments with different kernels and parameter settings lead to improvement of results. However, the performance did not come close to that of the neural networks. The  $\nu$ -svr variant showed improvement over  $\epsilon$ -svr with a best mean relative error of  $\approx 0.33$  %. Based on these results we decided to not further investigate the support vector regression technique and focus on neural networks for the remaining part of the research.

Finding a good way of producing the Jacobian was the greatest challenge in this research. The derivation of the Jacobian from a trained network failed to give good results. The insight that this was caused by the construction of the training set was essential for the understanding of the networks behaviour. The testing of this hypothesis by adding generated samples with Jacobian information proved our theory. The learning of the Jacobian by a composed architecture has shown to be a better way of producing the Jacobian. However, the accuracy never reached the level of the networks trained on the standard mapping.

Two techniques to further improve the performance of the selected models have been investigated in the form of principle component analysis and ensembles. Principle component analysis showed that it is possible

to reduce the dimension of the data set by a factor 2 without losing much accuracy. This was proved by experiments with neural networks. The training speed up gained from a reduced training set is welcome for the learning of the Jacobian. The combination of multiple neural networks in the form of an ensemble has shown to be a possible way of increasing the accuracy on the standard mapping. Further research should point out if this also holds for the Jacobian.

Analysis of ozone profile retrieval, with the neural network models integrated, on synthetic measurements showed that the lack of accuracy in the Jacobian caused trouble. Especially the noise-like behaviour of the network output resulted in a wrong interpretation in the regularisation part of the retrieval. A smoothing on the outputs fixed this interpretation problem and resulted in a much better retrieval. However, future research should focus on the improvement of the accuracy of the Jacobian, because the smoothing procedure is likely to remove possible valuable information as well.

Overall, the results look promising and we think that in the future an accelerated ozone profile retrieval by use of machine learning techniques could be used on real measurements.

# Appendix A

## Neural network experiments

---

The results of tests with neural networks described in chapter 4 are shown in the tables below. An interpretation of these results can be found in chapter 4.

### A.1 Normalisation

learn algo.	normalisation	train error (m.s.e.)	test error (m.s.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
SCG	NoNormalisation	0.022587	0.022274	0.111120	0.069793	2.485500
SCG	rowLinear	0.008977	0.009143	0.066372	0.051658	1.489000
SCG	rowStd	0.008455	0.009844	0.067616	0.055982	1.520900
SCG	columnLinear	0.000311	0.000341	0.012367	0.008755	0.275314
<b>SCG</b>	<b>columnStd</b>	<b>0.000056</b>	<b>0.000064</b>	<b>0.005366</b>	<b>0.003775</b>	<b>0.115740</b>
RPROP	NoNormalisation	0.019627	0.018725	0.10118	0.064133	2.315744
RPROP	rowLinear	0.008211	0.009831	0.068063	0.053909	1.531600
RPROP	rowStd	0.008249	0.009787	0.067923	0.053218	1.536300
RPROP	columnLinear	0.000339	0.000328	0.012907	0.007294	0.287714
RPROP	columnStd	0.000598	0.000682	0.017341	0.011949	0.369670

Table A.1: Hidden layer of size 20. Networks trained for 1000 epochs.

learn algo.	normalisation	train error (m.s.e.)	test error (m.s.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
SCG	columnLinear	0.000048	0.000086	0.005391	0.005434	0.098110
<b>SCG</b>	<b>columnStd</b>	<b>0.000037</b>	<b>0.000034</b>	<b>0.004059</b>	<b>0.002478</b>	<b>0.088911</b>
RPROP	columnLinear	0.000077	0.000128	0.006656	0.006735	0.147810
RPROP	columnStd	0.000125	0.000118	0.007530	0.004374	0.162610

Table A.2: Hidden layer of size 20. Networks trained for 5000 epochs.

### A.2 Hidden layer(s)

architecture	train error (m.s.e.)	test error (m.s.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
5	0.000258	0.000282	0.010355	0.008148	0.218873
10	0.000253	0.000278	0.010795	0.008744	0.233649
20	0.000058	0.000057	0.005218	0.003067	0.113580
50	0.000045	0.000046	0.004713	0.002961	0.103010
<b>100</b>	<b>0.000043</b>	<b>0.000053</b>	<b>0.005061</b>	<b>0.003104</b>	<b>0.111760</b>
5;5	0.000428	0.000414	0.014293	0.007982	0.305976
20;20	0.000105	0.000102	0.007159	0.003880	0.151832
30;50	0.000125	0.000155	0.008672	0.005263	0.186299
50;50	0.000147	0.000211	0.010329	0.006203	0.222015
100;100	0.000406	0.000578	0.016580	0.010434	0.351955

