University of South-Eastern Norway

www.usn.no

**FMH606 Master's Thesis 2020**
**Electrical Power Engineering**

# Thermal model parameter estimation of a hydroelectric generator using machine learning

Emil G. Melfald

Faculty of Technology, Natural Sciences and Maritime Sciences
Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

**Course:** FMH606 Master's Thesis 2020
**Title:** *Thermal model parameter estimation of a hydroelectric generator using machine learning*
**Pages:** *149*
**Keywords:** *Machine learning, Mechanistic thermal models, Digital twin, Parameter estimation*

**Student:** *Emil G. Melfald*
**Supervisor:** *Thomas Øyvang, Co-supervisor: Ola Marius Lysaker; Bernt Lie*
**External partner:** *Ingunn Granstrøm, Skagerak Energi*

**Summary:**
The synchronous generator have a central role in the power system, and is governed by multi-physics behaviour. A digital twin of the synchronous generator can lead to more efficient utilization in a more safe and reliable way. In this report, mechanistic thermal models of a synchronous generator and a heat exchanger will be defined in order to assess the feasibility of parameter estimations of mechanistic models using machine learning and recurrent neural networks (RNNs). Skagerak Energi provides with operational data that has been used with the mechanistic models and machine learning algorithms in this project. The data was collected using Azure Databricks, from a 12 MVA synchronous generator located at Grunnaai.

The results from parameter estimations shows that a recurrent neural network has the ability to estimate the three model parameters defined in the heat exchanger model with great success. Optimization of the model parameters with respect to prediction error showed similar results, with the machine learning approach being slightly more accurate. The neural network could not find good parameter solutions to the six model parameters in the generator model. However, the RNNs consistently predicted parameters in a narrow range of values. This may indicate that the RNN was trained on training data not representative to the measurements. The optimization with respect to the prediction error showed great accuracy in model parameter estimation.

# Preface

From this projects inception to realization and implementation, there have been several things not gone according to plan. One of those things are some objectives defined in this thesis task description, shown in Appendix A. During the semester, the task description was changed to not include a mechanistic electrical model of the generator. The minutes of meeting are shown in Appendix B, where this decision was taken.

I have been fortunate enough to be able to implement my favourite scientific topic into my field of research, and be able to implementing machine learning as a tool in my electrical engineering toolbox. At the start of this project, I had little knowledge of computer science, modeling and programming experience, and I had no idea how big the field of research was. This thesis has truly been a learning experience.

I want to thank my supervisor Thomas Øyvang for taking the initiative to make this project happen, and help integrate machine learning in an electric power engineering thesis. I also want to thank him for allowing me to participate at PTK in Trondheim, an experience that introduced me to the real industry of electrical engineering and several of the companies involved.

I also want to thank Kristian Nymoen from Skagerak Energi for active support and help with Azure Databricks. Thanks to Ingunn Granstrøm from Skagerak Energi, my sub-supervisors Ola Marius Lysaker and Bernt Lie from USN and Ole Magnus from USN for contributing ideas and knowledge in machine learning, parameter estimation, modeling and general information in this project.

A great appreciation goes to the data programs used in this thesis as well. The data tools used in this thesis are:

- Overleaf for writing in Latex
- Azure Databricks for accessing data and writing the Python Notebooks
- The programming language of Python 3.7, together with the modules Numpy, Scipy, Datetime, Pandas, Matplotlib, pyspark, Keras and Tensorflow.
- Simuling for making a lumped parameter diagram
- Microsoft Visio
- Mendeley for referencing

Porsgrunn, 14th May 2020

Emil G. Melfald

# Contents

# List of Figures

# List of Tables

# Nomenclature

bla

| Symbol | Explanation |
|---|---|
| MVA | Mega Volt Ampere |
| Mvar | Mega Volt Ampere Reactive |
| ML | Machine Learning |
| ReLU | Rectified Linear Unit |
| ELU | Exponential Linear Unit |
| NN | Neural Network |
| RNN | Recurrent Neural Network |
| CNN | Convolutional Neural Network |
| NODE | Neural Ordinary Drifferential Equations |
| SNN | Stacked Neural Network |
| LSTM | Long Short Term Memory |
| GRU | Gated Recurrent Unit |
| $\dot{Q}$ | Heat flow rate $[W]$ |
| $hA_{A2B}$ | Thermal Conductance $[W/^\circ K]$ from point A to point B |
| $T_A$ | Temperature $[^\circ C]$ in point A |
| $\Delta T$ | Temperature difference $[^\circ C]$ |
| $C_A$ | Capacitance, normally referred to as thermal capacitance $[J/^\circ K]$ |
| $\tau$ | Thermal time constant $[s]$ |
| $\dot{H}$ | Enthalpy flow $[W]$ |
| $\dot{W}$ | Thermal heat produced through mechanical work $[W]$ |
| $\hat{c_p}$ | Specific heat capacity of a material $[J/K/kg]$ |
| $P$ | Electrical Power $[W]$ |
| $Q$ | Electrical Reactive Power $[var]$ |
| $R$ | Electrical resistance $[\Omega]$ |
| $I$ | Referred to as current in formulas $[A]$ |
| $M$ | Capital M in formulas refer to a mass $[kg]$ |
| LPTN | Lumped Parameter Thermal Network |
| PT100 | A Resistance Temperature Detector, measuring the temperature through a current signal. |
| $uA_x$ | The heat exchanger thermal conductance from one medium to the other |
| $\dot{m}_a$ | The mass flow of air through the generator or heat exchanger |

| **Symbol** | **Explanation** |
|---|---|
| $\dot{m}_w$ | The mass flow of water through the heat exchanger |
| $ML - factor$ | The percent amount of heat from mechanical losses in the generator that contributes to heating up the air. |
| MSPE | Mean Squared Prediction Error |
| SCADA | Supervisory Control and Data Acquisition |
| NaN | Not a Number |
| DSX | Data Set number X |

# 1 Introduction

In Norway, as long as there have been electricity, there have been hydro power plans. The first hydro power plant in Norway was operational in 1882 [1], and since then, the main source of electricity has been hydro power plans [2]. This is a reliable and flexible method of generating energy, with huge dams and reservoir effectively working as large-scale energy storage systems. As the demand for electrical energy continues to increase, infrastructure must be built to utilize the available renewable resources. As of 12'th of May 2020, there exists 1651 hydro power plants in Norway [3]. The available locations for hydro power plans are limited, and increasing the efficiency of existing power plants will not scale to the increasing power demand long term. The energy companies and the government are therefore looking at alternative sources of energy. The utilization of intermittent energy sources, such as wind and solar energy have great potential in Norway [4], and the energy market should expect to see more of these energy sources in the future. However, the fact that they are intermittent means there is a requirement for large scale energy storage, and flexible hydro power plants. Although hydro power energy production is very flexible, uneven operation such as many starts and stops can noticeably reduce lifetime of equipment such as turbines and rotor shaft. Repetitive temperature fluctuations in equipment may also lead to thermal fatigue, which can cause several types of failures, such as insulation failure [5].

## 1.1 Background

When it comes to the operation of electrical machines, such as synchronous generators, there are usually strict temperature limits in the equipment. Although the machines are usually extensively tested at the factory, with numerous measurements, different operating conditions can cause deviations from expected temperature values. Often, installing additional temperature sensor is expensive or unpractical, making thermal mechanistic models a good alternative for temperature estimates inside the generator.

There are literature about case studies of designing mathematical thermal models for hydroelectric generators, including [6], [7] & [8]. Thermal modeling of hydroelectric generator have several challenges, such as the mathematical modeling and the estimation of model parameters. Some model parameters may have very different values, depending on generator size, manufacturer, equipment type etc.

Skagerak Kraft is a subsidiary company from Skagerak Energi AS, and they have averaged a production of 5.7 TWh each year for the last ten years [9]. It supplies power to over 200000 customers in the south-eastern part of Norway. They operates a decent amount of hydro power plants, one of which is named Grunnaai. In 2019, Skagerak Energi started the operation of a 12 MVA synchronous generator at Gruunaai. The generator is meant to facilitate research projects, such as this project [10]. Data collected from operation of this generator is logged in Microsoft Azure Databricks, where the data can be accessed through its cloud storage system. Skagerak Energi has the intention to develop a digital twin of the generator for more efficient utilization and flexible operation, and is the main reason for facilitate this project with their data.

## 1.2 Motivation

Digitization of the power production and power systems requires the modeling and collection of enough data to replicate a digital version of the physical system with all the variables of interest. A paradigm of a digital power systems may lead the way to a more efficient energy system with better control and stability. Achieving full digitalization of the power system requires accurate and robust digital twins of the electrical power equipment. A description of a digital twin is " virtual representation of a physical product or process, used to understand and predict the physical counterpart's performance characteristics" [11]. A digital twin of the synchronous generator can increase the utilization flexibility, while reducing mechanical and thermal fatigue because one would have more control over the internal states in the generator during operation. It can also be a great asset for preliminary detection of faults [12]. There is the mechanical, thermal electromagnetic and electric parts that may be of interest when developing digital twins of a generator. Tackling all of these aspects will require extensive research, modeling, measuring and testing. This report will focus only on the thermal aspect of this challenge.

The hydroelectric generators limiting factor for continuous power production is heavily dependent on its thermal design [13]. Measuring the temperatures of all equipment in a generator may not be feasible, and will require many expensive sensors. Mathematical models can be a great asset to this challenge, as the temperatures in generator is dependent on variables that can be measured more easily, such as currents, cooling air, cooling water temperatures etc.. Having a good model of the thermal properties in a synchronous generator is an important step towards a multi-physics digital twin.

The field of Machine Learning (ML), when used right, has huge potential in expanding and assisting with development and research projects. Lately, more and more data from power plants are being collected in an effort to use ML algorithms to teach us more about the power system and power generation [12]. Research from Open AI indicate that agents trained by reinforcement learning can learn to adapt to a variety of different scenarios,

shown by their multi-agent interaction research [14]. If this type of AI can prove to be reliable, robust and safe, it may play a key role in a new paradigm of generator and power system control, where the objective is to minimize losses, maximize voltage stability and generator lifetime and reduce risks to a minimum. For these types of models to work, digital twins need to be reliable and accurate.

## 1.3 Problem Statement

The main goal of this project is to assess how to use machine learning to predict model parameters for a mechanistic thermal generator model of the new 12 MVA synchronous hydro generator at Grunnaai. The objectives for working towards this goal are listed below. The first iteration of the task description for this thesis is shown in Appendix A.

1. Do a survey on the relevant machine learning technologies that can be used to solve the objectives.
2. Describe the sensor types and technology used at Grunnaai.
3. Set up a thermal mechanistic model of a synchronous generator.
4. Collect operational data from Skagerak Energis Grunnaai 12 MVA synchronous generator.
5. Use traditional optimization tools to fit the thermal model to operational data.
6. Generate neural network training data using the developed mechanistic model, and operational data.
7. Evaluate different neural network hyperparameters before fully committing to a machine learning model.
8. Train a neural network in Azure Databricks (provided by Skagerak) and make model parameter predictions using operational data.

The core activities in this project are the design of the mechanistic model, parameter estimation with traditional optimization algorithms, and parameter estimation with neural network algorithms. This project will contain two study cases for parameter estimation in two different thermal mechanistic models. Study case 1 will revolve around parameter estimation for a heat exchanger thermal model, based on data from Grunnaais 12 MVA synchronous generator. This study case will also work as an assessment for the performance and feasibility of parameter estimation using neural networks, while establishing methods that will be used for study case 2. Study case 2 is parameter estimation of a thermal mechanistic model of Grunnaais 12 MVA synchronous generator, where model parameters will be fitted based on the obtained data.

## 1.4 Tools, data, case and methods

The data will be provided by Skagerak Energi and accesses from Microsoft Azure Databricks. The data is collected from the SCADA-system for controlling and monitoring the generator. The data are acquired in this project spans from 15. October 2019 to 14. February 2020. Skagerak Energi owns their data, and a condition of using their data is to keep the programming environment inside Azure Databricks. In addition, as a requirement from Skagerak, the data will not be publicly accessible.

The programming language used in this project will be Python version 3.7 inside the Microsoft Azure system. The python code will be written as notebooks. Several important modules in Python will be used, such as Pandas, Numpy, Scipy, Matplotlib, Datetime, Tensorflow and Keras. Pandas are used for data processing and analysis. Numpy is used for data processing and calculations. Scipy is used for solving sets of differential equations and optimizing parameter values. Matplotlib is used for presenting data and results in figures and graphs. Datetime is used to keep track of the timestamps in the data. Tensorflow and Keras are providing the machine learning algorithms, together with all the necessary utilities for this, such as activation functions, optimization algorithms, dropout and much more.

All python commands are executed through Azure Databricks, and the hardware used for training the neural network are the ”Standard_NC6” configuration in Azure [15].

Machine learning algorithms learn from learn from labeled data, and the model parameters are not known. The way to generate training data in order to use machine learning is illustrated in Figure 1.1, where figure (a) represents a mechanistic model, which needs some input data and model parameters to make a prediction (output). The neural network will turn this process on its head by attempting to learn what types of *model parameters* is present while given the *model outputs*, and *model inputs*. This process is illustrated in Figure 1.1 (b). Making the training data for the neural network to estimate model parameters is trivial, as random model parameter guesses can be used to simulate some output temperature from the models.

