Thesis for the degree of Master of Science in Complex Adaptive Systems

Modeling and optimization of response surfaces using an Artificial Neural Networks

Axel Jarenfors

Department of Applied Physics
Chalmers University of Technology
Göteborg, Sweden, 2011

Modeling and optimization of response surfaces using an Artificial Neural Networks
AXEL JARENFORS

Department of Applied Physics
Chalmers University of Technology
412 96 Göteborg, Sweden
Telephone: +46 (0)31 772 1000

This report has been produced with support from Volvo 3P, and Jönköping University, School of Engineering

Cover:
Final response surface of an artificial neural network trained of the Rosenbrock function. The final error is evaluated at $\approx 10^{-3}$. The points indicate the original data set. For more details see 4.1.2.1 on page 23

Chalmers Reproservice
Göteborg, Sweden, 2011

Modeling and optimization of response surfaces using an Artificial Neural Networks

AXEL JARENFORS
Department of Applied Physics
Chalmers University of Technology

ABSTRACT

In a world where new products are developed using computer simulations, and where every aspect can be measured, and refined with extreme precision, most optimisation algorithms still rely on the existence of a clearly defined function to optimize. In reality however this function is often defined through a FEM-calculation, which may require hours to evaluate each individual point. In order to save time, only a small discrete set of points can be evaluated and are then used to construct a mathematical model, or response surface, of the data. Actual optimisation of the problem can then be done on this model instead.

This project focuses on using an artificial neural network (ANN) to construct such a model. The objective is to build a generalised software tool that can take a set of data, construct a response surface, and find the optimal point on it. The tool must also be able to do this with high accuracy and within reasonable time.

Because the method requires many mathematical formulations, the tool was written in MATLAB. The structure of the ANN used is limited to feed-forward networks with two hidden layers, where the number of hidden neurons ischosen such that overfitting is avoided. The training of the ANN uses backpropagation and the results are evaluated using the response surface of a quadratic regression model (QRM) for comparison.

Testing of the final product shows that the ANN is in most cases able to outperform the QRM, and sometimes with several orders of magnitude. It is also clear that the ANN is more versatile than the QRM when it comes to modelling non-symmetric functions. When the number of input parameters increases, the difference becomes less distinct. However the ANN has the advantage that its adaptability can be easily improved upon if the data set is increased.

The applicability of the tool developed in this project is immediate. It can be used as it is to help R&D-staff with their work. The tool does require some experience to be used fluently, and it still has potential for further improvements. It does however illustrate once again that advanced mathematical concepts can be translated into industry-useful aids.

Keywords: optimization, response surface, error function, artificial neural networks, backpropagation, Levenberg-Marquart

Contents

1. Introduction

1.1. Problem

The central part of design optimization revolves around the problem of locating the minimum of a given parameter, such as tension or weight, within specified boundaries, such as as size or strength. In order to approach this problem it is necessary to determine the exact way that changes in the design will effect the the final value of whatever is being optimized, i.e. what the mathematical function between them is. The two most common ways of obtaining data about this function is by physical measurements, or by computer simulation.

Unfortunately, these two methods can only provide discrete data about specifically tested points, and it is usually a very time consuming process. In order to perform accurate optimization it is necessary to be able to evaluate new points very rapidly. The solution is to use the available discrete data, and construct a continuous approximation of the function, known as the *response surface.* The idea of this is that if the approximation is accurate, the minimum of the real function will coincide with the minimum of the response surface. Since the function describing the response surface will be well known, its minimum can can be obtained using a number of well established techniques. The focus of this thesis however, lies in the construction of an accurate approximation.

1.2. Background

When constructing an approximation of a function, the starting point is always a general function with a number of coefficients that need to be fine-tuned in order to fit the given data. A greater number of coefficients usually means the function can be more accurate, however it also means it will be more difficult to fine-tune.

In its simplest version the response surface is taken to be a linear plane. A plane would of course not have a minimum without the boundaries that limit the design variables, and as such the minimum will always lie in a corner. After locating it, the boundaries are moved closed towards it and a new set of points with greater resolution around this corner is tested. Using the new data the process is repeated until convergence at a function minimum is reached. The trouble with this method is that the linear plane is rarely, if ever, an accurate approximation. Therefore this method may end up requiring a large number of points to be tested before any results can be found. It also means that the final response surface cannot be used to determine the function's behavior around the minimum, which may be very important if more than one parameter is to be optimized.

1.3. Objectives

The purpose of this thesis is to test a different way of utilizing the available data to construct a more accurate response surface. Specifically, a program employing an Artificial Neural Network (ANN) will be written, and fine-tuned. This should be able to take more of the non-linear aspects of the real function into consideration. Once completed the program will be tested using known benchmark functions to determine its accuracy and robustness.

1.4. Outline

The thesis is divided into several parts. The first is a theoretical description of the mathematical concepts employed when designing the ANN. The second depicts the implementation of the concepts as a MATLAB program. The final section covers the testing that was employed during the development.

2. Theory

2.1. Artificial Neural Networks

An ANN is a method of performing calculations in a way that attempts to mimic the processes of the biological brain. In order to understand how this is accomplished, it is necessary to understand how how an ANN is setup. [1]

2.1.1. Basic aspects of training

The most significant aspect of an ANN is its ability to learn. Just like the brain it attempts to simulate, it is often terrible at its task when it first starts out. In order for it to improve it requires training, which consists of carefully adjusting the coefficients of the network. Training may be either supervised or unsupervised depending of the task at hand. For the purposed of this project, supervised learning is the way to proceed.

Supervised learning is used when the operator already possesses a number of input points, $\mathbf{x}_q$, as well as their corresponding function value, y_q. This is referred to as an *input/output-pair* (I/O-pair). During training the inputs are fed into the network and by comparing the network's outputs to the known function values, appropriate adjustments can be made.

2.1.2. The Single Neuron

The first step when constructing an ANN is to understand the basic building block, i.e. the neuron. The artificial neuron is modeled after the biological neuron, which is illustrated in figure 2.1. Via the dendrites, the neuron receives a number of input signals, the strength of which is determined by the strength of the connections, also known as synapses. The combination of signals is then processed by the cell body, and the resulting output signal is sent along the axon to be received by other neurons.

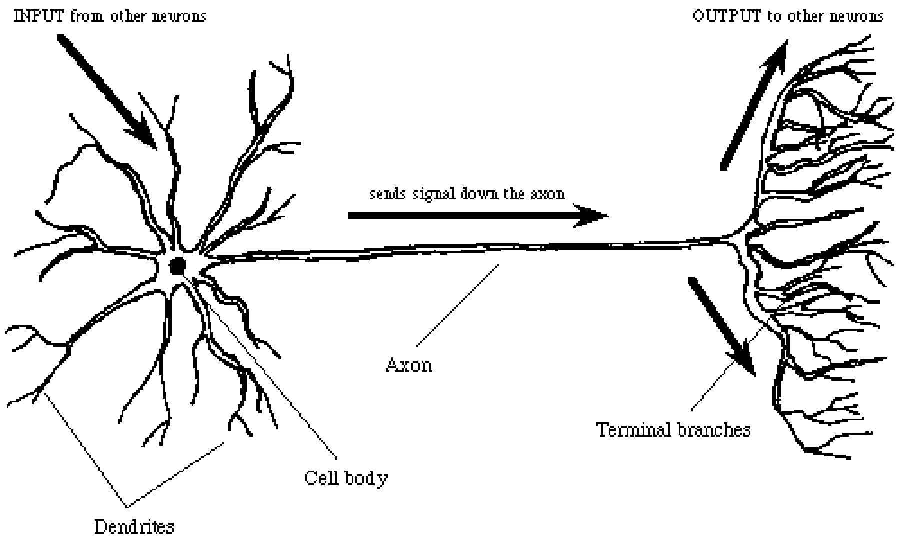

Figure 2.1.: Schematic drawing of a biological neuron

In the case of the artificial neuron, the dendrites are replaced by a number of slots through which a set of numerical *inputs*, a_j, can be introduced to the neuron. Each input is multiplied by a *weight* coefficient, w_j, depending on which dendrite it arrives in. These weight are as such equivalent to the different synaptic strengths. The weighted inputs are summed and an *offset* coefficient, c, is added to give the *net input*, n.

$$n = \sum_{j=1}^{m} w_j \cdot a_j + c \tag{2.1}$$

This notation is often simplified by considering the offset as an additional weight for an input slot that always receives a one, i.e. $w_0 = c$ and $a_0 = 1$.

$$n = \sum_{j=0}^{m} w_j \cdot a_j = \mathbf{W} \cdot \mathbf{a} \tag{2.2}$$

where $\mathbf{W} = \begin{bmatrix} w_0 & \cdots & w_m \end{bmatrix}$

and $\mathbf{a} = \begin{bmatrix} a_0 & \cdots & a_m \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} a_0 \\ \vdots \\ a_m \end{bmatrix}$

are written in matrix notations.

The net input is passed through a *transfer function*, $f(x)$, in order to arrive at the neuron *output*, b.

$$b = f(n) = f(\mathbf{W} \cdot \mathbf{a}) \tag{2.3}$$

This is the value that the neuron transmits to other neurons in the network, equivalent to the signal sent along the axon.

2.1.3. The Transfer Function

The transfer function of the neuron is of particular interest as it determines both the range and the distribution of the neuron's output. One widespread transfer function is the binary classifier:

$$f_{\text{bin}}(n) = \begin{cases} 0 & n < 0 \\ 1 & n \geq 0 \end{cases} \tag{2.4}$$

In fact, a neuron using this even has its own name, perceptron.

The simplest transfer function may be the linear function:

$$f_{\text{lin}}(n) = n \tag{2.5}$$

Compare (2.5) to the equally common log-sigmoid:

$$f_{\text{sig}}(n) = \frac{1}{1 + e^{-n}} \tag{2.6}$$

In some cases the transfer function can include stochastic properties. For example, a function of the net input may be used to set the probabilities of a randomly generated output. This however lies outside the scope of this project.