Table A.3: Networks trained for 1000 epochs with SCG and columnStd normalisation.

architecture	train error (m.s.e.)	test error (m.s.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
5	0.001859	0.001914	0.031559	0.018565	0.715980
10	0.000355	0.000403	0.013395	0.009820	0.295867
20	0.000339	0.000328	0.012907	0.007294	0.287714
50	0.000292	0.000358	0.012766	0.008360	0.285816
100	0.000515	0.000706	0.017614	0.012071	0.400633
5,5	0.000649	0.000712	0.018427	0.011669	0.393929
20,20	0.000216	0.000305	0.011225	0.008820	0.249249
<b>30,50</b>	<b>0.000121</b>	<b>0.000129</b>	<b>0.008077</b>	<b>0.004493</b>	<b>0.178868</b>
50,50	0.000130	0.000171	0.008819	0.005617	0.197161
100,100	0.000148	0.000277	0.010896	0.007887	0.247912

Table A.4: Networks trained for 1000 epochs with RPROP and columnLinear normalisation.

architecture	learning algo.	train error (m.s.e.)	test error (m.s.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
<b>20</b>	<b>SCG</b>	<b>0.000020</b>	<b>0.000073</b>	<b>0.003702</b>	<b>0.006222</b>	<b>0.083915</b>
20	RPROP	0.000077	0.000128	0.006656	0.006735	0.147810
30;50	SCG	0.000027	0.000080	0.004218	0.006227	0.094415
30;50	RPROP	0.000049	0.000104	0.005755	0.006167	0.128420

Table A.5: Best networks trained for 5000 epochs.

## Appendix B

# Support vector regression experiments

---

The results of tests of the different parameter configuration of the support vector regressors are shown in the tables below. An interpretation of these results can be found in chapter 5.

### B.1 $\epsilon$ -svr

training set size	normalisation	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
300	columnStd	0.032427	0.047356276	0.014809	1.065059
1000	columnStd	0.029377303	0.036751996	0.009888	0.853489
2000	columnStd	0.027061368	0.030803155	0.005925	0.717828
<b>2448 (complete)</b>	<b>columnStd</b>	<b>0.02622716</b>	<b>0.02964163</b>	<b>0.006946</b>	<b>0.451636</b>

Table B.1: Training set size.  $\epsilon$ -svr, with  $C=0.1$ ,  $\epsilon=0.01$ , kernel = poly 3

training set size	normalisation	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
1000	columnStd	0.029377303	0.036751996	0.009888	0.853489
<b>1000</b>	<b>columnLin</b>	<b>0.01993823</b>	<b>0.02159993</b>	<b>0.005040</b>	<b>0.532740</b>
1000	rowStd	0.042083458	0.066350638	0.027869	1.502935
1000	rowLin	0.045925492	0.062448813	0.026870	1.414610

Table B.2: Normalisation results.  $\epsilon$ -svr, with  $C=0.1$ ,  $\epsilon=0.01$ , kernel = poly 3

$\epsilon$	C	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
0.0001	0.001	0.019134165	0.022568899	0.004984	0.542979
0.0001	0.1	0.019322011	0.021060181	0.004797	0.516755
0.001	0.0001	0.021121442	0.022870857	0.004831	0.554499
<b>0.001</b>	<b>0.001</b>	<b>0.01857633</b>	<b>0.02025434</b>	<b>0.004208</b>	<b>0.499386</b>
0.001	0.1	0.019349361	0.019349361	0.004802	0.517269
0.01	0.00001	0.04061064	0.041521355	0.013065	0.971313
0.01	0.0001	0.02179842	0.023540966	0.004925	0.572772
0.01	0.001	0.019060165	0.020741986	0.004484	0.513543
0.01	0.01	0.019830573	0.021358796	0.004826	0.527545
0.01	0.1	0.019938233	0.021599935	0.005040	0.532740
0.1	0.1	0.031374908	0.033305665	0.010575	0.847922
0.1	1	0.030888745	0.032795229	0.008388	0.993739

Table B.3: Varying  $\epsilon$  and C. Training set size = 1000, kernel = poly 3

training set size	kernel type	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
1000	poly 2	0.020589417	0.023103532	0.005273	0.582131
<b>1000</b>	<b>poly 3</b>	<b>0.01993823</b>	<b>0.02159993</b>	<b>0.005040</b>	<b>0.532740</b>
1000	poly 4	0.020579465	0.022787841	0.004956	0.556207
1000	rbf 1	0.02300433	0.026489006	0.005834	0.616280
<b>1000</b>	<b>rbf 2</b>	<b>0.019334026</b>	<b>0.02097325</b>	<b>0.004230</b>	<b>0.474793</b>
1000	rbf 3	0.022525024	0.023531816	0.006068	0.509841

