Linköping University | Department of Computer Science
Master thesis, 30 ECTS | Datateknik
2016 | LIU-IDA/LITH-EX-A--16/036--SE

Automatic fine tuning of cavity filters

Automatisk finjustering av kavitetsfilter

Anna Boyer de la Giroday

Supervisor : Cyrille Berger
Examiner : Ola Leifler

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Cavity filters are a necessary component in base stations used for telecommunication. Without these filters it would not be possible for base stations to send and receive signals at the same time. Today these cavity filters require fine tuning by humans before they can be deployed. This thesis have designed and implemented a neural network that can tune cavity filters. Different types of design parameters have been evaluated, such as neural network architecture, data presentation and data preprocessing. While the results was not comparable to human fine tuning, it was shown that there was a relationship between error and number of weights in the neural network. The thesis also presents some rules of thumb for future designs of neural network used for filter tuning.

Contents

1 Introduction

Base stations are used for mobile phone communication. The base stations receive signals from mobile phones and retransmit received signals to the intended recipient or to a base station closer to the recipient. This wireless communication could be phone conversations, text messages or internet traffic. This communication is increasing, and also becoming a more and more central part of our society. Therefore it is important that the systems that support this communication are as cheap as possible and can be deployed quickly, when the existing infrastructure fails. This thesis will look into automating the fine tuning of filters used in base stations.

1.1 Motivation

A base station needs to be able to send and receive signals at the same time. Different signals are sent at the same time, but at different frequencies. By using a filter that removes some frequencies and lets other frequencies through, the different signals can be separated and handled individually.

As an illustration of this, consider a person trying to communicate in Morse code by playing a C on a piano. At the same time, another person wants to communicate in Morse by playing an A on the same piano. The difference between a C and an A is that they have different frequencies. So in order to understand any of these messages a listener could use a filter that removes all notes except C or A, depending on which message they want to listen to.

A filter in a base station would work in much the same way, removing all frequencies except the frequency of the signal or signals it is responsible for. It is important that the filters used in base stations are capable at separating signals that are close in the frequency band as this increases the amount of data the base station can handle simultaneously.

Today these filters need to be fine tuned after they have been produced. This is because the production method is not exact enough. Instead the filters are designed in a way that makes it possible to adjust which frequencies will be let through. Every year several thousands of these filters are produced and each of them has to be manually fine tuned by a human expert, which can take more than 40 minutes. This makes the fine tuning process a bottleneck during production.

A RBS 2206 [26] which is a radio base station developed by Ericsson, can use up to three filters that needs this type of fine tuning. Each of theses filter is in turn able to handle about 24 simultaneous users.

1.2 Aim

The aim of this thesis project is to train a neural network to fine tune a 5-pole cavity filter. Michalski [14] have also trained neural networks to fine tune cavity filters, and many of their findings will be used as guidance in this thesis. The thesis will also look into what can be done to improve the results, by comparing different back-propagation algorithms, neural network architectures and how the input data can be transformed before it is used to train the neural network. Finally a few rules of thumb, for how to design a neural network to tune a filter will be presented.

1.3 Research questions

The following questions will be used to guide the work in this thesis. The questions will be answered in section 5.1.

- **Can a neural network fine tune a 5-pole cavity filter to meet its specification?**
  A filters specification is usually based on how well it removes unwanted frequencies, and also how much of the desired frequencies' energy is preserved after it has passed through the filter. A low energy, in terms of sound, would mean quiet. The frequencies that the filter should not remove should keep as much of its energy as possible. Similarly unwanted frequencies should lose as much as possible of their energy.

  A specification usually explains how much of the unwanted frequencies' energy have to be removed, as well as how much of the wanted frequencies' energy can be allowed to be lost.

  Michalski [14] trained a neural network to fine tune a 6-pole and a 11-pole cavity filter. Later Michalski [15] managed to improve the method enough for the filters to not need further fine tuning. Is it possible to use these findings to train a neural networks to fine tune a 5-pole cavity filter?

- **How does the architecture of the neural network affect performance?**
  Will the back-propagation algorithm used by Michalski [14] be the best choice for a 5-pole cavity filter as well, or will a different back-propagation algorithm work better? What effect does the number of hidden nodes and the transfer function have on the performance?

- **How can the data be presented to the neural network to improve performance?**
  Often the data used to train neural networks is preprocessed in some ways to improve performance. Can performance be improved by transforming the input data to the neural network in a different way than Michalski [14] did?

- **Can the number of examples needed to reach a certain error be estimated?** Michalski [14] presented a function for estimating the number of examples needed in order to keep the same error, when the output space changed. As will we shown later, this function does not work well on smaller filters. Can a better estimation be made?

1.4 Delimitations

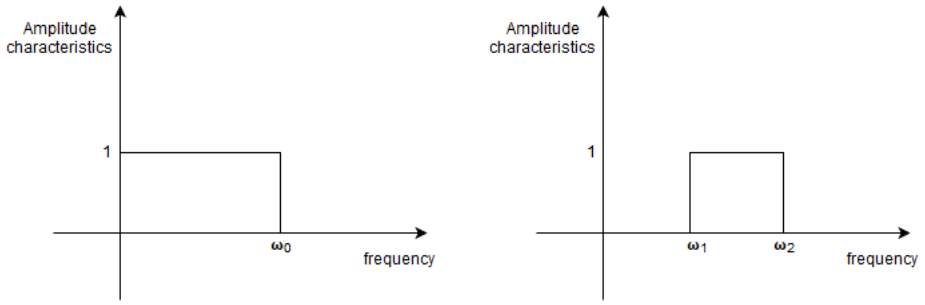
Under normal circumstances a filter may have more than 8 poles, and several cross couplings. However, in this thesis a solution for a 5-pole band-pass filter without cross couplings will

be developed. Normally cavity filters are connected in pairs. In this thesis a single filter that is connected to another tuned filter will be used. The neural network will be developed with the neural network tool in Matlab, and the neural network will be designed using built in functions only, which puts some limitations on what can and can not be done.

2 Theory

This chapter will start with a short description of how a cavity filter works in section 2.1. Section 2.2 contains a description of the manual techniques available for fine tuning these cavity filters. In section 2.3 automation techniques others have tried are presented. Section 2.4 has an overview of relevant machine learning techniques. Lastly, in section 2.5, a more in depth description of techniques that are used in this thesis will be described.

2.1 Microwave cavity filter



(a) An ideal low pass filter lets low frequencies through but not high.

(b) An ideal band-pass filter blocks frequencies that are too low or too high.

Figure 2.1: Amplitude characteristics of a low pass and band-pass filter.

A common way of designing band-pass filters is by starting with a low pass filter (figure 2.1a) and then transforming it to a band-pass filter (figure 2.1b). This can be done by changing every inductor to an inductor and capacitor connected in series, and each capacitor into a capacitor and inductor connected in parallel (a *LC-circuit*). The LC-circuit is also called *resonator* [19].

If an inductor is connected to a charged capacitor as in figure 2.2, the capacitor will discharge. This will cause a current through the inductor until the capacitor has been discharged. This current will create a magnetic field in the inductor. As the capacitor discharges, the cur-

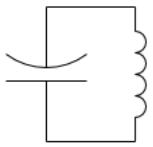


Figure 2.2: Example of a simple LC-circuit.

rent will become weaker and weaker. When the current decreases, the magnetic field will to. According to Lenz's law [11], the change in the magnetic field will create an induced current that opposes this change. That is the induced current will move in the same direction as the current from the capacitor in order to prevent the magnetic field from decreasing. This extra current will charge the capacitor with a voltage opposite what it had before. This will result in an oscillation that is mathematically equivalent to that of a weight on a spring [11].

If the band-pass filter has a bandwidth that is less than 10% of the centre frequency, the LC-circuit can not be realized. Instead a different technique is used where a number of LC-circuits are *coupled*. This way the magnetic field of each resonator is allowed to affect the neighbouring resonators magnetic field. In cavity filters this coupling is usually done with an inductor [19]. An example of a filter circuit can be seen in figure 2.3.

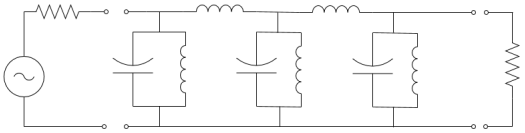


Figure 2.3: Example of a circuit equivalent to a cavity filter.

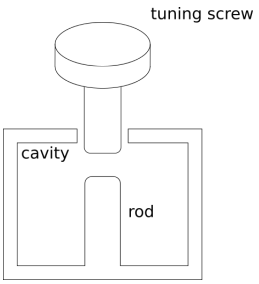


Figure 2.4: When the distance between the screw and the rod changes, the capacitance changes as well.

A cavity filter consists of one or several cavities connected together. In each cavity a cylindrical rod is placed. The cavity and rod acts as a LC-circuit [24]. The cavity and rod is also called a *pole*. A magnetic field will surround each rod and affect neighbouring rods in other cavities.

A cavity or pole is tuned by changing the length of the rod, which changes the inductance, or by changing the distance between the rod and the walls or lid of the cavity which changes the capacitance [24]. When the capacitance or inductance is changed, the *resonance frequency* also changes. A resonance frequency [11] in a LC-circuit is the frequency of the oscillating current in the circuit where the current reaches its maximum value. For a different frequency the maximum value of the current is not as large as the currents maximum value for the resonance frequency. By changing the resonance frequency, a different frequency of the input signal will be let through the LC-circuit with least loss of energy.

One method for tuning the filter is to have a screw placed over the rod. When the distance between the screw and the rod changes, so does the capacitance [24]. Figure 2.4 illustrates this situation.

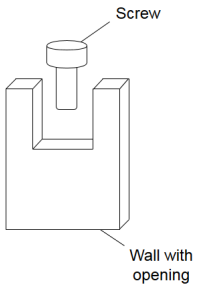


Figure 2.5: The screw in the wall opening acts as an inductor.

Between some poles or cavities is a wall with an opening (figure 2.5). The wall will prevent the magnetic field of the poles from passing through the wall. If a wall does not have an opening, the magnetic fields of two poles on either side of this wall will not interfere. The bigger the opening in a wall is the more the poles will be *coupled*, that is, the more the poles magnetic fields affect each other. In some designs a screw is inserted in the opening [24]. This screw will act as an inductor (just like the cavities). The longer the screw the higher the coupling is. This screw can also be turned to change the coupling between resonators. In practice the position of these coupling screws have often been set beforehand so that a tuner should not have to change them. The coupling screws only need to be tuned if something is wrong with the filter, for example if the mould has been used so many times that its size have changed.

A high coupling between the poles will make the electromagnetic signal pass through the poles without losing a lot of energy. The energy that has been lost is called *insertion loss*. A high coupling will also lead to a filter that is hard to fine tune. if the coupling between the poles is low, the filter will be easier to tune, but the insertion loss will be higher.

Vector Network Analyser

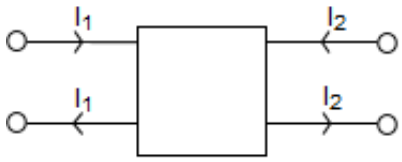


Figure 2.6: Example of two-port electrical network.

An electric network can be classified based on how many connections it has. These connections often come in pairs, where each pair shares current and voltage (see figure 2.6).

A cavity filter is a type of two-port electric network. One of the ports in the cavity filter is used as the input port and the other port is used as the output port. When the filter is being used, the signal that should be filtered will be sent into the filter via the input port, and the resulting filtered signal can be read from the output port.

Each pair, or port, has its own incident and reflection variable (a_1 and b_1 for port 1, and a_2 and b_2 for port 2) as depicted in figure 2.7.

They are related in the following way:

$$\begin{cases} b_1 = S_{11}a_1 + S_{12}a_2 \\ b_2 = S_{21}a_1 + S_{22}a_2 \end{cases} \tag{2.1}$$

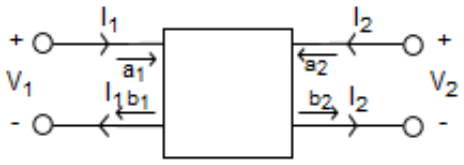


Figure 2.7: Two-port electrical network with incident and reflection variables.

S_{11}, S_{12}, S_{21} and S_{22} are called *scattering parameters* [23].

These scattering parameters can be measured by a *Vector Network Analyser* or VNA. The scattering parameters or S-parameters take on different values for different frequencies and can therefore be seen as functions of frequency. Figure 2.8a and 2.8b shows a vector network analyser that displays the S-parameters S_{11} (blue) and S_{21} (yellow). S_{11} and S_{21} are complex, but usually viewed in a logarithmic format as in figure 2.8a and 2.8b. The complex values are converted to dB with the following formula [9]:

$$magnitude(S) = 20 * Log_{10}(\sqrt{Re(S)^2 + Im(S)^2}) \text{ dB} \tag{2.2}$$

2.8a shows an example of a well tuned 5-pole filter and figure 2.8b shows a frequency response from a slightly detuned 5-pole filter. The difference between the tuned and detuned filter is that there is an extra "lump" on S_{21} for the detuned filter, and the width of the flat region of S_{21} is shorter compared to the tuned filter. In a tuned filter S_{11} should consist of a number of arches with about the same height, in the frequency region where S_{21} is flat, as is the case in the tuned filter in figure 2.8a.

S_{11} measures how much of the sent signal gets reflected back to the input port. S_{21} measures the insertion loss, that is, how much of the sent signal's energy have been lost during its way through the filter. More specifically S_{21} measures how much of the sent sent signal reach the output port. A filter specification is usually based on S_{11} and S_{21}. For example a requirement could be that S_{11} is at least 17dB lower than what was sent in, in the pass band region. This would mean that very little of the sent signal is reflected back to the input port. A requirement on the S_{21}-signal could be that the output signal can not be more than 1.2dB lower than what was sent in, in the pass band region. This would mean that most of the sent signal reach the output.

The S-parameter that will be looked at in this thesis is S_{11}. S_{11} is also called the input reflection coefficient [23]. The reason only S_{11} will be used in this thesis is because Michalski [14] only used S_{11} and the advice that can be taken from their results is based on using S_{11} only.