The transfer function may be selected individually for each neuron, however it is common practice that groups of similar neurons uses the same function. Different transfer function are useful for different purposes.

2.1.4. The Network Topology

As implied, a network usually consists of more than one neuron. Each neuron can have any number of inputs, and it can transmit its result to any number of receiving neurons, including itself. It is therefore critical to consider the structure or *topology* of a neural network in order to understand it.

There are several categories of topologies but this thesis considers only one, the feed-forward neural network (FFNN). The main feature of a FFNN is that it is set up in distinct *layers*. While each layer may have any number of neurons in it, each neuron may only receive input from neurons in the previous layer, and can similarly only transmit its output to neurons in the upcoming layer, see figure 2.2. As such a FFNN contains no loops, and the output of a neuron cannot influence the input the neuron might receive at a later time. This makes a FFNN time-consistent in that a given set of network input signals fed to the first layer of the network it will always produce the same network outputs.

Defining the topology of a FFNN comes down to two parameters. The first is the number of layers, K, to be used. The second is the number of neurons in each layer, m_k. The last layer, known as the *output layer*, must have the same number of neurons as the number of desired outputs for the entire network. The other layers, known as *hidden layers*, may have any number of neurons and they do not have to equal the number of network inputs, the number of network outputs, or the be the same in all hidden layers.

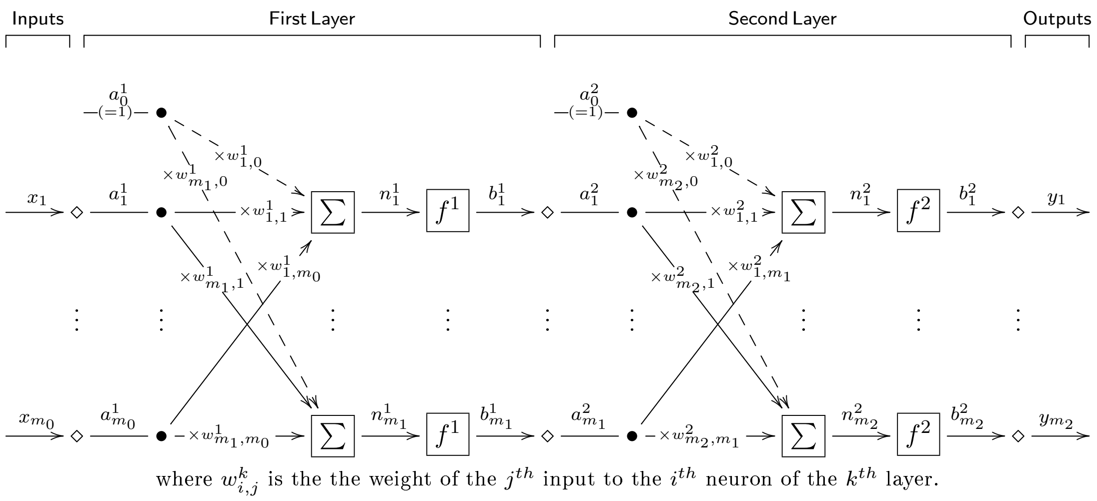

Figure 2.2.: A two-layered artificial neural network

2.1.5. Error function

One important aspect training is to determine whether of not a change made to the weights was an improvement or not. This is achieved by considering the *error function*, E, and attempting to minimize it. For most training algorithms it is a requirement that the error function must be scalar, i.e. there may only be one neuron in the output layer. A commonly used error function is the summed squared error between the network's final output , $b_{1,q}^K$, and the known corresponding function value, y_q.

$$E = \sum_{q=1}^{Q} \left(b_{1,q}^K - y_q\right)^2 \tag{2.7}$$

The idea is to consider all Q of the I/O-pairs as constants for the duration of the training, and instead attempt to express the error as a function of the network's weights.

$$E = F(\mathbf{w}) \tag{2.8}$$

where $\mathbf{w} = \left[\begin{array}{cccccccc} w_{1,0}^1 & \cdots & w_{1,m_0}^1 & w_{2,0}^1 & \cdots & w_{m_1,m_0}^1 & w_{1,0}^2 & \cdots & w_{m_K,m_{K-1}}^K \end{array}\right]^{\mathsf{T}}$

Another useful way of writing the error function is:

$$E = \mathbf{e}(\mathbf{w})^{\mathsf{T}} \cdot \mathbf{e}(\mathbf{w}) \tag{2.9}$$

with the *error vector*, $\mathbf{e}(\mathbf{w}) = \left[\begin{array}{ccc} e_1(\mathbf{w}) & \cdots & e_Q(\mathbf{w}) \end{array}\right]^{\mathsf{T}}$
and $e_q(\mathbf{w}) = b_{1,q}^K(\mathbf{w}) - y_q$.

The summed square error, E, while being very useful for actual training, is not very suitable for comparison between different topologies, different sets of I/O-pairs or different functions. The problem is that as Q increases, the error will increase for the simple reason that there are more error components to sum. This in spite of the fact that more I/O-pairs will lead to a generally more accurate response surface. The solution is to instead use the root-mean-square error for comparison:

$$E_{\text{RMS}} = \sqrt{\frac{1}{Q} \cdot E} \tag{2.10}$$

This error measure will decrease (or increase) with E, it is more independent of the number of I/O-pairs, and the square root gives it a real interpretation as a measure of the average distance between the network's response surface and the real function.

2.1.6. Determinacy of a system

The training of neural network, regardless of what method is used, is a matter of adjusting a number of coefficients, $w^k_{i,j}$, to fit a number of I/O-pairs, $(\mathbf{x}_q, y_q)$. In order to say something about the determinacy of the system, it is necessary to known the number of weights in the entire network.

For each layer, k, there are $m_{k-1}+1$ input signals to each one of the layer's m_k neurons. Therefore the total number of weights can be calculated as:

$$P = \sum_{k=1}^{K} m_k \cdot (m_{k-1} + 1) \tag{2.11}$$

There are now three possibilities

- $P > Q$ which is referred to as an underdetermined system. For a system like this there is not enough information in the I/O-pairs to uniquely determine the optimal values of the weights. As a result there may be several solutions that are equally accurate for the given points, but at the same time very different between these points.
- $P = Q$ which is referred to as a determined system. A system like can have exactly one solution that is accurate for all given I/O-pairs. The chance of this situation occurring in a real scenario however, is minuscule.
- $P < Q$ which is referred to as an overdetermined system. This system has more information than coefficients to set. As a result there will most likely be no perfect solution. There will however be a solution that optimizes whatever error function has been selected. This solution will also be more robust against changes in the I/O-pairs, which means it is usually better at generalizing to the function as a whole.

2.2. Training with Backpropagation

Backpropagation (BP) is a precise method of calculating favorable weight adjustments for a FFNN in training. It does so first and foremost by considering the derivatives of the error function with regards to the network's weights. Unfortunately, the error function of equation (2.7) is too complex to minimize analytically. It is instead necessary to use an iterative approach. This consists of calculating the error for a fixed set of weights, $\widetilde{\mathbf{w}}$, and determining an *update vector*, $\mathbf{u}$, such that the new set of weights, $\widetilde{\mathbf{w}}_{t+1} = \widetilde{\mathbf{w}}_t + \mathbf{u}_t$, has a lower error. This process can be repeated until the error is sufficiently small.

2.2.1. Steepest decent

The steepest decent method is the most straight forward of the iterative methods. The first step is to make a first order approximation of F in the vicinity of $\widetilde{\mathbf{w}}$.

$$\widetilde{F}(\mathbf{w}) = F(\widetilde{\mathbf{w}}) + \mathbf{g}(\widetilde{\mathbf{w}})^{\mathsf{T}} \cdot (\mathbf{w} - \widetilde{\mathbf{w}}) \tag{2.12}$$

where $\mathbf{g}(\mathbf{w}) = \nabla F(\mathbf{w})$ is the *gradient* of the original error function.

$\nabla = \left[\frac{\partial}{\partial \bar{w}_1} \ \cdots \ \frac{\partial}{\partial \bar{w}_P} \right]^{\mathsf{T}}$ is the column derivative operator, where $\bar{w}_p$ is the p^{th} element of the weight vector $\mathbf{w}$.

This approximated function is linear and as such, it does not have a minimum. However the direction in which the error will most rapidly vary is the gradient. As such the update is set according to $\mathbf{u} = \lambda \cdot \mathbf{g}(\widetilde{\mathbf{w}})$, where λ is a scalar factor. Since the real error function is most likely not linear, a line search must be performed to determine the optimal value of λ. In order to avoid excessive computation, λ is often kept fixed or varied according to set rules, rather than optimized for every step. This can often lead to the final result oscillating around a minimum.

2.2.1.1. Calculation of the gradient vector

In order to determine the numerical value of the gradient vector, it helps to consider the error function in the form of equation (2.9).

$$\begin{aligned}
\mathbf{g}(\mathbf{w}) &= \nabla \left(\mathbf{e}(\mathbf{w})^{\mathsf{T}} \cdot \mathbf{e}(\mathbf{w}) \right) \\
&= \nabla \left(\sum_{q=1}^{Q} \left(e_q(\mathbf{w}) \right)^2 \right) \\
&= 2 \cdot \sum_{q=1}^{Q} \left(\nabla e_q(\mathbf{w}) \right) \cdot e_q(\mathbf{w}) \\
&= 2 \cdot \left[\nabla e_1(\mathbf{w}) \ \cdots \ \nabla e_Q(\mathbf{w}) \right] \cdot \mathbf{e}(\mathbf{w}) \\
&= 2 \cdot \mathbf{J}(\mathbf{w})^{\mathsf{T}} \cdot \mathbf{e}(\mathbf{w})
\end{aligned} \tag{2.13}$$

where $\mathbf{J}(\mathbf{w})$ is the *Jacobian matrix* of $\mathbf{e}(\mathbf{w})$.