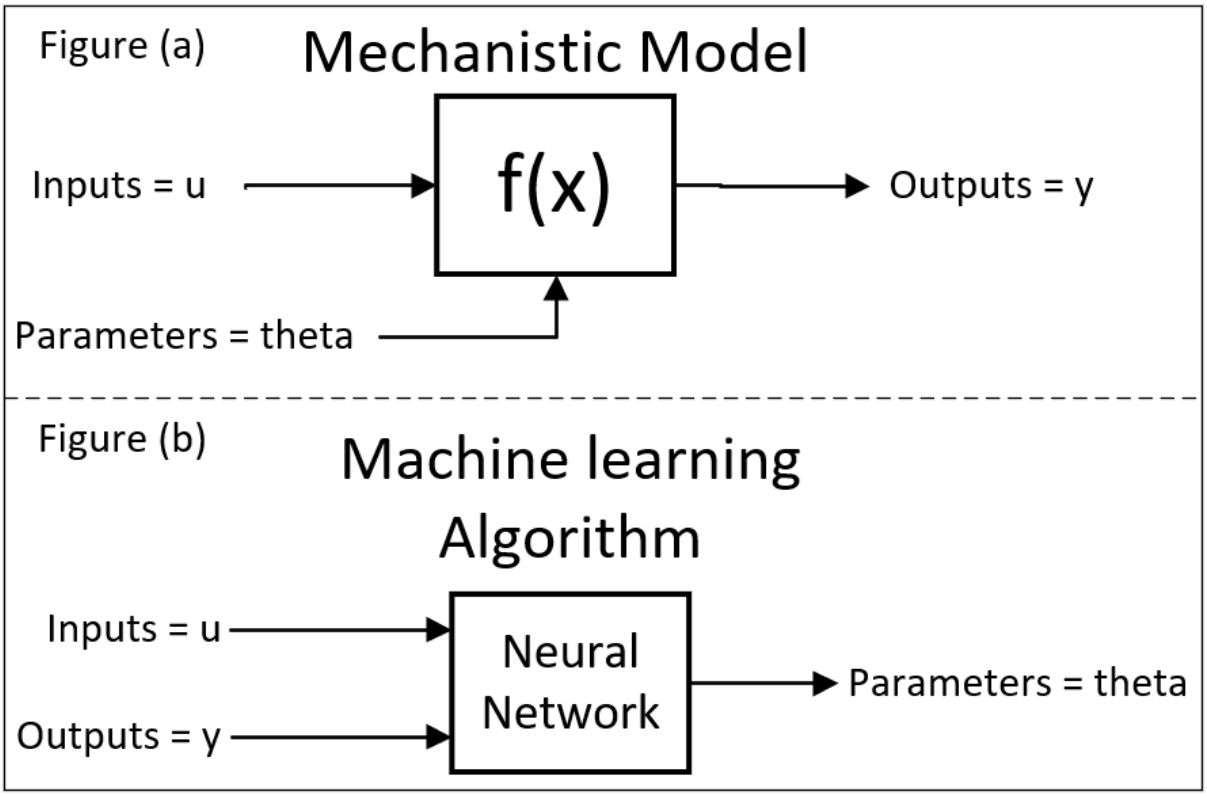



Figure 1.1: Functional diagram of the mechanistic model and the neural network model.

# 2 Machine Learning Technology

Nowadays, artificial intelligence and machine learning has been introduced to a number of industries, enabling technologies such as self-driving cars, facial recognition software, recommendation algorithms and forecasting to name a few [16].

Machine learning is a broad term, and in general it means that computers have the ability to learn from data. From a engineering perspective, machine learning can be defined with the following statement: "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. - Tom Mitchell, 1997" [17].

Although machine learning systems have a broad usage, it is not a "miracle cure" that will solve all our problems. For machine learning to work, real effort must go into data acquisition, deciding machine learning type, assumptions and robustness [18]. The learning objective must also be well defined to work with machine learning's strengths, which can be pattern recognition and imitation of historic data [16]. One main weakness of machine learning is that it can be prone to noise, in the sense that it can get really good at a particular tasks. But when implemented in the real world, the algorithm can perform poorly because of noise or other environmental factors that it hasn't been trained for. The best remedy for this is large quantities of quality data that will represent real-life data [19].

In this chapter, a short overview of some important machine learning models will be presented, and the project-relevant technologies will be presented.

## 2.1 Machine learning systems

When applying machine learning algorithms to problems, one has to assess the type of problem, and the goal for the algorithm. In literature, there are commonly defined four groups of machine learning types that has some clear distinctions from one another. These types are different in what data they learn from and capabilities [17]. The main types are defined as:

- Supervised Learning
- Unsupervised Learning

- Semi-supervised Learning
- Reinforcement Learning

Semi-supervised learning can be seen as a hybrid of supervised and unsupervised learning, and will not be discussed in this report.

### 2.1.1 Supervised Learning

Supervised learning is used when the objective is to classify or predict one or several values (labels or targets) based on input data (features). The machine learning algorithm will achieve this by learning through labeled training data. There are two main types of supervised learning methods, namely classification and regression. The main difference between the two methods is that regression predicts a numeric value as the target, while classification predicts the most likely feature, from a set of choices [17].

An example of a classification problem is when the input features are the color values of all pixels in a picture, and the algorithm should classify wherever that picture is a cat or a dog (or any label in practice). The algorithm has in this example two choices, a cat or a dog, and outputs a score of what label it thinks the picture is. The highest score is the algorithms pick, since it can only choose from a discrete amount of choices.

An example of a regression algorithm is to predict the price (target) of a computer based on its features (specifications). Another example could be to predict parameters in a mathematical equations (targets), given the output of that given equation (labels).

### 2.1.2 Unsupervised Learning

The objective in unsupervised learning is not to predict some target value based on features, but rather to find connections and correlations between some given input features. The data given to the unsupervised learning algorithms are unlabeled.

A usage of unsupervised learning may be for the super market owners to look at sales data from customers. The unsupervised algorithm may find connections such as people buying soda is more likely to buy chocolate in the same shopping trip. Therefore the owners of the store can plan the product placements such that these two products are placed closer to each other. It might also be used as recommendation algorithms, such that it can cluster people with similar preferences together in groups. If the case is that many people who likes sci-fi books also like jazz music, the recommendation algorithm already may know a persons likely music preferences based on movie genre preferences [20]. This is a very interesting topic that will not be discussed further in this report.

### 2.1.3 Reinforcement Learning

Reinforcement learning is special in that it don't require data to learn. The approach is to make an instance of the learning algorithm, called an agent, that should be a part of an environment. This could be for instance a car in the traffic environment or a player in a game. The goal of the agent is to learn what actions will lead to the highest score, where score is based on the performance of the agent in the given environment. The main challenges in reinforcement learning is to design the scoring-function, and it can be very computationally heavy. This is because it may needs millions of iterations of agents to show behaviour that increases score.

Arguably, the most publicly known case of a reinforcement learning algorithm is Deepmind's Alpha Zero. By letting the algorithm know only the rules of a given game, and by playing against itself for a time period, it can learn the game Chess and Go to a "super-human level" [21]. On March 2016, the algorithm Alpha Go from Deepmind was the first algorithm in history to beat a world champion in the game of Go. Reinforcement learning shows great potential to the machine learning world, and will probably be the right direction for achieving artificial general intelligence [22]. Reinforcement learning will not be used as a method in this project, and will therefore not be discussed further.

## 2.2 Supervised Learning Algorithms

In this project, supervised learning will be the applied method for machine learning. There is several supervised learning algorithms, and many of them are different approaches for solving the same problems types. The list below is a selection of a few well-known algorithms, but are not by any means an exhaustive list [17]. This project will focus on neural networks, because this is the machine learning algorithm most suitable for interacting with time-series data.

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines
- Decision trees and Random Forest
- Neural Networks

## 2.3 Artificial Neural Networks

The term "Neural Network" refers to a collection/system of neurons that changes state based on some input. The artificial-term arises when neurons are structured programmatically [23]. What a neural network is, on a basic level, is a was to process input data and map it to some output data through a number of intertwined connections. In machine learning terminology, the artificial neural network is normally shorted to just neural network (NN).

In Figure 2.1, a graphical representation of a basic neural network is displayed. The input features are displayed as the x's, which are connected to the neural network through so-called weights. These weights are unique element-wise multiplication from all the input features to all the nodes in the next layer (the hidden layers). In addition, a bias is added to each node, represented by the yellow circles with 1's, and the weights connecting this node to the rest of the layer. The sum of all inputs and bias in a node is then passed into an activation function. There are many types of activation functions, but will for now be represented as $A(x)$.

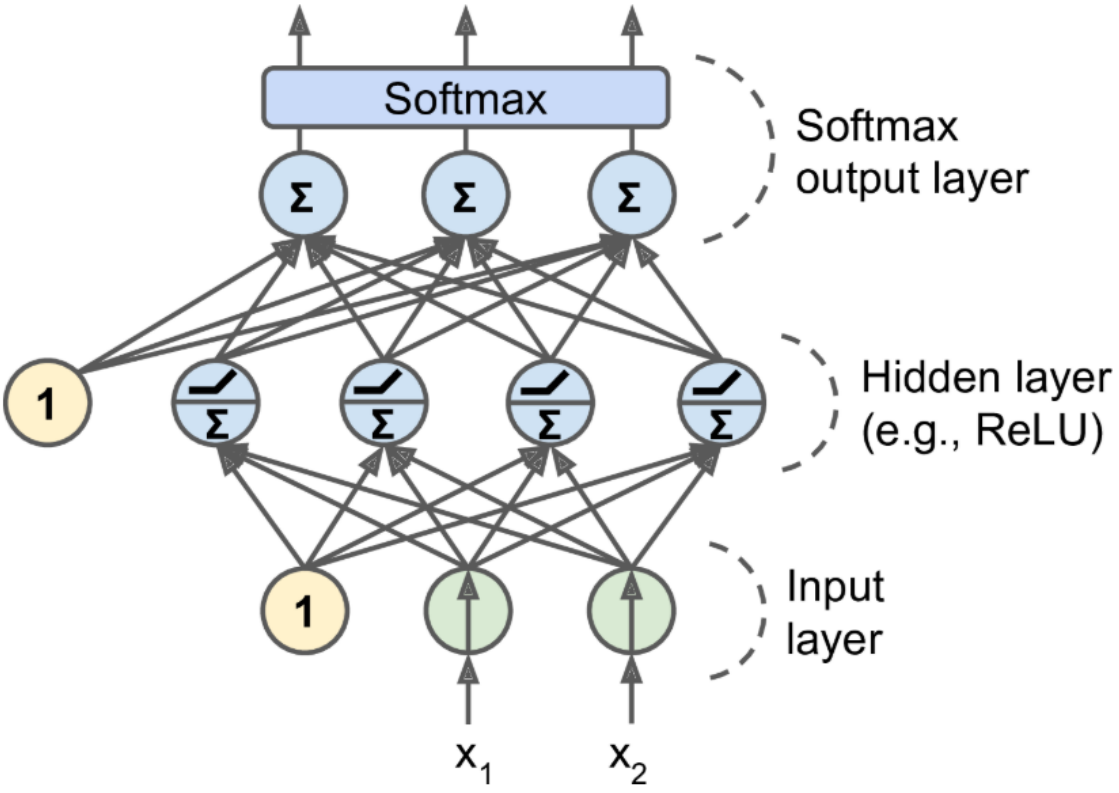



Figure 2.1: A basic neural network architecture. Picture taken from [17].

The equation for calculating the value of each node from one layer to the next is shown in

Equation 2.1, where all variables are in either matrix or vector form, with the matrix/vector dimension shown in the subscript. $Y_{N\times 1}$ is the values in each node in the subsequent layer, with N nodes. $X_{M\times 1}$ is the node values from the previous layer, with M nodes (can be either an input layer or a previous hidden layer). $W^x_{N\times M}$ is the weight matrix from one layer to the next, while $W^b_{N\times 1}$ is the weights that calculates the bias in each node.

$$Y_{N\times 1} = A(W^x_{N\times M} \cdot X_{M\times 1} + 1 \cdot W^b_{N\times 1}) \tag{2.1}$$

There are a number of different activation functions, used in the nodes of the neural network. Some commonly used activation functions are presented in Equations 2.2 - 2.4, and are a way of introducing non-linearity to an otherwise linear transformation model. The following activation functions are also shown graphically in Figure 2.2 [24] & [25].

$$\text{ReLU - Rectified Linear Unit:} \quad A(x) = max(0,x) \tag{2.2}$$

$$\text{ELU - Exponential Linear Unit:} \quad x < 0 : A(x) = \alpha(e^x - 1);\ x \geq 0 : A(x) = x \tag{2.3}$$

$$\text{Sigmoid:} \quad A(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

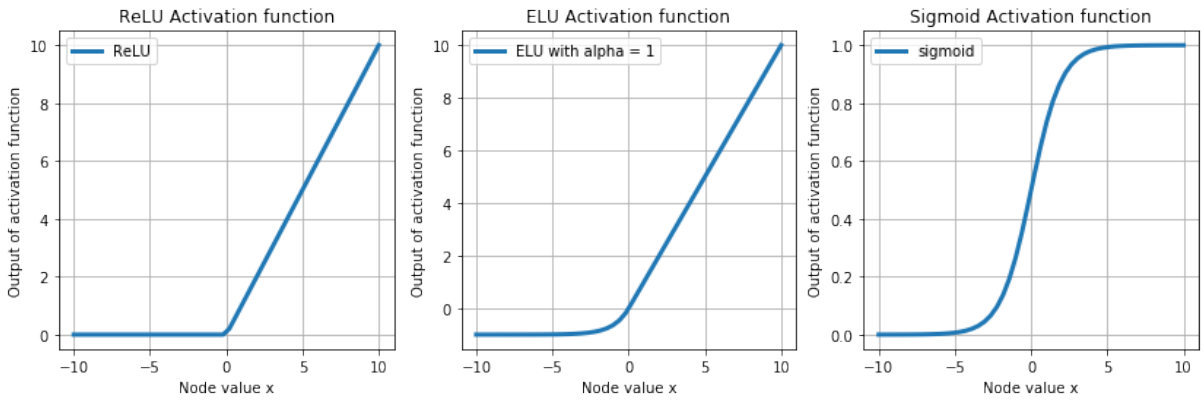



Figure 2.2: Illustration of three different activation functions.

The example model in Figure 2.1 is a basic sequential, or feed-forward model. It is sequential because the calculations occur sequentially from the input to the output layer. The neural network architecture can differ from a sequential model. For instance, there can be several neural networks working in parallel, so-called stacked neural networks [26]. There are many more architectures that will not be discussed in this report, but which are beautifully illustrated by The Asimov Institute [27].

The neural network introduced so far has been the feed-forward type, with input-layers, hidden-layers, output-layers, nodes, weights, biases and activation functions. There are

however several types of neural networks, each with either one or several strengths and weaknesses. Some of the neural network types are shown below.

- Fully Connected Neural Networks (FCNN)
- Recurrent Neural Networks (RNN)
- Convolutional Neural Networks (CNN)
- Neural Ordinary Differential Equations (NODE) [28]

The system in which data is collected, and its underlying assumptions can help to select a suitable machine learning algorithm. The recurrent neural networks have a memory state, which can store information about important events in a sequence of data. This sequence can represent time-dependent data, where each step in the sequence is a time step. Because of this memory state, the recurrent neural network will be the choice of network in this project.

## 2.4 Recurrent Neural Network Algorithms

Recurrent neural networks (RNNs) is a type of neural network that is most often used in predicting sequences of data, like stock-prices or load-demand in power systems, and with input data of arbitrary input length [17]. A RNN consists of RNN-cells, which takes input data $x_1$ for one time step at a time. The cell then calculates some outputs $y_1$ and a hidden state $h_1$. Then the next time step input $x_2$ gets sent to the RNN-cell, which will calculate the next sequence output $y_2$ while also considering data from the previous hidden state $h_1$. This process is illustrated in Figure 2.3.