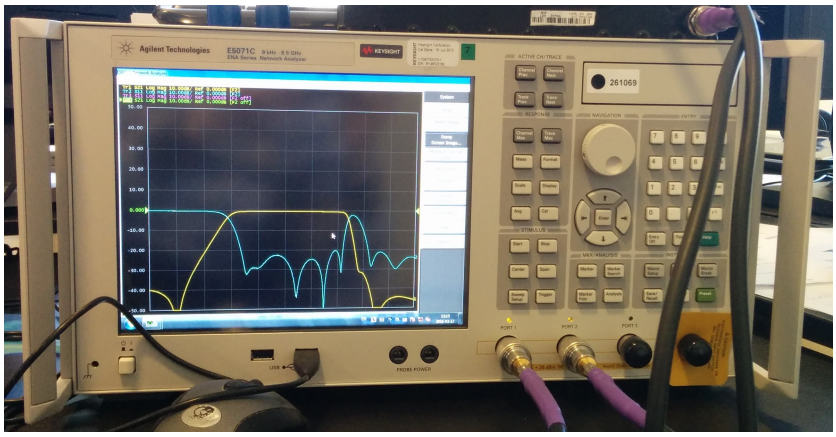
predictions based on what has happend earlierrevious

2.2 Manual techniques for fine tuning cavity filters

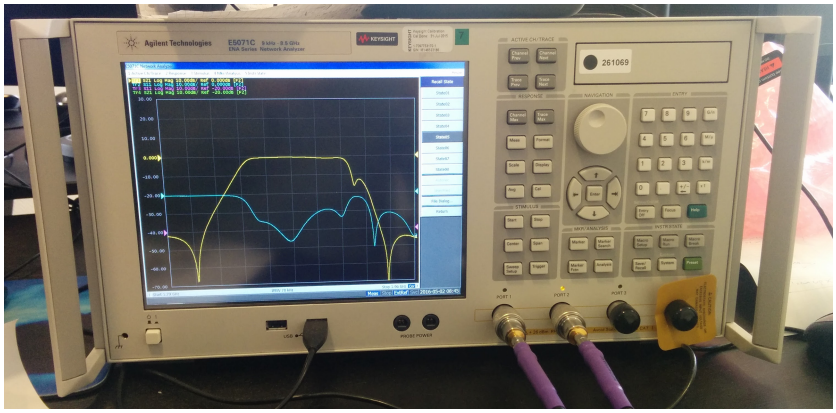
Lindner and Biebl [13] describe a process of fine tuning a coupled cavity filter. They developed the method by looking at how experts fine tune these filters. This process can be divided into the following seven steps:

Step 0: Before starting the tuner should pre tune the filter so that all *reflection poles* are recognizable as *dips* on the S_{11} plot. This is a requirement to use the described method. A reflection pole is a place in the S_{11} frequency response where very little of the energy at that frequency gets reflected back. These reflection poles are recognised by the γ-shaped dips they cause in the S_{11} frequency response. In figure 2.8a a number of theses dips can bee seen.

Step 1: in the VNA, S_{11} should be displayed in linear format as the logarithmic format will give a misleading view on the sensitivity of the tuning elements. S_{21} should still be displayed in logarithmic format.



(a) A VNA showing S_{11} (blue) and S_{21} (yellow) on a well tuned filter.



(b) A VNA showing S_{11} (blue) and S_{21} (yellow) on a detuned filter.

Figure 2.8: Two examples of frequency response from a filter.

Step 2: The goal of this step is to have all dips in S_{11} as low and equal as possible. To achieve this make sure to only use one half of the tuning elements and the associated couplings.

Step 3: Make all the curves in S_{11} symmetric around the centre frequency of the band-pass region. Make sure to only tune the resonators in a symmetric fashion. That is, tune each symmetric pair of resonators the same amount and in the same direction. Do not tune the couplings.

Step 4: Tune the filter to have the desired centre frequency by adjusting all resonators the same amount. Sometimes step 2 needs to be repeated because the couplings are frequency sensitive.

Step 5: Make the filter have the desired bandwidth. This is done by tuning the couplings. Start by tuning them the same amount, and by tuning the input and output coupling symmetrically. Do not tune the cross couplings

Step 6: Change the height of the ripple (that is, the arches between the dips in S_{11}) with the couplings so that they are even. If the input and output couplings are tuned, retune the adjacent resonators. At this step the filter should have the correct return loss, ripple bandwidth and centre frequency. As an example, look at the arches in the S_{11} frequency response in figure 2.8a. In the band-pass region the arches are fairly even.

Step 7: Tune the cross couplings so that the transmission zeros are at the right frequencies. The transmission zeros can be seen as the two dips in the S_{21} frequency response in figure 2.8a on either side of the band-pass region. At these two frequencies no energy is transmitted. This

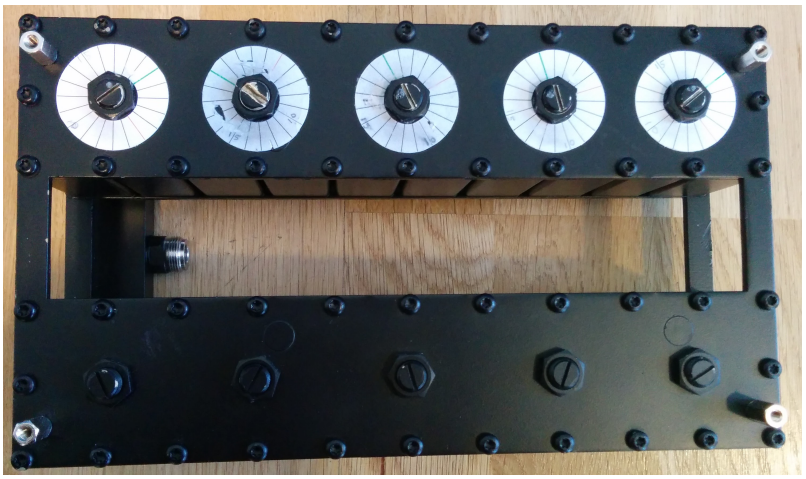


Figure 2.9: An example of two 5-pole filters. The two filter have a shared antenna on the left, but different band-pass regions. The screw are used to tune the filter and the screw nuts will lock the screws once the filter is tuned.

step will slightly detune the adjacent resonators, and these may therefore need to be retuned. To do this, repeat step 2-7 until the filter fulfils its specification.

The process could be used on filters with one or several cross couplings and with a different number of poles. The process itself is simple to use and does not require a lot of experience.

Note [19] shows how the time domain can be used when fine tuning a filter. When looking at the time domain response a correctly tuned filter will have one dip in the time domain response per resonator. By tuning the filter from the outside in, i.e. starting by tuning the outermost resonators and then moving inwards, one can see when each resonator is correctly tuned by making sure the corresponding dip is as low as possible.

2.3 Automated techniques

With fuzzy logic [25] a variable with continuous values can be divided into overlapping regions or sets. Each value can belong to several sets to different degrees.

For example, if one were to classify days as either "sunny" or "cloudy" a day that was partially cloudy would belong to both sets. If it were fairly cloudy it would belong to the cloudy set more than it belonged to the sunny set. The sum of how much a value belongs to all sets should be one. For example a partially cloudy day could belong to the the sunny set with the value 0.2 and to the cloudy set with the value 0.8. In other words that day would be 20% sunny and 80% cloudy

A function is used to determine how much a particular value belongs to a fuzzy set. For example this function could be triangular with the top at the centre of the region. Figure 2.10 shows some example triangular functions for four fuzzy sets. X is the variable that has been divided into fuzzy sets and Y is a value indicating how much a value belongs to a particular set.

If both input and output variables are divided into fuzzy sets, one can create a mapping from input sets to output sets. Once this have been done, output values for a given set of input values can be calculated by taking into account how much the input values belongs to different fuzzy sets. This way the output from the algorithm can take any value in the output space, rather than just a few discrete values.

Miraftab and Mansour [17] used a *Fuzzy Logic Controller* (FLC) as a replacement for a human tuner. This controller was made up by several small *Fuzzy Logic Systems* (FLS).

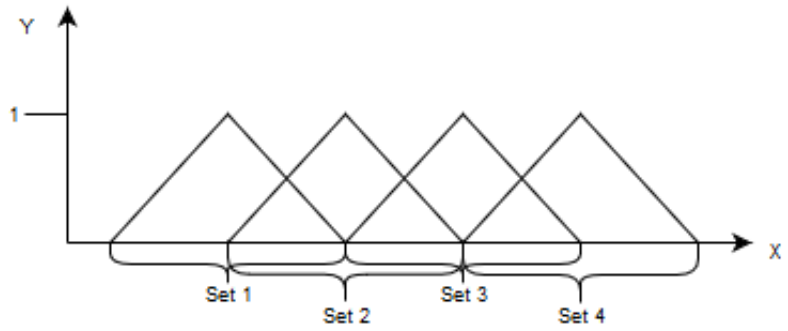


Figure 2.10: Triangular functions for four fuzzy sets

When creating a FLS a human expert is asked to fine tune a filter. The S_{11} frequency response of the filter and its comparison to the desired S_{11} frequency response is the input for the FLS. The human expert will then create an output in the form of a change made to a particular parameter. The human output to a specific input is recorded and saved as a input/output pair in a FLS. A number of FLS are created this way by giving the human expert a number of different starting scenarios.

When the machine is then used to auto tune a new filter it will treat each FLS as a fuzzy set. It will selects the fuzzy set that the input scenario is most similar to. This set corresponds to a FLS that then select an appropriate tuning instruction given the input. Applying this instruction will generate a new scenario. This is then repeated until the filter meets its specification or a number of iterations have been made without reaching the filters specification. In the second case the current scenario is given to a human expert that will tune the filter manually. This tuning will then be turned into a new FLS.

Michalski [14] describes a method where a neural network is used to fine tune a cavity filter. Neural networks will be described in more detail in section 2.5. When training the network a measurement from a VNA (Vector Network Analyser) was used together with a vector indicating how much each screw had been modified from their correct position. As output the network would return how much each screw should be turned. The tests of the neural network were successful although not good enough for making the filter completely tuned. Only one filter was used, both for gathering example data, and for evaluating the neural network.

Michalski [15] later used the same neural network and trained it on several filters in an attempt to minimize the generalisation error. When more filters were used the neural network managed to tune a new filter so well that it did not need further fine tuning.

Zhou, Duan, and Huang [27] use expert tuners to generate data pairs that consists of a vector of how much each screw have been turned, and the corresponding change in the coupling matrix, which can be generated from the S-parameters. These data points are then used to create a model of the relationship between the coupling matrix and the screw deviation with least square support vector regression (LSSVR). With this model it is possible to estimate the current screw positions given the filter response and also to estimate the position the screws should have to get the desired filter response.

2.4 Machine learning

An *agent* is anything that can perceive its environment, through sensors, and also act in the environment via actuators [21]. An agent can be a robot, perceiving the world through a camera. The actuators could be wheels allowing it to move about. An agent could also be a computer program, that senses its environment through input parameters and act upon

this environment with return values. In this thesis the agent is the program that implements a neural network. The environment is the filter, which the program can sense through the frequency response from the VNA. The program can control the environment by deciding which screws will be turned.

People that work in the field of machine learning, try to make agents learn from experience. The agent is using data to draw conclusions on how the environment works. One reason for creating a learning agent is that it is not necessary to anticipate all possible inputs, which is not always possible to do. Another reason is that it is not always evident how a problem can be solved. With learning an agent can find a solution to a previously unsolved problem.

Three types of learning exists, and will be presented below. Which type is used depends on what kind of feedback the agent have access to [21].

Supervised learning

In supervised learning, data that consists of input/output pairs are used. The goal is to find a function that can map from a given input to a correct output, using the input/output examples during training [21].

An example could be trying to recognize faces in images. The agent would be given an input in the form of an image that may or may not contain a face, and as output whether this image contain a face or not. The agent will then try to find a pattern in the examples, so that it can correctly classify images it has never seen before [2].

Reinforcement learning

In the reinforcement model, an agent is placed in an environment which the agent perceives with *input signals*. The agent can change the state of the environment with actions. In each time step the agent will perform an action and then receive some input that indicates the current state as well as a reward for the state transition. The agent's goal is to maximize its reward over time [8].

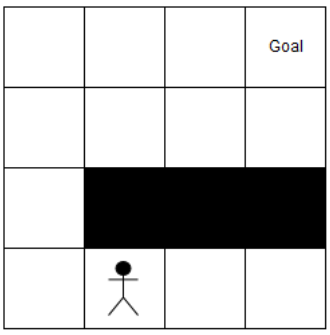


Figure 2.11: The agent should learn to find the goal in the grid map

For example, if an agents goal is to find the goal position on a grid map (see figure 2.11), the states would be the positions in the map. The actions would be the possible moves the agent could make in each state i.e. up, down left and right (note that not all actions will be available in all states) The reward could be the inverse of the distance to the goal position. As the agent tries to move around the map it will learn a *policy* that maps states to actions such that following this policy will maximize the agents reward.

For the reinforcement model to work, the environment does not have to be deterministic. That is, taking the same action in the same state does not have to lead to the same new state or reward. The environment does however need to be stationary. This means that the

probabilities of a state transition or reward to happen given an action have to stay the same [8].

Unsupervised learning

In unsupervised learning the agents task is to find patterns in data. Since there is no "answer" to the data, no feedback is given to the agent. Common problems in this area is clustering data, such as trying to divide the provided data into groups based on how similar they are[21].

2.5 Neural networks

Artificial Neural Networks (ANN) are inspired by biological neural networks. ANNs can find patterns in non-linear data, and is robust against error. They can also be updated if the environment changes. ANNs are among other things, useful for classification problems, clustering, and function approximation [1]. ANNs do not need a model of the environment to work, but they require a large training set.

Normally these neurons are organized into layers where each neuron is connected to all neurons in the succeeding and preceding layer. The input received by a particular neuron is converted to the neuron's output via a *transfer function*. This output will be input to the neurons in the next layer.

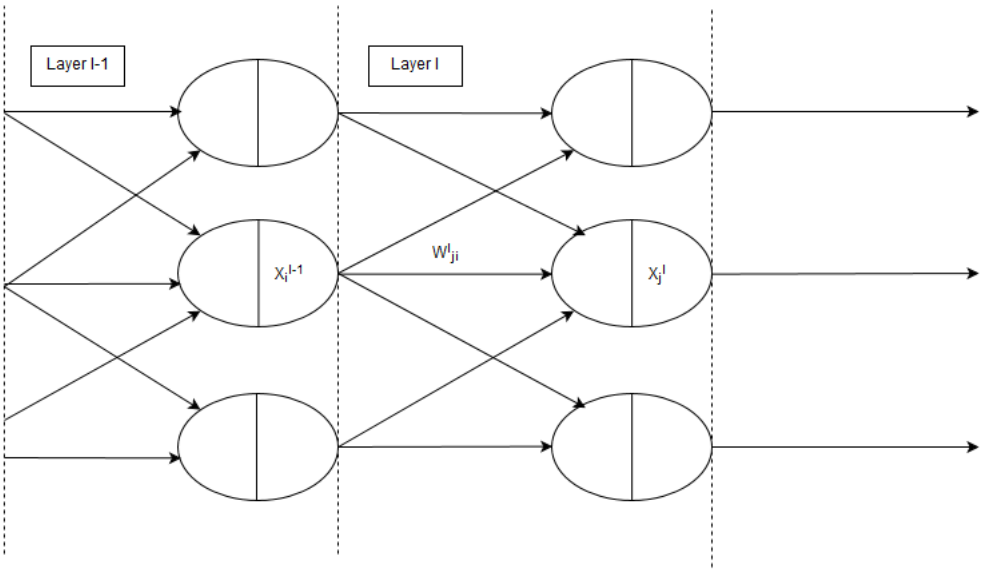


Figure 2.12: example of neural network

Figure 2.12 shows an example of a neural network organized in layers, and those neurons input connections. Each input connection has an associated weight w^l_{ji} where l indicates that the weight is in layer l and j is the neuron it connects to in layer l and i is the neuron in the previous layer (layer $l-1$). The net signal a neuron receives is calculated as:

$$S^l_j = \sum_{i=1}^{N_{l-1}} (w^l_{ji} x^{l-1}_i) \tag{2.3}$$

where N_{l-1} is the number of neurons in layer $l-1$ and x^{l-1}_i is the output from neuron i in layer $l-1$. This net signal S^l_j is then used as input to the neurons transfer function $\sigma(S)$. The transfer function maps S to a real value in a bounded interval. A common choice for transfer

function is a *sigmoid function* [1]. A sigmoid function [4] returns a value from a bounded interval, for all values between $[-\infty, \infty]$, and has a positive derivative at all points. Some example sigmoids can be seen in figure 2.14a and 2.14b.