The numerical value of $\mathbf{e}(\mathbf{w})$is easily obtained from the results of running the network. The difficulty lies in evaluation the Jacobian matrix.

2.2.1.2. Calculation of the Jacobian matrix

In order to determine the numerical value of the Jacobian matrix, it helps to consider the generalized element of this matrix, $\frac{\mathrm{d}e_q}{\mathrm{d}w^k_{i,j}}$.

The chain rule splits this expression as follows:

$$\frac{\partial e_q}{\partial w^k_{i,j}} = \frac{\partial n^k_{i,q}}{\partial w^k_{i,j}} \cdot \frac{\partial n^K_{1,q}}{\partial n^k_{i,q}} \cdot \frac{\partial e_q}{\partial n^K_{1,q}} \tag{2.14}$$

The third term is simply the derivative of the output layer's transfer function:

$$\begin{aligned} \frac{\partial e_q}{\partial n^K_{1,q}} &= \frac{\partial}{\partial n^K_{1,q}} \left(b^{K,(0)}_{1,q} - y_q \right) \\ &= b^{K,(1)}_{1,q} \end{aligned} \tag{2.15}$$

where $b^{k,(\nu)}_{i,q} \equiv \frac{\partial^\nu}{\partial\left(n^k_{i,q}\right)^\nu} f^k\left(n^k_{i,q}\right)$.

If f^k is the linear function then,

$$\begin{aligned} b^{k,(1)}_{i,q} &= \frac{\partial}{\partial n^k_{i,q}} \left(n^k_{i,q} \right) \\ &= 1 \end{aligned}$$

while if f^k is the log-sigmoid function,

$$\begin{aligned} b^{k,(1)}_{i,q} &= \frac{\partial}{\partial n^k_{i,q}} \left(\frac{1}{1+e^{-n^k_{i,q}}} \right) \\ &= e^{-n^k_{i,q}} \cdot \left(1+e^{-n^k_{i,q}}\right)^{-2} \\ &= b^{k,(0)}_{i,q} \cdot \left(1+e^{-n^k_{i,q}} - 1\right) \cdot \left(1+e^{-n^k_{i,q}}\right)^{-1} \\ &= b^{k,(0)}_{i,q} \cdot \left(1 - b^{k,(0)}_{i,q}\right) \end{aligned}$$

The first term is evaluated in accordance with equation (2.2).

$$\begin{aligned} \frac{\partial n^k_{i,q}}{\partial w^k_{i,j}} &= \frac{\partial}{\partial w^k_{i,j}} \left(\sum_{l=0}^{m_{k-1}} w^k_{i,l} \cdot a^k_{l,q} \right) \\ &= b^{k-1,(0)}_{j,q} \end{aligned} \tag{2.16}$$

The numerical values of both these terms are easily obtained from the results of running the network.

The second term of equation (2.14) is referred to as the *sensitivity*:

$$s^k_{i,q} = \frac{\partial n^K_{1,q}}{\partial n^k_{i,q}} \tag{2.17}$$

.

The sensitivity for the output layer ($k = K$) is obviously equal to 1.
For earlier layers ($k < K$) the chain rule is applied.

$$\begin{aligned}
s^k_{i,q} &= \sum_{l=1}^{m_{k+1}} \left(\frac{\partial n^{k+1}_{l,q}}{\partial n^k_{i,q}} \cdot \frac{\partial n^K_{1,q}}{\partial n^{k+1}_{l,q}} \right) \\
&= \sum_{l=1}^{m_{k+1}} \left(\frac{\partial}{\partial n^k_{i,q}} \left(\sum_{j=0}^{m_k} w^{k+1}_{l,j} \cdot a^{k+1}_{j,q} \right) \cdot s^{k+1}_{l,q} \right) \\
&= \sum_{l=1}^{m_{k+1}} \left(\frac{\partial}{\partial n^k_{i,q}} \left(\sum_{j=1}^{m_k} w^{k+1}_{l,j} \cdot b^{k,(0)}_{j,q} + w^{k+1}_{l,0} \right) \cdot s^{k+1}_{l,q} \right) \\
&= \sum_{l=1}^{m_{k+1}} \left(\frac{\partial}{\partial n^k_{i,q}} \left(b^{k,(0)}_{i,q} \right) \cdot w^{k+1}_{l,i} \cdot s^{k+1}_{l,q} \right) \\
&= b^{k,(1)}_{i,q} \cdot \sum_{l=1}^{m_{k+1}} \left(w^{k+1}_{l,i} \cdot s^{k+1}_{l,q} \right)
\end{aligned} \tag{2.18}$$

Using this and working recursively, all sensitivities can be expressed in terms of the first order derivatives of the layers' transfer functions. For this method to be viable it is therefore necessary that each transfer function has a first order derivative. Discontinuous function such as the binary classifier may not be used.

2.2.2. Newton's method

The Newton method is based on a second order approximation of F in the vicinity of $\widetilde{\mathbf{w}}$.

$$\widetilde{F}(\mathbf{w}) = F(\widetilde{\mathbf{w}}) + \mathbf{g}(\widetilde{\mathbf{w}})^{\mathsf{T}} \cdot (\mathbf{w} - \widetilde{\mathbf{w}}) + \frac{1}{2} (\mathbf{w} - \widetilde{\mathbf{w}})^{\mathsf{T}} \cdot \mathbf{H}(\widetilde{\mathbf{w}}) \cdot (\mathbf{w} - \widetilde{\mathbf{w}}) \tag{2.19}$$

where $\mathbf{H}(\mathbf{w}) = \nabla \cdot \nabla^{\mathsf{T}} F(\mathbf{w})$ is the *Hessian matrix* of the original error function.

This approximated function is quadratic and as such, it does have a minimum. After considering the derivative of equation (2.19), the update is set to solve:

$$\mathbf{H}(\widetilde{\mathbf{w}}) \cdot \mathbf{u} = -\lambda \cdot \mathbf{g}(\widetilde{\mathbf{w}}) \tag{2.20}$$

Unfortunately the update vector given by the equation does not always constitute an reduction for the error function, E. As a result a line search is often performed to determine the optimal value of its magnitude.

2.2.2.1. Calculation of the Hessian matrix

In order to determine the numerical value of the Hessian matrix, it helps to consider it as the gradient of the transposed gradient vector:

$$\begin{aligned}
\mathbf{H}(\mathbf{w}) &= \nabla \cdot \left(\mathbf{g}(\mathbf{w})^{\mathsf{T}}\right) \\
&= \nabla \cdot \left(2 \cdot \sum_{q=1}^{Q} \left(\nabla^{\mathsf{T}} e_q(\mathbf{w})\right) \cdot e_q(\mathbf{w})\right) \\
&= 2 \cdot \sum_{q=1}^{Q} \left(\left(\nabla \cdot \nabla^{\mathsf{T}} e_q(\mathbf{w})\right) \cdot e_q(\mathbf{w}) + \left(\nabla e_q(\mathbf{w})\right) \cdot \left(\nabla^{\mathsf{T}} e_q(\mathbf{w})\right) \right) \\
&= 2 \cdot \left(\mathbf{G}(\mathbf{w}) + \mathbf{J}(\mathbf{w})^{\mathsf{T}} \cdot \mathbf{J}(\mathbf{w})\right) \qquad (2.21)
\end{aligned}$$

As such the Hessian can be split into two terms. The second consists solely of the Jacobian matrix, which has already been calculated during the evaluation of the gradient.

The first term on the other hand, contains the second derivatives and does therefore present a significant difficulty. For a full evaluation see appendixA.1.

2.2.3. Approximation of the Hessian matrix

Due to the complexity of the full analytical solution, the amount of computations needed to give the exact Hessian matrix is substantial. It therefore becomes a necessity to reduce said amount, by constructing an approximate Hessian.

2.2.3.1. Gauss-Newton

The Gauss-Newton algorithm (GN) is very straight forward. It is based on the approximation that:

$$\left| \left(\nabla \cdot \nabla^{\mathsf{T}} e_q(\mathbf{w})\right) \cdot e_q(\mathbf{w}) \right| \ll \left| \left(\nabla e_q(\mathbf{w})\right) \cdot \left(\nabla^{\mathsf{T}} e_q(\mathbf{w})\right) \right| \qquad (2.22)$$

This is valid as long as the values of the error, e_q, are close to zero. Since the minimum of the error is zero (or almost zero), this can be assumed to be the true in the vicinity of the minimum. The approximation leads to the expression:

$$\mathbf{H}(\mathbf{w}) = 2 \cdot \mathbf{J}(\mathbf{w})^{\mathsf{T}} \cdot \mathbf{J}(\mathbf{w}) \qquad (2.23)$$

In order for equation (2.20) to have a unique solution it is required that the Hessian is of full rank. With this approximation, that translates to the necessity of the number of I/O-pairs, Q, being greater than the network's total number of weights, P.