Table B.4: Varying type of kernel. Training set size = 1000, C=0.1,  $\epsilon=0.01$

kernel type	C	$\epsilon$	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
<b>poly 3</b>	<b>0.001</b>	<b>0.001</b>	<b>0.017848</b>	<b>0.018782</b>	<b>0.004726</b>	<b>0.455350</b>
rbf 2	0.1	0.01	0.018111	0.019253	0.005160	0.463390

Table B.5: Optimal configurations  $\epsilon$ -svr. Training set size = complete,

## B.2 $\nu$ -svr

$\nu$	C	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
0.1	0.01	0.016459457	0.018438086	0.003978	0.449418
0.5	0.01	0.014364499	0.016043197	0.002851	0.383358
<b>1</b>	<b>0.01</b>	<b>0.01418153</b>	<b>0.015746114</b>	<b>0.002699</b>	<b>0.378937</b>

Table B.6: Different values for  $\nu$ . Training set size = 1000

$\nu$	C	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
1	0.001	0.014281	0.015889	0.002881	0.383420
<b>1</b>	<b>0.1</b>	<b>0.014125</b>	<b>0.015690</b>	<b>0.002725</b>	<b>0.377063</b>
1	1	0.014203	0.015813	0.002860	0.381965
0.5	0.001	0.014835	0.016080	0.002858	0.385744
0.5	0.1	0.015041	0.016043	0.002851	0.383358
0.5	1	0.014653	0.016376	0.002966	0.392008

Table B.7: Varying the C parameter for two values for  $\nu$ . Training set size = 1000

$\nu$	C	train error (m.a.e.)	test error (m.a.e.)	std dev. test set	rel. error (%)
1	0.1	0.012621	0.013912	0.002370	0.333606

Table B.8:  $\nu$ -svr optimal configuration. Training set size = complete



# Appendix C

## Matlab source code

---

### C.1 Jacobian derivation algorithm

Matlab source code to calculate the Jacobian from a trained neural network.

```
0001 function [jacobian]=calcJacobOzoneProfile(net, inputPattern)
0002     if( ~isa(net, 'network') )
0003         load('31-20-151 (3000 cycles).mat');
0004     end
0005
0006     % init to size of network
0007     inputNodes = size(net.iw{1},2);
0008     hiddenNodes = size(net.lw{2,1},2);
0009     outputNodes = size(net.lw{2,1},1);
0010
0011     % outputJacobDeltas are equal to 1 for linear output activation function
0012     outputJacobDelta = ones(1,outputNodes);
0013     hiddenJacobDelta = zeros(1,hiddenNodes);
0014     jacobian = zeros(outputNodes, inputNodes);
0015     sumDeltaWeight = 0;
0016
0017     inputVector = inputPattern;
0018     inputHiddenNode = 0;
0019     outputVector = 0;
0020
0021     for j=1:outputNodes
0022         % calculate hidden jacob deltas
0023         for k=1:hiddenNodes
0024
0025             % calc input hidden node k
0026             inputHiddenNode = net.iw{1}(k,:) * inputVector';
0027
0028             % calc derivSigmoid of input hidden node
0029             derivSigmoidHiddenNode = dSig(inputHiddenNode + net.b{1}(k));
0030
0031             % calc sum of W_jk * outputJacobDelta(j), in case of linear
0032             % output, outputDeltas are all equal to 1
0033             sumDeltaWeight = sum( net.lw{2,1}(j,k) );
0034
0035             hiddenJacobDelta(k) = sumDeltaWeight * derivSigmoidHiddenNode;
0036
```

```

0037         inputHiddenNode = 0;
0038
0039     end
0040
0041     % calculate part derivs of output node j, multiply jacobDelta's
0042     % with W_ki (input weights)
0043     jacobian(j,:) = hiddenJacobDelta * net.iw{1};
0044 end
0045 end

```

## C.2 Ensemble meta method

This is the source code written to train a multi-output regression ensemble. It consists of a meta method 'trainMultipleOutput', which calls the 'trainSingleOutput' method, that in turn initialises and trains a single-output ensemble.