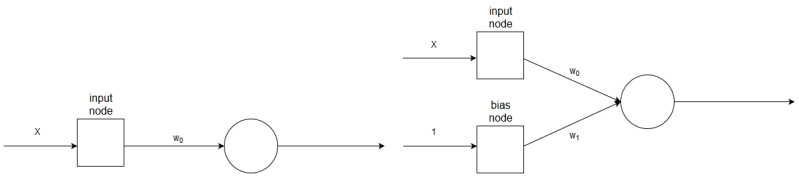
A neural network is trained by incrementally adjusting the weights in the network, until the desired output is received.

ANNs can have different topologies and functions for updating the weights. The architecture and update function used in an ANN will affects its performance in different tasks. A Kohonen network [18] or self organizing map, can be used for analysing high dimensional data. It is a unsupervised learning method, that can be used for pattern recognition, clustering and classification. Adaptive Resonance Theory (ART) [1] networks are also trained by unsupervised learning. When the network is presented with a new pattern it will either match it to a previously stored similar pattern or store it as a new pattern if no stored pattern was similar enough. They can be used for pattern recognition and classification.

Other examples of networks are Hopfield networks [6], counter propagation networks [5] and recurrent networks [1], where some neurons output are sent to itself or to neurons in a preceding layer.

The most common ANN architecture however, is a feed forward network that uses the back-propagation (BP) algorithm. It can among other things be used for data modelling, forecasting and pattern recognition. The back-propagation network consists of several layers [1]: one input layer, one or several *hidden layers* and one output layer. Each neuron in the input layer represents one input variable, and each neuron in the output layer represents one output variable. The number of hidden layers and the number of hidden neurons in those layers may vary. If no hidden layer is used, the ANN can not handle non-linear mapping between input and output values.

Back-propagation networks use supervised learning. The input layer will receive some input that is fed forward to the first hidden layer. Each layer will send their output forward until it reaches the output layer which will produce the networks collective output. An error is calculated based on the difference between the networks output and the desired output. This error is *back propagated* to all layers starting with the output layer and moving backwards. This back-propagation algorithm will be described below.



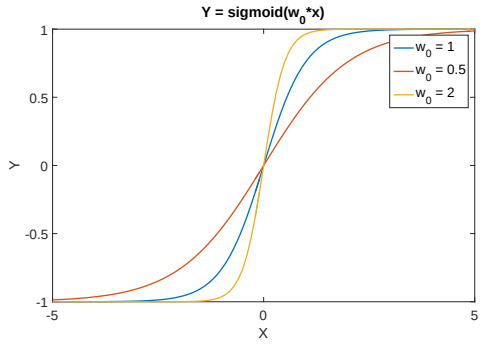
(a) An example network with one input and one output signal.

(b) An example network with one input, one bias and one output signal.

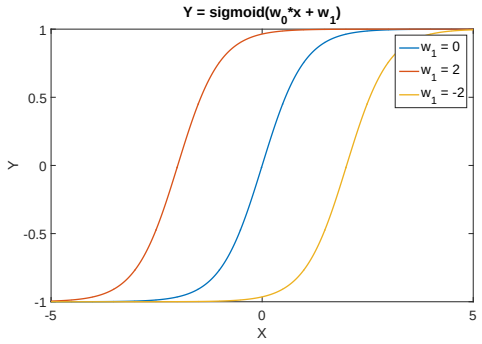
Figure 2.13: Example networks with bias

A common feature in back-propagation networks are *bias*. A bias is a neuron that will always have 1 as its output. The reason for having such neurons is that it makes it possible for a neuron that receives this signal as input, to move their transfer function "sideways". so that it is not centred around 0. As an example consider a network with one input neuron and one output neuron (figure 2.13a). The output of the network is calculated as $sigmoid(w_0 * x)$. When the weight w_0 changes, the "steepness" of the output signal will change as is shown in figure 2.14a. If a bias neuron is added so that the network look like in figure 2.13b, the output will instead be calculated as $sigmoid(w_0 * x + w_1 * 1)$. Figure 2.14b shows how the output will change with different values for w_1.

As figure 2.14b shows, the bias weight allows the output curve to be shifted to the left or right. This can help improve the results on some problems.



(a) A sigmoid function for different values of w_0.



(b) An sigmoid function for different values of w_1.

Learning representation by back-propagation error

A weight w_{ji}^l is updated from its previous state (t-1) with the following equation:

$$w_{ji}^l(t) = w_{ji}^l(t-1) + \Delta w_{ji}^l(t) \tag{2.4}$$

where $\Delta w_{ji}^l(t)$ is the change that will be made to the weight. Using the modified delta rule this change is calculated as:

$$\Delta w_{ji}^l(t) = L_r \frac{\delta E}{\delta w_{ij}^l} + \mu \Delta w_{ji}^l(t-1) \tag{2.5}$$

where L_r is the *learning rate*, $\frac{\delta E}{\delta w_{ij}^l}$ is the derivative of the error with respect to the weight that is being updated, μ is the *momentum coefficient* and $\Delta w_{ji}^l(t-1)$ is the change made to the weight the previous update.

L_r controls the update step size. If it is too large the ANN will oscillate around the solution because the change to the weights is so big that the correction overshoots the goal. If L_r is to small the training will be slow because the improvement after each iteration is small.

The momentum term μ will help direct the search by adding a part of the previous updates magnitude and direction to the current update. if μ is large the ANN is less inclined to getting stuck in a local minima, but the risk of overshooting the solution is increased. if μ is small the training will take more time and the ANN is more likely to end up in a local minima.

Assuming the error function is defined as the sum of squared errors:

$$E = \frac{1}{2} \sum_{k=1}^{N} (x_k - y_k)^2 \tag{2.6}$$

the error derivative can be written as:

$$\frac{\delta E}{\delta w_{ij}^l} = \delta_k^{l+1} * x_i^{l-1} \tag{2.7}$$

For the output layer δ_k^{l+1} will be calculated as [1]:

$$\delta_k^{l+1} = (x_j^l - y_j)\sigma'(S) \tag{2.8}$$

and for the other layers:

$$\delta_k^{l+1} = \sigma'(S) \sum_{k=1}^{N_{l+1}} (\delta_k^{l+1} w_{kj}^{l+1}) \tag{2.9}$$

x_j^l is the output from output neuron j and y_j is the desired output for output neuron j, δ_k^{l+1} is the weight change made to neuron k in the succeeding layer, and $\sigma'(S)$ is the first derivative of the transfer function. In order to calculate δ_j^l, the δ value for all the connection links in the succeeding layer must be calculated. Because of this, the weight change is first calculated for the output layer, then for the last hidden layer and so on, such that the weight change is made backwards, from the output layer to the input layer. This is why it is called *back-propagation*.

Other back-propagation algorithms

The following back-propagation algorithms are described because they will be compared with the method chosen by Michalski [14]. The reason for this is that these back-propagation algorithms have performed well for others with similar problems.

Conjugate gradient [12] is an update method that uses a linear search to decide the step size. A descent direction is picked (for example the gradient direction). Then the minimum value along this line is found, using linear search. From that point a new line search will be done in the conjugate direction. The conjugate direction is a direction such that the gradient direction does not change if one follows it, only the gradients length will change. (see figure 2.14). The reason for choosing the conjugate direction for the next line search is that doing so will not remove the improvement made in the previous update. For quadratic functions, it can be proved that conjugate gradient search will converge within N steps, where N is the number of variables.

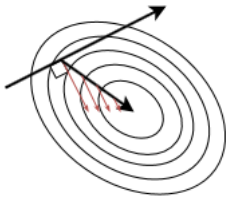


Figure 2.14: The red arrows indicate gradients, and the black arrows are two conjugate directions. As can be seen on the image, the gradient direction does not not change on the conjugate direction of the first search direction.

In **RPROP** [22], unlike in most back-propagation algorithms, the size of the update in a weight is not dependent on the size of the gradient. Instead the step size is determined by the sign of the gradient. If the sign of the gradient has not changed since the previous update, the step size is increased. If the sign has changed, the step size is decreased. Each weight will have its own step size that is updated by the sign of that weights gradient.

Schiffmann, Joost, and Werner [22] have evaluated a number of different back-propagation update algorithms. In their experiments they find that using the back-propagation algorithm with momentum performs very well. The only algorithms that performed better was those that used a local learning rate. That is, each weight in the network had their own learning rate variable. The best performing algorithm that also exists in the Matlab neural network toolbox is RPROP. It was less then 0.5% worse than the best performing algorithm.

Data and Training

Some of the collected data needs to be used for evaluating how well the ANN will work on data that it has not seen before. Therefore the data should be divided into three subsets:

- **Training.**
  The training data is used for updating the weights in the network.

- **Test.**
  The test set is used to check the networks response to new data and is not used to update the weights. The goal is to find parameter values that leads to the smallest error in the test set.

- **Validation.**
  The validation set is used to further confirm the networks accuracy. Unlike the test set, the validation set is not used for optimizing the network, but rather to see how well the network performs on data it has neither seen nor been optimized for.

It is important to make sure that the data in all three subsets covers the entire problem domain, and that the three datasets do not contain the same examples.

When presenting examples to the ANN there exists two methods that could be used in combination or alone: *example-by-example training* (EET) or *batch training* (BT).

In EET the weights are updated after each example. Every example will be presented over and over, until the error is low enough or after a certain amount of iterations. Then the second example will be presented the same way.

If BT is used all training examples will be presented one after another before the weights are updated. The error will be calculated as the average over all examples.

EET is less likely to get stuck in a local minima compared to BT, but a bad first example may lead the search in a bad direction. BT's advantage is that it will have a more representative measurement of the necessary weight change and a better estimate of the error gradient [1].

Schiffmann, Joost, and Werner [22] found that all variants of EET outperform batch training on their classifying problem. Using EET will usually yield results faster[12], especially on large data sets. Because larger data sets often are redundant EET can achieve the same results as batch without having to do as many calculations.

3 Method

In this chapter the methodology used for gathering results will be described. Section 3.1 will summarize the method used by Michalski [14], as this method will largely be used in this thesis as well. Section 3.2 will go through some environmental factors that will affect decisions and results. Section 3.3 will describe how the neural network was designed in this thesis as well as take up differences between this work and the work by Michalski [14]. Lastly section 3.5 will explain how the data used for training was gathered.

3.1 The method used by Michalski

The automation methods described in section 2.3 all require data to be gathered from the filter to use as learning examples. The method described by Michalski [14] is however the only one that does not require expert tuners to gather the data. The method also treats the filter as a "black box" the developer does not need to understand. For this thesis experts tuners were not available for data gathering, and the time limit did not allow for acquiring a thorough understanding of the filter. Therefore the method used by Michalski [14] should be a good choice.

Michalski [14] used a neural network to create a mapping from a filter's S_{11} frequency response to how much each screw deviates from its tuned position. This is done by collecting data pairs where the input example is a filter characteristic $S_{11}(x)$ and the output example is how much each screw deviates from its tuned position. Given some filter characteristic, the neural network should output how much each screw position need to change for the filter to be tuned. This is similar to what an expert tuner does when they use the filter characteristics to determine which screw to turn and how much.

The output space has one dimension for each screw. The origin of this space is defined as the tuned filter. The maximum deviation is defined to be $\pm K$ for each screw, which makes the length of each dimension $2K$.

Michalski [14] set the maximum screw deviation to $\pm 360^\circ$, and each screw adjustment is made in steps of $\pm 18^\circ$. The unit $[u]$ is defined as *number of screw adjustments*. For example, a screw deviation of 36° can be written as a deviation of $2u$.

The value of K can be calculated as $\frac{360}{18} = 20u$.

The procedure used for gathering examples for the neural network was as follows:

1. Start with a correctly tuned filter.
2. Read the tuned filters screw positions. This will be used to calculate the difference in screw positions when the filter has been detuned.
3. Randomly detune the filter with the following formula:

$$W(j) = W_0(j) + \delta W(j) \tag{3.1}$$

   where

$$\delta W(j) = RND[2 * K] - K \tag{3.2}$$

   K is as explained earlier the maximum screw deviation as measured in values of u. W(j) is the position of screw j, and $RND[X]$ a function that returns a random integer value between 0 and 1.
4. Read the corresponding S_{11}-parameter of the now detuned filter.
5. Store the S_{11}-parameter as an input example and the screw adjustment as an output example. The output examples need to be normalized.
6. Repeat from point 3 until enough learning samples have been gathered.

The same procedure will be used to gather data in this thesis. The only exceptions is that Michalski [14] gathered data with a robot and that the input examples were not normalized. In this thesis the data is gathered by hand, and the input examples are normalized. The effect manual data gathering have on performance will be discussed in chapter 5.

The neural network that was trained by Michalski [14] was a feed forward network with three layers. There were 512 input neurons, and 50 hidden neurons. Two filters of different sizes were used, so the number of output neurons was 6 or 11 depending on which filter was being used.

A feed forward neural network will be used in this thesis as well, but the number of input nodes and hidden nodes to use will be determined experimentally.

Michalski et al. [16] calculated that the number of input points necessary for the neural network to learn is:

$$L = 2(N + 1) \tag{3.3}$$

where N is the number of tunable elements. The input values are complex and therefore two neurons need to be used per input value.

These results will be taken into account when deciding which values for number of input nodes will be selected.

Michalski [15] trained a neural network on 5 filters. After this the neural network was able to properly tune a filter it had not seen previously. In this thesis only one filter will be used due to time and resource constraints. The effects of this will be discussed in chapter 5

3.2 Environment

As was explained in section 2.1 There are 4 scattering parameters that can be measured by the VNA. Each parameter varies with the frequency and is complex. The VNA can sample up to 1601 points for each scattering parameter at a selected frequency range. The filter used in this thesis is supposed to only let through signals with a frequency between 1.850 GHz and 1.910 GHz.

Michalski [14] used 256 sample points (and consequently 512 input neurons), but later Michalski et al. [16] presents a formula that shows that as few as 12 sample points should be enough to learn a mapping from frequency response to screw deviations on a 5-pole filter.

Since the VNA will sample 1601 points the number of data points needs to be reduced before they are presented to the neural network. Section 3.5 will explain how this was done in more detail.

When tuning filters, experts usually chose to view a wider frequency range than the filter is supposed to work on. This is because it helps detect poles whose resonance frequencies are outside the intended range and because the band-pass region may become shifted during the fine tuning process. As was explained in section 2.1, a poles resonance frequency is the frequency which the pole will let through best. These resonance frequencies should all be inside the band pass region. In general, experienced tuners view a smaller frequency range than novice tuners. If the range used is too small, it is hard to adjust large errors, since the poles that are badly placed may be outside the viewed range. If the range is to big, small changes can not be detected because the sampling is not dense enough.

The range used as input for the neural network was 1.79 GHz to 1.96 GHz. Normally human tuners select a range that suits them. In general experienced tuners selects a smaller range than novice tuners. Since Michalski [14] did not mention what frequency range was used in their project, the range used in this thesis was selected in accordance to advice from a person with experience in tuning filters.