2.2.3.2. Broyden-Fletcher-Goldfarb-Shanno

The Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS) is a different way of obtaining the Hessian matrix needed for Newton's method. The idea being that by continuously adjusting the Hessian at each iteration, a close approximation can be made.[2]

Before the BFGS can begin, the matrix $\mathbf{H}_t$ must be initialized. The simplest place to start is the identity matrix, i.e. $\mathbf{H}_0 = \mathbf{I}$. Each iteration now consists of three steps:

1. Calculating of the current update vector, $\mathbf{u}_t$, in accordance with equation (2.20).
2. Using the updated weight vector to calculate the new gradient, and storing the difference as $\mathbf{h}_t = \mathbf{g}_{t+1} - \mathbf{g}_t$.
3. Updating the Hessian according to the formula:

$$\mathbf{H}_{t+1} = \mathbf{H}_t + \frac{\mathbf{h}_t \cdot \mathbf{h}_t^\mathsf{T}}{\mathbf{h}_t^\mathsf{T} \cdot \mathbf{u}_t} - \frac{(\mathbf{H}_t \cdot \mathbf{u}_t) \cdot (\mathbf{H}_t \cdot \mathbf{u}_t)^\mathsf{T}}{\mathbf{u}_t^\mathsf{T} \cdot \mathbf{H}_t \cdot \mathbf{u}_t} \tag{2.24}$$

2.2.4. Levenberg-Marquart method

The Levenberg-Marquart method (LM) is based on a variation of Newton's method. The LM, also known as the trust region method, is an alternative for the basic linear search, which has been shown to be far more robust, when finding an optimal update vector.[3]

Instead of only searching along the direction of the update vector, an adjustment is made to rotate the update vector back towards the direction of steepest decent. The modification lies in changing equation (2.20) to:

$$\left(\mathbf{H}\left(\widetilde{\mathbf{w}}\right) + \lambda \cdot \mathbf{I}\right) \cdot \mathbf{u} = -\mathbf{g}\left(\widetilde{\mathbf{w}}\right) \tag{2.25}$$

where $\mathbf{I}$ is the identity matrix. This is equivalent to the Newton method for $\lambda = 0$, and the steepest decent method as $\lambda \to \infty$.

Finding the optimal λ starts with a small value. During each iteration, if the update vector, $\mathbf{u}$, does not results in a reduction of the error function then it is not accepted. Instead λ is increased and an new $\mathbf{u}$ is generated. This is repeated until a satisfactory $\mathbf{u}$ has bee found. When the update vector does result in a reduction of the error function, it is applied to give the $\widetilde{\mathbf{w}}$ for the next iteration. The λ is also reduced somewhat in preparation for the next iteration.

2.3. Training without Backpropagation

The greatest problem with any BP algorithm is the need to calculate the derivatives of the error function. This tends to be a very time consuming process. One way to avoid this is to use a zeroth order algorithm such as Particle Swarm Optimization (PSO), Evolutionary Algorithms (EA), etc. These consider only the value of the target function or *fitness*, (such as the reciprocal of the error function) for a given point in its input space. The idea behind all zeroth order algorithms is to improved the convergence of the random search. This is done by employing a population of points, as opposed to a single staring point. At every iteration the fitness of each point in the population is evaluated and a new population is generated based on that information.[4]

2.3.1. Particle Swarm Optimization

PSO is based on the idea of a flock of birds. Each particle (bird) has two aspects associated with it, a position in the input space, $\mathbf{x}_q$, and a velocity through it, $\mathbf{v}_q$. The velocity of each particle is what determines its position during the next iteration. The velocity in itself is affected by three factors.

1. The particle's inertia, i.e. the velocity that the particle still has since the previous iteration.
2. The particle's self-confidence, i.e. the degree to which the particle trusts that the optimal point lies close to fittest point it has itself visited, $\mathbf{x}_q^{\mathrm{pb}}$.
3. The particle's social trust i.e. the degree to which the particle trusts that optimal point lies close to the fittest point visited by any particle in the swarm, $\mathbf{x}^{\mathrm{sb}}$.

Both the last two terms determine how strongly the particle will try to steer towards those points. Furthermore, in order to prevent divergence, there is an imposed speed limit on the particle's velocities.

The algorithms begins with the random initialization of the vectors $\mathbf{x}_q$ and $\mathbf{v}_q$.
At the start of each new iteration, the fitness of each particle's position is evaluated and stored as $e_{q,t}$.
If $e_{q,t} > e_q^{\mathrm{pb}}$ then $\mathbf{x}_q^{\mathrm{pb}} = \mathbf{x}_{q,t}$.
If $e_{q,t} > e^{\mathrm{sb}}$ then $\mathbf{x}^{\mathrm{sb}} = \mathbf{x}_{q,t}$.
The velocity vectors are updated according to:

$$v_{p,q,t} = \omega_t \cdot v_{p,q,t-1} + 2 \cdot \varphi \cdot \left(x_{p,q}^{\mathrm{pb}} - x_{p,q,t}\right) + 2 \cdot \varrho \cdot \left(x_p^{\mathrm{sb}} - x_{p,q,t}\right) \tag{2.26}$$

where $v_{p,q,t}$ is the p^{th}component of $\mathbf{v}_{q,t}$, ω_t is the inertia weight that may vary over time, andφ and $\varrho$are uniform random numbers in $[0,1]$.
Once the update is completed, it is verified that no velocities violate the speed limit.
The final step for each iteration is to update the position vectors, i.e. $\mathbf{x}_{q,t+1} = \mathbf{x}_{q,t} + \mathbf{v}_{q,t}$

3. Implementation

3.1. Basic structure

From the very start of the project the basic structure of the programs has remained the same. It is best described divided into three parts:

1. The initiation of all necessary variables. This firstly includes the setting of important topological and computational parameters. Since these are often modified by the operator they are accessibly located at the top. Next comes the introduction of all the training data. Lastly the data is reformatted in order to make the later training process more efficient.

2. The training loop. Before the loop can being the weights as well as a few needed parameters must be initialized. Since nothing is known about what would be suitable weights, they are randomly generated. The first step of each iteration begins with the calculation of the network's outputs and the comparison of them with the known function values to find the current error vector. The error vector is then used to determine the sensitivities and with the addition of a simple line search, a suitable update vector is calculated. Finally the necessary information about the current iteration is saved in fitting parameters.

3. The displaying of the results. One of the most important pieces of information is the evolution of the network's total error, which is both highly significant as well as easily plotted. (For examples see chapter 4) Another important piece of information is the minimum point of the trained network's response surface. For comparison, the minimum point of a differently calculated response surface is also presented. As a finishing treat, if the dimensionality of the test allows it, the response surface itself is plotted with the I/O-pairs juxtaposed.

One thing that quickly became apparent is that there is much flexibility to be gained by keeping commonly used functions as separate files. The benefit of course is that the central program becomes less cluttered. It also makes it possible to make adjustments to the functions that immediately apply everywhere. For that reason, all the code for the data initiation has been placed in its own subroutine. The same thing goes for the code regarding most of what is to be displayed as results.

3.2. Initiation of variables

3.2.1. Topology settings

From the start of the project it had been decided that any ANN under investigation would have a fixed number of two hidden layers. The neurons in these layers will all employ a log-sigmoid transfer function and the output layer will use a simple linear transfer function. However the number of neurons in each hidden layer, m_1 and m_2, remains to be determined.

One option is to leave this setting to the judgment of the operator. This however requires that the operator has a lot of experience or else a lot of time will be wasted using basic trial and error. It would be far more preferable if a mathematically robust method of calculating a suitable number of hidden neurons could be found.

3.2.1.1. Optimal over iterations

The purpose of the ANN is to find a response surface with as low an error as possible. For that reason the first attempt at determining a suitable topology was set up to loop through a number of different topologies. For each it would train a network using the same I/O-pairs, and then compare the networks' errors after a given number of iterations.

While this did produce clear results, it became obvious that something was wrong with the way the task had been formulated. As had been expected, the larger networks were able to produce much smaller total errors, but the time it took them to do so however, was staggering. Several attempts to create a weighted error index, where the networks would be punished for holding more neurons, proved futile. Most of all the approach could not be generalized to work well across different functions.

3.2.1.2. Optimal over time

The failure of the previous attempt led to some consideration. The conclusion was that what would actually be optimal was the ability to achieve a low error in a set amount of time. For that reason a timer was introduced into the program, and it was instructed to run training iterations on each topology until a set amount of time had passed. This meant that the smaller networks, for which each iteration took less time, was able to perform a greater number of iterations compares to the larger networks.

Using this approach provided a clear optimal structure each time it was performed. Unfortunately, that structure was not necessarily consistent over consecutive runs. The apparent reason for this is that basing the optimization on the timer means that the results are affected by what other processes are being run simultaneously on the computer. It would also give very different results on computers with different processing speeds. While this method had its merits it is not robust enough for the current application.

3.2.1.3. Optimal in determinacy

In the end, the approach of evaluating several topologies every time the program was run was deemed too time consuming. It was not until the issues of determinacy of the system was considered, that a solution was found. Because one way to safeguard against the ambiguities of an underdetermined system is to set a topology making that impossible.

Firstly the decision was made to always set $m_2 = m_1$. The inverse of equation (2.11) then gives that:

$$m_1 = \sqrt{P - 1 + \frac{1}{4}\left(m_0 + 3\right)^2} - \frac{1}{2}\left(m_0 + 3\right) \tag{3.1}$$

In order to have a definitely overdetermined system $P < Q$. Since it is also a requirement that $m_1$is and integer, the final formula used becomes:

$$m_1 = \left\lfloor \sqrt{\frac{3}{4}Q - 1 + \frac{1}{4}\left(m_0 + 3\right)^2} - \frac{1}{2}\left(m_0 + 3\right) \right\rfloor \tag{3.2}$$

Since both Q and m_0 are properties of the input data, this formula can be quickly applied to calculate the largest still overdetermined topology.

In some of the test cases the situation may be reversed. That is that the operator specifies the topology before the program generates the I/O-pairs. Then a modified equation (2.11) can be used to calculate the lowest needed value of Q.

$$Q = \left\lceil \frac{4}{3}\left(m_1^2 + \left(m_0 + 3\right) \cdot m_1 + 1\right) \right\rceil \tag{3.3}$$

3.2.2. Reformatting of data

Within the initiation subroutine, all the data is saved as a single matrix. Each I/O-pair is stored as a row with in this matrix, where the input variable each have their own column and the last column contains the function values. This matrix never modified during the remainder of the run. Instead a copy, $\mathbf{X}$, is taken that can be modified freely.

One challenge that arose was the limited sensitivity of the transfer functions. The log-sigmoid functions used in the hidden layers is almost flat some distance away from the origin. Since training a network consist of amplifying or dampening the randomly initiated aspect of the network, it is important that the data points lie close to where these disturbances first appear, i.e. the origin.