### trainMultipleOutput.m

```

0001 % train ensemble and return performance on test set
0002 function [ensVec, ensOut, desiredOut]= trainMultipleOutput(Xtr, Ytr, ...
                                Xtst, Ytst, cv, mixedModels, savename)
0003     % train ensemble on training set Xtr
0004     for i=1:151
0005         disp(['training ensemble:' num2str(i)]);
0006         ensVec(i) = trainSingleOutput(Xtr, Ytr(:, i), cv, mixedModels);
0007     end
0008
0009     save(savename, 'ensVec');
0010
0011     % calculate ensemble output for test set
0012     for i=1:size(Xtst,1)
0013         for j=1:151
0014             ensOut(i,j) = calc(ensVec(j), Xtst(i,:));
0015         end
0016     end
0017
0018     desiredOut = Ytst;
0019 end

```

### trainSingleOutput.m

```

0001 % Function trains single output ensemble on training set Xtrain with
0002 % outputs Ytrain. The parameters cv and mixedModels respectively configure
0003 % the number of crosstraining folds and mixture of models.
0004 function [ens]= trainSingleOutput(Xtrain, Ytrain, cv, mixedModels)
0005
0006     ens = crosstrainingensemble;

```

```

0007     tpe = get(ens, 'trainparams');
0008
0009     if( mixedModels == 1 )
0010         modelParams1 = get(perceptron3, 'trainparams');
0011         modelParams1.maxiter = 1000;
0012
0013         tpe.modelclasses = {'perceptron3', modelParams1, {}};
0014                             'vicinal', [], {};
0015                             'rbf', [], {};
0016     elseif( mixedModels == 2 )
0017         modelParams1 = get(perceptron3, 'trainparams');
0018         modelParams1.maxiter = 1000;
0019
0020         tpe.modelclasses = {'perceptron3', modelParams1, {}};
0021                             'vicinal', [], {};
0022                             };
0023     elseif( mixedModels == 3 )
0024         modelParams1 = get(perceptron3, 'trainparams');
0025         modelParams1.maxiter = 1000;
0026
0027         tpe.modelclasses = { 'vicinal', [], {};
0028                             'rbf', [], {};
0029     elseif( mixedModels == 4 )
0030         modelParams1 = get(perceptron3, 'trainparams');
0031         modelParams1.maxiter = 1000;
0032
0033         tpe.modelclasses = {'perceptron3', modelParams1, {}};
0034                             'rbf', [], {};
0035     else % neural network only
0036         modelParams = get(perceptron3, 'trainparams');
0037         modelParams.maxiter = 1000;
0038
0039         tpe.modelclasses = {'perceptron3', modelParams, {}};
0040     end
0041
0042     tpe.use_models = 0.8;
0043     tpe.nr_cv_partitions = cv;
0044     tpe.scale_data = 0;
0045
0046     ens = set(ens, 'trainparams', tpe);
0047     ens = train(ens, Xtrain, Ytrain, [], tpe, 0.01);
0048 end

```

### C.3 Export network

The following code has been written to export neural networks from Matlab to a text file. These are later imported by the fortran code that is part of the ozone profile retrieval (appendix D).

## exportNetwork.m

```
0001 function exportNetwork(net, fileName, normVec1, normVec2)
0002
0003 if( ~isa(net, 'network') )
0004     'wrong argument type, must be a network!'
0005 else
0006     % write number of networks
0007     fid = fopen(fileName,'w+');
0008     fprintf(fid, 'networks: 1\n');
0009
0010     % retrieve (node) layer size of network
0011     layerSizes(1) = size(net.iw{1},2);
0012     for i=2:net.numLayers
0013         layerSizes(i) = size(net.lw{i,i-1},2);
0014     end
0015     layerSizes(net.numLayers+1) = size(net.lw{net.numLayers,net.numLayers-1},1);
0016     fprintf(fid, ['layers: ' num2str(net.numLayers) '\n']);
0017     %write layerSizes to file
0018     for i=1:size(layerSizes,2)
0019         fprintf(fid, ['l' num2str(i) ': ' num2str(layerSizes(i)) '\n']);
0020     end
0021
0022     fprintf(fid, 'normalisation: li2\n');
0023     fprintf(fid, ['normvalue1: ' normVec1 '\n']);
0024     fprintf(fid, ['normvalue2: ' normVec2 '\n']);
0025
0026     fclose(fid);
0027
0028     % get weight layers; reshape to 1-dim vector; write to file
0029
0030     % input layer
0031     tmpWeights = net.iw{1};
0032     tmpWeights = reshape(tmpWeights, 1, size(tmpWeights,1) * size(tmpWeights,2));
0033     dlmwrite(fileName, tmpWeights, '-append', 'delimiter', ' ', 'precision', '%19.18e');
0034     dlmwrite(fileName, net.b{1}', '-append', 'delimiter', ' ', 'precision', '%19.18e');
0035
0036     % hidden layer(s): weights + bias weights
0037     for i=2:net.numLayers
0038         tmpWeights = net.lw{i, i-1};
0039         tmpWeights = reshape(tmpWeights, 1, size(tmpWeights,1) * size(tmpWeights,2));
0040         dlmwrite(fileName, tmpWeights, '-append', 'delimiter', ' ', 'precision', '%19.18e');
0041         dlmwrite(fileName, net.b{i}', '-append', 'delimiter', ' ', 'precision', '%19.18e');
0042     end
0043 end
```