The requirements on a filter are usually placed on S_{11} and S_{21}. Therefore these are the parameters looked at by experts. When the filter is designed, it is made sure that the filters frequency response far outside the band-pass region is acceptable when the filter is tuned. Therefore there is no need for a tuner to look far outside the band-pass region.

S_{11} and S_{21} have complex values, but they are usually viewed in a logarithmic scale. The complex values of S_{11} and S_{21} are converted to a logarithmic scale measured in dB with the following function [9]:

$$magnitude(S) = 20 * Log_{10}(\sqrt{Re(S)^2 + Im(S)^2}) \text{ dB} \tag{2.2}$$

Michalski [14] only used the S_{11}-parameter and did not convert the measurements to logarithmic scale. I will try both using S_{11} as Michalski [14] did and also to use S_{11} in logarithmic format as expert tuners do today.

When gathering data the screw deviation was always a multiple of 18°. The neural networks output is continuous, so it should still be able to handle input where the screw deviation is not in multiples of 18°. However it is possible that some pattern are lost because of how the data was sampled. For example, The neural network should have a hard time with deviations much smaller than 18° since it has not seen any such examples. In practice an expert tuner should need to be able to make smaller adjustments than 18°.

The step size in screw deviation has been chosen to be the same as for Michalski [14], as that value was found to work well for them. An issue with smaller step sizes is that the measurement error would be more significant with smaller step sizes. Using the same step size as Michalski [14] will also help make the results more comparable.

3.3 Set-up

$$E = \sum_{l=1}^{L} \sum_{n=1}^{N} \frac{|o_{ln} - a_{ln}|}{LN} \tag{3.4}$$

Since the different update algorithms have their own parameters that need to be selected, this section is organized into three subsections. Section 3.3 will go through the parameter choices and design decisions that are common to all three update algorithms. Section 3.3-3.3 will explain the parameter choices that are specific for the different update algorithms as well as motivate why these particular update algorithms were selected.

Shared parameter choices and design decisions

- **Transfer function $\sigma(S)$:**
  For transfer function both a logarithmic sigmoid and a tanh sigmoid function will be used. A logarithmic sigmoid maps input values to values between 0 and 1, and a tanh sigmoid maps input to values between -1 and 1. Michalski [14] do not mention which transfer function was used, but the sigmoid function is the most common choice [1], therefore a sigmoid function will be used. The normalization made by Michalski [14] fits a logarithmic sigmoid, but LeCun et al. [12] suggest that a sigmoid centred around 0 is a better choice so that will be tried as well.

- **Number of hidden neurons:**
  Michalski [14] states that the number of hidden neuron used was 50, but also that the number of neurons should be chosen experimentally. Therefore between 20 and 80 hidden neurons will be tested. Presumably, there should not be a large difference in what number of hidden neurons suits the filter used in this thesis and the filters used by Michalski [14].

- **The ratio between the training, test and validation subset:**
  70% of the gathered data will be used for training, and 15% for the test and validation set respectively. The suggestions found by Basheer and Hajmeer [1] is that 20 to 35 percent of the data should be used for testing and evaluation. The values picked in this thesis is within this range and the default values used by Matlab's neural network toolbox.

- **Number of data points gathered:**
  Michalski [14] found that more than 500 data points did not improve the results significantly for them. Since the manual data gathering in this thesis should mean that the examples are not as good as those gathered by Michalski [14], a few more data points (700 to be exact) were gathered for this problem. The data was copied and noise was added to the copies in order to increase the number of data points. Adding noise also tends to increase the neural networks robustness for measurement errors. In the end, 2800 data points were available for training and evaluation.

- **Batch training or EET:**
  Michalski [14], does not mention whether batch training or EET was used. Schiffmann, Joost, and Werner [22] and LeCun et al. [12] found that update algorithms trained with EET performed better than all update algorithms trained with batch training.

  The conjugate gradient update algorithm only works with batch training. To make the update algorithms results more comparable, and also to save computing time most of the tests where done using batch training. The iteration of examples for EET training had to be implemented manually, making training with EET much slower than batch training where the entire training procedure already existed in the toolbox.

- **Error function:**
  From the description by Michalski [14] it is clear that they used mean absolute error, as their error function. This will be used in this thesis as well, as it will simplify comparisons.

- **Bias:**
  Michalski [14] did not mention having any bias, but bias will still be used in this thesis for the hidden and output neurons. This bias makes it possible to shift the sigmoid function, which allows the neurons to learn more functions.

- **Number of input variables:**
  The number of input variables can be changed by changing how many data points from the S-parameters are used. As was mentioned in section 3.1, Michalski et al. [16] found that the number of input points necessary for the neural network to learn is

$$L = 2(N + 1) \tag{3.3}$$

  N is the number of tunable elements. Since the data points are complex two neurons per point is needed. Michalski et al. [16] also found that using more input neurons than necessary, up to some point, decreases training time. Therefore between 24 to 204 input neurons will be tested.

  The more input neurons are used, the more complex patterns should be findable in the data, but the more redundant information should be found as well. If the input is too large the network should have trouble learning well because there are so many weights to optimize, whereas to few input neurons should mean that there are not enough patterns in the data to learn from.

In the Matlab tool-kit a number of stopping criteria can be defined for batch training. If any of the following is true the training will stop.

- Number of epochs have reached 1000.
- The error function is 0 for the training set.
- The gradient of the back-propagation algorithm is less than $1E^{-7}$.
- The validation error has increased 6 times in a row.

The exact values of the different stopping criteria can be changed. The values stated in the list are the values used in this thesis. An epoch is one iteration of presenting all examples to the neural network. For batch training all the examples will be presented once and then the weights of the neural network will be updated based on the total error. With EET the neural network will be updated after each example, and when all examples have been presented the epoch is finished.

For EET training no built in stopping criteria exist, instead the following stopping criteria has been implemented:

- Number of epochs have reach 800.
- The test error has increased 6 times in a row.
- The test error has not decreased more than 0.1 in the last 100 epochs.

Back-propagation with momentum

The general update function that was presented in section 2.5 is:

$$w_{ji}^{l}(t) = w_{ji}^{l}(t-1) + \Delta w_{ji}^{l}(t) \tag{2.4}$$

w_{ji}^{l} is the weight being updated and Δw_{ji}^{l} the change that is made to the weight. How this change is calculated depends on the update algorithm.

For back-propagation with momentum, the change update Δw_{ji}^{l} is calculated as:

$$\Delta w_{ji}^{l}(t) = L_r \frac{\delta E}{\delta w_{ij}^{l}} + \mu \Delta w_{ji}^{l}(t-1) \tag{2.5}$$

where L_r is the *learning rate*, $\frac{\delta E}{\delta w_{ij}^l}$ is the error derivative with respect to the weight that is being updated. μ is the *momentum coefficient* and $\Delta w_{ji}^l(t-1)$ is the change made to the weight the previous update.

The back-propagation with momentum algorithm was used by Michalski [14] and will therefore be tested in this thesis as well. The following parameter choices have been made for the neural network using back-propagation with momentum as its update algorithm.

- **Learning rate L_r:**
  In general, the lower the learning rate the better. A learning rate that is too large will cause the network to miss a solution by overstepping it. However if the learning rate is to low, the learning time will be longer. The default value in Matlab is 0.01, and according to Basheer and Hajmeer [1], common suggestions are in the range 0 to 1. In this thesis 0.001, 0.01 and 0.1 will be tested, as there is an order of magnitude in difference between the values. This should make the results from using each value differ noticeably.

- **Momentum coefficient μ:**
  If the momentum coefficient is too low, the risk of finding a local minima is increased, and the training will take more time. If the momentum coefficient is too high, the risk of ending up in a local minima is reduced, but the risk of overshooting the global minima is increased. Using a $\mu > 1$ may cause instability in the search. The Matlab default value is 0.9. In order to get a wide range of values 0.1, 0.5 and 0.9 will be tested.

RPROP

Just as for back-propagation with momentum the update formula for RPROP is:

$$w_{ji}^l(t) = w_{ji}^l(t-1) + \Delta w_{ji}^l(t) \tag{2.4}$$

Unlike back-propagation with momentum RPROP only uses the sign of the error gradient when deciding the size of its weight change update Δw_{ji}^l. As mentioned in section 2.5 RPROP makes use of a weight update $\Delta_i^l j(t)$ which is increased when the current error gradient has the same sign as the previous error gradient. Each weight in the network has its own weigh update $\Delta_i^l j(t)$, which is updated as follows [20]:

$$\Delta_i^l j(t) = \begin{cases} \eta^+ * \Delta_{ij}^l(t-1) & \text{if } \frac{\delta E(t-1)}{\delta w_{ij}} * \frac{\delta E(t)}{\delta w_{ij}} > 0 \\ \eta^- * \Delta_{ij}^l(t-1) & \text{if } \frac{\delta E(t-1)}{\delta w_{ij}} * \frac{\delta E(t)}{\delta w_{ij}} < 0 \\ \Delta_{ij}^l(t-1) & \text{else} \end{cases} \tag{3.5}$$

η^+ and η^- are the increase and decrease factors. The weight update $\Delta_i^l j(t)$ will then decide the change update Δw_{ji}^l in the following way:

$$\Delta w_{ji}^l = \begin{cases} -\Delta_{ij}^l(t-1) & \text{if } \frac{\delta E(t-1)}{\delta w_{ij}^l} * \frac{\delta E(t)}{\delta w_{ij}^l} < 0 \\ -\Delta_{ij}^l(t) & \text{if } \frac{\delta E(t)}{\delta w_{ij}^l} > 0 \\ +\Delta_{ij}(t) & \text{if } \frac{\delta E(t)}{\delta w_{ij}^l} < 0 \\ 0 & \text{else} \end{cases} \tag{3.6}$$

Verbalized, Δw_{ji}^l will be negative if the error gradient is positive (meaning the error has increased) and the weight will be decreased. If the error gradient is negative, the weight will increase. If the error gradient have changed sign since the last step, the previous weight update will be reverted (as per the first case in 3.6). This will undo the previous update. If

this happens, the gradient is bound to change its sign in the next turn. In order to avoid this double "backtracking", $\Delta_i^l j(t)$ should not be changed in the in the succeeding step.

In the tests performed by Schiffmann, Joost, and Werner [22] RPROP was the best performing update algorithm, that was also available in Matlab's neural network tool-kit. Therefore it will be tested in this thesis. The following parameter choices was made for RPROP:

- **Learning rate L_r:**
  For RPROP the learning rate changes depending on the sign of the gradient. However a starting learning rate will still need to be set. The starting learning rates that will be tested are 0.001, 0.01 and 0.1 as there is an order of magnitude in difference between the values. This should make the results from using each value differ noticeably.

- **increase factor η^+ and decrease factor η^-**
  Riedmiller and Braun [20] recommends using 0.5 for η^- and 1.2 for η^+. In this thesis η^- was set to 0.5 and for η^+ the values 1.1, 2 and 5 is tested. η^+ has to be larger than 1, and in order to test a wide range of values, some values had to be very large. η^- will not be tested in order to save computing time.

Conjugate gradient

As explained in 2.5, the conjugate gradient algorithm works by selecting a direction and doing a line search to find a minimal point along that direction. A new direction is selected such that moving in that direction will not destroy the updates done earlier.

Conjugate gradient was the update algorithm recommended by LeCun et al. [12]. They recommend it when the training set is not large or when the neural network is used for function approximations rather than for classification. Since both of these conditions are true for the problem in this thesis, conjugate gradient will be tested as an update algorithm. There are no parameters that will be set for the conjugate gradient update algorithm other than those parameters that are shared for all update algorithms.

3.4 How many examples are needed?

Michalski [14] presents the density function 3.8. They also state that for the generalisation error to stay at a certain level when changing the search space, the density must stay the same. The volume of the search space is:

$$V = (2K)^N \tag{3.7}$$

where N is the number of dimensions of the search space and 2K is the length of each dimension. On a filter N corresponds to the number of tunable screw in that filter and K is the maximum number of screw deviations. In both this thesis and in the work by Michalski [14], the maximum screw deviation is $\pm 360^\circ$ and each adjustment was made in steps of 18°. Since $\frac{360}{18} = 20$, K is 20 as well.

The density function presented by Michalski [14] is calculated by dividing the number of learning examples with the volume of the search space:

$$d_0 = \frac{L_0}{(2K)^N} \tag{3.8}$$

Michalski [14] also derives equation 3.9 which can be used to estimate the number of examples needed to keep the same generalisation error when the number of dimensions in the output space increases.

$$L = L_0(2K)^{(N_{new} - N_{known})} \tag{3.9}$$

L is the number of examples needed for the new output space. N_{known} is the number of dimensions in the "old" output space, where it is already known how many examples are needed to reach a desired generalisation error. N_{new} is the number of dimensions in the "new" output space, for which we want to estimate the number of examples needed to reach a certain generalisation error. L_0 is the number of examples used to reach the desired generalisation error on the "old" output space with N_{known} dimensions.

Michalski [14] then creates a test for determining the number of learning vectors needed before the generalisation error stops improving. They used a filter with $N = 6$ tuning elements.They collected 2000 learning vectors for the test case. The experiments showed that using more than 500 learning vectors, did not improve performance. As was mentioned earlier, $K = 20$ both in the work byMichalski [14] and in this thesis.

With these results it is now possible to use equation 3.9 to estimate the number of learning examples needed for the filter used in this thesis:

$$L = 500(2 * 20)^{(N_{new}-6)} \tag{3.10}$$

The filter used in this thesis will have 5 tunable elements so the number of learning elements can be estimated as:

$$L = 500(2 * 20)^{(-1)} = 12.5 \tag{3.11}$$

Clearly the size of the search space is not the only thing that affects the number of learning vectors needed.

Michalski [14] did not test equation 3.9 in the article, and from the description it is clear that equation 3.9 was intended to be used when increasing the search space, not decreasing it.

3.5 How the examples were gathered

The data will be gathered by connecting a filter with a VNA. The procedure is as follows:

1. A random screw deviation is determined by using the following Matlab formula:

   $$s = rand(1, 5) * (2 * k) - k;$$
   $$screw = int64(s);$$

   The function will generate random integers between $-20u$ and $20u$. Since the example outputs given to the neural network are in the unit u, the output produced by the neural network will also be in the unit u.

2. The screws are manually turned according to the generated values and the resulting S_{11}-parameter is saved.

This procedure will be repeated until enough data has been gathered. The VNA will save 1601 points on the S_{11}-parameter graph. To use this as input the data points will be reduced with the following algorithm:

$$N = floor(len/numElements); \tag{3.12}$$
$$s11 = s11(1 : N : len); \tag{3.13}$$
$$s11 = s11(1 : 1 : numElements); \tag{3.14}$$

Where len is the length of S_{11}, and numElements is the number of elements S_{11} should be reduced to. It is worth noting that if the number of elements desired can not be selected

evenly, the rightmost sample point will be removed until the desired number of elements have been selected. The purpose of this was to make the algorithm as simple as possible. Since the points removed will be outside the band-pass, they should be less interesting than points inside the band-pass.