The simplest way of circumventing this problem is to translate and rescale the data. For each column in $\mathbf{X}$, the statistical mean, μ_j and standard deviation, σ_j, is computed. The columns are then transformed according to:

$$\mathbf{x}_j = \frac{\mathbf{x}_j - \mu_j}{\sigma_j} \tag{3.4}$$

The network is then trained on this data instead. Important to remember is of course that the final output of the network need to be reversely transformed in order to find the correct result. However, this transformation will in no way effect the accuracy of finding the minimum.

In some cases further transformations may be needed. The case of the 5-member truss, (see section4.2) for example provided very inconsistent result in early trials. This was resolved by applying the transformation:

$$\mathbf{x}_j = \log_{10}(\mathbf{x}_j) \tag{3.5}$$

as well, before applying the transformation of (3.4).

3.3. Training loop

3.3.1. Initiating the weight vector

The first step of the training loop is to determine a suitable staring point for the weight vector. This is a difficult task, since the vector space may include many local minima, and there is no clear way of knowing where they are or how to avoid them. Only after training has been carried out can the results be used to determine whether or not the run was successful.

To combat this a loop has been set up around the actual training loop. The weights are randomly initiated within this loop. Training is performed and the lowest comparable error achieved is computed. If this is lower than the previously lowest comparable error the the current weight vector is saved. By performing many such independent training runs, the chance that one of them will find the global minimum increases.

3.3.2. Overfitting

As the dimensionality of the inputs grows larger, it becomes necessary to increase the number of neurons in each hidden layer. More neurons does mean more flexibility for the response surface, but it also means that the total number of adjustable weights grows rapidly. This can lead to one of the most troublesome challenges of constructing any function estimator, namely *overfitting*.

Overfitting is the process where the training is so focused on reduced the error for the I/O-pairs given, that it sacrifices the general shape of the response surface. This is especially dangerous if data from real measurements are used. Since such data will always contain measurement errors, even the true response surface will not be error free. Due to the risk of overfitting is it vital to monitor not only the progress of the error of the I/O-points used for training, but also how well the network performs in other points.

The simplest way to monitor for overfitting is to split the available data into two sets, the learning set and the test set. A reasonable ratio is to use 80% of the data for learning and 20% for testing. This of course has the downside that even less data is available for training the network. In this program the split is performed by sorting the rows of

$\mathbf{X}$ according to their function values. Starting with the third row, every fifth row is extracted and placed in a separate matrix, $\mathbf{X}_0$.

The training of the network it carried out using the data still remaining in $\mathbf{X}$. At the end of each iteration however, the comparable error is computed for both sets combined. The result of this will be an initial decrease in the comparable error as the network correctly adapts to the data. After a certain point however, if and when overfitting occurs, the comparable error will begin to increase again as the contributions of the test set starts to grow. By recording the network's weight vector at the minimum of this process a reasonably accurate response surface devoid of overfitting can be determined.

3.4. Display of results

3.4.1. Finding the minimum

While it is not the main focus of this thesis, a significant part of the task is to locate the minimum point of the estimated function. There are many ways of finding a minimum, and most of them are quite complicated. If any method involving the function's derivatives is to be used, then those derivatives must first be calculated. Since such methods tend to be iterative, this would have to be done repeatedly, a task in and of itself. There is also the risk of getting stuck in local minima.

A 0^{th}-order method, such as PSO, does not require derivatives and during the development of this program it was for a while used to great success. The time-consumption of an iterative process still remain however.

The final version of the program makes use of MATLAB proficiency in handling large matrices. Due to the nature of the neural network, the final output can be written as:

$$b^3 = f_{\text{lin}}\left(\mathbf{W}^3 \cdot f_{\text{sig}}\left(\mathbf{W}^2 \cdot f_{\text{sig}}\left(\mathbf{W}^1 \cdot \mathbf{a}^1\right)\right)\right) \tag{3.6}$$

where $\mathbf{W}^k = \begin{bmatrix} w^k_{1,0} & \cdots & w^k_{1,m_{k-1}} \\ \vdots & \ddots & \vdots \\ w^k_{m_k,0} & \cdots & w^k_{m_k,m_{k-1}} \end{bmatrix}$

Let $\mathbf{a}^1$instead of representing a single input point as a column vector, be a matrix where each column specified a different input point. The final output of the network could then be calculated for all these points simultaneously, and with great speed. This allows for the possibility of placing a tight grid of the domain of interest, evaluating the function at all these points, and thereby finding the minimum by a method similar to an exhaustive search. While this method will not be the most precise, the inaccuracy of the response surface itself makes the pursuit of much further precision redundant.

An added bonus of this method is that the same function values found in the exhaustive search can be reused for the purpose of plotting the response surface itself.

3.4.2. Quadratic Regression Model

The quadratic regression model (QRM) is an alternative way of constructing a response surface to estimate a function. The reason it is included here is to serve as a reference for the network's response surface to be evaluated against. The basis of the QRM is to consider the function to be estimated as a multivariate, second order polynomial:

$$y\left(\mathbf{x}\right)=\sum_{i=1}^{m_0}\left(\sum_{j=1}^{i}\left(\alpha_{i,j}\cdot x_i\cdot x_j\right)\right)+\sum_{i=1}^{m_0}\left(\beta_i\cdot x_i\right)+\gamma \tag{3.7}$$

Such a function has $P_{\text{QRM}}=\frac{1}{2}\left(m_0+2\right)\left(m_0+1\right)$ variable coefficients. As a result the QRM will tend to remain overdetermined longer for cases where less data is available. As long as it is overdetermined, all the coefficients can be optimized in one step using the least square method.

Once the parameters have been set, is it a simple matter to calculates the QRM's value at any point within the domain. When comparing the two response surfaces at the end of the program's run, the parameters of the QRM are recomputed before the I/O-pairs are used to calculate the models comparable error. The same method as for the network's response surface is then used to determine the minimum point of the QRM. In most cases where larger amounts of data are available, the superiority of the neural network is expected to be evident.

4. Testing

The results displayed in this chapter are all gathered using the final version of the program. In this version, 50 randomly initiated networks are each trained over 500 iterations. The single network with the lowest comparable error is taken as the final result, and compared to the response surface of a quadratic regression model 3.4.2.

4.1. Benchmarks functions

The first tests were carried out using basic benchmark functions. The purpose of this is to determine that the method of training an ANN can be a significant improvement compared to the QRM.

In order to generate the I/O-pairs, a simple square grid is placed over the domain of interest. The number of hidden neurons is specified by the operator, and the program then calculates how many I/O-pairs are necessary to prevent the system from being underdetermined. In the 2D cases, the number I/O-pairs deemed necessary have been rounded upward to the nearest square number.

4.1.1. 1-dimensional

4.1.1.1. $y(\mathbf{x}) = \sin(x_1)$

This network uses 22 I/O-pairs, since it requires the use of 2 neurons in each hidden layer.

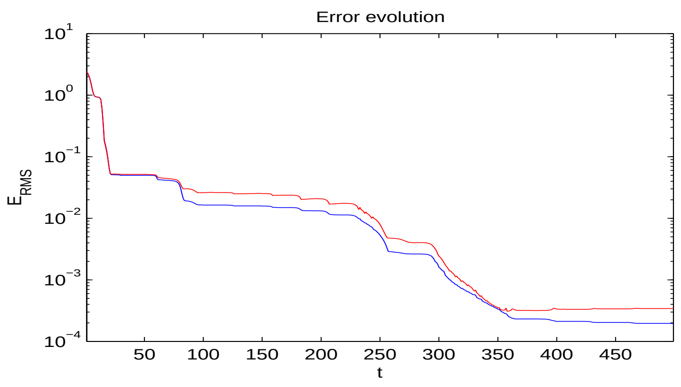

Figure 4.1.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

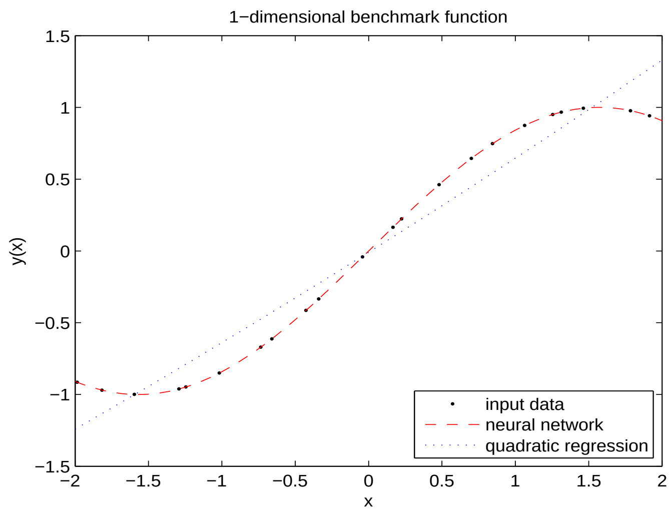

Figure 4.2.: Final response surface of the trained ANN, as well as of the QRM. The I/O-pairs used have also been included.

Table 4.1.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	3.10×10^{-4}	0.217
x_1^*	$-\frac{\pi}{2} \approx -1.57$	-1.56	-2
y^*	-1	-1.00	-1.24

The comparable errors between the two models differ with a factor of $\approx 10^3$.

This function involves a sign shift in the curvature as the function passes through zero. The quadratic regression is unable to cope with this shift, and it's limitations can be easily seen. The network's response surface on the other hand, is able to account for the data much more accurately.

4.1.1.2. $y(\mathbf{x}) = e^{x_1}$

This network uses10 I/O-pairs, since it requires the use of only a single neuron in each hidden layer.