### 2.4.1 The RNN-cell

Calculating the outputs and states of the RNN-cell is similar to the calculations for a neuron in the feed-forward NNs. The main difference is that there is an added term inside the activation function in Equation 2.1. This is the hidden state variables and is calculated using the previous output values of an RNN-cell and a unique set of weights $W^h_{N\times N}$. The hidden state is calculated using Equation 2.5, and the RNN-cell output is calculated using Equation 2.6. At the first time step of in a sequence the hidden state is usually set to 0 in all elements.

$$\text{The hidden state for a single cell:} \quad h^{i-1}_{N\times 1} = W^h_{N\times N} \cdot Y^{i-1}_{N\times 1} \tag{2.5}$$

$$\text{The output of a RNN-cell:} \quad Y^i_{N\times 1} = A(W^x_{N\times M} \cdot X^i_{M\times 1} + 1 \cdot W^b_{N\times 1} + h^{i-1}_{N\times 1}) \tag{2.6}$$

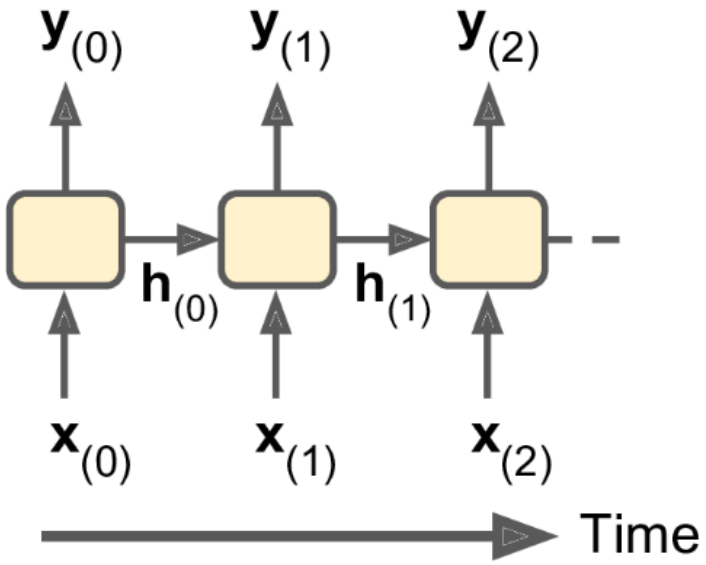



Figure 2.3: Illustration of the RNN-cell working on sequence data. Image from [17].

In this project, the whole sequence of outputs $Y$ from the RNN-cells are used when having two or more RNNs in sequence, as shown to the left in Figure 2.4. On the other hand, when the RNN transitions to a feed-forward NN, only the last output $y_n$ of the RNN-sequence are obtained, as shown on the right side in Figure 2.4.

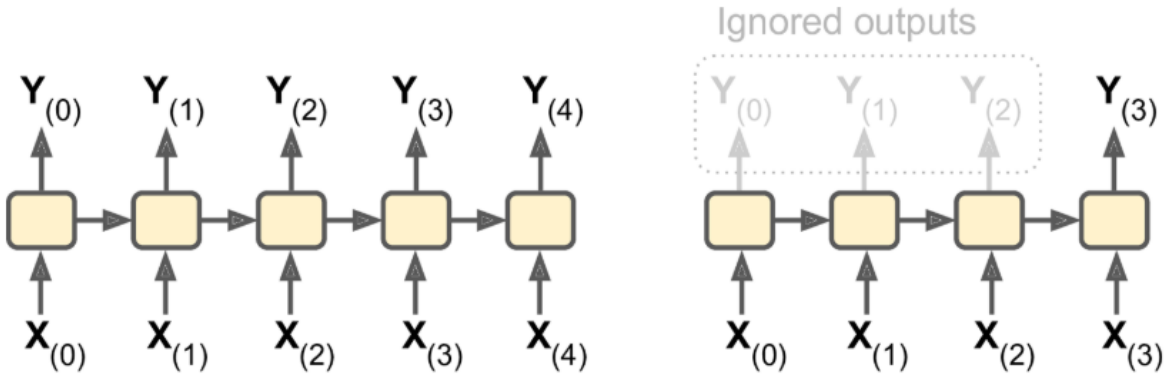



Figure 2.4: Two different methods of collecting the RNN outputs. Image from [17]

The recurrent cell enable sequential data to be processed by neural networks. The sequence can in principle be any length, but there are some challenges in regards to training RNNs with long sequences of data (more than 100 steps) [17]. These challenges relates to the optimization of the weights and biases of the neural network. When iterating through the sequential input data, the optimization algorithm should give an indication to how much the NN-parameters should be changed to improve performance. If the data has many time steps, the parameter gradients calculated will accumulate over all the

time steps until the gradient either vanishes or explodes. Lets say the input data is 100 sequences long, and the optimization algorithm calculates that each parameter values throughout the neural network be multiplied by 0.9. If this happens over all the 100 sequences, the parameter gradients at the first step will end up being $0.9^{100} = 0.0000266$. This will lead to the parameters being virtually untrained because of the vanishing gradient. On the contrary, if the gradient values are e.g. 1.1, then the gradient values in a sequence will end up being $1.1^{100} = 13780.6$. This is the so-called exploding gradient problem, and can lead to difficulties training the neural network.

There are a few remedies to this challenge, listed below [17].

- LSTM-type of RNN-cell
- GRU-type of RNN-cell
- Gradient clipping
- Initialization techniques of the weights and biases
- Use good activation functions

### 2.4.2 LSTM and GRU cells

There are special types of RNN-cells that has shown to reduce the vanishing and exploding gradients while training, called the LSTM- and GRU-cells. They work in principle the same ways as the RNN-cell, but with some additions. The "Hands-on machine learning with Scikit-learn and Tensorflow" book has done a great job illustrating and explaining the internal workings of the LSTM- and GRU-cells [17].

LSTM is an acronym for "Long Shorth Term Memory", and has its name because of how the internal states is transferred throughout the data sequence. In addition to the *hidden state* $h$, it also has a *cell state*, indicated as $c$. One way to view the LSTM states are that $h$ is short-term memory and $c$ is "long-term memory" [17]. In Figure 2.5, the LSTM-cell is shown when processing data from time step $t$. The input data $x(t)$ combines with the *hidden state* from the previous cell four times in parallel in some fully-connected neural networks (FCNN). Read from left to right, the first FCNN is combining with the cell-state from the previous node to determine what the cell-state should "forget". The second and third FCNN is determining if there is any new information worth storing in the *cell state*. The fourth FCNN, combined with the *cell state* $c$ determines the new output and *hidden state*.

The LSTM-cell has the ability to catch important features early on in a sequence, and remember that for the rest of the sequence. This is especially important in language processing where one word can change the meaning of an entire sentence [29].

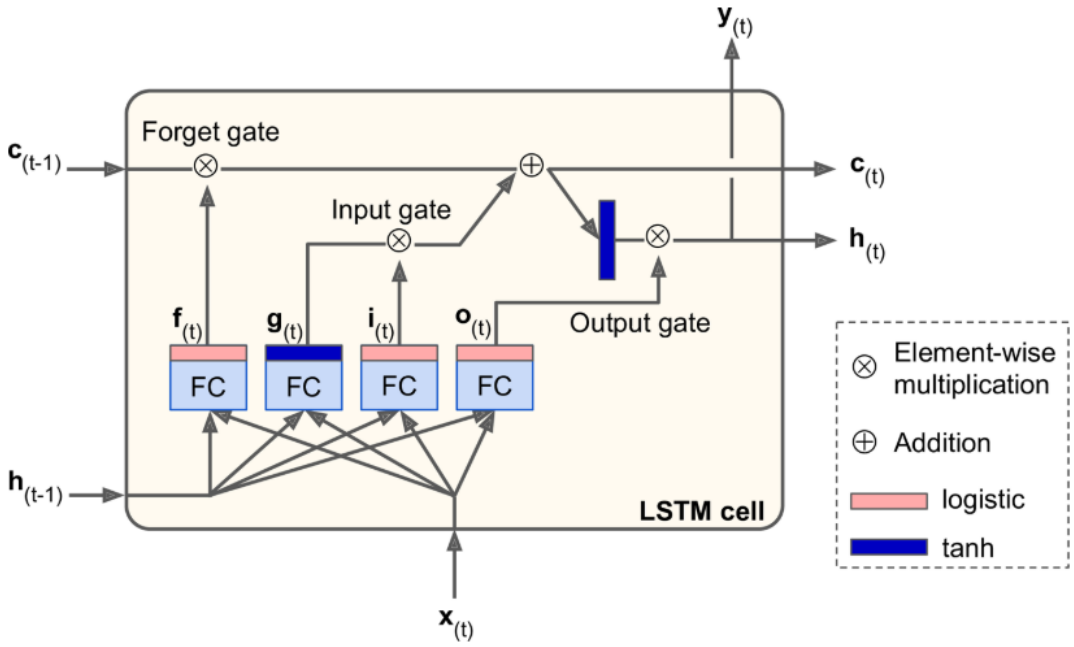



Figure 2.5: The inner structure of a LSTM-cell. Image from [17]

GRU is an acronym for "Gated Recurrent Unit", and is a different variant to the LSTM-cell. It takes the concept of "long-term memory" from the LSTM-cell, and implements it into the *hidden* state $h$ instead of having the *cell state* $c$. This makes for less computations per cell, and has shown to perform close to as well as the LSTM-cell [30]. Figure 2.6 shows the internal components of the GRU-cell. The first FCNN helps determines what "memories" from $h$ should be considered for calculating the new output. The second FCNN determines what memories should be completely forgot from the sequence, in addition to determining what new information should be allowed to pass through the third FCNN. The -1 circle means that the input array given gets inverted, meaning that all 1's becomes 0, and all 0's becomes 1.

## 2.5 Training Neural Networks

In this report, two metrics will be used for determining the performance of a neural network, namely loss and accuracy. Loss is the neural networks prediction error, and is the value that optimization algorithms minimizes during training [25]. Accuracy is the measure of how often the neural networks predicted a target correctly [25].

There are several neural network design features that can be implemented to improve the neural network metrics, and make a more generalized neural network. The methods used in this report are presented in Chapters 2.5.1 - 2.5.4.

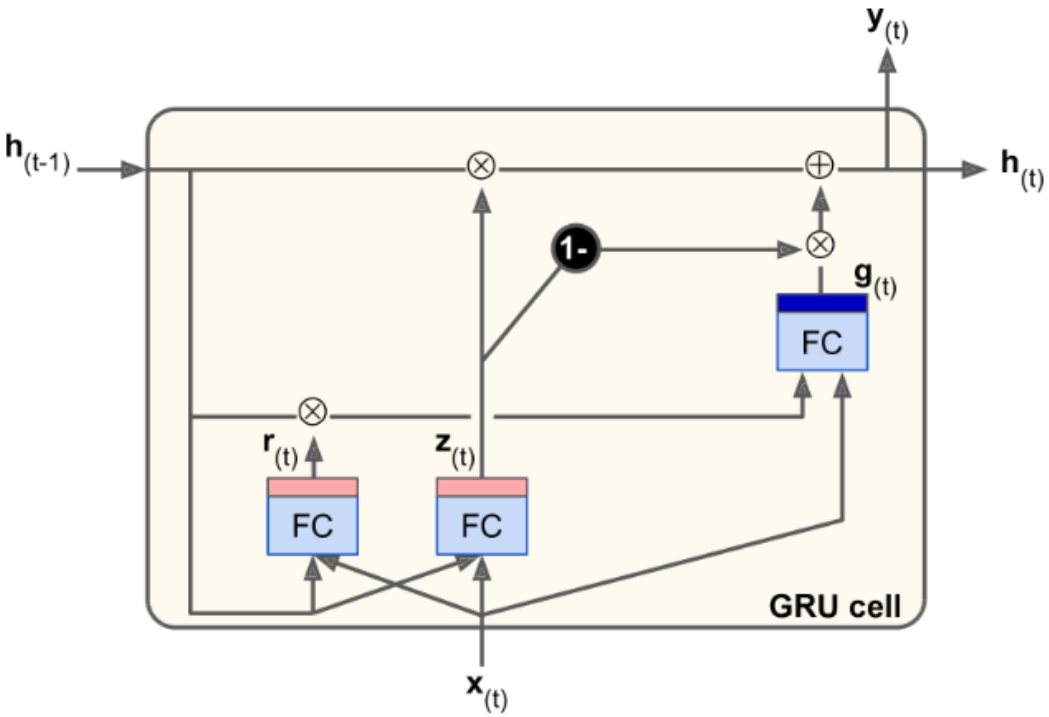



Figure 2.6: The inner structure of a GRU-cell. Image from [17]

### 2.5.1 Optimization function

Training a Neural Network (NN) is the process of tuning the NN-parameters (weights and biases) such that NN predictions matches the targets as close as possible, given some input data. The optimization algorithms objective is to find how the NN-parameters should be adjusted to reduce training loss. Because of its efficiency and ability to work with many parameters, the Adam optimization is used in this report [25].

### 2.5.2 Regularization

Overfitting is a common challenge when training a neural network. It occurs when the NN has been so accustom to the training set that it no longer generalizes to unseen data. This is a challenge because the loss don't detect overfitting. Regularization is applied to neural networks to reduce the change of overfitting. The most common regularization method is called Dropout. Dropout is active only during the training phase, and effectively deactivates a portion of the weights inside the NN. The basis for doing this is that the NN will become less reliant on only a few sequence of nodes for prediction. Often, the result will be a more generalized neural network [17].

### 2.5.3 Learning Rate

The learning rate describes how fast the neural network parameter should be tuned in the directions specified by the optimization algorithm. A high learning rate means that the NN parameters will be tuned in large steps towards the minimum. The consequence may be that the neural network parameters never converges to an optimum because it always skips the minimum. This process is excellently illustrated by [31] in Figure 2.7. A too low learning rate is also shown in the figure, together with a learning rate that is just right. In the figure, $J(\boldsymbol{\theta})$ is the prediction error for the NN estimations based on the NN model parameters, referred to as loss.