The Matlab tool-kit contains methods for preprocessing data before it is used in the neural network. A function that maps the input and output values to values between 0 and 1 will be used when the transfer function is a logarithmic sigmoid. When a tanh sigmoid is used the input and output values will be normalized to have zero mean and a standard deviation of 1 in accordance to the recommendations given by LeCun et al. [12].

4 Result

Section 4.1, will explain how to interpret the results. In Section 4.2-4.7 the effects of different design parameters will be presented. Section 4.8 will look at what the gathered data can tell about how many examples are needed to learn to tune a filter. Lastly, in section 4.9 the results in this thesis are compared to those of Michalski [14].

4.1 Understanding the results

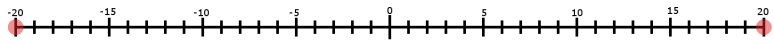
As explained in section 3.1, the maximum screw deviation is $\pm 360^\circ$ and each screw adjustment is made in steps of $\pm 18^\circ$. The unit $[u]$ is defined as *number of screw adjustments*. For example, a screw deviation of 36° can be written as a deviation of $2u$. The maximum screw deviation, measured in u is 20. The output of the neural network is also measured in u. As a consequence of this, the error will also be measured in u. An error of $4u$ in this case, would mean that on average, the neural network will make a guess that is $4u(= 72^\circ)$ from the correct value for each screw. All the output examples given to the neural network was made in integer values. That is, an output example can not have the value 2.3 or 5.7, since these values are not integers. The neural network outputs floating point values, and therefore it may output 2.3 or 5.7, even though these values could never be correct.

In order to evaluate how good a result is, it can help to compare it to how good a program that randomly selects a value between $-20u$ and $20u$ for each screw would be. In order to calculate this it is necessary to know how likely each "distance" from the correct answer is.

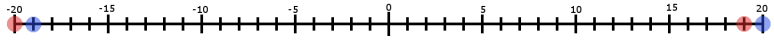
There are only two ways to guess $40u$ from the right answer (by guessing -20 when the answer is 20, and by guessing 20 when the answer is -20). Figure 4.1a illustrates this. The red circle at -20 is $40u$ away from the red circle at 20 and the other way around. There are no other ways of placing circles so that the circles are $40u$ away from each other.

There are four ways to guess $39u$ from the right answer, as is illustrated by figure 4.1b. The two red circles are $39u$ away from each other and the two blue circles are also 39 steps away from each other. As the illustration shows, there are therefore four ways to be 39 steps from the right answer.

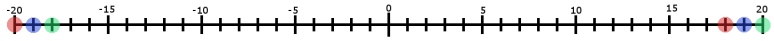
There are six ways to guess $38u$ wrong. Figure 4.1c illustrates this scenario. the two green circles are $38u$ from each other. The same is true for the two blue circles and the two red circles.



(a) There are two ways to guess $40u$ wrong.



(b) There are four ways to guess $39u$ wrong.



(c) There are six ways to guess $38u$ wrong.

Figure 4.1: theses figure illustrates in how many different ways a guess can be $40u$, $39u$ and $38u$ from the right answer

This pattern, where the number of ways to guess increases by two each time the error decreases by one, continues all the way down to guessing $1u$ wrong from the right answer, which can be done in 80 different ways. The only exception is guessing exactly right which can only be done in 41 different ways (one way for each value).

By adding $2 + 4 + 6 + ... + 78 + 80 + 41$ together we get the total number of different guesses.

In order to get the mean absolute error we also have to calculate the total "distance" we get from all these different guesses and divide by the number of guesses. That is, calculate

$$\frac{40 * 2 + 39 * 4 + 38 * 6 + ... + 2 * 78 + 1 * 80 + 0 * 41}{2 + 4 + 6 + ... + 78 + 80 + 41} \approx 13.7u \tag{4.1}$$

This means that a program with an error rate close to $14u$ is pointless since simply random guessing would be just as good.

The method used for calculating the error is mean absolute error. The error is calculated according to this function:

$$E = \frac{\sum_{j}^{L}\sum_{i}^{N}|target_{ij} - output_{ij}|}{L * N} \tag{4.2}$$

N is the number of tunable elements (five in this thesis). L is the number of examples.

As was explained in section 2.5 the examples should be divided into three sets. One that is used for training, and two sets that is used for evaluating the neural network. The first evaluation set, the test set, is used to see what error the neural network has on examples it has not seen before. When training the neural network, the goal is to make sure that the error

Update algorithm	Transfer function	Data type	Input neurons	Hidden neurons	best error
Back-propagation with momentum	logarithmic sigmoid	complex	144	50	2.57
RPROP	logarithmic sigmoid	complex	164	80	2.09
Conjugate gradient (twice as many examples)	logarithmic sigmoid	complex	124	70	1.44

Table 4.1: The table shows which design decisions led to the best result for each update algorithm, as well as what the best validation error was.

on the test set is as low as possible. The second evaluation set, the validation set, is used to see what error the neural network has on examples it has neither seen before nor been optimized to perform well on. All the errors shown in the results will be validation error, except for the error regarding EET training. Generalization error (that is the error from the first evaluation set, the test set), will be used instead, as validation error was not implemented for EET.

In table 4.1 the best design for each update algorithm is presented as well as the best validation error achieved for that update algorithm.

4.2 Back-Propagation Algorithm

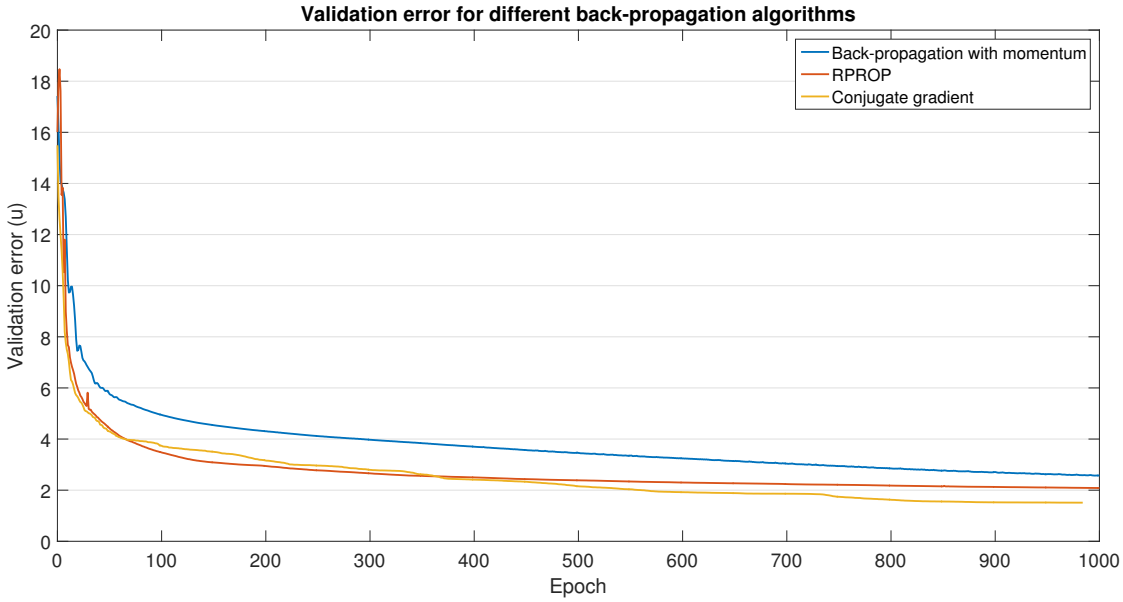


Figure 4.2: Comparison between back-propagation with momentum, RPROP and conjugate gradient. Logarithmic sigmoid was used.

In figure 4.2 three different update algorithms are compared. In the figure, the validation error after different number of epochs is shown. An epoch is one iteration of presenting all learning examples to the network and using the total error to update the weights in the neural network. A figure 4.2 shows, the conjugate gradient update method performed best and back-propagation with momentum performed worst.

The parameters and neural network architecture that lead to the best result were used for each of the update algorithm. All neural networks were trained with batch training.

4.3 Example-by-example training

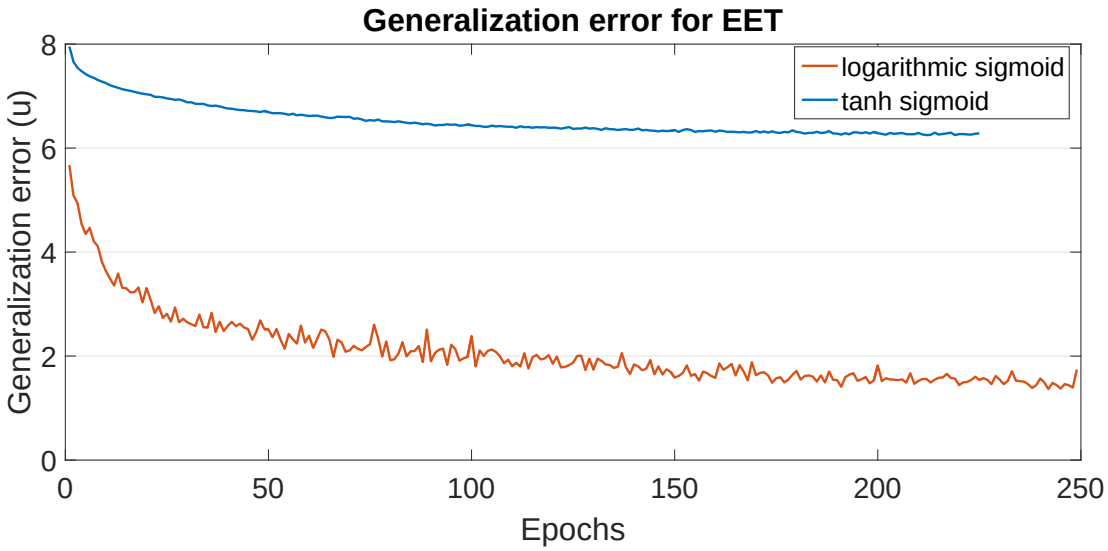


Figure 4.3: Comparison between a logarithmic and tanh sigmoid as transfer function when training with EET. Back-propagation with momentum was used as update algorithm

Figure 4.3 shows the result of using EET when training the neural network. The neural network used back-propagation with momentum as its update algorithm and for transfer function, both a logarithmic sigmoid and a tanh sigmoid was tested. For the figure the combination of parameters that lead to the best result was selected. As figure 4.3 shows, a tanh sigmoid did not perform well compared to a logarithmic sigmoid. EET in combination with a logarithmic sigmoid however worked well. The best generalisation error was 1.36 which is slightly lower than the best validation error (1.44). Normally one can expect the generalisation error to be slightly larger than the validation error. However as will be shown later, the best validation error was achieved with twice as many input examples.

4.4 Number of hidden neurons

When gathering data for the performance of the conjugate gradient update algorithm, different numbers of input neurons and hidden neurons were tested. In figure 4.4 a box plot shows how the number of hidden neurons affects the error. Each result from a particular number of hidden neurons has a different number of input neurons. The logarithmic sigmoid was used as the transfer function in all neural networks. The best value as well as the best median value was achieved with 70 hidden neurons. That group also had the widest span between the 25th percentile and 75th percentile, which indicates that the number of input neurons have a larger effect on the results for this group than for other groups.

4.5 Number of input neurons

When testing the effect of changing the number of input neurons, conjugate gradient was used as the update algorithm. The logarithmic sigmoid was used as the transfer function. The number of hidden nodes used was 70. These parameters were selected because this combination yielded the best result for the conjugate gradient update algorithm. In all other experiments the number of input neurons varied between 12 and 102, with increments of 10. In this case the number of input neurons varied from 11 to 1601, with increments of 10.

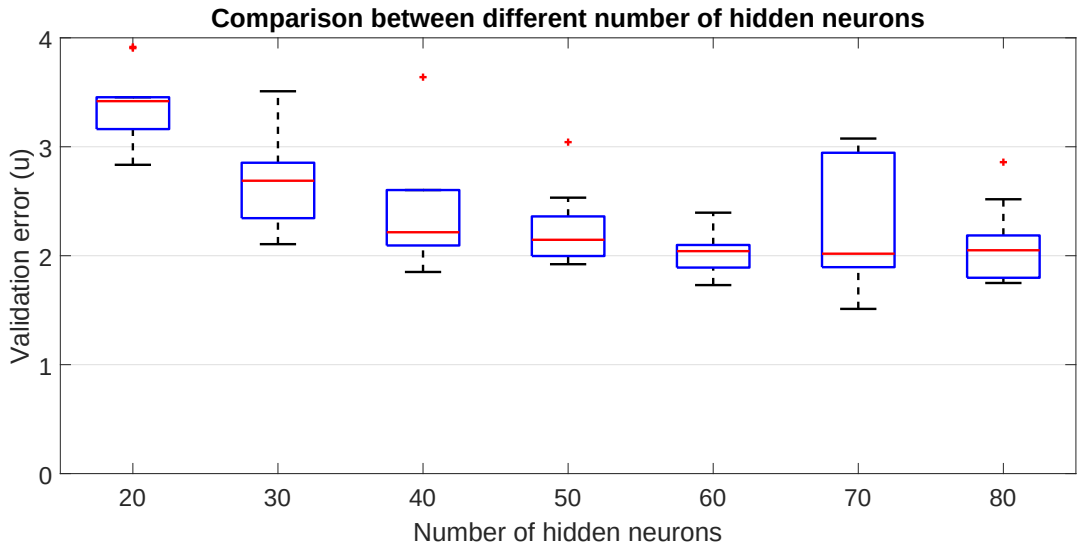


Figure 4.4: A box plot of how the number of hidden neurons affect the error. The red line in each box is the median value in that group. the top and bottom line of the box is the 75th respective 25th percentile of the group. The two black lines above and below the box is the maximum and minimum value of that group. The red crosses are outliers. Some outliers have been removed in order to better present the data

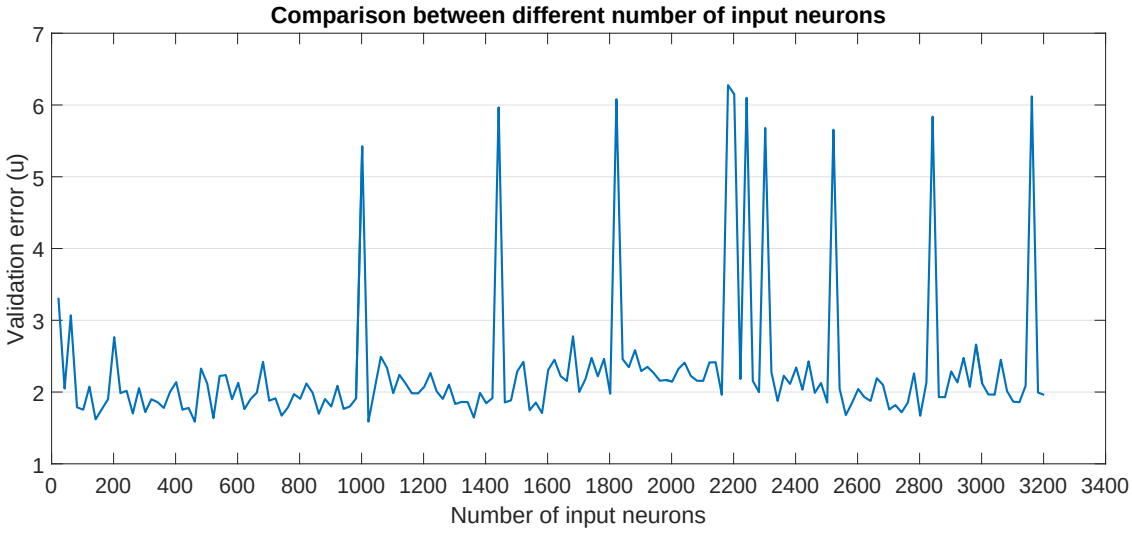


Figure 4.5: This figure shows the relation between the error and the number of input neurons in the neural network.