## exportMultiNet.m

```
0001 function exportMultiNet(multiNet, normVec1, normVec2)
```

```
0002
0003 number = 0;
0004
0005 for i=1:151
0006     net = multiNet(i).net;
0007
0008     exportNetwork(net, ['net' num2str(number + i) '.txt'], normVec1, normVec2);
0009 end
0010
0011 end
```



# Appendix D

## Fortran source code

---

This appendix contains the source code written in Fortran that is used to import and simulate neural networks exported from Matlab. The 'simSingleNet' and 'simMultiNet' functions are invoked from the ozone profile retrieval procedure instead of the forward model calculation.

### main.f90

```
program main

use loadnetwork ! module contains functionality of loading network weights from a file
use normalise ! module implements the normalisation functions
use calcoutput ! module implements the functions to produce the network output

implicit none

! Main

type(multiNetInfo), pointer :: multiNet
type(NeuralNetInfo), pointer :: net
double precision, dimension(:), pointer :: inputVector

allocate(inputVector(32));

! Example single net (standard mapping)
call initSingleNet(net, "3d set (s02 s05) albedo.txt")
call readInputVector("input_tst.txt", inputVector)
call simSingleNet(net, inputVector)

! print output
print *,net%outputVector

! Example multi net (Jacobian)
call initMultiNN(multiNet, "jacob (3d set (small) - column)")
call readInputVector("input_tst.txt", inputVector)
call simMultiNN(multiNet, inputVector);

! save output to file
call writeJacobToFile(multiNet%outputMatrix, "jacobout.txt")

! Subroutines and functions
```

contains

*! ----- Single - Net ----- !*

*! Subroutine that has to be called for initialisation of single net*

subroutine initSingleNet(net, fileName )

type(neuralNetInfo), pointer :: net  
character(len=\*) :: fileName;  
double precision, pointer, dimension(:) :: dummy1

call readNetworkInfo(fileName, net, dummy1)

end subroutine initSingleNet

*! Subroutine simulates single network and fills the outputvector of the network*

subroutine simSingleNet(net, inputVector)

type(neuralNetInfo), pointer :: net  
double precision, dimension(:), pointer :: inputVector;

call normaliseInputs(net, inputVector)

call calcNetOutput(net, inputVector)

end subroutine simSingleNet

*! ----- Multi - Net ----- !*

*! Subroutine that has to be called for initialisation of multi net*

subroutine initMultiINN(multiNet, dirName )

type(multiNetInfo), pointer :: multiNet;  
character(len=\*) :: dirName;

call readMultiNetworkInfo(dirName, multiNet)

end subroutine initMultiINN

*! Function encapsulates the calcMultiNetOut function.*

*! The matrix constructed by the outputs of all networks is returned*

*! as a 2d-array pointer*

subroutine simMultiINN(multiNet, inputVector )

type(multiNetInfo), pointer :: multiNet;  
double precision, dimension(:), pointer :: inputVector  
integer :: i

*! normalisation parameters of 1st network in vector are used*

call normaliseInputs(multiNet%neuralNetVector(1)%net, inputVector)

call calcMultiNetOut(multiNet, inputVector)

*! denormalise the output (hard coded!)*

```

    multiNet%outputMatrix = multiNet%outputMatrix * 1.0e-12;
end subroutine simMultiNN

! ----- Test Functions ----- !

! subroutine reads an input vector from a file.
! note: only for testing purposes
subroutine readInputVector(inputVectorFile, inputVector)

    double precision, dimension(:), pointer :: inputVector
    character(*) :: inputVectorFile

    open(unit = 1, file = inputVectorFile)
    read(1,*) inputVector

    close(1)

end subroutine readInputVector

subroutine writeJacobToFile(matrix, fileName)

    double precision, dimension(:,:): matrix
    character(len=*) :: fileName
    integer :: i,j

    open(unit=2, file = fileName)

    do i=1, 151
        write(2, *) matrix(i,:)
    end do

    close(2)

end subroutine writeJacobToFile

end program main

loadnetwork.f90

module loadNetwork
    implicit none

    ! Global type layerWeights contains the weight matrix for a certain layer
    type layerWeights
        double precision, dimension(:,:), allocatable :: weights
    end type layerWeights

    ! Global type neuralNetInfo contains all the info of the loaded neural network
    type neuralNetInfo
        character(3) :: normalisationType
        double precision, dimension(:), pointer :: normVec1 ! 1st value used in normalisation
        double precision, dimension(:), pointer :: normVec2 ! 2nd value used in normalisation
        integer :: nrOfLayers
    end type neuralNetInfo
end module loadNetwork