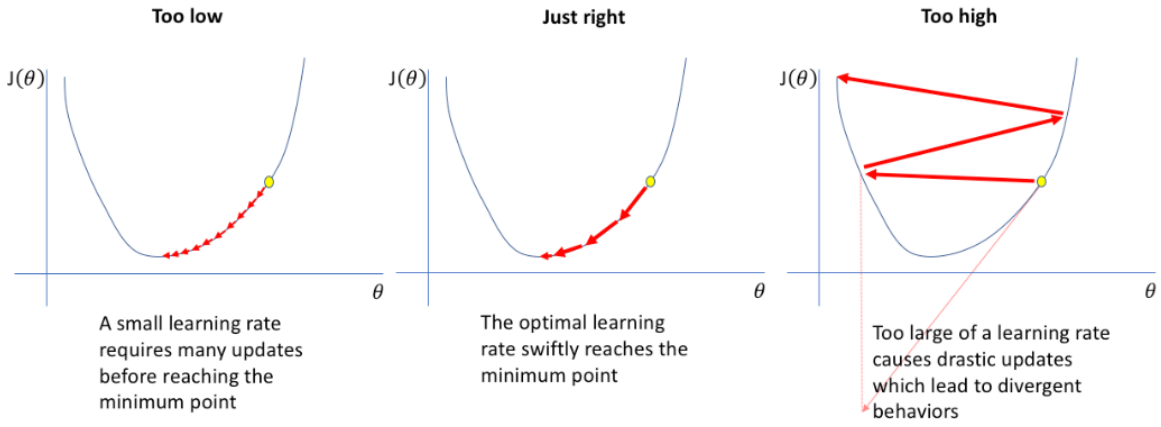



Figure 2.7: An illustration of a low (left), great (middle) and high learning rate. Image from [31]

### 2.5.4 Early Stopping

Training neural networks is computationally heavy, and can take a long time. In the Python module Keras, a specified amount of epochs must be defined before training begins. EarlyStopping is a function that stops the training of a neural network if a specified metric doesn't improve after a specified number of epochs. As an example, Early stopping can be set to monitor training loss, with a patience of 10. This means that the training stops if the training loss doesn't improve after 10 epochs.

## 2.6 Neural network hyperparameters

The neural network architecture is an important factor for deciding how suitable a neural network model is for a given task. The list below are some of the factors that defines the NN architecture.

- Number of layers
- Number of nodes in any layers
- The activation function used for each layer
- Optimization function
- Dropout rate
- Learning rate

Determining the hyperparameters that perform well can be difficult and computationally heavy [32]. There are strategies for finding good hyperparameters. A simple method is to use uninformed random search, which is random guessing of the hyperparameters for several iterations in the hope of hitting a good architecture.

# 3 Thermal physics related to synchronous generators

When designing a synchronous machine, there are several branches of physics that needs careful considerations. The thermal physics are important because it can model how temperatures changes in the generator. This is important because high temperature rises may have the risk of considerable reduction of lifetime on the electrical insulation and other equipment [13]. The thermal design is also determining the maximum steady-state output power from the generator, as it is the thermal properties that determine the steady state temperatures, and the thermal time constants. In rotating electrical machines there are several components generating heat, and with the high power density of machines today, natural heat dissipation is not sufficient. Measures have to be taken to ensure that the heat sources is sufficiently cooled down during operation of the generator.

While heat is generated in several parts of the generator model, heat is also transferred throughout other parts of the generator. Modeling the heat generation and heat transfer mechanisms requires an understanding of the basic physics of thermal energy balances, heat transfers and the heat generation mechanisms. This chapter will focus on the physics required to make a simple mechanistic thermal model of the generator. The mechanistic model will then be presented in Chapter 4.

## 3.1 Thermal Conductance

If an object, whatever the material or size, has a higher temperature than the surrounding, this object will cool down over time due to some heat transfer. The entropy of the universe will always increase, asserting a never-ending effort to even out the temperatures over time, described by the second law of thermodynamics [33]. Heat transfer is the mechanism of transporting energy stored as heat in one object to the surroundings, and adjacent objects and medium with lower temperatures. This phenomena is mathematically described by Fourier's Law of heat conduction, shown in Equation 3.1. The $\dot{Q}$ is the transferred heat in Joules per second, or Watts $[W]$. k is the materials ability to transport heat from one place to another, formally called thermal conductivity in $[W/m/^{\circ}K]$. A is the crossectional

surface area between the temperature gradient [$m^2$], while $\frac{dT}{dx}$ is the temperature gradient in the x-direction. [$^\circ K$/m]

$$\dot{Q} = -k \cdot A \cdot \frac{dT}{dx} \tag{3.1}$$

Some assumptions can be made to make this formula simpler to work with. One can assume that the temperature of an object is homogeneously distributed, meaning that the temperature in any subsystem will be homogeneous. Another assumption is that the boundary walls between two sub systems have infinitesimally small width, such that the surface between two systems are responsible for causing the heat conduction, and not the conductivity and length. With these assumptions, Equation 3.1 can be simplified to merge the length, area and heat conductance into one term, $hA$. These simplification is implemented in Equation 3.2 $\dot{Q}_{A2B}$ expresses how much heat is flowing through the surface from subsystem A to B, while $hA_{A2B}$ is the thermal conductance between the surface of subsystem A and B [$W/^\circ K$], and $T_A$ and $T_B$ is the temperatures in the two subsystems [13][8].

$$\dot{Q}_{A2B} = hA_{A2B}(T_A - T_B) \tag{3.2}$$

This assumption is leads to the so-called lumped parameter modeling, where a system is divided into discrete subsystems with homogeneous temperature distributions. Using this assumption, the heat flow in/out of a subsystem can be calculated using only temperature differences and the thermal conductance as described in Equation 3.2. It is common in literature to describe the thermal transfer capability as thermal resistance instead of thermal conductance. Thermal resistance is the inverse of thermal conductance, and will make no difference is the results of the calculations. The thermal conductance is a useful tool for modeling and describing different heat transfer phenomenons, such as conduction, convection and radiation [13][6].

## 3.2 Heat Transfer mechanisms

When heat flows from one system to another, there are several physical aspects that contributes to the heat transfer. These heat transfer mechanisms are modeled similarly, but are governed by different physical phenomenons. The list below is the three most basic heat transfer mechanisms used for calculating heat transfer between two systems. These are heat transfer through:

- Conduction
- Convection
- Radiation

### 3.2.1 Thermal Conduction

Heat transfer through conduction happens when two solid objects are in contact with each other. The contact surface transports heat from one object to the other when there is a temperature gradient. There are two phenomenons that causes heat flow by conduction. There is the transition between adjacent molecules kinetic energy through lattice vibrations, that causes sections of higher kinetic energy (higher temperatures) to transfer some kinetic energy to regions of lower kinetic energy (lower temperatures). The second phenomenon of heat transfer through conduction is through free electrons in the material. The first phenomenon can occur in any materials, regardless of state (solid, liquid, gas), but the second phenomenon occurs where there is free electrons in the materials. This is usually in metals, and is a good explanation of why metals that is a good electrical conductor is also a good thermal conductor. There is of course exceptions to this, as electrically insulating materials can be good thermal conductors, such as oxidised metals or diamonds [13]. Modeling heat conduction can be done using Equation 3.2 assuming a lumped parameter model.

### 3.2.2 Convection

Heat transfer through convection is the phenomena where heat is transported from/to a solid object through adjacent flow of a fluid (gas or liquid). There are two types of convection, namely natural and forced convection. Natural convection occurs when fluid adjacent to a warm object gets heated, and the heated fluid gets displaced by colder fluid because of buoyancy [34]. Forced convection forces the motion of fluid past an object, cooling it down more efficiently. This is because the local fluid has less time to heat up, and therefore causes a continuously higher temperature difference between the fluid and the solid object [13].

Heat transfer through convection can be estimated the same way as with conduction, shown in Equation 3.2. However, when using lumped parameter modeling with conduction, the thermal conductor $hA$ is a function of the peripheral fluid velocity of a solid object [13].

### 3.2.3 Heat loss from radiation

Temperature is described as the average kinetic energy of particles. Heat loss through radiation means that an object experiences heat loss by emitting electromagnetic waves (light). With higher energy levels, the electrons in the material has a high kinetic energy. The electron usually ”wants” to reach a lower energy state in an atom, and therefore it releases its kinetic energy through photons. When the material has a high emissivity (close to 1), this phenomenon occurs often, while a low emissivity means this happens

more rarely. All things hotter than the surrounding temperature releases heat through radiation. The thermal conductivity for radiation between two surfaces can be calculated using Equation 3.3, where $\sigma$ ($5.67 \cdot 10^{-8}$ $[W/m^2/K^4]$) is the Stefan-Boltzmann constant, the $\varepsilon$ is the surfaces emissivity and F is the view factor which describes how perpendicularly faced the two areas are to each other [6]. The heat loss from radiation can be calculated using Equation 3.2 if the heat conductance is replaced with the radiation conductance in Equation 3.3.

$$hA_{rad} = \frac{\sigma \cdot \varepsilon \cdot F_{1,2} \cdot (T_1^4 - T_2^4)}{T_1 - T_2} \tag{3.3}$$

## 3.3 Lumped Thermal Capacitance

So far, the heat transfer in and out of a subsystem has been explained for different physical phenomenons. The lumped capacitance method is a way of relating the change in temperature to the mass of the subsystem and the heat flow in and out. For the lumped capacitance method to be valid, the thermal conductivity must be much smaller than the thermal capacity, referred to as the Biot's number [35]. The validity of the lumped capacitance model is dependent on a low Biot's number ($Bi \ll 1$), which is assumed in this project to be the case.

Lumped capacitance method is a thermal representation of the dynamics of the voltages/currents in electrical capacitors. Voltage ($V$) is analogous to temperature difference ($\Delta T$), capacitance ($C_{el}$) is analogous to thermal capacitance ($C_{th}$) and current ($I$) is analogous to heat flow ($\dot{Q}$). Equation 3.4 is the electrical expression for voltage change in a capacitor (assuming no resistance), and Equation 3.5 is the thermal counterpart [35]. The heat capacitance $C_{th}$ can be calculated as the product of the objects mass and its specific heat capacity.

$$\text{Voltage over a capacitor:} \quad C_{el} \cdot \frac{dV}{dt} = I \tag{3.4}$$

$$\text{Temperature in a thermal capacitance:} \quad C_{th} \cdot \frac{d\Delta T}{dt} = \dot{Q} \tag{3.5}$$

### 3.3.1 Thermal time constant

The thermal time constant is a measure of how fast the temperature in a system changes. It is measured as the time it takes the temperature change from steady state to reach 63.2 % of its next steady state value [36]. The time constant does not have a direct impact on how much the temperatures will change in a step response. In an RC-thermal circuit, the time constant $\tau$ is the time constant to the heat capacity temperature, and can be

calculated using Equation 3.6 [8]. Notice that the thermal time constant is independent of the adjacent temperatures in the system, because it is a property of the system, and not a consequence of variables such as heat flow or temperatures. However, one should be careful to use this equation in systems where adjacent temperatures change over time. Equation 3.6 assumes that the adjacent temperatures are constants.

$$\tau = \frac{C}{hA} \tag{3.6}$$

## 3.4 Thermal energy balance

The lumped capacitance model is a useful tool for modeling the temperature inside a subsystem based on a lumped parameter model. A more general form is the thermal energy balance, which will help calculate the cooling air temperature of the generator as it heats the generator metal parts. Assuming that the temperature is evenly distributed in a volume, and that the volume and pressure stays constant, the temperature change can be described using thermal energy balance. The thermal energy balance can be simplified to Equation 3.7, based on the assumptions taken. In the equation, $\rho$ is density, $V$ is volume, $\dot{H}$ is enthalpy flow, $\dot{Q}$ is heat transfer or heat production in/out of the system and $\dot{W}$ is the mechanical work converted into heat (e.g. friction) [37].

$$\rho V \frac{dT}{dt} = \dot{H}_{in} - \dot{H}_{out} + \dot{Q} + \dot{W} \tag{3.7}$$

Enthalpy flow is the energy flow into or out of a system carried by mass, such as any fluid. If a hot fluid enters a system and the same amount of fluid exits the system with a lower temperature, the system gains thermal energy from enthalpy flow. The enthalpy flow can be expressed using Equation 3.8, where $\dot{m}$ is the mass flow, $\hat{c_p}$ is the specific heat capacity of the fluid, and the total mass balance in the system is zero.

$$\dot{H} = \hat{c_p} \cdot \dot{m} \cdot T \tag{3.8}$$

### 3.4.1 Steady state thermal energy balance

If the system is dominated by the effect of convective heat transfer, such as in the air gap of the generator, one can assume that the enthalpy into the system plus added thermal energy equals enthalpy out of the system. This is a so-called steady-state model. This is usually a reasonable assumption where the thermal time constant in a system is very low. Equation 3.9 shows the steady-state form of the thermal energy balance [37].

$$\dot{H}_{out} = \dot{H}_{in} + \dot{Q} + \dot{W} \tag{3.9}$$

## 3.5 Heat generation in a synchronous generator

High currents, rotation and changing magnetic fields all contributes to losses and heat generation on the synchronous machine. Optimizing the generator efficiency has several advantages, as it leads to less wasted energy, and less heat generated as losses in the generator. Literature categorizes generator losses in four groups [13][6]:

- Copper losses
- Iron losses
- Mechanical and windage losses
- Stray/additional losses

### 3.5.1 Copper losses

Copper losses is the resistive power loss in the stator and rotor windings in the generator. These losses are usually easy to predict, as one to only know the current ($I$) and resistance ($R$) through the conductor to calculate the power losses according to Equation 3.10 [6]. It should be said that the resistance is dependent on temperature, but will be assumed constant for this report.

$$P_{loss} = R \cdot I^2 \tag{3.10}$$

### 3.5.2 Iron losses

The iron losses is mainly present in the iron core at the stator side of the generator. Iron losses comes mainly from hysteresis and eddy-currents in the core , caused by the changing magnetic field in the iron. Both of these factors are a function of frequency [13]. This loss is known to be difficult to calculate precisely, but since the operational speed should be constant, the iron losses are assumed constant in this report.

### 3.5.3 Mechanical losses

The mechanical losses are caused mainly by the friction in the bearings of the generator. There is in addition some windage losses in the rotor blades and cooling fan. These contributions prove difficult to estimate analytically, but is mainly affected by the rotational speed of the rotor [13]. Since the rotor should always be rotating at constant speed in normal operation, the mechanical and windage losses can be approximated to a constant value.

### 3.5.4 Stray losses

The stray losses can be calculated by estimating the difference between the lost energy in the generator during operation, and subtracting all losses discussed so far. The discrepancy between the power in and out are called the stray losses and may be caused e.g. losses in the damper bars across rotor laminations, eddy current losses in the stator conductors and several other aspects of both the magnetic and electrical circuits [6].

# 4 Mechanistic thermal models

As discussed in Chapter 3, lumped parameter models can be a great tool for modeling of thermal behaviour. The theory presented in Chapter 3 will serve as a basis for defining the mechanistic thermal model of the generator, represented as a lumped parameter thermal network (LPTN). There will be two different case studies in this report. Study case 1 is for the thermal heat exchanger model in Chapter 7. Study case 2 is for the mechanistic thermal generator model, without the heat exchanger in Chapter 8. This means that two mechanistic thermal models must be defined, and this is done in this chapter. The generator model will be based on much of the work done by [8], with some simplifications. The heat exchanger will be based on equations developed for a counter-current heat exchanger in [37].