In figure 4.5 there is no apparent connection between number of input neuron and the error. Some input sizes result in a very large error, but this could just be a result of a bad weight initialization.

However, if one removes the extremely large errors in figure 4.5, as have been done in figure 4.6, one can see that there is some pattern in the data. Somewhere between 200 and 1000 input neurons gives the best performance.

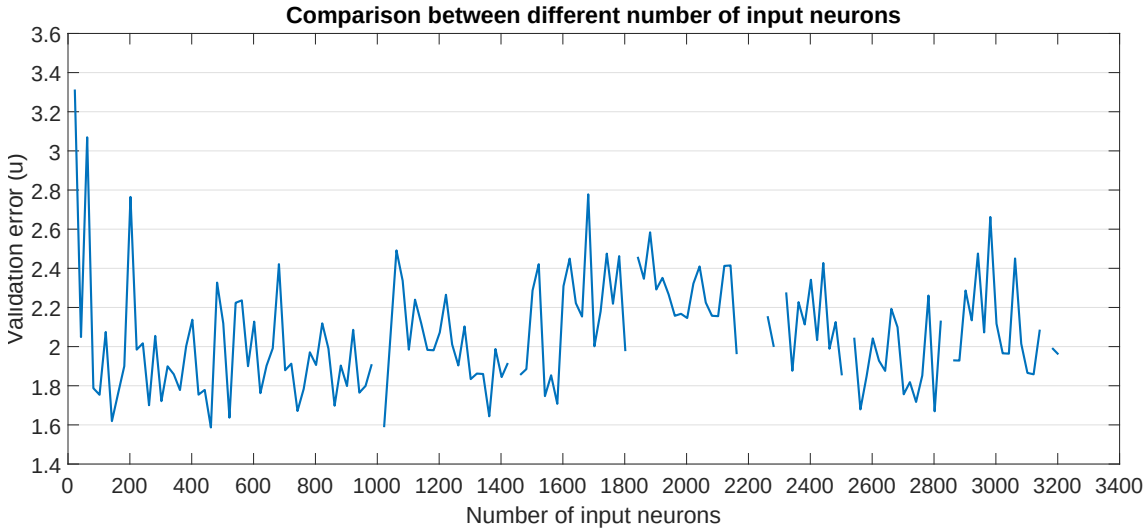


Figure 4.6: This figure shows the relation between the error and the number of input neurons in the neural network, when outliers have been removed.

4.6 Type of input

S_{11} has a complex value for each frequency. In order to visualize S_{11}, each complex value is converted to a logarithmic format with the following function:

$$magnitude[dB] = 20 * Log(\sqrt{Re^2 + Im^2}) \tag{2.2}$$

These converted points are then plotted against the frequency (An example can be seen in figure 2.8a).

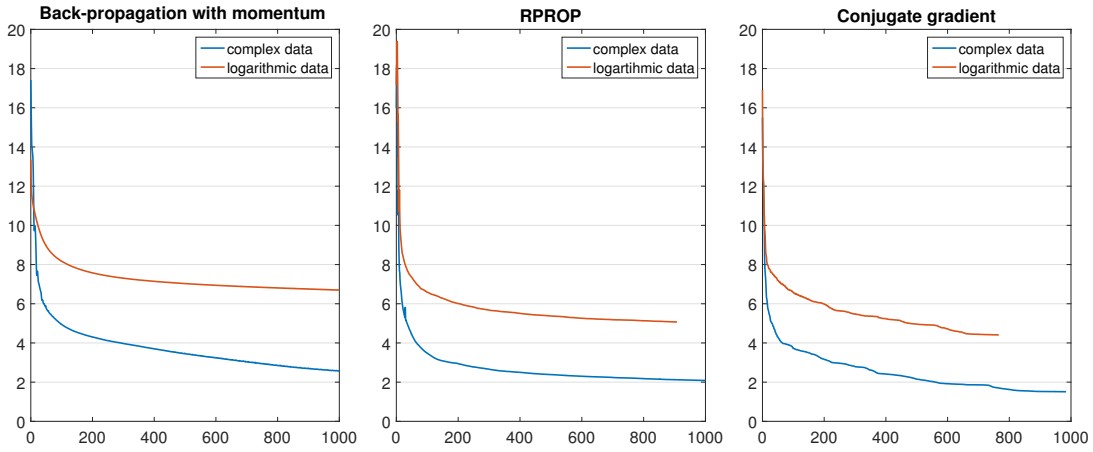


Figure 4.7: Comparison between logarithmic and complex input

As was explained in section 3.2, human experts view S_{11} in a logarithmic format when fine tuning. Therefore it would be interesting to see if presenting the input in this format would be beneficial for the neural network as well.

Figure 4.7 shows the difference between using logarithmic input and complex input for the three different update algorithms. For both the complex and the logarithmic case the transfer function that yielded the best error was selected. The number of input neurons and

the number of hidden neurons was also chosen based on which combination had the best result. All neural networks were trained with bath training.

Figure 4.7 shows that using complex values give better results than using logarithmic values, regardless of which update algorithm is used. While it is not shown in figure 4.7, the best transfer function for both complex and logarithmic input was a logarithmic sigmoid.

4.7 Transfer function

When comparing the transfer functions, the combination of number of input neurons and number of hidden neurons that performed best was used. All neural networks were trained with batch training.

Figure 4.8 shows the effect of using a logarithmic sigmoid and a tanh sigmoid for the three different update algorithms. As was explained in section 3.3, LeCun et al. [12] suggests using a sigmoid centred around 0. The results shows that a logarithmic sigmoid as the transfer function performs better in all three cases, despite the suggestions of LeCun et al. [12].

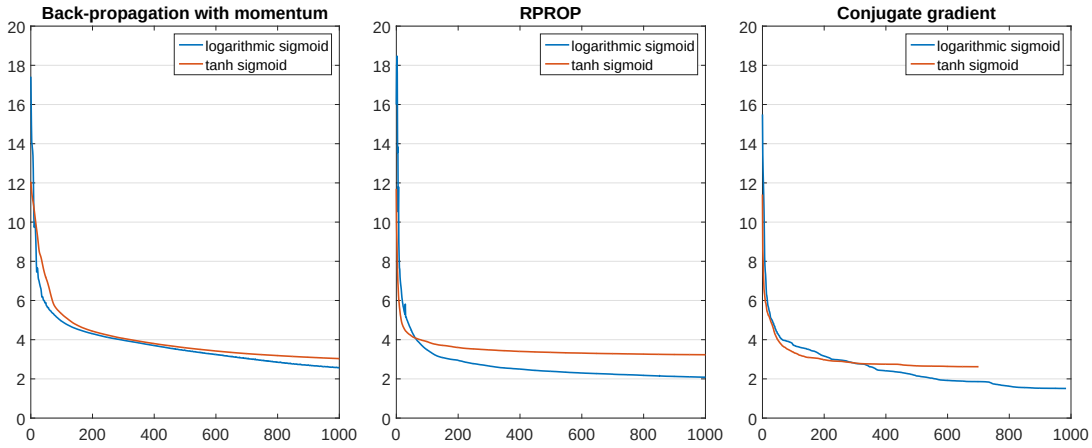


Figure 4.8: Comparison between a logarithmic sigmoid with range [0, 1] and a tanh sigmoid with range [-1. 1]

4.8 How many examples are needed?

Michalski [14] presented a function that relates the number of examples needed with the size of the output space. As was shown in section 3.3, this approximation does not work well for a smaller filter. The results in this thesis and the results by Michalski [14] can not be used together and therefore it is not possible to estimate a better approximation between number of examples and output size.

Instead lets look at the data that can be gathered to see what conclusions can be drawn from them.

As was shown in figure 2.12, for each connection between two neurons, there is an associated weight w_{ij}^{l}. Since the goal of training a neural network is to find a suitable value for each weight it can be interesting to see if the number of weights in the neural network affects the error. The number of weights in a neural network can be calculated with the following formula:

$$weights = input * hidden + hidden * output \tag{4.3}$$

In figure 4.9 the number of input neurons and hidden neurons have been varied to create neural networks with a different number of weights. The conjugate gradient update algorithm was used in combination with a logarithmic sigmoid as the transfer function. The best validation error achieved for a different number of weights in the neural network is shown, after outliers have been removed.

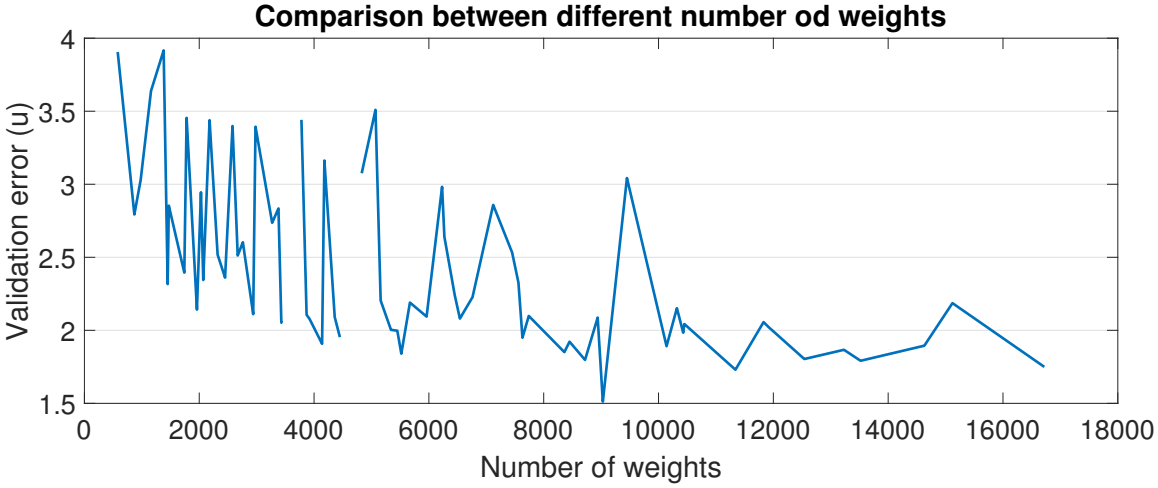


Figure 4.9: This figure shows the relation between the error and the number of weights in the neural network

As figure 4.9 shows, the error goes down as the number of weights increases. There is also more variation in the error when the number of weights is lower. although this could be because there are fewer neural network with many weights.

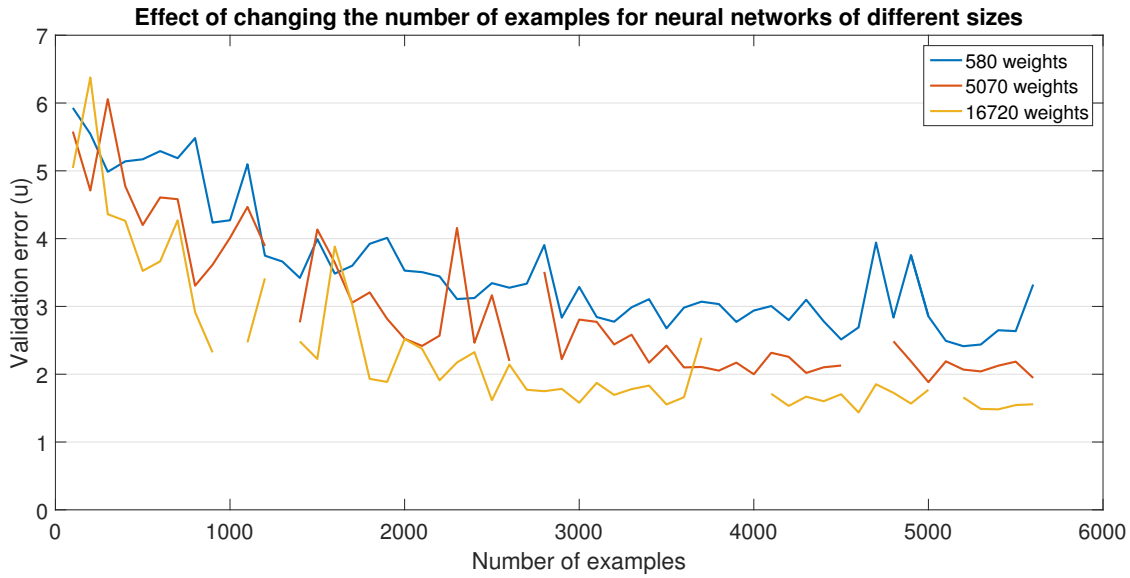


Figure 4.10: This figure shows the relation between the error and the number of examples for three different values for the number of weights in the neural network.

Figure 4.10 shows how the validation error is affected by the number of examples used. The conjugate gradient update algorithm was used with a logarithmic sigmoid. In order to test up to 5600 examples it was necessary to also increase the number of examples that have had noise added to it. Basheer and Hajmeer [1] suggests that the number of learning examples

should increase if the number of weights in the neural network increases. Therefore it can be interesting to see how the error for neural networks with a different amount of weights will change when the number of examples used is changed.

The three neural networks with the smallest, largest and median number of weights is shown in figure 4.10 after outliers have been removed. Neural networks with a different number of weights had a similar result to those shown in figure 4.10.

Figure 4.10 shows that increasing the number of examples improves the error, regardless of how many weights the neural network has. It is also clear that the more examples are added the less the error improves regardless of the number of weights used. Figure 4.10 also shows that a neural network with more weights will reach a lower error, as the number of examples increases, compared to a neural network with fewer weights.

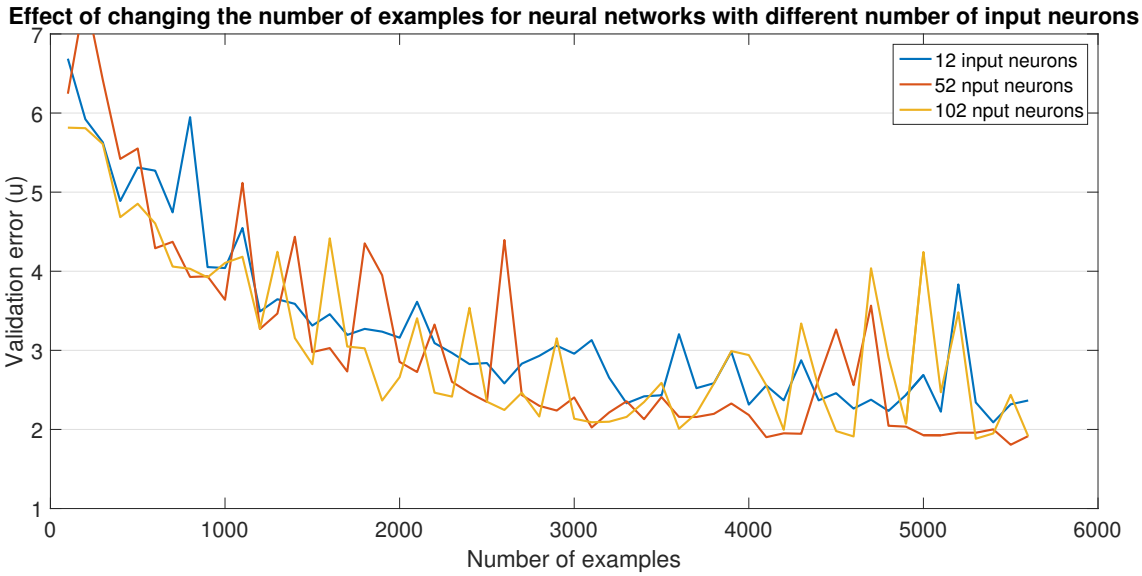


Figure 4.11: This figure shows the relation between the error and the number of examples. Three different values for the number of input neurons in the neural network is shown.