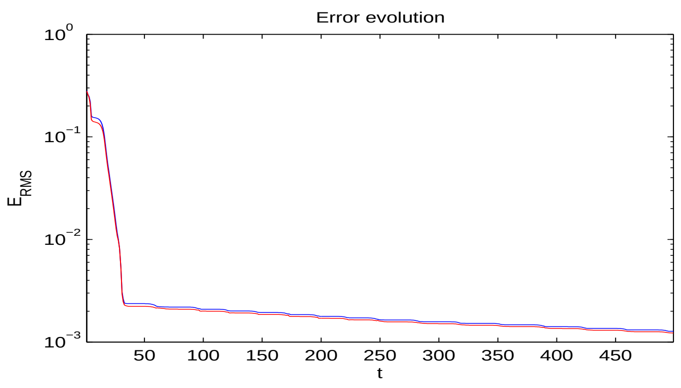

Figure 4.3.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

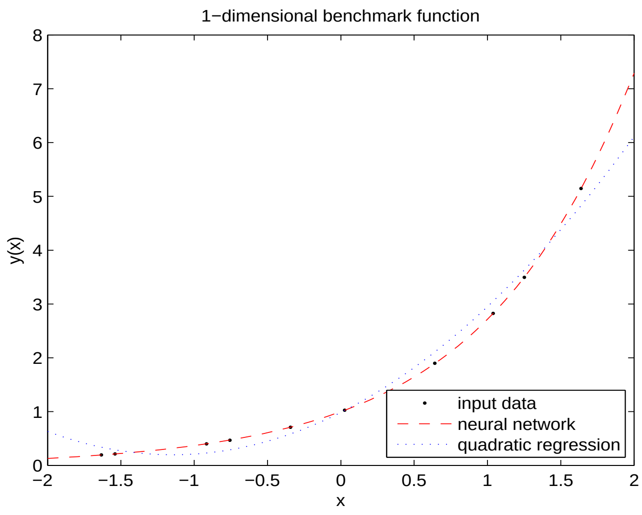

Figure 4.4.: Final response surface of the trained ANN, as well as of the QRM. The I/O-pairs used have also been included.

Table 4.2.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	1.35×10^{-3}	0.106
x_1^*	$\min(x) = -2$	-2	-1.16
y^*	0.135	0.132	0.199

The comparable errors between the two models differ with a factor of $\approx 10^2$.

This function is monotonically increasing and as such it's minima will always be the lowest end of the specified domain. The network was able to capture this behavior, while the the quadratic regression was not.

This test also illustrates the fact that, depending on the situation, the number of hidden neurons needed to build an accurate response surface can be lowered. Since this lowering leads to less data being needed to build an accurate model, this is an important result.

4.1.2. 2-dimensional

4.1.2.1. Rosenbrock function

The function is formulated as:

$$y(\mathbf{x}) \quad = \quad \sum_{i=1}^{n-1} \left(100\left(x_i^2 - x_{i+1}\right)^2 + (x_i - 1)^2\right) \tag{4.1}$$

Since this network requires the use of 2 neurons in each hidden layer, it necessitates 25 I/O-pairs.

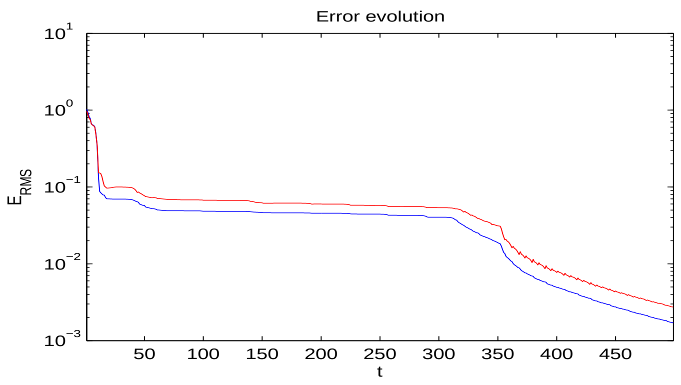

Figure 4.5.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

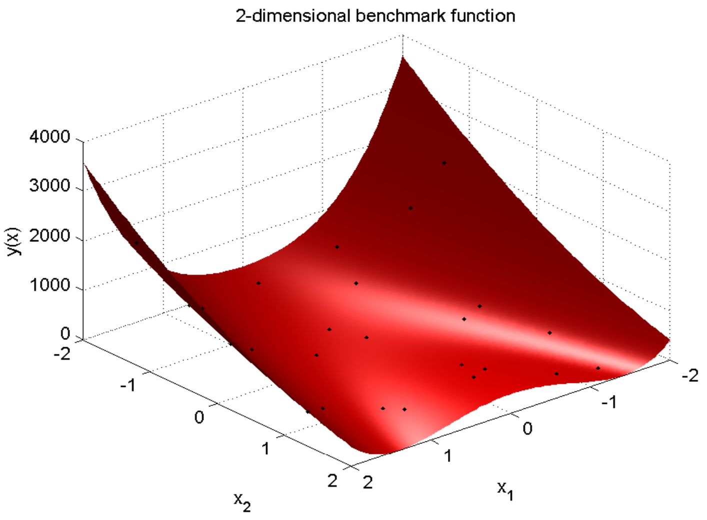

Figure 4.6.: Final response surface of the trained ANN. The I/O-pairs used have also been included.

Table 4.3.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	2.73×10^{-3}	0.439
x_1^*	1	1.4	0.08
x_2^*	1	2	1.52
y^*	0	$5.61. \times 10^{-3}$	-3.50

The comparable errors between the two models differ with a factor of $\approx 10^2$.

This function is tricky because it involves a curved trench within which the gradient is comparably slight. This makes the exact location of the minimum within the trench difficult to determine. The network's response surface did in fact indicate a point within the trench as the minimum. The real function value at this point is $y\left(\mathbf{x}_{\text{neur}}^*\right) = 0.32$. Considering that the range of the data reaches values $> 10^4$, this must be considered reasonably accurate. For comparison, the minimum of the quadratic regression has a real function value of $y\left(\mathbf{x}_{\text{quad}}^*\right) = 230$.

4.1.2.2. Sine square function

The function is formulated as:

$$y(\mathbf{x}) = \frac{1}{2} + \frac{\left(\sin|\mathbf{x}|\right)^2 - \frac{1}{2}}{\left(1 + \frac{|\mathbf{x}|^2}{1000}\right)^2} \tag{4.2}$$

Since this network requires the use of 4 neurons in each hidden layer, it necessitates 64 I/O-pairs.

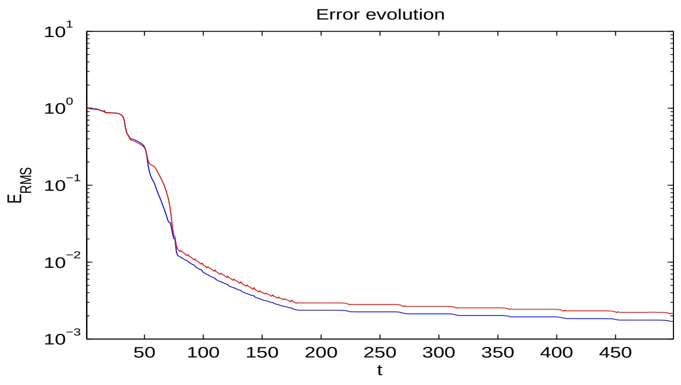

Figure 4.7.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

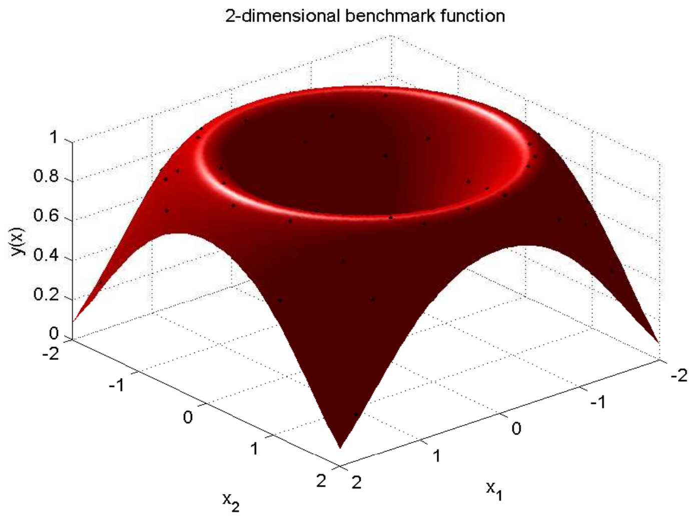

Figure 4.8.: Final response surface of the trained ANN. The I/O-pairs used have also been included.

Table 4.4.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	2.13×10^{-3}	0.983
x_1^*	0	0	-1.84
x_2^*	0	0	-2
y^*	0	$4.54. \times 10^{-4}$	0.706

The comparable errors between the two models differ with a factor of $\approx 10^2$.

This function is complicated by it's many ridges and valleys. As the domain increases and more of these are exposed, the number of neurons needed increase rapidly. Without them the network will in some cases include distortions near the domain's edge. If these happen to include a sharp drop then the network will be unable to correctly locate the minimum.

4.2. Ideal truss

4.2.1. Description

This is a test to illustrate how the comparable error can vary when the number of hidden neurons are increased. The object of study is a 5-member truss that can be seen in figure 4.9.

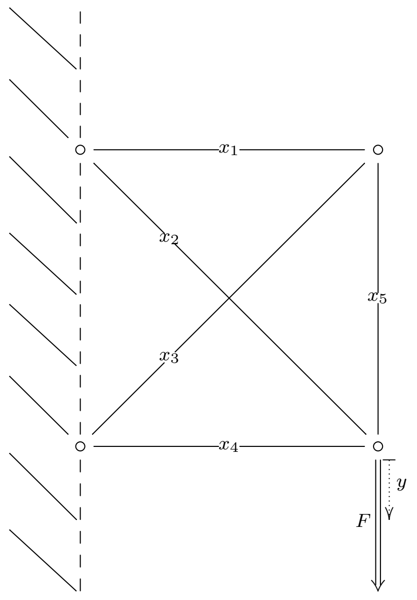

Figure 4.9.: Diagram of the 5-member truss that is to be used for the following test cases.