## 4.1 Generator structure

The mechanistic thermal generator model must be able to capture the basic dynamics of the generator, with a basis of the physical generator structure. Figure 4.1 shows a sketch of the generator sliced in the middle by the vertical axis. In the figure, the cold air cools down the stator iron and rotor copper, heating up the air. The hot air gets cooled down via the heat exchanger, and the cold air circulates back into the generator.

From Figure 4.1, the mechanistic thermal model requirements can be listed. These requirements serve as both an overview of the expected behaviour of the model, and thereby gives solid testing criterias when testing the model with inputs.

- The generator model assumes normal operation at all times while predicting temperatures.
- *Cold air* is coming out of the heat exchanger.
- *Cold air* is cooling down *rotor copper* and *stator iron*.
- The *heated cooling air* is leaving the generator and going into the heat exchanger.
- The heat exchanger is cooling down the *hot air* via *cooling water*.
- The *stator copper* is cooled through the *stator iron*.

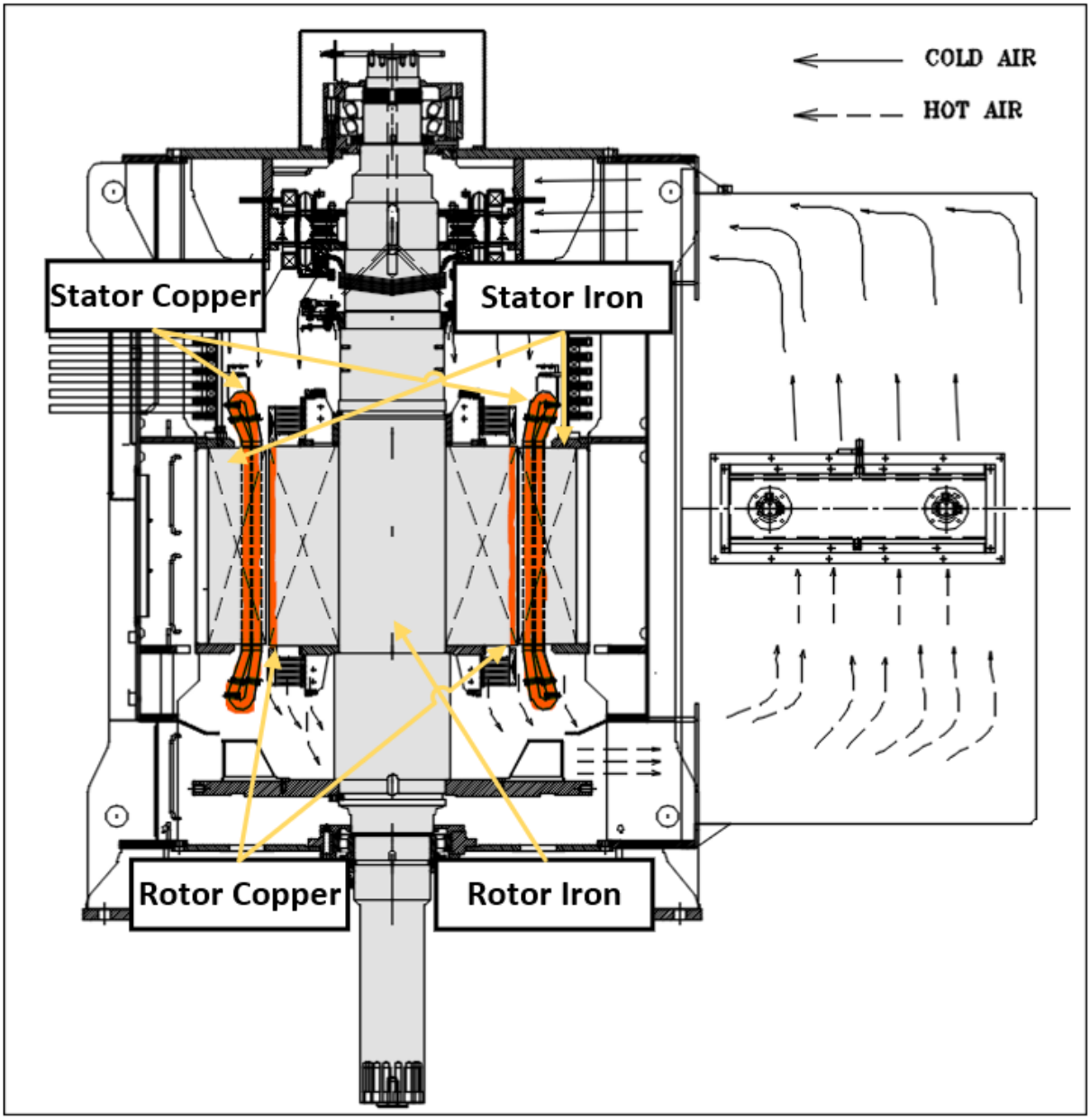



Figure 4.1: A basic sketch of the generator cooling method. Figure modified from the Manual from Grunnaais generator.

- The *rotor iron* is thermally isolated from the *rotor copper*.
- The generator metal part will be modeled as one lumped heat capacitor each.
- There is no volume work or change in pressure.
- Electrical resistances is assumed to be constant and independent on temperatures.

- The model will only be valid for generator in normal operating condition.
- The stray losses is not considered.
- The mechanical losses is set to a constant, and a parameter $ML-const$ is deciding how much of the mechanical losses that contributes to heating the *cooling air*. The mechanical loss value is obtained from the heat run test.
- The iron losses is considered constant, and its value is taken from the generator heat run test.

Based on these requirements, the LPTN-model is made and illustrated using Simulink in Figure 4.2. In the figure, the three modeled metal parts contains a thermal capacitance and a current source, representing the heat capacitance and heat generation. Each metal systems has a voltage measurement units which monitors the respective voltage (temperature). Color coding is noting which symbols are model parameter (dark red), model input (green) and model output (cyan). More detailed information about the model input, output, parameters and constants is presented in Chapter 4.3. Note that the model was not programmed in Simulink or Matlab as displayed, but in Python, and the figure is only an illustration of the LPTN.

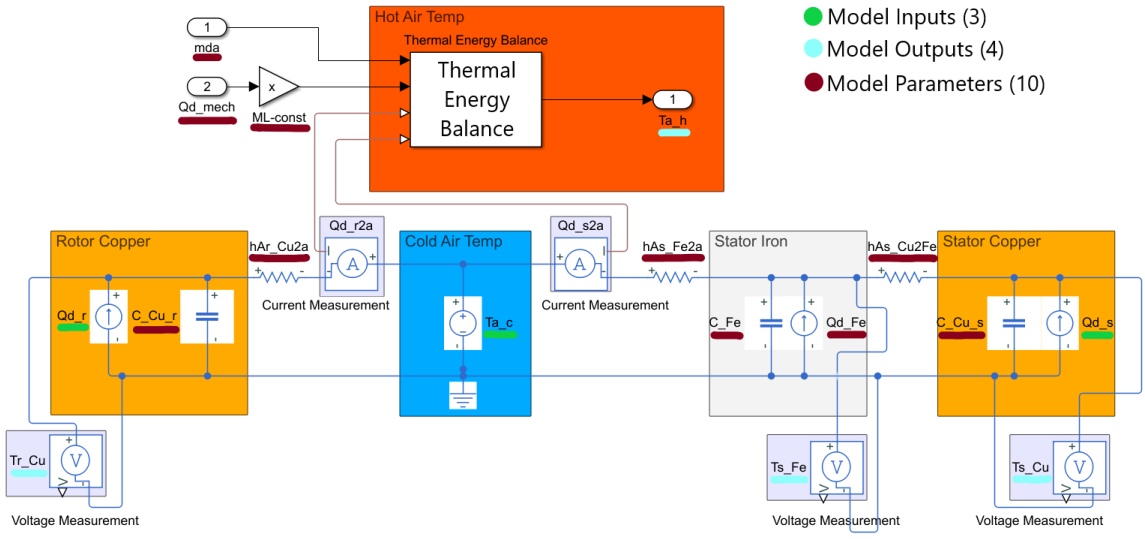



Figure 4.2: Illustration of the lumped parameter thermal network of the generator.

### 4.1.1 Thermal Measurements in the generator

At Grunnaais 12 MVA synchronous generator there are in total four different parts that contain temperature measurements. The stator copper temperature is measured by a

PT100 stator resistance thermometer from Technocontrols. The cold water, and cold and hot air temperatures are measured by a PT100 element from Wika [38]. The PT100 element uses the property that material resistance is dependent on the temperature. Instead of measuring temperature "directly" the probe measures the current through the element. Since the resistance in the element changes, so does the current, assuming that the voltage is constant. Therefore a current measurement can be extrapolated to a temperature measurement [39].

## 4.2 Heat Exchanger mechanistic thermal model

Heat exchangers are devices which transport heat from one flowing medium to another without having the fluids mix [40]. In the generator, cooling air is flowing in a closed loop as illustrated in Figure 4.1, and the heat exchanger is a water to air type. Heated air ($T_h^a$) is cooled by cold water through the high surface area in the heat exchanger, and the cold air ($T_c^a$) then gets pushed back into the generator by rotor fans, closing the loop. The heat exchanger provides a region of high surface area between the air and the water to maximize the transferred heat from the air to the cooling water.

The heat exchanger model is an algebraic function that expresses the cold air temperature by giving it the hot air temperature and cold water temperature as inputs. The model has a few underlying assumptions:

- The Heat exchanger model assumes normal operation at all times
- The heat exchanger is of type counter-current [37].
- The mass balance is in steady state at all times (mass in = mass out)
- There is no heat loss through friction in the heat exchanger
- There is no change in volume or pressure in the heat exchanger
- The equation is based on no internal heat stored in the heat exchanger, which means $\sum \dot{H}_{in} = \sum \dot{H}_{out}$, where $\dot{H}$ is enthalpy flow.

The last assumption effectively means that any change in either of the input temperatures (hot air or cold water) will instantly affect the output temperatures. This is not an unreasonable assumption, since the air is flowing through the generator at rather high speed, and the thermal time constant is small compared to the thermal time constants in the generator. The equations for the specified heat exchanger are shown in Equation 4.1 - 4.3 [37], and its graphical representation of the model parameters, $[uA_x, \dot{m}_a, \dot{m}_w]$, is shown in Figure 4.3. In [37], the heat exchanger model consists of two algebraic equations, one describing the cold air temperature, and the other describing the hot water temperature. The hot water temperature equation is not used in this project, but is illustrated in Figure

4.3. The used variables are written in in blue, while the unused temperature is written in red. The model parameters for the heat exchanger are:

- $uA_x$: Heat transfer coefficient between the air and water $[W/K]$
- $\dot{m}_a$: Mass flow rate of air through the heat exchanger $[kg/s]$
- $\dot{m}_w$: Mass flow rate of water through the heat exchanger $[kg/s]$

$$\text{Air temperature out:} \quad T_c^a = \frac{(\alpha_a - \alpha_w)T_h^a - \alpha_w(e^{(\alpha_w - \alpha_a)} - 1)T_c^w}{(\alpha_w - \alpha_a e^{(\alpha_a - \alpha_w)})} \tag{4.1}$$

$$\text{Constant 1:} \quad \alpha_a = \frac{uA_x}{\hat{c_{p,a}}\dot{m}_a} \tag{4.2}$$

$$\text{Constant 2:} \quad \alpha_w = \frac{uA_x}{\hat{c_{p,w}}\dot{m}_w} \tag{4.3}$$

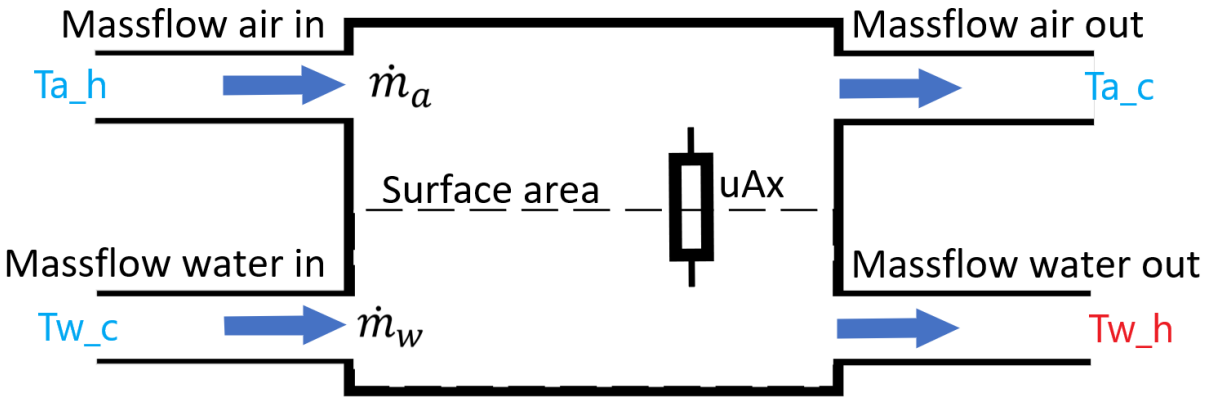



Figure 4.3: A basic sketch of the heat exchanger, and its model parameters.

## 4.3 Generator thermal mechanistic model

The LPTN model presented in Figure 4.2 will be programmed in Python. The model will be a first order model, governed by three differential equations, one for each metal part. These differential equations are derived from the thermal energy balance discussed in Chapter 3.4. This model and approach of modeling the generator is inspired by [8].

### 4.3.1 Generator input variables

The input variables are the rotor copper losses, the stator copper losses and the cold air temperature out of the heat exchanger. The copper losses are a direct consequence of the current through the conductors. These losses will be calculated based on current measurements and resistances from the heat run test of the generator. In Chapter 6 the losses will be estimated based on data from the generator. The input variables is shown in Table 4.1 with its associated symbol.

Table 4.1: Input variable to the mechanistic thermal generator model.

| Symbol | Unit | Description |
|---|---|---|
| $T_c^a$ | $^\circ C$ | Cold cooling air entering the generator, coming from the output of the heat exchanger. |
| $\dot{Q}_r$ | W | The calculated copper losses in rotor copper that is contributing to heating the copper metal. |
| $\dot{Q}_s$ | W | The calculated copper losses in stator copper that is contributing to heating the copper metal. |

### 4.3.2 Generator model outputs

The outputs of the generator model is the variables that gets affected by the inputs, and that are of interest when simulating the model. The generator model should estimate the temperatures in the metal parts and the hot air out of the generator. Two out of four model output variables are measured, which can be an asset to determine if the model parameters are tuned correctly or not. The four model outputs are listed in Table 4.3.2 together with their associated symbol and unit.