Figure 4.11 was created with the same data set as figure 4.10, but this time the data was ordered based on the number of input neurons. The graphs show the mean value of all neural networks with that number of input neurons. As can be seen, varying the amounts of input neurons only has a small effect on the error. The difference does not appear to change much when the number of examples increases either.

Figure 4.12 has also been created with the same dataset as figure 4.10 and 4.11. The data has now been ordered based on the number of hidden neurons. The graph shown in figure 4.12 shows the mean value for all neural networks with that amount of hidden neurons.

Adding more hidden neurons has a larger effect on the error than adding more input neurons does. It also looks like adding more examples improves the error more, when more hidden neurons are used, even though the gain is not very large.

The best validation error (1.44) was achieved with the data set used to create figure 4.10, 4.11 and 4.12.

4.9 How the results compare to Michalski

In order to compare the results in this thesis with the results of Michalski [14], one first has to make sure that the errors are calculated in the same way. Michalski et al. [16] introduces a generalisation error that is said to be the same as the one introduced by Michalski [14]. It is also shown that the error is measured in units of u, just like the error in this thesis.

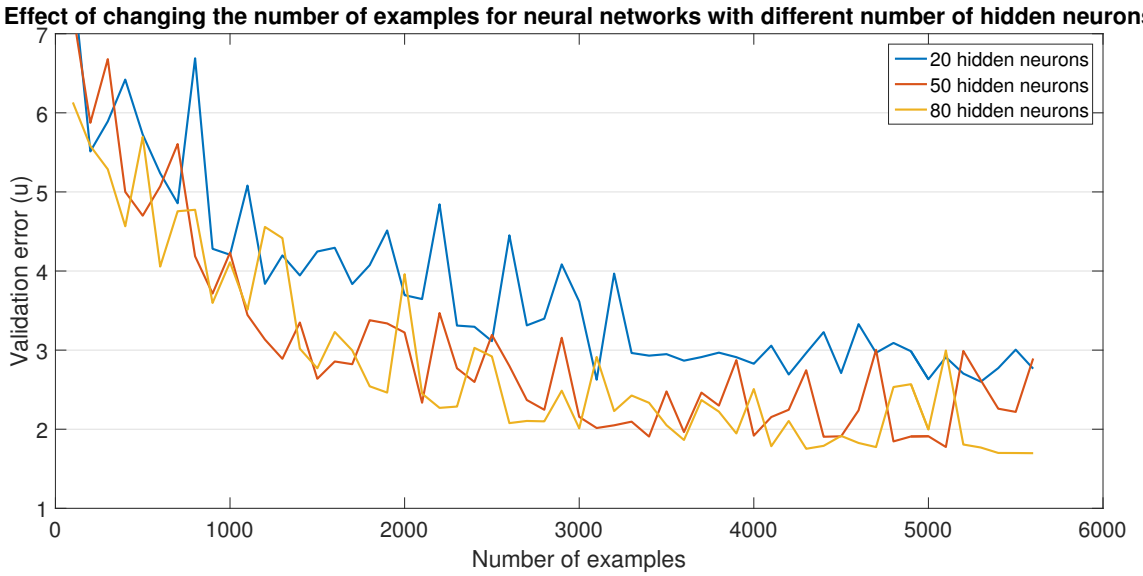


Figure 4.12: This figure shows the relation between the error and the number of examples. Three different values for the number of hidden neurons in the neural network is shown.

However, the error function presented by Michalski [14], (assuming the variable used in the error formula has the same meaning as it had earlier in the article) should be in the unit $2K * u$. The intended unit is not shown explicitly. Kacmajor and Michalski [7] does not explicitly state the unit used for the error, but from the units used when calculating the error one can conclude that the error is measured in units of u. In that article the authors also compare their results with the results by Michalski [14] and by comparing the figures made by Kacmajor and Michalski [7] and by Michalski [14] it does not seem like the values have been converted in any way. Based on this it will be assumed that all errors presented by Michalski [14] and Michalski et al. [16] are in the unit u.

Michalski [14] did not calculate the validation error for their neural network, instead they calculated the generalisation error. The difference between the two errors is that the neural network is optimized to have the lowest generalisation error, and validation error is used to see how well the neural network perform on examples it has not been optimized for. Michalski [14] managed to get a generalisation error for a 6-pole filter that was slightly below $0.5u$, and for a 11-pole filter the generalisation error was slightly below $1.5u$. In this thesis the best validation error achieved was ($1.44u$) which is very similar to the error Michalski [14] had for the 11-pole filter.

5 Discussion

Section 5.1 will try to answer the research questions presented in 1.2. In section 5.2 issues with the chosen method will be evaluated. In section 5.3 potential future work will be discussed, and finally section 5.4 contains a conclusion of the results and the work in a wider context.

5.1 Result

- **Can a neural network fine tune a filter to meet its specification?**

  The lowest error achieved in this thesis is not small enough to be used in a commercial product, but it is small enough to show that the method itself works.

  Michalski [14] had a generalisation error for a 6-pole filter that was slightly below $0.5u$. For an 11-pole filter the generalisation error was slightly below $1.5u$. This was not enough to completely tune the filter. Michalski [15] later showed that using more filters was enough to tune the filters to meet their specification. The best error ($1.44u$) achieved in this thesis is worse than the result for the 6-pole filter it is most comparable to. While the filter used in this thesis was smaller and therefore should be easier to tune, there are a number of reasons for why the error was still worse than that of Michalski [14]:

  - Michalski [14] explains that in their work the examples were gathered using a robot, whereas in this thesis the data was gathered by hand. This means that the data in this thesis likely contains wrongly labelled examples, and that the measurement error is larger, both of which should decrease performance. The number of examples used are also fewer, although the number of examples have been based on what Michalski [14] found to be enough.
  - In order to make sure a given result could be generated again, the random generator in Matlab (which was used to set the initial weights in the neural network) was set to a default value. Training several neural networks with randomly initialized weights and then selecting the network that performed best, is one way to improve the results. Michalski [14] does not mention doing so, but that does not mean that they did not.

  - In this work the validation error was used for all tests except when testing EET, where generalisation error was used. Michalski [14] only used generalisation error when evaluating their results. In general the validation error should be slightly higher than generalisation error. In this work the difference between generalisation error and validation error has been on the scale of 0.1

Despite these drawbacks it was still possible to reach a relatively low error, which shows that the method used by Michalski [14] is feasible for more filter that those tested in their work.

- **How does the architecture of the neural network affect performance?**

Update algorithm
Figure 4.2 showed that conjugate gradient was the best update algorithm for this problem. The figure also showed that both conjugate gradient and RPROP performed better than back-propagation with momentum, as the results by LeCun et al. [12] and Schiffmann, Joost, and Werner [22] suggested.

Transfer function
The results shown in figure 4.8 is more unexpected. Against the advice from LeCun et al. [12] a positive sigmoid performs better for all update algorithms tested. It is possible that which transfer function is better varies with the update method. Another reason for the result could be that with the tanhsig transfer function the data was normalized to have a zero mean and a standard deviation of 1, whereas the data was mapped between 0 and 1 when using the logsig transfer function. It is possible that in this particular case mapping all values to have a range within the transfer function is a better choice, even though LeCun et al. [12] suggest otherwise.

Number of input neurons
The effect of varying the number of input neurons is small but noticeable. From figure 4.5 it is clear that between 200-1000 input neurons performs best, even if the variation is still big.

Number of hidden neurons
In figure 4.4 the effect of varying the number of hidden neurons was shown. There is a tendency that more hidden neurons is better. It was also shown that, with the exception of 70 hidden neurons, the span between the 25th percentile and 75th percentile was fairly small despite a change in number of input neurons. This shows, together with figure 4.6, that the number of input neurons does not have an as large effect on the results as the number of hidden neurons

Number of weights
Figure 4.10 and figure 4.9 both shows that increasing the number of weights in the neural networks improves the performance. Figure 4.10 also shows that the benefits of using more weights is not visible until enough examples have been added. In theory a larger neural network should be able to generalize at least as well as a smaller neural network. The larger neural network could "emulate" a smaller neural network by setting some weights to zero. Given enough time, a larger neural network should therefore either perform better than a smaller neural network or set its weights in such a way that it performs very similarly to a smaller network.

Function 4.3, which shows how to calculate the number of weights in a neural network, can be rewritten as:

$$W = hidden(input + output) \tag{5.1}$$

5.1 shows more clearly that in order to change the number of weights as much as possible, the smaller value of *hidden* and $(input + output)$ should be increased. This can explain why there were a clear correlation between validation error and more hidden neurons, while no clear correlation was found between the validation error and the number of input neurons.

In figure 4.4 between 20 and 80 hidden neurons were tested and the number of input neurons varies between 24 and 204. In other words, the number of hidden neurons were fairly close to the number of input neurons. This meant that increasing the number of hidden neurons would increase the number of weights in the network noticeably. In figure 4.6 the number of input neurons varies between 22 and 3202. For the larger values, the number of hidden neurons is so small comparatively, that increasing the number of input neurons will barely affect the number of weights. This means that an increase in number of hidden neurons in figure 4.4 is related to a similar increase in number of weights to a larger degree than an increase in number of input neurons in figure 4.6.

- **How can the data be presented to the neural network to improve performance?**

  Preprocessing
  As figure 4.7 shows, using complex data is better than using logarithmic data, despite the fact that the logarithmic format suits human experts best. The reason for this is probably that the complex format contains more information than the logarithmic format, which combines the real and imaginary values into one new value. This is detrimental to a neural network but beneficial to a human that prefers data that can be easily visualized.

  Training type
  Generalisation error was used when testing EET. In all other cases validation error was performed. I general the validation error should be slightly higher than the generalisation error. However the difference between generalisation error and validation error in the tests using batch training was on the order of 0.1. Therefore the generalisation error for training with EET (1.36) should still be somewhat comparable to the validation error used with batch training. Figure 4.3 showed that using back-propagation with momentum, trained with EET performs comparable to the conjugate gradient update algorithm trained with batch training and twice as many examples (which had an validation error of 1.44). This is in accordance with LeCun et al. [12] who explained that when using EET, one will often get the same result, but with fewer calculations. Schiffmann, Joost, and Werner [22] found that algorithms trained with EET performed better than algorithms trained with batch. It is not certain if that was the case in this thesis as well, as generalisation error was used in the EET case, but it is fairly certain that back-propagation with momentum will perform better if trained with EET rather than batch, since the difference in generalisation error was about 0.1.

  According to LeCun et al. [12], EET can handle slowly changing functions. For example functions changing because of machine wear and tear. This should be useful for a neural network that tunes filters. The mould that is used when casting the filters will change after being used many times. This affects the filters being cast, and changes how the filters needs to be tuned. With EET it would be possible to add examples from new filters that have been tuned by humans after the neural network have been deployed. This way the neural network can learn how the tuning needs to change, when the mould changes.

- **Can the number of examples needed to reach a certain error be estimated?**

Effect of number of tuning parameters
Michalski [14] got a much worse generalisation error on an 11-pole filter (c.a. $1.5u$) compared to a 6-pole filter (c.a. $0.5u$). Which suggest that the size of the output space have a large effect on the performance.

However, none of the results in this thesis indicates that changing the size of the neural network by slightly increasing the number of output neurons would have a large negative effect on the error results. This suggests that it is the complexity of the filter, rather than the size of the neural networks output space, as Michalski [14] suggested, that decides what amount of examples will be needed to reach a certain error.

I have learned from expert tuners that the more poles a filter have, the harder it is to fine tune it. This may not have to do with increasing the number of screws in itself, but rather because the relationship between the frequency response and the screws positions is more complex compared to a filter with fewer poles. Even though each pole have its own resonance frequency, one can not identify an individual pole by looking at the frequency response, because the response at a particular frequency will depend on all the poles. For a filter with fewer poles, moving the resonance frequency of one pole should therefore have a larger and "easier" to understand effect, compared to a filter with many poles.

Another thing I learned about cavity filters was that for a human tuner, the challenge when tuning a filter, is making sure that the poles are in the correct order. That is, placing the poles such that the pole that is supposed to have the lowest resonance frequency, actually does have the lowest resonance frequency, and so on. Once the poles have been "sorted" in this way, it is relatively easy to tune the filter so that it fulfils its specification. Before the poles have been placed in the right order, the frequency response is usually bad, and might not improve much until all poles are placed in the correct order. This could explain why it is much harder for a neural network to learn a filter with more tuning parameters, even though the number of tuning parameters is not much larger. The number of ways the poles could be ordered is $N!$, where N is the number of poles, or tuning parameters. If the reason a filter with more tuning parameter are harder to tune, is mostly because it is harder to find the correct ordering of the poles, the number of examples needed, should have to increase with the number of permutations of pole ordering.

Effect of changing neural network size
Figure 4.10 shows that the error rate exponentially decays as the number of examples increases. The figure also shows that a neural network with more weights will achieve a smaller error if enough examples have been used. From this, one can learn that if the error stops improving, despite adding more examples, adding more weights to the network might improve the error. In this particular problem, it was possible to select both the number of hidden neurons, and the number of input neurons. Figure 4.11 and 4.12 shows how the error rate varies with the number of examples for neural networks with different number of input neurons and hidden neurons. In figure 4.11 we can again see that the error decays exponentially as the number of examples increases. The number of input nodes does not seem to have an as large effect on the error as the number of weights did in figure 4.10.

Figure 4.12 also shows that the error decays exponentially with the number of examples, and that the number of hidden nodes does not appear to have a large effect on the error. 20 hidden neurons gives a slightly worse result than using 50 or 80 hidden neurons, which have very similar errors. This is probably because too few hidden neurons makes it harder for the neural network to learn. It is also worth noting that the difference in weight is very large in figure 4.10, while the difference is not so large in figure 4.11 and

4.12. It is possible that the difference in error would have been larger if the difference in number of input neurons and hidden neurons had been larger.

These results further support the conclusion that the number of weights is more important to the error rate than the number of hidden neurons or input neurons alone.

Summary
What the results have said so far is that, in general, increasing the number of weights in the neural network will decrease the error. But in order to take advantage of the extra weights, the number of examples may need to be increased. If increasing the size of the neural network does not improve the error, adding more examples should help.

If each neural network architecture was tested several times with different weight initializations, together with even larger neural networks, it might be possible to estimate how much lower the error would be if the number of examples stayed the same, but the number of weights increased. It might also have been possible to estimate how many examples would be needed before the error stops improving, given the number of weights stay the same.