The input variables in this case, are the widths of the members. Each width can be varied individually, however the total area must remain fixed. The output value is the vertical displacement of bottom right node when a downward force is applied. The process for calculating this using matrix multiplication is well known,[5] albeit not short when written out.

In order to generate the I/O-pairs, a $3 \times 3 \times \ldots$ grid containing 243 points is placed over the domain of interest. Due to the area constraint, 32 of these are duplicates and are therefore removed, giving a total of 211 points. For each test on the truss, the program calculates how many I/O-pairs are necessary to prevent the system from being underdetermined, and randomly picks that amount from the 211. The function value for these points are evaluated and combined to produce the I/O-pairs.

4.2.2. Using different topologies

4.2.2.1. $m_1 = 3$

Table 4.5.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	0.231	0.228
x_1^*	0	1.01×10^{-2}	4.14×10^{-5}
x_2^*	0.414	0.403	0.414
x_3^*	0	4.03×10^{-3}	4.14×10^{-5}
x_4^*	0.414	0.403	0.414
x_5^*	0	1.01×10^{-2}	4.14×10^{-5}
y^*	9.24	8.63	8.64×10^{-2}

As these results show, the network is not able to achieve the same low error as it did in previous tests. The most noticeable aspect of this is that it was not able to fully disregard the 1^{st} and 5^{th}member, as can be seen in figure 4.11 below.

In this particular scenario the network is not able to outperform the quadratic regression, which may be an indication that the QRM is better suited for this particular problem. It is however likely that the QRM benefited from the fact that the minimum lies in a corner of the domain. One feature that stills hints at the ANN being the better model, is the fact that its minimum value lies significantly closer to the true minimum value.

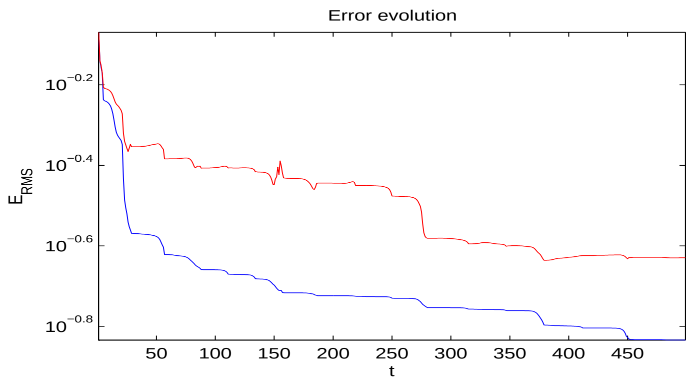

Figure 4.10.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

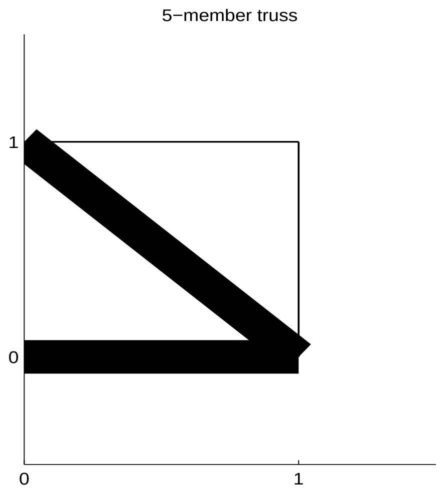

Figure 4.11.: Schematic drawing of the final minimum of the trained ANN's response surface.

4.2.2.2. $m_1 = 5$

Table 4.6.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	0.143	0.239
x_1^*	0	4.08×10^{-5}	4.14×10^{-5}
x_2^*	0.414	0.408	0.414
x_3^*	0	1.03×10^{-2}	4.14×10^{-5}
x_4^*	0.414	0.408	0.414
x_5^*	0	4.08×10^{-5}	4.14×10^{-5}
y^*	9.24	7.26	0.636

As these results show, with more hidden neurons to work with the network is able reduce it's error below that of the quadratic regression. Once again the network is not able to disregard all of the unneeded members, (see figure 4.13) in this case member 3 .

Important to note is that it required several attempts before the program was able to find this solution. Since the solution depends on the random initiation of the weight vector, this indicates that there are many local minima in the vector space and that the attraction area of the global minimum is smaller than in previous tests.

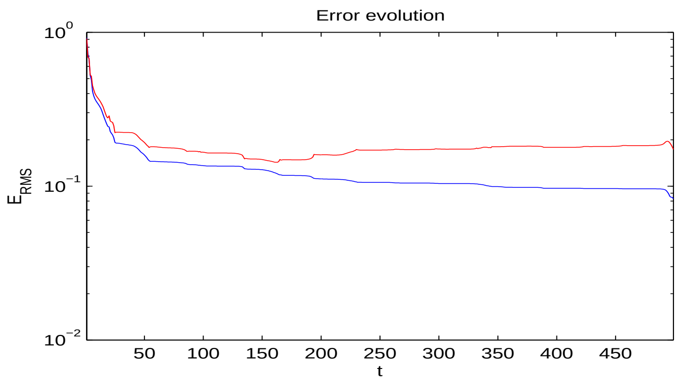

Figure 4.12.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

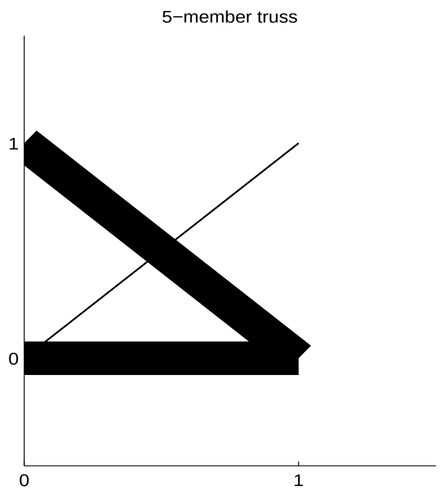

Figure 4.13.: Schematic drawing of the final minimum of the trained ANN's response surface.

4.2.2.3. $m_1 = 7$

Table 4.7.: Final comparable error of the trained ANN, as well as of the QRM. The minimum point of each function have also been included.

	$f(x)$	ANN	QRM
E_{RMS}	/	0.190	0.249
x_1^*	0	6.45×10^{-4}	4.14×10^{-5}
x_2^*	0.414	0.407	0.414
x_3^*	0	1.02×10^{-2}	4.14×10^{-5}
x_4^*	0.414	0.407	0.414
x_5^*	0	1.62×10^{-3}	4.14×10^{-5}
y^*	9.24	9.27	0.661

As these results show, there has not been much improvement compared to the previous test. There does come a point when a continued increase in the number of hidden neurons is no longer beneficial. As figure 4.15 illustrates the point of overfitting has been reached. The only true way of avoiding this without altering the topology, is to further increase the number of I/O-pairs. Since this test already uses 57 I/O-pairs, a must larger number is not feasible in a practical application.

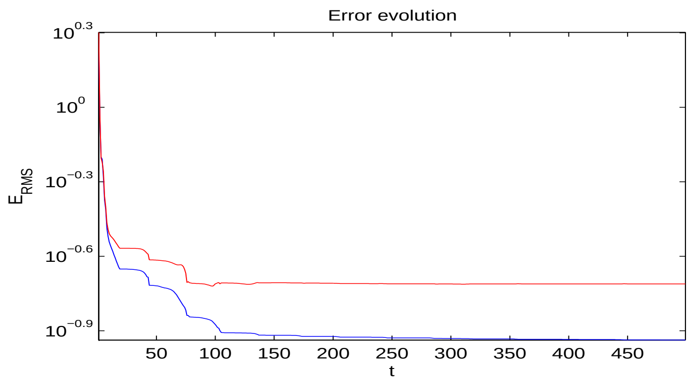

Figure 4.14.: Evolution of the comparable error over 500 iterations. The blue is for the learning set only, and the red for the learning and test sets combined.

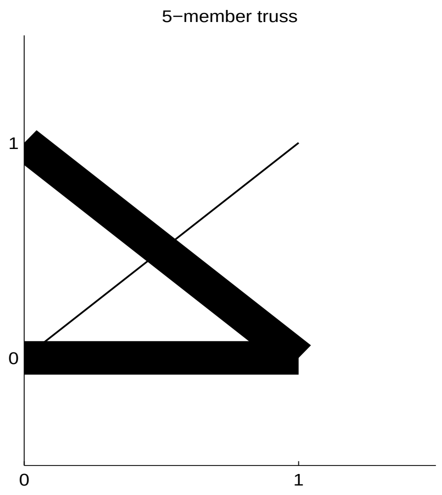

Figure 4.15.: Schematic drawing of the final minimum of the trained ANN's response surface.

5. Conclusions

5.1. Discussion

In this thesis the concept of the artificial neural network has been explored and implemented. The focus has been on the collection of training methods known as backpropagation. It has been demonstrated that the variant known as the Newton-Gauss method with the Levenberg-Marquart variation can be highly successful at training networks to a low comparable error, within short time spans. At the same time the there were several challenges encountered along the way. A significant one was the setting of the networks topology. Throughout most of the testing period this was done by manual input, which placed high demands on the operator's experience and aptitude. The automation of this process can be considered a major breakthrough.

Further challenges includes the high level of dependency that the final solution of each training cycle has on its initial weight vector. The method of training a number of independently initiated networks before selecting the most accurate one has been effective. It is however not an ideal way to proceed and it greatly increases the time-consumption of the program as whole. Time that can be much better spent.