Table 4.2: Model outputs

| Symbol | Unit | Description |
|---|---|---|
| $T_h^a$ | $^\circ C$ | Hot air temperature leaving the generator and entering the heat exchanger. |
| $T_{Cu}^r$ | $^\circ C$ | Average rotor copper temperature. |
| $T_{Fe}^s$ | $^\circ C$ | Stator iron temperature |
| $T_{Cu}^s$ | $^\circ C$ | Stator copper temperature |

### 4.3.3 Generator model parameters and constants

Several values in the generator model equations don't change over time. Some of these values are governed by the generator design, type and ratings, and are called parameters. Some values are constants based on physical properties, and will not change regardless of the generator type. Parameters and constants in this project are assumed constant values, unchanged by the environment. Some of these constants or parameters may in reality be dependent on properties such as temperature, pressure etc. Table 4.3.3 lists the generator model parameters, together with the associated symbol and unit. In addition, for later clarification, it lists which parameter should be estimated in Chapter 8, and which parameters are known beforehand. Table 4.3.3 lists the model constants used for the defined generator model [41].

Table 4.3: Generator model outputs

| Symbol | Unit | Known | Description |
|---|---|---|---|
| $hA^r_{Cu2a}$ | $\frac{W}{^\circ K}$ | No | Thermal conductivity from rotor copper to air |
| $hA^s_{Cu2Fe}$ | $\frac{W}{^\circ K}$ | No | Thermal conductivity from stator iron to stator copper. |
| $hA^s_{Fe2a}$ | $\frac{W}{^\circ K}$ | No | Thermal conductivity from stator iron to air. |
| $C^r_{Cu}$ | kg | Yes | Rotor copper mass. |
| $C^s_{Fe}$ | kg | No | Stator iron mass. |
| $C^s_{Cu}$ | kg | Yes | Stator copper mass. |
| $ML-factor$ | None | No | The percentage of how much mechanical loss is contributing to heating the cooling air. |
| $\dot{m}_a$ | $\frac{kg}{s}$ | No | Mass flow of cooling air through the generator |
| $\dot{Q}_{Fe}$ | W | Yes | Stator Iron losses |
| $\dot{Q}_{mech}$ | W | Yes | Mechanical losses |

Table 4.4: Generator model constants [41]

| Symbol | Value | Unit | Description |
|---|---|---|---|
| $\hat{c_{p,Cu}}$ | 385 | $\frac{J}{kg \cdot K}$ | Specific heat capacity of copper. |
| $\hat{c_{p,Fe}}$ | 449 | $\frac{J}{kg \cdot K}$ | Specific heat capacity of iron. |
| $\hat{c_{p,a}}$ | 1005 | $\frac{J}{kg \cdot K}$ | Specific heat capacity of dry air at 1 atmosphere pressure. |
| $\hat{c_{p,w}}$ | 4200 | $\frac{J}{kg \cdot K}$ | Specific heat capacity of water. |

### 4.3.4 Rotor copper modeling

The rotor is heated up by copper losses $\dot{Q}_r$ caused by the magnetization current ($I_m$). Cooling of the rotor is done by the cold air ($T^a_c$) passing through the rotor air gaps. The

rate at which the rotor copper temperature changes is described by Equation 4.4.

$$\text{Change in rotor copper temperature:} \quad \frac{dT_{Cu}^{r}}{dt} = \frac{\dot{Q}_r - \dot{Q}_{r2a}}{C_{Cu}^{r}} \tag{4.4}$$

$$\text{Heat from rotor copper to cooling air:} \quad \dot{Q}_{r2a} = hA_{r2a}(T_{Cu}^{r} - T_c^{a}) \tag{4.5}$$

### 4.3.5 Stator copper modeling

The stator copper temperature is dependent on the copper losses in the stator windings and the heat flow from the stator copper to stator iron core. The rate at which the stator copper temperature changes are given in Equation 4.6.

$$\text{Change in stator copper temperature:} \quad \frac{dT_{Cu}^{s}}{dt} = \frac{\dot{Q}_s - \dot{Q}_{Cu2Fe}}{C_{Cu}^{s}} \tag{4.6}$$

$$\text{Heat from stator copper to stator iron:} \quad \dot{Q}_{Cu2Fe} = hA_{Cu2Fe}^{s}(T_{Cu}^{s} - T_{Fe}^{s}) \tag{4.7}$$

### 4.3.6 Stator iron modeling

The stator iron is cooled down by the cooling air flowing through the generator. Heating is caused by iron losses and heat from the stator copper. The rate at which the stator iron temperature changes is given in Equation 4.8.

$$\text{Change in stator copper temperature:} \quad \frac{dT_{Fe}^{s}}{dt} = \frac{\dot{Q}_{Fe} + \dot{Q}_{Cu2Fe} - \dot{Q}_{Fe2a}}{C_{Fe}^{s}} \tag{4.8}$$

$$\text{Heat from stator iron to cooling air:} \quad \dot{Q}_{Fe2a} = hA_{Fe2a}^{s}(T_{Fe}^{s} - T_c^{a}) \tag{4.9}$$

## 4.4 Tests of model requirements

Mathematical expressions have been defined for the heat exchanger and the generator model. To verify that they work as intended, some small tests will be done to check that both models fulfill their defined requirements. Appendix C shows the Python code for defining and testing the heat exchanger model, and Appendix D shows the Python code for defining and testing the generator model, where both input step responses and sensitivity analysis will be done. In this chapter, only the input step responses will be tested as it is sufficient to confirm the model requirements.

### 4.4.1 Step changes in inputs of heat exchanger model

The heat exchanger is a relatively simple system, with two inputs and one output. There is no dynamics in the system, and therefore a changes in inputs will instantly affect the output. The model step response will be displayed when one of the inputs are changing and the other remains constant. In Figure 4.4 (a) and (b), the first 60 time steps remains constant because no input variables are changing, with cold water temperature set to 10 $°C$ and hot air temperature set to 50 $°C$. After the 60'th time step, each figure shows seven different step responses of one of the input variables, where the new input value is labeled in each figure. Figure 4.4 (a) shows the step responses from water temperature changes and Figure 4.4 (b) shows the step responses from hot air temperature change. While simulating these step responses, the model parameters were set to constant values throughout the test, with values $[\mu Ax, \dot{m}_a, \dot{m}_w] = [10000, 10, 10]$.

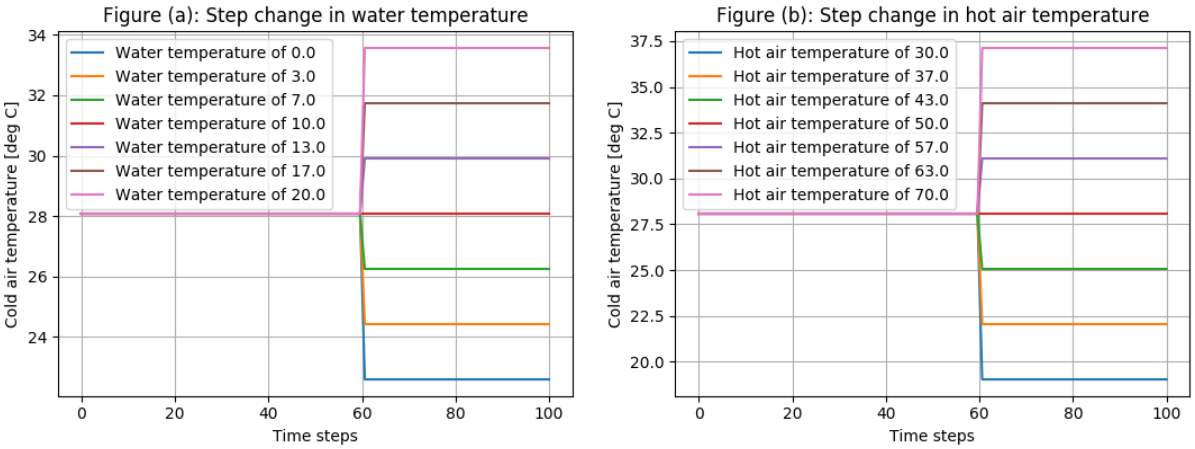



Figure 4.4: Heat exchanger responses to step changes in the input variables.

### 4.4.2 Step changes in inputs of the generator thermal model

The generator model is a dynamic system, where there are three inputs and four outputs. The step responses of the model are verified based on the requirements in Chapter 4.1. A step change in the cooling air temperature should affect all four outputs from the model. A step change in rotor copper losses should only affect the rotor temperature and the hot air temperature. A step change in stator copper losses should affect the stator copper, stator iron and the hot air temperature. This is what is shown in Figure 4.5, where figure (a) shows response from a step in the cold air temperature, figure (b) shows response from a step change in rotor copper loss, and figure (c) shows the response from a step change in stator copper loss. The test case is designed such that the first 10 minutes of

the simulation reaches steady state. After 10 minutes, step changes in inputs occurs. The model parameters used for this simulation is listed in Table 4.4.2.

Table 4.5: Generator model outputs

| Symbol | Value |
|---|---|
| $hA^r_{Cu2a}$ | 3000 |
| $hA^s_{Cu2Fe}$ | 7000 |
| $hA^s_{Fe2a}$ | 5000 |
| $C^r_{Cu}$ | 200000 |
| $C^s_{Fe}$ | 700000 |
| $C^s_{Cu}$ | 500000 |
| $ML-factor$ | 0.5 |
| $\dot{m}_a$ | 10 |
| $\dot{Q}_{Fe}$ | 37888 |
| $\dot{Q}_{mech}$ | 53696 |

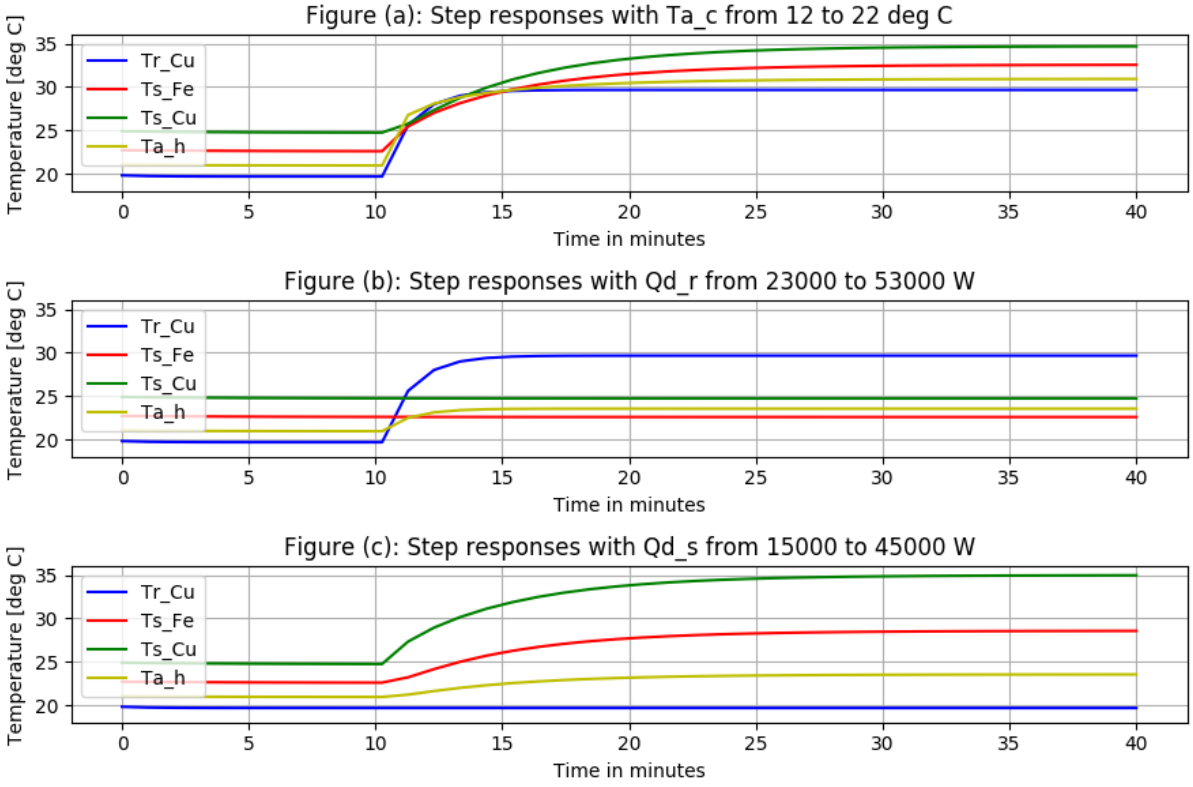



Figure 4.5: Generator temperature responses to step changes in the input variables.

## 4.5 Evaluate the prediction error

Limitations of mathematical modeling has been long understood, and even though computing power today is many times more that only a few decades ago, the limitations of modeling are still just as valid today. From [42], an operational definition of a model is: *"a useful, practical description of a real-world problem, capable of providing systematic mathematical predictions of selected properties"*. Since one cannot hope to obtain a perfect model, a metric must be defined to indicate how well the model performs compared to real data. Given some inputs, the model should produce some outputs. The model outputs can be compared with real measurement values, and this comparison happens in an error function. Figure 4.6 illustrates the concept of a squared prediction error function, where the error is calculated as the squared difference between model output and measurement. The error function used for quantifying the time-series prediction error is through the