```

```

double precision, dimension(:), pointer :: outputVector
integer, dimension(:), allocatable :: layerSize
type(layerWeights), dimension(:), allocatable :: biasWeights
type(layerWeights), dimension(:), allocatable :: weightsPerLayer
end type neuralNetInfo

type neuralNetVecMember
type(neuralNetInfo), pointer :: net
end type neuralNetVecMember

type multiNetInfo
integer :: nrOfNetworks;
double precision, dimension(:), pointer :: inputVector;
double precision, dimension(:, :), allocatable :: outputMatrix;
type(neuralNetVecMember), dimension(:), allocatable :: neuralNetVector
end type multiNetInfo

contains
! subroutine initialises the 'neuralNet' object and the input and
! output vectors.
subroutine readNetworkInfo(networkFile, neuralNet, inputVector)

type(neuralNetInfo), pointer :: neuralNet
character(len=*) :: networkFile
double precision, pointer, dimension(:) :: inputVector
integer :: i, nrOfNetworks
character(10) :: lineDescriptor
integer, dimension(:), allocatable :: layerSize
double precision, dimension(:), allocatable :: tmpWeights

allocate(neuralNet)

! open network file
open (unit = 1, file = networkFile)

! read network architecture info
read(1,*) lineDescriptor, nrOfNetworks
read(1,*) lineDescriptor, neuralNet%nrOfLayers

allocate(neuralNet%layerSize(neuralNet%nrOfLayers+1))
allocate(neuralNet%weightsPerLayer(neuralNet%nrOfLayers))
allocate(neuralNet%biasWeights(neuralNet%nrOfLayers))

do i = 1,neuralNet%nrOfLayers+1
read(1,*) lineDescriptor, neuralNet%layerSize(i)
end do

! allocate input vector and normalisation vectors
allocate(inputVector(neuralNet%layerSize(1)))
allocate(neuralNet%normVec1(neuralNet%layerSize(1)))
allocate(neuralNet%normVec2(neuralNet%layerSize(1)))

! read normalisation info
read(1,*) lineDescriptor, neuralNet%normalisationType

```

```

read(1,*) lineDescriptor, neuralNet%normVec1
read(1,*) lineDescriptor, neuralNet%normVec2

! allocate the weightLayers
do i = 1,neuralNet%nrOfLayers
  ! normal weights (2d)
  allocate(neuralNet%weightsPerLayer(i)%weights(neuralNet%layerSize(i+1),
    neuralNet%layerSize(i)))

  ! bias weights (1d)
  allocate(neuralNet%biasWeights(i)%weights(neuralNet%layerSize(i+1), 1))
end do

! read the weights, using a tmpWeights array that is reshaped
do i = 1,neuralNet%nrOfLayers
  ! normal weights
  allocate(tmpWeights(neuralNet%layerSize(i+1) * neuralNet%layerSize(i)))
  read(1,*) tmpWeights
  neuralNet%weightsPerLayer(i)%weights = reshape(tmpWeights, (/neuralNet%layerSize(i+1),
    neuralNet%layerSize(i)/))
  deallocate(tmpWeights)

  ! bias weights
  read(1,*) neuralNet%biasWeights(i)%weights
end do

close(1)
end subroutine readNetworkInfo

subroutine readMultiNetworkInfo(networkDir, multiNet)

character(len=*)          :: networkDir
type(multiNetInfo), pointer :: multiNet
type(neuralNetInfo), pointer :: tmpNet;
integer                   :: i, j
double precision, pointer, dimension(:) :: dummy1
character(len=32) :: charNumber;

allocate(multiNet)

! set by default to 31
multiNet%nrOfNetworks = 31

allocate(multiNet%neuralNetVector(multiNet%nrOfNetworks))

do i=1, multiNet%nrOfNetworks
  write (charNumber,*) (i)
  charNumber = adjustl(charNumber)
  call readNetworkInfo(networkDir//'/net'//trim(charNumber)//'.txt',
    multiNet%neuralNetVector(i)%net,dummy1)
end do

multiNet%inputVector => dummy1