5.2 Method issues

The main issue in this work is the quality and size of the data. The data had to be gathered by hand and this set a limit to how much data could be gathered and also on how exact the measurements were. Michalski [14] recommended selecting the value u (how many degrees one step change in the screws position is) experimentally. However, this was not possible, since there was not enough time to gather enough data points to do that. Instead the choice was made to use the same value for u as Michalski [14] did (18°). It would have been interesting to see if the neural network would produce better results with a smaller u. But as long as the data is gathered by hand it is not possible to have a much smaller u, than 18°, because the measurement error would be to big.

Only one filter was used when gathering data. But as is shown by Michalski [15], using more than one filter to train the neural network was necessary to tune the filter well enough to not need further fine tuning. The reason for only using one filter was that only one filter was available for use, and also that there were not enough time to gather learning samples from more than one filter.

The quality of the data and the number of filters used in this thesis meant that it was never possible to reach an error low enough to be useful as a commercial product.

The number of examples gathered was based on the findings by Michalski [14] on how many examples could be removed without affecting performance. Basheer and Hajmeer [1] suggests adding noise to data in order to make the network more robust towards measurement error. Because of this noise was added so that 2800 examples were used to train the networks. Basheer and Hajmeer [1] also mention suggestions on how many examples are needed to train a network successfully. Two suggestions are an example to weight ratio of more than 4, and more than 10. If the examples with added noise counts as examples, the example to weight ratio would be between $[0.17 - 4.8]$ depending on the number of input and hidden neurons. The best performing network had a ratio of 0.45. Therefore it is possible that not only better quality data, but simply more data could improve the results as well.

Other than increasing the number of examples, it might also have been worthwhile to consider different frequencies of the S_{11}-graph. It seems reasonable to assume that the most interesting aspect of S_{11} are those values within the band-pass region. In this thesis a slightly wider range was used.

Lastly the time limit had an effect on the results as well. If there had been more time available it would have been possible to test more back-propagation algorithms. A more exhaustive pre-study could also have yielded different suggestions as to what back-propagation algorithms should have been tested first.

5.3 Further work

The most obvious thing to be done is to try to gather higher quality data and see if that improves the performance. This could be done by using a motor to turn the screws automatically. A motor would be able to rotate the screws more precisely than a human can. The risk of mismatching input examples and output examples would also be lower if the data gathering was automatic.

More examples should also help improve performance. Although it is worth noting that the filter may get worn out if the screws are turned too often, this would make the measurements less reliable, especially if the neural network is supposed to be used for tuning new filters.

Aside from improving the data there are a number of techniques that could be added to this work as well:

- Kim and Guest [10] describes how to change the back-propagation algorithm to accept complex values. Using this method would cut the number of input neurons in half and, unlike in this thesis, the real and imaginary parts of a data point would not be treated as two unrelated variables by the neural network. This should mean that patterns in the data are easier to find for the neural network.
- I have been told that a human tuner should look at both S_{11} and S_{21} when tuning. It would be interesting to see if S_{21} is more useful as input than S_{11} and if using both S_{11} and S_{21} would improve the results. This could be especially interesting to use in combination with the complex back-propagation algorithm. When looking at a VNA, I have found the changes in S_{21} to be more subtle than the changes in S_{11}, but this does not mean that a neural network will have a harder time finding patterns in the S_{21} graph.
- Kacmajor and Michalski [7] have tried combining a neural network with fuzzy logic. They reported a 20% improvement compared to the results by Michalski [14]. It would be interesting to see what design choices could have been made differently and how they affect the result.
- A human expert can both handle a filter where the screw deviation is much larger than 360° and correctly adjust a filter where the screw deviation needed are very small. Rather than increasing the output space, one could train several neural networks with different screw deviation sizes. This way a filter could first get a "rough" tuning from a neural network using a large screw deviation, and then a finer tuning from a neural network, that uses a smaller screw deviation. The second neural network would also be able to assume that the maximum error for a screw is much smaller than the previous neural network.
- More experienced human tuners often chose to view a smaller frequency range compared to a novice tuner, when tuning a filter. Evaluating the effect of using different frequency ranges could be interesting. If the frequency range is small, some misplaced poles may be outside the frequency ranged looked at. But unless the pole is very far away from the viewed frequency range, it should still have an effect on the frequency response in the viewed range. One example to test could be only using the band-pass range as input. In this range S_{21} should be as "flat" as possible without having a too low value. This might be something a neural network can learn easily.
- The requirements of S_{11} and S_{21} differ depending on whether the frequency is inside the band-pass region or outside it. Therefore one approach could be to divide the input so that not all hidden nodes receives input from all input nodes. Maybe it would be better to use different hidden neurons for the input values from the band-pass frequencies and the input values from the frequencies outside the band-pass.

- If it is possible to train a neural network to estimate the screw deviation for all screws in a filter, it should also be possible to train a neural network to estimate the screw deviation of a single screw. A different approach to this problem could be to train one neural network per screw. If each network is allowed to "specialize" on one particular screw, the end result might be better.

- The output examples used in this thesis only had discrete values, but the neural networks output was continuous. It would have been possible to use the neural network as a classifier instead. If one neural network per screw was used, each possible step adjustment could be treated as a class, and the neural network would then have to determine which class its screw belonged to.

- Something that this thesis did not answer, is how the error would change if a smaller step change than 18° was used. If u was defined to be for example 9° How would the error change? If the error stayed the same, that is, the best error achieved was still $1.44u$, the filter tuning would be better, since an average error of $1.44u$ would translate to an average error of $1.44 * 9 = 12.96°$.

5.4 Conclusion

This thesis was not able to reach a low enough error for the neural networks tuning to be enough on its own. However it has been able to show that the results achieved by Michalski [15] should be reproducible on filters with a different amount of tunable elements than the filters used by Michalski [15] in their work.

It has also been discussed that the method for estimating the number of examples needed, that were presented by Michalski [14] did not make a good estimation for the filter used in this thesis. Instead it has been suggested that it is the complexity of the filter (which depends on the number of tunable elements) rather than the size of the output space that decides what number of examples is necessary to reach a certain error.

The main work in this thesis however was comparing the effect of different design parameters, a summary of the results regarding the design are presented below:

- **Update algorithm**
  For the architecture of the neural network, it was found that conjugate gradient was the best update algorithm for this problem and that it should be used in combination with a logarithmic sigmoid transfer function. A logarithmic sigmoid as transfer function performed best on all the update algorithm used.

- **Neurons**
  When it comes to the number of neurons, there does not seem to be a clear connection between validation error and number of input neurons. There is a correlation between more hidden neurons and a lower validation error, but this may have been caused by hidden neurons being closely related to the number of weights in the experiments performed. Increasing the number of weights in the network was shown to improve the validation error.

- **Training**
  When training the neural network, using EET rather than batch training is likely to be preferred. This improved the back-propagation with momentum update algorithm enough to perform comparable to the conjugate gradient update algorithm, while also using half as many examples. This suggests that EET should be used if the update algorithm allows it, and also that it may be worthwhile to change update algorithm if it does not support EET.

- **Data preprocessing**
  Presenting the neural network with complex input examples was shown to be far superior to presenting the input examples in logarithmic format. This is probably because the logarithmic format combines the real and imaginary values into one new value, and thus reducing the amount of information.

From the results the following rules of thumb for training a neural network to fine tune a cavity filter can be presented:

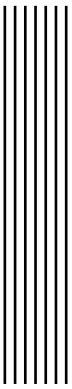
- When the error stops improving, add more weights to the neural network, If this does not help, add more examples.
- As suggested by Schiffmann, Joost, and Werner [22] one should train the neural network with EET. If this is not possible, conjugate gradient should be used as the update algorithm as was suggested by LeCun et al. [12].
- The input examples should be presented in complex rather than logarithmic format as this preserves more information.

The work in a wider context

Today filters are tuned by human experts. This would soon change if a product that could tune these filters automatically was available, as the filters could then be produced faster for a lower price. This work is a part of an ongoing change where human work is automated and done by computers and robots.

Frey and Osborne [3] explains that in the past few decades, the computerization has lead to a reduction in middle income jobs, as computers have been able to perform more and more routine tasks. Instead the workers have moved to low-skilled, low-income jobs and high-skill, high-income jobs. The authors further predicts that more and more low-skill, low-income jobs will be replaced by computers and robotics. People working in those field will instead have to find jobs requiring creative and social skills.

A future without mundane and dangerous jobs would of course be a good thing. However we do not yet know how the transition into such a future will work, nor if the human workforce will be able to pick up new skills fast enough. It is not hard to find articles predicting a future where robots are taking all our jobs, leaving the vast majority unemployed. In these scenarios, very few people, if any, will be able to benefit from the automation. Of course these doomsday descriptions about automation have existed since before the industrial revolution [3], and may very well be as true now as they were then.



Bibliography

[1] I.A Basheer and M Hajmeer. "Artificial neural networks: fundamentals, computing, design, and application". In: *Journal of Microbiological Methods* 43.1 (2000). Neural Computting in Micrbiology, pp. 3–31. ISSN: 0167-7012. DOI: http://dx.doi.org/10.1016/S0167-7012(00)00201-3. URL: http://www.sciencedirect.com/science/article/pii/S0167701200002013.

[2] Neural network-based face detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.1 (Jan. 1998), pp. 23–38. ISSN: 0162-8828. DOI: 10.1109/34.655647.

[3] C. B. Frey and M. A. Osborne. "The future od employment: how susceptible are jobs to computerization?" In: (Sept. 2013).

[4] Jun Han and Claudio Moraga. "From Natural to Artificial Neural Computation: International Workshop on Artificial Neural Networks Malaga-Torremolinos, Spain, June 7–9, 1995 Proceedings". In: ed. by José Mira and Francisco Sandoval. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. Chap. The influence of the sigmoid function parameters on the speed of backpropagation learning, pp. 195–201. ISBN: 978-3-540-49288-7. DOI: 10.1007/3-540-59497-3_175. URL: http://dx.doi.org/10.1007/3-540-59497-3_175.

[5] Robert Hecht-Nielsen. "Counterpropagation networks". In: *Appl. Opt.* 26.23 (Dec. 1987), pp. 4979–4984. DOI: 10.1364/AO.26.004979. URL: http://ao.osa.org/abstract.cfm?URI=ao-26-23-4979.

[6] J. J. Hopfield. "Hopfield network". In: *Scholarpedia* 2.5 (2007). revision #91362, p. 1977.

[7] T. Kacmajor and J. J. Michalski. "Neuro-fuzzy approach in microwave filter tuning". In: *Microwave Symposium Digest (MTT), 2011 IEEE MTT-S International*. June 2011, pp. 1–4. DOI: 10.1109/MWSYM.2011.5972771.

[8] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285. DOI: 10.1613/jair.301.

[9] Keysight. *What are the equations that relate Real and Imaginary data pairs (as read from either a disk save, the GPIB interface, or a display marker) to Log Magnitude, Polar, Phase, and Smith Chart formats?* URL: http://www.keysight.com/main/editorial.jspx?cc=

SE&lc=swe&ckey=566084&nid=-32496.1150429&id=566084 (visited on 04/13/2016).

[10] M. S. Kim and C. C. Guest. "Modification of backpropagation networks for complex-valued signal processing in frequency domain". In: *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. June 1990, 27–31 vol.3. DOI: 10.1109/IJCNN.1990.137820.

[11] Randall D. Knight. *Physics For Scientists and Engineers - a strategic approach*. 3rd ed. Pearson Education, Inc, 2013. ISBN: 0321824083.

[12] Yann LeCun et al. "Neural Networks: Tricks of the Trade". In: ed. by Genevieve B. Orr and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. Chap. Efficient BackProp, pp. 9–50. ISBN: 978-3-540-49430-0. DOI: 10.1007/3-540-49430-8_2. URL: http://dx.doi.org/10.1007/3-540-49430-8_2.

[13] A. Lindner and E. Biebl. "A Manual Tuning Method for Coupled Cavity Filters". In: *Microwave Conference, 2006. 36th European*. Sept. 2006, pp. 1340–1342. DOI: 10.1109/EUMC.2006.281264.

[14] J J Michalski. "Artificial neural networks approach in microwave filter tuning". In: *Progress In Electromagnetics Research M* 13 (2010), pp. 173–188.

[15] J.J. Michalski. "Artificial neural network algorithm for automated filter tuning with improved efficiency by usage of many golden filters". In: *Microwave Radar and Wireless Communications (MIKON), 2010 18th International Conference on*. June 2010, pp. 1–3.

[16] JJ Michalski et al. "Consideration on artificial neural network architecture in application for microwave filter tuning". In: vol. 7. 3. 2011, pp. 271–275.

[17] V. Miraftab and R.R. Mansour. "Tuning of microwave filters by extracting human experience using fuzzy logic". In: *Microwave Symposium Digest, 2005 IEEE MTT-S International*. June 2005. DOI: 10.1109/MWSYM.2005.1517011.

[18] Kohonen network. In: *Scholarpedia* 2.1 (2007). revision #122029, p. 1568.

[19] Application Note. "1287-8: simplified filter tuning using time domain". In: *Agilent Technologies Corp* (2001).

[20] M. Riedmiller and H. Braun. "A direct adaptive method for faster backpropagation learning: the RPROP algorithm". In: *Neural Networks, 1993., IEEE International Conference on*. 1993, 586–591 vol.1. DOI: 10.1109/ICNN.1993.298623.

[21] Norvig P Russell S. *Artificial Intelligence A Modern Approach*. 3rd ed. Edingburgh Gate, Harlow, Essex CM20 2JE, England: Pearson Education Limited, 2014. ISBN: 1292024208.

[22] W. Schiffmann, M. Joost, and R. Werner. *Optimization of the Backpropagation Algorithm for Training Multilayer Perceptrons*. Tech. rep. 1994.

[23] S Söderkvist. *Tidskontinuerliga signaler & system*. 3rd ed. 2000.

[24] Torbjörn Ahl Uno Henningsson Patrik Lindell. "Skruvanordning samt trimanordning innefattande en sådan skruvanordning för trimning av ett kavitetsfilters frekvensförhållande eller kopplingsgrad". Patent SE 519554 (SE). Oct. 2000. URL: http://was.prv.se/spd/pdf/oH1HxgSNJyPWS3oljenFlQ/SE519554.C2.pdf.

[25] L.-X. Wang and J.M. Mendel. "Generating fuzzy rules by learning from examples". In: *Systems, Man and Cybernetics, IEEE Transactions on* 22.6 (Nov. 1992), pp. 1414–1427. ISSN: 0018-9472. DOI: 10.1109/21.199466.

[26] Per Wilén. *The RBS 2206 - A flexible ticket to third-generation wireless systems*. URL: https://www.ericsson.com/ericsson/corpinfo/publications/review/2000_02/files/2000024.pdf (visited on 06/14/2016).

[27] Jinzhu Zhou, Baoyan Duan, and Jin Huang. “Influence and tuning of tunable screws for microwave filters using least squares support vector regression”. In: *International Journal of RF and Microwave Computer-Aided Engineering* 20.4 (2010), pp. 422–429. ISSN: 1099-047X. DOI: 10.1002/mmce.20447. URL: http://dx.doi.org/10.1002/mmce.20447.