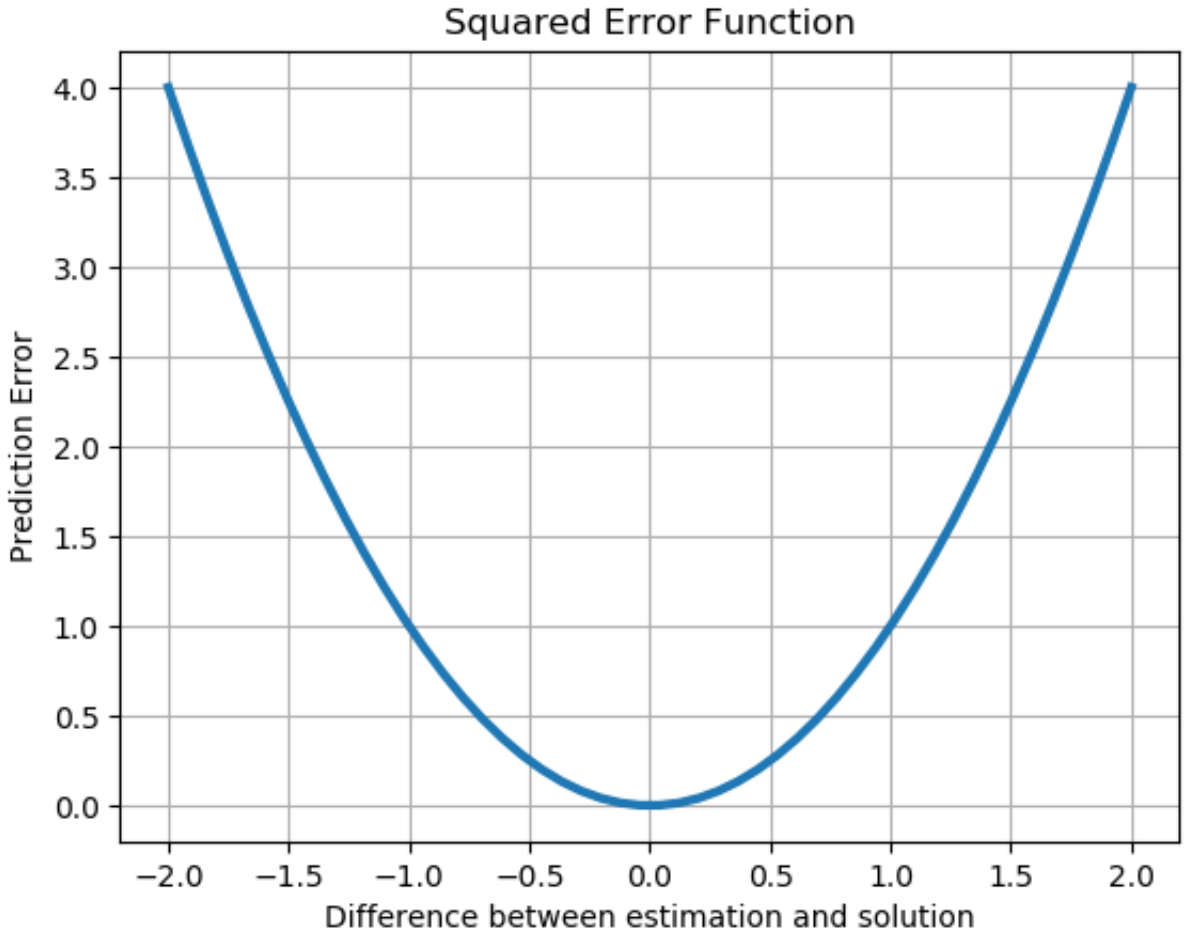



Figure 4.6: Illustration of the squared prediction error function.

mean of the squared prediction errors each time step, so-called Mean Squared Prediction Error (MSPE) [43]. There exists several error functions, each with its own use, with the squared error being the most used one. The MSPE will be the metric for evaluating model performance in this report. Some other alternatives is the absolute error, absolute error with saturation and squared error with dead-zone, illustrated by [44]. These error functions will not be discussed in this report.

# 5 Data Acquisition and Preparation

One of the most influential factors for the success of supervised learning algorithms is the amount of quality data available for training. Bad quality or non-representative data used for training usually leads to the algorithm performing poorly during implementation. Figure 5.1 shows a general relationship between the amount of data and test accuracy, with several different ML algorithms. This graph implies that a small data set leads to poorer accuracy that with more data. It also implies that large quantities of data can saturate the accuracy, meaning that more data won't do anymore good for performance [17]. Although the figure shows the data in the context of natural language processing, the same principle is valid for other types of ML tasks.

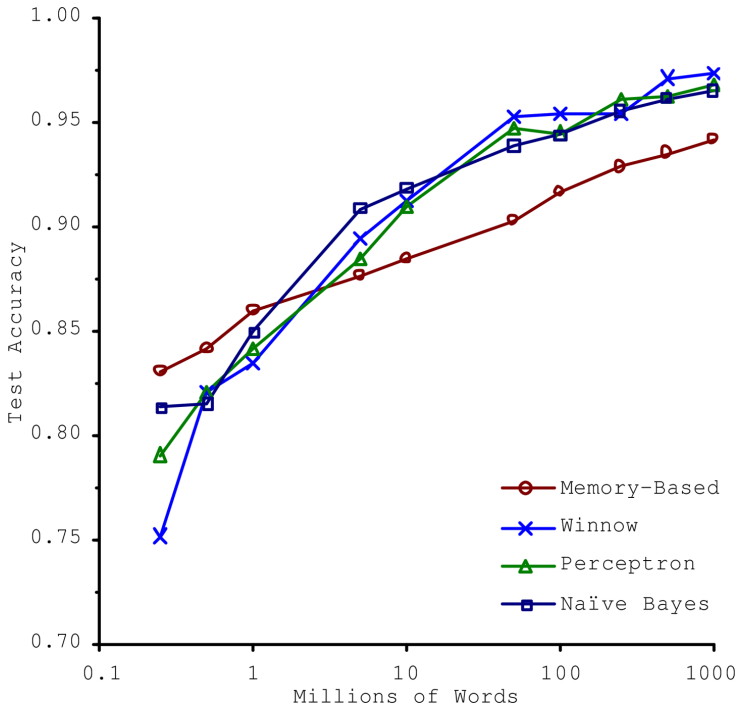



Figure 5.1: The importance of data vs algorithm. Picture taken from [17]

By courtesy of Skagerak Energi, this project got the permission to use operational data from Grunnaais 12 MVA brushless, synchronous generator. The main focus of this chapter is to extract operational data from Azure Databricks, and filter out unwanted data. This data will be used as model inputs and parts of the training data for the neural networks.

## 5.1 Data storage and SCADA

SCADA is an acronym for "Supervisory Control and Data Acquisition", and is used for controlling and monitoring industrial processes [45]. In this case, this system is used for monitoring and controlling Grunnaais 12 MVA generator. The monitored SCADA-data are sent to Azure Databricks, and stored in Databricks, which is a cloud-based storage system. With permission from Skagerak Energi, this data contributes to parameter estimation in this project.

### 5.1.1 Data type and storage format

The storage file format is avro, which is commonly used to store timestamped data values. Avro-format is a binary, open sourced serialization format, and works well with json, which will be used to decode the files [46]. From Grunnaai, data is collected in 1-hour batches, and then collected and converted into a single avro-file. The avro-files are stored in separate folders, structured in a hierarchy of: year → month → day → hour → avro-file. Each hour folder contains a single avro-file with data over a 1-hour period.

The avro-files used to store SCADA-data has three attributes. It has an ID, value and timestamp, illustrated with some fictive values in Table 5.1. Therefore, this file format can store many different measurement values with several unique ID's, collected at many different times.

Table 5.1: Illustration of the data stored in an avro-file.

| ID | Value | Time stamp |
|---|---|---|
| Data1 | 1 | 2019-11-22T12:03:55.333000 |
| Data2 | 31 | 2019-11-22T12:10:22.740000 |
| Data3 | -340 | 2019-11-22T12:10:23.210000 |
| ... | ... | ... |

The time stamp values have the format YYYY-MM-DDTHH:MM:SS.xxxxxx, where:

- YYYY - Year stamp
- MM - Month stamp
- DD - Day stamp
- T indicates that the following sequence is time
- HH - Hour stamp
- MM - Minute stamp

- SS - Second stamp
- xxxxxx - Microsecond stamp

## 5.2 Collecting and reading the avro-files

Although storing the SCADA-data in hourly batches might be a good system for long-term storage, hourly batches will not be sufficient for viewing the thermal responses during model parameter estimation of the generator. In addition, going through each avro-file separately is an inefficient process when filtering out bad data, such as nonoperational generator data. To collect the data, the first step is to collect it into monthly batches. Then the data will be filtered for nonoperational data.

Appendix E1 shows the Python notebook used for extracting data from Azure Databricks. The steps for achieving this is listed below.

1. Define the folder that avro-files should be collected from in Azure Datalake. This can be defined as for instance a month folder, or a day folder. A Python function then automatically collects all avro-file paths in that specified folder, ordered by date. The function returns a Python list of all the avro-file paths in that folder.
2. Define what IDs/tags should be extracted from the avro-files while decoding them. Skagerak Energi provided with a tag list that could be used to find the relevant variables for this project. However, this list is confidential and cannot be shown in this thesis.
3. Iterate through the Python list of avro file paths. Each iteration will decode a single avro-file using the pyspark.sql module in Python, and add the values and time stamps of each specified ID to a Python dictionary. When several avro-files are read in succession, the values and time stamps will be appended to this dictionary, making ordered lists of values and time stamps for each variable.
4. The Python dictionary containing all decoded data will now be converted into a Pandas DataFrame. However, before this can be done, variables must have the same time axis and the same time span. This is not the case for the majority of the variables because of the irregular data sampling from SCADA. This is solved by removing the microsecond stamp on all variables and filling all data with a ”NaN”-value for each second where there are no data points. By this, all variables have a time step of one second, where all time steps without any value is filled with NaN. This means that all variables can be moved to a common time axis. This conversion is illustrated in Figure 5.2, where the left table illustrates the raw data set from the avro-files, and the right table is the new data structure in a Pandas DataFrame, as a result of these steps.

Data structure originally

| | Timestamp (YYYY-MM-DD HH:MM:SS.µs) | Value |
|---|---|---|
| Variable 1 | 2020-01-11 00:03:44.928000 | 10 |
| | 2020-01-11 00:03:47.888000 | 12 |
| | 2020-01-11 00:03:50.270000 | 13 |
| | 2020-01-11 00:03:52.125000 | 14 |
| Variable 2 | 2020-01-11 00:03:45.928000 | 23 |
| | 2020-01-11 00:03:48.888000 | 22 |
| | 2020-01-11 00:03:49.270000 | 21 |
| | 2020-01-11 00:03:51.125000 | 20 |
| ... | | |

Formatted data frame

| Timestamp (YYYY-MM-DD HH:MM:SS.µs) | Var 1 | Var 2 | ... |
|---|---|---|---|
| 2020-01-11 00:03:44 | 10 | NaN | |
| 2020-01-11 00:03:45 | NaN | 23 | |
| 2020-01-11 00:03:46 | NaN | NaN | |
| 2020-01-11 00:03:47 | 12 | NaN | |
| 2020-01-11 00:03:48 | NaN | 22 | |
| 2020-01-11 00:03:49 | NaN | 21 | |
| 2020-01-11 00:03:50 | 13 | NaN | |
| 2020-01-11 00:03:51 | NaN | 20 | |
| 2020-01-11 00:03:52 | 14 | NaN | |

Figure 5.2: Illustration of converting raw avro-data to a Pandas DataFrame format.

## 5.3 Storing extracted data in batches

The process described in Chapter 5.2 can be used to collect data in any specified length of time, being for instance one hour, three day, one month etc.. The data being collected spans from 15. October 2019 to 14. February 2020, in batches of specified lengths as listed i Table 5.2. The batches are divided such because the avro-files didn't always contain the data in the expected format, or there simply was no avro-files stored in some dates. Sometimes, in a few avro-files, some of the variables had single data points far away from the expected time span. This occurred if the SCADA-system was restarted at any point, and if a variable hadn't been logged for very long. The SCADA-system, as start-up initialization, locates the previous logged value of all variable, and makes these the initial data points. For some variables, that initial data point may be several days ago, or in some cases months ago. The fix for this was to remove all data points before the expected time frame, before using the defined Python methods to collect the data as usual. The extracted variables are listed below. These variables will be processed in Chapter 6, but are for now just extracted and collected into batches.

- Stator current L1 [A]
- Stator current L2 [A]
- Stator current L3 [A]
- Excitation current in stator [A]
- Active Power [MW]
- Reactive Power [Mvar]
- Rotor speed, in relation to nominal speed [%]

Table 5.2: Overview of the batches of collected data, by date.

| Batch number | Data collection date |
|---|---|
| Batch 1 | October 13th to 31th 2019 |
| Batch 2 | November 8th to 19th 2019 |
| Batch 3 | November 20th to 30th 2019 |
| Batch 4 | December 1th to 16th 2019 |
| Batch 5 | December 17th to 31th 2019 |
| Batch 6 | January 1th to 10th 2020 |
| Batch 7 | January 11th to 20th 2020 |
| Batch 8 | February 16th to 29th 2020 |

- Cold cooling air temperature [°C]
- Hot cooling air temperature [°C]
- Cold cooling water temperature [°C]
- Stator winding temperature L1 [°C]
- Stator winding temperature L2 [°C]
- Stator winding temperature L3 [°C]

## 5.4 Selecting and filtering data

In the batches listed in Table 5.2, there are often several places where the Grunnaais generator temporarily stops. However, data is usually still being collected during these time spans and must therefore be removed. The reason for this is that the assumptions made in Chapter 4 requires the generator to be operative for the models to be valid.

Python notebook that are used for filtering out non-operational data is shown in Appendix E2. The approach taken for accomplishing this is to plot rotational speed for the full batch of data. Such a plot can be seen in the upper part of Figure 5.3. All times that the speed deviates significantly from 100 % should not be included when preparing the fully collected data set, illustrated in the lower part of Figure 5.3.

The next step is to split all good data into 4-hour samples from all batches in Table 5.2. When data is filtered for a given month, all the 4-hour batches will be collected into one data frame and saved for later processing. In Figure 5.4, all the time interval containing ”good data” are showed on a timeline from October 2019 to February 2020, together with the number of 4-hour data batches. The resulting files are one file for each month containing the 4-hour batches, namely the months of October, November and December

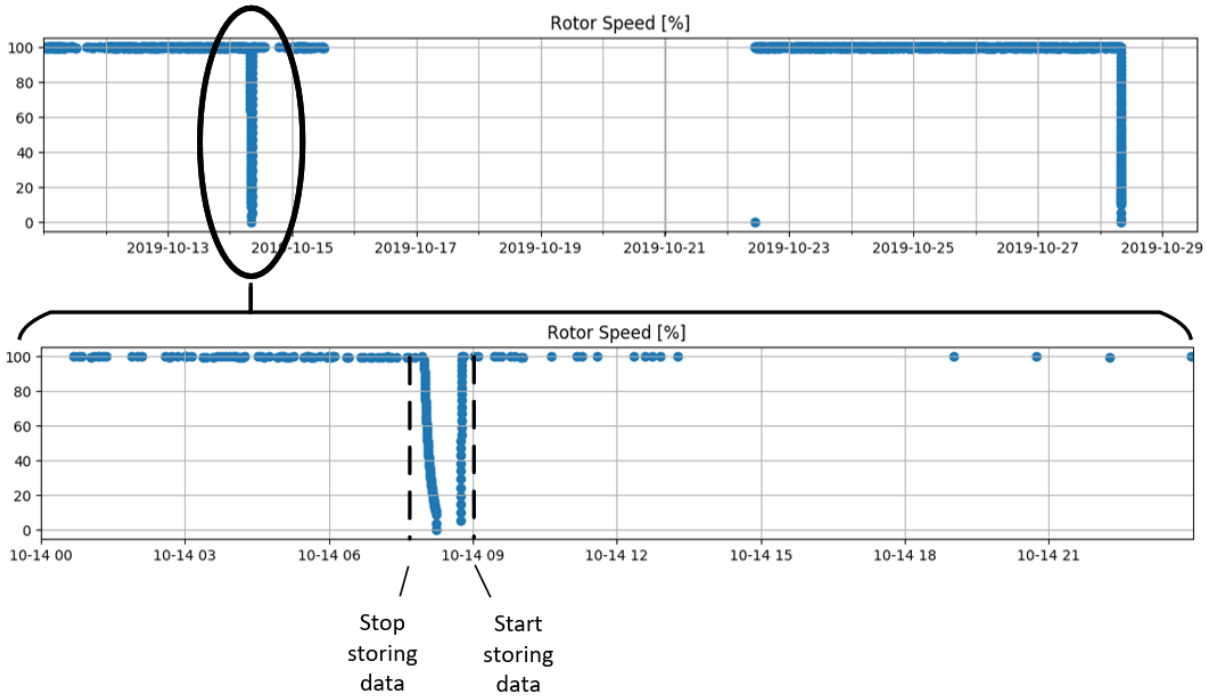



Figure 5.3: Illustrates the process of marking data that shouldn't be included in the data set