```

```

        allocate(multiNet%outputMatrix(151, multiNet%nrOfNetworks))

    end subroutine readMultiNetworkInfo

end module loadNetwork

normalise.f90

module normalise

    use loadnetwork
    implicit none

contains

    ! subroutine normalises the inputVector to domain [0,1]
    subroutine normaliseLinear(min, max, inputVector)

        integer :: i
        double precision, pointer, dimension(:) :: min, max
        double precision, pointer, dimension(:) :: inputVector

        do i = 1, size(inputVector)-1 ! albedo (last input) should not be normalised
            inputVector(i) = (inputVector(i) - min(i)) / (max(i) - min(i))
        end do

    end subroutine normaliseLinear

    ! subroutine normalises the inputVector to domain [-1, 1]
    subroutine normaliseLinear2(min, max, inputVector)

        integer :: i
        double precision, pointer, dimension(:) :: min, max
        double precision, pointer, dimension(:) :: inputVector

        do i = 1, size(inputVector)-1 ! albedo (last input) should not be normalised
            inputVector(i) = (2*((inputVector(i) - min(i)) / (max(i) - min(i)))) - 1
        end do

    end subroutine normaliseLinear2

    ! subroutine normalises input to std = 1, mean = 0
    subroutine normaliseStd(mean, std, inputVector)

        integer :: i
        double precision, dimension(:), pointer :: mean, std
        double precision, dimension(:), pointer :: inputVector

        do i = 1, size(inputVector)
            inputVector(i) = ((inputVector(i) - mean(i)) / std(i))
        end do

    end subroutine normaliseStd

```

```

! selection of normalisation mode
subroutine normaliseInputs(neuralNet, inputVector)

  type(neuralNetInfo), pointer :: neuralNet;
  double precision, pointer, dimension(:) :: inputVector

  if( neuralNet%normalisationType == 'li1') then
    call normaliseLinear(neuralNet%normVec1, neuralNet%normVec2, inputVector)
  else if(neuralNet%normalisationType == 'li2') then
    call normaliseLinear2(neuralNet%normVec1, neuralNet%normVec2, inputVector)
  else if(neuralNet%normalisationType == 'std') then
    call normaliseStd(neuralNet%normVec1, neuralNet%normVec2, inputVector)
  else if(neuralNet%normalisationType == 'sdC') then
    call normaliseStd(neuralNet%normVec1, neuralNet%normVec2, inputVector)
  end if

end subroutine normaliseInputs

end module normalise

```

## calcoutput.f90

```

module calcOutput
use loadnetwork

implicit none

contains
! subroutine calculates the network output
subroutine calcNetOutput(neuralNet, inputVector)

  type(neuralNetInfo), pointer :: neuralNet
  double precision, pointer, dimension(:) :: inputVector
  double precision, pointer, dimension(:) :: inputNodeValues
  double precision, pointer, dimension(:) :: hiddenNodeValues
  integer :: i,j
  double precision :: tmp

  inputNodeValues => inputVector

  ! loop through the layers
  do i = 1, neuralNet%nrOfLayers
    allocate(hiddenNodeValues(neuralNet%layerSize(i+1)))
    hiddenNodeValues = matmul(inputNodeValues,
      transpose(neuralNet%weightsPerLayer(i)%weights));

    ! add bias weight and apply activation function (not for output layer)
    if( i < neuralNet%nrOfLayers ) then
      do j=1, neuralNet%layerSize(i+1)
        hiddenNodeValues(j) = activationFunction(hiddenNodevalues(j) +
          neuralNet%biasWeights(i)%weights(j,1))
      end do
    end if
  end do

```

```

else
    do j=1, neuralNet%layerSize(i+1)
        hiddenNodeValues(j) = hiddenNodeValues(j) +
            neuralNet%biasWeights(i)%weights(j,1)
    end do
end if

! init for next round
if( i > 1) deallocate(inputNodeValues)
inputNodeValues => hiddenNodeValues
end do

neuralNet%outputVector => hiddenNodeValues;

end subroutine calcNetOutput

subroutine calcMultiNetOut(multiNet, inputVector)

type(multiNetInfo), pointer :: multiNet;
double precision, dimension(:), pointer :: inputVector;
integer :: i,j

do i=1, multiNet%nrOfNetworks

    call calcNetOutput(multiNet%neuralNetVector(i)%net, inputVector)

    multiNet%outputMatrix(:,i) = multiNet%neuralNetVector(i)%net%outputVector

end do

end subroutine calcMultiNetOut

! function returns the standard sigmoid activation function value for 'input'
function activationFunction( input )

double precision :: activationFunction
double precision :: input

activationFunction = 1 / (1 + exp(-input))

end function activationFunction

end module calcoutput