Another major problem, that is still largely unsolved, is that of overfitting. As soon the complexity of the network increases, so does its propensity to include patches of turbulence in the final response surface. The only real cure for this is a proper balance between the number of neurons and the number of I/O-pairs. As it happens the aforementioned computerized process of determining a suitable topology is based on the idea of reducing overfitting. Despite this however, overfitting can still often be found in error evaluation plots, especially when running tests of the 5-member truss. Not surprisingly, more input dimensions appear to aggravate the condition.

On the positive side, the program in its current state was sufficiently equipped to perform a comparison with the alternate method of quadratic regression. In most scenarios where there is a sufficient number of I/O-pairs, the network method stands out as far more accurate. In remaining cases the comparisons are inconclusive. The method has also shown promise both at locating minimum points, and at closely approximating the minimum function values.

5.2. Suggested improvements

There are several ways to improve the final program. Many of which include finding better solutions for the challenges mentioned in the previous section. Particularly the question of finding suitable initial weight vectors should be addressed. The reason for its high priority is the possibly huge benefit of significantly reduced time-consumption if the issue was resolved. This thesis has not focused on the problem and there could therefore exist a simple solution ready for implementation. For starters the implementation of some kind of evolutionary algorithm.

Further improvement could be made if a time-efficient implementation of the full analytical solution, (see appendix A.1) was achieved. A comprehensive comparison between it and other method of backpropagation has not been carried out. It is possible that the rate of convergence during training could be greatly improved. It might also have an impact on the issue of overfitting.

A different method of training that was implemented but not fully evaluated was the BFGS method. An idea that was never fully realized was to construct a hybrid of this method together with the GN. It stems from the fact that the different algorithms are better suited for different situations. Testing has shown that the BFGS method is more robust for handling situations where the error is large, and is less likely to get stuck in local minima. At the same time the GN has a faster convergence rate, but is more prone to overfitting. A good strategy could therefore be to begin training the network with the BFGS-method, and once a decent results has been achieved switch for final refinement.

5.3. Recommendation

The method of using artificial neural networks for the purpose of aiding in design optimization has shown great promise. It is especially useful in cases where a lot of data, either from calculations or actual measurements are available. The program described above and the algorithms it contains are able to take the data with minimal formatting, and quickly propose a solution. Since the judgment of the operator remains important it would be beneficial to develop a graphic user interface. This would allow operators unfamiliar with the details of the algorithms to still use the program to great effect. The program has so far only been trained on one real data set, but there is not reason why it should apply itself equally well to others. Building a base of experience in using the application is necessary to determine precisely what improvements need to be address, and in what order.

The investigations of this thesis has shown that the generation of accurate response surfaces is possible. The method of using response surfaces for optimization is a necessity whenever only limited data is available. Rather then using up the time of designers, an application is set to train a network to generalize the data. It should be possible for it to return within minutes with a clear and mathematically accurate description of the case at hand.

Bibliography

[1] Martin T. Hagan, Howard B. Demuth & Mark Beale. *Neural Network Design.* PWS Publishing Company, 1996.

[2] Niclas Andéasson, Anton Evgrafov & Michael Patriksson. *An Introduction to Continuous Optimization.* Lund; Studentlitteratur, 2005.

[3] Jorge Nocedal, Stephen J. Wright. *Numerical Optimization.* Springer Science + Business Media LLC, 2000.

[4] Mattias Wahde. *Biologically Inspired Optimization Methods.* Southampton; WIT Press, 2008

[5] Peter W. Christensen, Anders Klarbring. *An Introduction to Structural Optimization.* Springer Science + Business Media B.V., 2009.

A. Appendix

A.1. Full analytical Hessian

This is a continuation of the evaluation of the analytical Hessian. In accordance with section 2.2.2.1, it has been shown that the Hessian can be written as:

$$\mathbf{H}(\mathbf{w}) \quad = \quad 2 \cdot \left(\mathbf{G}(\mathbf{w}) + \mathbf{J}(\mathbf{w})^{\mathsf{T}} \cdot \mathbf{J}(\mathbf{w}) \right) \tag{A.1}$$

The analytical value of the Jacobian is known so what remains is to evaluate the matrix:

$$\mathbf{G}(\mathbf{w}) = \sum_{q=1}^{Q} \left(\nabla \cdot \nabla^{\mathsf{T}} e_q(\mathbf{w}) \right) \cdot e_q(\mathbf{w}) \tag{A.2}$$

The difficult part of this formula is the double derivative. Note that in the following equations, the q-index has been dropped. However the final results need to be applied on all Q errors to find the full Hessian. The best way to proceed is now to apply the chain rule to each double derivative:

$$\begin{aligned}
\frac{\partial^2 e}{\partial w^{\theta}_{\zeta,\eta} \partial w^{k}_{i,j}} &= \frac{\partial}{\partial w^{\theta}_{\zeta,\eta}} \left(\frac{\partial e}{\partial w^{k}_{i,j}} \right) \\
&= \frac{\partial n^{\theta}_{\zeta}}{\partial w^{\theta}_{\zeta,\eta}} \cdot \frac{\partial}{\partial n^{\theta}_{\zeta}} \left(b^{k-1,(0)}_{j} \cdot s^{k}_{i} \cdot b^{K,(1)}_{1} \right) \\
&= b^{\theta-1,(0)}_{\eta} \cdot \Bigg(\tfrac{\partial}{\partial n^{\theta}_{\zeta}} \left(n^{k-1}_{j} \right) \cdot b^{k-1,(1)}_{j} \cdot s^{k}_{i} \cdot b^{K,(1)}_{1} + b^{k-1,(0)}_{j} \cdot \tfrac{\partial}{\partial n^{\theta}_{\zeta}} \left(s^{k}_{i} \right) \cdot b^{K,(1)}_{1} \\
&\quad + b^{k-1,(0)}_{j} \cdot s^{k}_{i} \cdot s^{\theta}_{\zeta} \cdot b^{K,(2)}_{1} \Bigg) \\
&= b^{\theta-1,(0)}_{\eta} \cdot \Big(\sigma^{\theta,k-1}_{\zeta,j} \cdot b^{k-1,(1)}_{j} \cdot \sigma^{k,K}_{i,1} \cdot b^{K,(1)}_{1} + b^{k-1,(0)}_{j} \cdot \tau^{\theta,k,K}_{\zeta,i,1} \cdot b^{K,(1)}_{1} \\
&\quad + b^{k-1,(0)}_{j} \cdot \sigma^{\theta,K}_{\zeta,1} \cdot \sigma^{k,K}_{i,1} \cdot b^{K,(2)}_{1} \Big)
\end{aligned} \tag{A.3}$$

where $\sigma^{k,K}_{i,I} \equiv \frac{\partial n^{K}_{I}}{\partial n^{k}_{i}}$ and $\tau^{\theta,k,K}_{\zeta,i,I} \equiv \frac{\partial^2 n^{K}_{I}}{\partial n^{\theta}_{\zeta} \partial n^{k}_{i}}$.

Note that the second line of equation (A.3) assumes that $\theta \leq k$. However, the symmetry of the double derivative allows the results to be extended to fill the entire Hessian matrix. The exact values for the quantities $\sigma_{i,I}^{k,K}$ and $\tau_{\zeta,i,I}^{\theta,k,K}$ can be be evaluated using recursive relations similar to those described for the sensitivity.

$$\begin{aligned} & \text{for} \quad K < k \\ \sigma_{i,I}^{k,K} &= 0 \end{aligned} \tag{A.4}$$

$$\begin{aligned} & \text{for} \quad K = k \\ \sigma_{i,I}^{k,K} &= \delta_{i,I} \end{aligned} \tag{A.5}$$

$$\begin{aligned} & \text{for} \quad K = k+1 \\ \sigma_{i,I}^{k,K} &= b_i^{k,(1)} \cdot w_{I,i}^{K} \end{aligned} \tag{A.6}$$

$$\begin{aligned} & \text{for} \quad K > k+1 \\ \sigma_{i,I}^{k,K} &= \sum_{l=1}^{m_{k+1}} \left(\sigma_{i,l}^{k,k+1} \cdot \sigma_{l,I}^{k+1,K} \right) \end{aligned} \tag{A.7}$$

$\tau_{\zeta,i,I}^{\theta,k,K} = \tau_{i,\zeta,I}^{k,\theta,K}$ i.e. this quantity is inherently symmetric. Therefore all the following assumes $\theta \leq k$.

$$
\begin{aligned}
&\text{for} \quad K < k \\
\tau_{\zeta,i,I}^{\theta,k,K} &= 0 \qquad \text{(A.8)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{for} \quad K = k \\
\tau_{\zeta,i,I}^{\theta,k,K} &= \frac{\partial}{\partial n_{\zeta}^{\theta}} \left(\delta_{i,I}\right) \\
&= 0 \qquad \text{(A.9)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{for} \quad K = k+1 \\
\tau_{\zeta,i,I}^{\theta,k,K} &= \frac{\partial}{\partial n_{\zeta}^{\theta}} \left(b_i^{k,(1)} \cdot w_{I,i}^{K}\right) \\
&= \sigma_{\zeta,i}^{\theta,k} \cdot b_i^{k,(2)} \cdot w_{I,i}^{K} \qquad \text{(A.10)}
\end{aligned}
$$

$$
\begin{aligned}
&\text{for} \quad K > k+1 \\
\tau_{\zeta,i,I}^{\theta,k,K} &= \frac{\partial}{\partial n_{\zeta}^{\theta}} \left(\sum_{l=1}^{m_{k+1}} \left(\sigma_{i,l}^{k,k+1} \cdot \sigma_{l,I}^{k+1,K}\right)\right) \\
&= \sum_{l=1}^{m_{k+1}} \left(\tau_{\zeta,i,l}^{\theta,k,k+1} \cdot \sigma_{l,I}^{k+1,K} + \sigma_{i,l}^{k,k+1} \cdot \tau_{\zeta,l,I}^{\theta,k+1,K}\right) \qquad \text{(A.11)}
\end{aligned}
$$