```

# Bibliography

- [Bellman, 1961] R. Bellman. Adaptive control processes: A guided tour. *Princeton*, 1961.
- [C. Merkwirth, 2003] J. Wichard C. Merkwirth. A short introduction to entool, 2003.
- [Chance and Schneider, 1997] J. P. Burrows D. Perner Chance, K. V. and W. Schneider. Satellite measurements of atmospheric ozone profiles, including tropospheric ozone, from ultraviolet/visible measurements in the nadir geometry: a potential method to retrieve tropospheric ozone. *J. Quant. Spectroscop. Radiat. Transfer* 57, pages 467–476, 1997.
- [Chandra and Yao, 2006] Arjun Chandra and Xin Yao. Multi-objective ensemble construction, learning and evolution, 2006.
- [Chang and Lin, 2002] Chih-Chung Chang and Chih-Jen Lin. Training nu-Support Vector Regression: Theory and Algorithms. *Neural Comp.*, 14(8):1959–1977, 2002.
- [Cherkassky and Ma, 2003] Vladimir Cherkassky and Yunqian Ma. Practical selection of svm parameters and noise estimation for svm regression. *Neural networks*, 17:113–126, 2003.
- [Cortes and Vapnik, 1995] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
- [Cybenko, 1989] G. Cybenko. Approximation by superposition of a sigmoidal function. *Mathematics of control, signals and systems*, pages 303–314, 1989.
- [Fortuin and Kelder, 1998] Fortuin and Kelder. An ozone climatology based on ozonesonde and satellite measurement. *Journal of geophysical research*, 1998.
- [Hertz *et al.*, 1990] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the theory of neural computation*. Santa Fe institute studies in the sciences of complexity. Westview, 1990.
- [Hinton, 1989] G. Hinton. Connectionist learning procedures. *Artificial Intelligence* 40, pages 185–234, 1989.
- [Hornik *et al.*, 1989] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.
- [Kohavi, 1995] Ron Kohavi. *A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection*. 1995.
- [Lazaro *et al.*, 2004] M. Lazaro, I. Santamar, F. Perez-Cruz, and A. Art es Rodriguez. Support vector regression for the simultaneous learning of a multivariate function and its derivatives, 2004.
- [Lee and Oh, 1997] Jeong-Woo Lee and Jun-Ho Oh. Hybrid learning of mapping and its jacobian in multilayer neural networks. *Neural Computation*, 9(5):937–958, 1997.
- [Moller, 1993] M. Moller. A scaled conjugate gradient algorithm for fast supervised learning. 6, pages 525–533, 1993.
- [Muller and Kaifel, 2003] Martin D. Muller and Anton K. Kaifel. Ozone profile retrieval from some data using a neural network inverse model, 2003.

- [Riedmiller M., 1992] Braun H. Riedmiller M. Rprop- a fast adaptive learning algorithm. *Technical Report (Also Proc. of ISCIS VII), Universitat Karlsruhe., 1992.*
- [Schwander *et al.*, 2001] H. Schwander, A. Kaifel, A. Ruggaber, and P. Koepke. Spectral Radiative-Transfer Modeling with Minimized Computation Time by Use of a Neural-Network Technique. *Applied Optics*, 40:331–335, January 2001.
- [Shewchuk, 1994] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.
- [Smola and Schoelkopf, 1998] A.J. Smola and B. Schoelkopf. A tutorial on support vector regression, 1998.
- [Vazquez and Walter, 2003] Emmanuel Vazquez and Eric Walter. *Multi Output Support Vector Regression*. IFAC, Rotterdam, aout 2003.
- [Vytautas Vysniauskas, 1993] Ben. J.A. Krose Vytautas Vysniauskas, Frans C.A. Groen. The optimal number of learning samples and hidden nodes in function approximation with a feedforward network, 1993.
- [Witten and Frank, 2005] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques, 2nd Edition*. Morgan Kaufmann, San Francisco, 2005.

# Acknowledgements

First of all I would like to express my gratitude to Jochen Landgraf from Space Research Netherlands (SRON) for the many times he took the time to explain the principles of ozone profile retrieval to me. Our meetings to interpret and discuss the (intermediate) results were very useful and fun at the same time. Further I would like to thank Joost Kok for providing me with useful tips and feedback during this research and the writing of this thesis. Naturally I am also thankful to my family and friends. Often did the distraction they gave me provide me with new energy to keep working on this thesis.