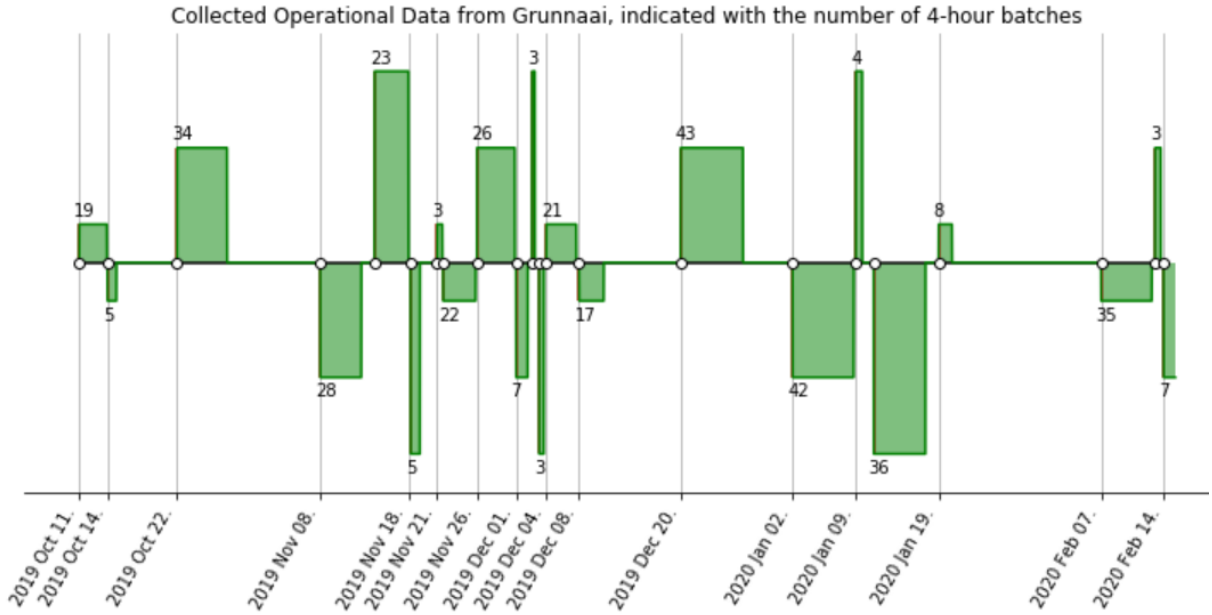



Figure 5.4: Visualizes the collected data on the timeline, with the number of 4-hour batches.

2019, and January and February 2020. The next chapter will process these data batches further.

### 5.4.1 Interpolating data

At the start of Chapter 5.4 the SCADA-data had a time step throughout the data of 1 second. However, the data points with no original value were set to NaN. For the mechanistic models and neural network to work with the data, values must be present at every time step. By using interpolation, a curve drawn through every registered data point can be made. Figure 5.5 shows the difference between the raw SCADA-data and the interpolated data. Before the data is split into 4-hour samples, all batches gets interpolated to fill inn the NaN values. The interpolation method used in this project is "pchip", implemented and done through the Pandas module in Python. The Pandas interpolates using Scipys interpolation algorithms, which is documented in [47] and [48]. After the data is interpolated, the monthly batches are stored as csv-files for later preparation, presented in Chapter 6.

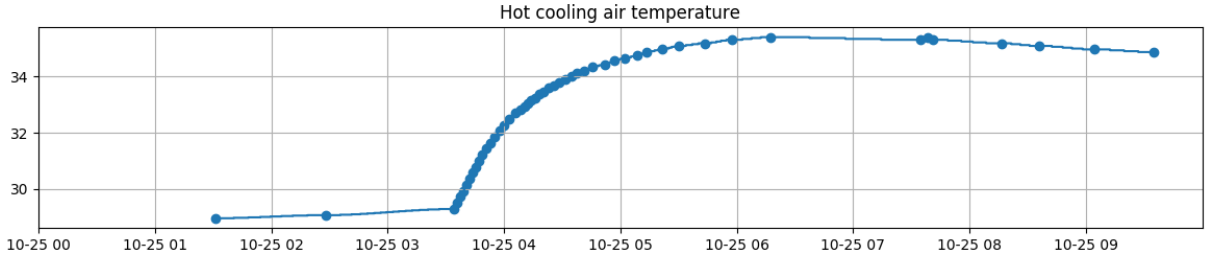



Figure 5.5: Shows how the interpolation fills in the missing values in a way that is convincingly realistic.

# 6 Prepare and generate training data

In Chapter 5, five months worth of SCADA-data was collected, interpolated, filtered and sliced into 4-hour batches. In this chapter, this data will be prepared such that it can be used by the mechanistic models and neural networks. This is an essential step towards creating the neural network training data for both the thermal heat exchanger and thermal generator model. This chapter is split into three parts: Collecting and preparing the SCADA-data; Generate the heat exchanger training data; Generate the generator model training data.

## 6.1 Preparing and collecting data

The reason that the data must be prepared further is because currently, data is not compatible with the generator model. The generator model are defined such as it needs the copper losses generated in both the rotor $\dot{Q}_r$ and stator $\dot{Q}_s$ as inputs in order to work. These variables can be estimated by obtaining the generator resistances in rotor and stator.

The copper losses in any conductor can be calculated using ohms law. Calculating the copper losses in rotor and stator is however slightly different, since there are one winding in the rotor and three winding sets in the stator. The equations for the copper losses are shown in Equation 6.1 and 6.2, where $I$ is the current through the winding, and R is the ohmic resistance in the winding.

$$\text{Rotor copper losses:} \quad \dot{Q}_r = I^2 \cdot R \tag{6.1}$$

$$\text{Stator copper losses:} \quad \dot{Q}_s = 3 \cdot I^2 \cdot R \tag{6.2}$$

Both current and resistance in rotor and stator are required for estimation the copper losses, but none of them are given directly. From the SCADA-data, the currents that are obtained are not the correct currents to use in these two equations. The collected rotor current is the excitation current ($I_m$), which is present on the stator-side. This current is what induces a voltage in the rotor, that in turn enable for a magnetization current ($I_F$) in the rotor. The collected stator current is not the stator armature current ($I_a$)

directly, but rather the line currents at the secondary side of the generator transformer ($I_{a2}$). These currents in the context of the physical system are shown in Figure 6.1. Before calculating the rotor and stator copper losses, these two currents needs to be converted into the appropriate current values $I_F$ and $I_a$.

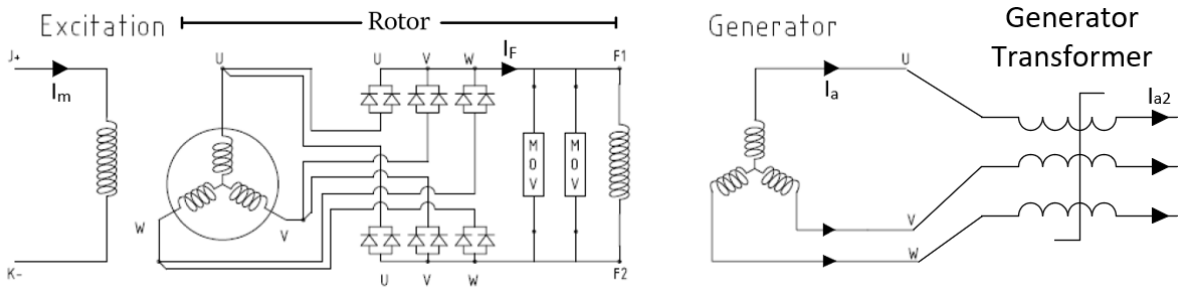



Figure 6.1: The currents in the contexts of the generator and transformer circuits.

### 6.1.1 Rotor copper loss estimation

For the copper loss to be estimated correctly, the excitation current values must be converted into the magnetization current values before using Equation 6.1. The heat run test contains a table where both exciter and magnetization current are given. These values are shown in Figure 6.2, together with the estimated magnetization current calculated using Equation 6.3. The assumption is that the magnetization current scales linearly for all excitation currents. The scaling factor from $I_m$ to $I_F$ is calculated by taking the average value of all $I_F$ / $I_m$ ratios given in 6.2. The resulting averaged ratio were calculated 47.0376.

$$I_F = 47.0376 \cdot I_m \tag{6.3}$$

The rotor resistance is given in the heat run test to be $0.30937\Omega$. This is verified by plotting an estimated rotor copper loss using the estimated magnetization current $I_F$, against the actual values given in the heat run test. The estimated and given copper losses are identical. The Python code for these plots and calculations are given in Appendix E3.

### 6.1.2 Stator copper losses

From SCADA, the secondary generator transformer currents are given in all three phases L1, L2 and L3. For the purpose of this project, these currents are averaged to one current. This averaged current will from now on be referred to as $I_2^{avg}$. Since the transformer ratio is

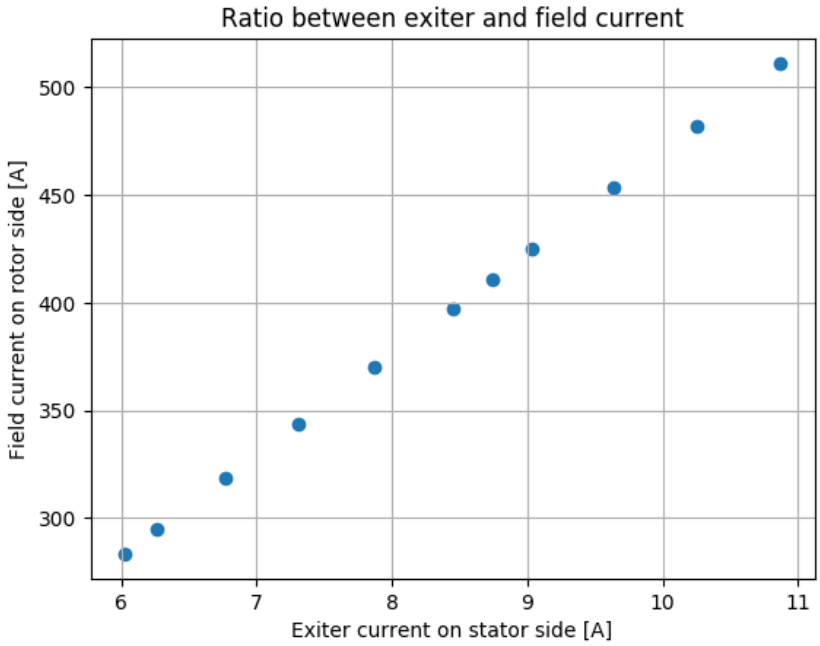



Figure 6.2: Relationship between the exciter current and field current.

not known in this project, the armature current is calculated based on the apparent power production in the generator at any time, named $S$. The armature currents in the generator is assumed to be inversely proportional to the apparent power from the generator since the voltage is stepped up. This means that the relationship between power and current can be used with rated values to calculate the armature current, given the apparent power. Since the apparent power is known at all times, Equation 6.4 can estimate the average armature current $I_1^{avg}$ in the generator. This assumes that the power into the transformer is equal to the power out of the transformer, with no losses.

$$\frac{I_1^{avg}}{I_{rated}} = \frac{S_{rated}}{\sqrt{P_g^2 + Q_g^2}} \tag{6.4}$$

Calculating the stator copper losses requires the armature winding resistances. Since the stator copper losses at nominal load is given in the heat run test, together with the armature currents, this resistance can be calculated using Equation 6.2. The resistance in one winding is calculated to $0.0143\Omega$.

### 6.1.3 Collecting the data

The final step of preparing the SCADA-data is to calculate the rotor and stator copper losses, and save the 4-hour batches from all the months as one DataFrame. All the

collected variables used in the rest of this report are listed below.

- Cold cooling air temperature $T_c^a$ [°C]
- Hot cooling air temperature $T_h^a$ [°C]
- Cold cooling water temperature $T_c^w$ [°C]
- Average stator winding temperature $T_{Cu}^r$ [°C]
- Rotor copper losses $\dot{Q}_r$ [W]
- Stator copper losses $\dot{Q}_s$ [W]

The final preparation to the data set is to down-sample the data to contain one sample per minute instead of one sample per second. Pandas resample function was used for this operation [49]. The operation took values from 60 data points, and calculated the average value of these to make a single point representing the variable value at one time step. This was done throughout the data set, and resulted in a sufficient and representative down sampled version of the data set.

## 6.2 Heat exchanger training data

This section is dedicated to generate neural network training data for the heat exchanger in Chapter 7. The objective of the neural network is to achieve good parameter estimation for the mechanistic model given a set of real temperature measurement data from SCADA. To get there, the heat exchanger must first learn how the model parameters affect the input and output temperatures in the heat exchanger model. Therefore, training data should represent the heat exchanger behaviour in different operation scenarios, such that the neural network learns the model behaviour for different inputs and model parameters.

The approach for generating training data for the heat exchanger is to collect a set of model inputs to the mechanistic model from the prepared SCADA-data. Then, the heat exchanger model will estimate the cold air temperature, based on the given inputs and randomly selected model parameters in the given ranges in Table 6.1. The training data for the neural network will then be stored such that the input and output variables are the neural network input features, and the model parameters will be the targets. This process is illustrated in Figure 6.3.

The RNN input features given to the neural network will be structure as a three dimensional array in the Python module Numpy, as shape (A, B, C). A is the number of total training sets that will be generated, B is the number of time steps in each training set. And C is the number of variables in each time step. The heat exchanger case has three variables, namely $T_c^w$, $T_h^a$ and $T_c^a$. The number of time steps have been set to 30 in this project, meaning that all training data samples should contain 30 minutes of data. Since

Table 6.1: Boundaries for the heat exchanger parameter values.

| Parameter | Minimum | Maximum |
|---|---|---|
| $uA_x$ | 1000 | 100000 |
| $\dot{m}_a$ | 5 | 100 |
| $\dot{m}_w$ | 5 | 100 |

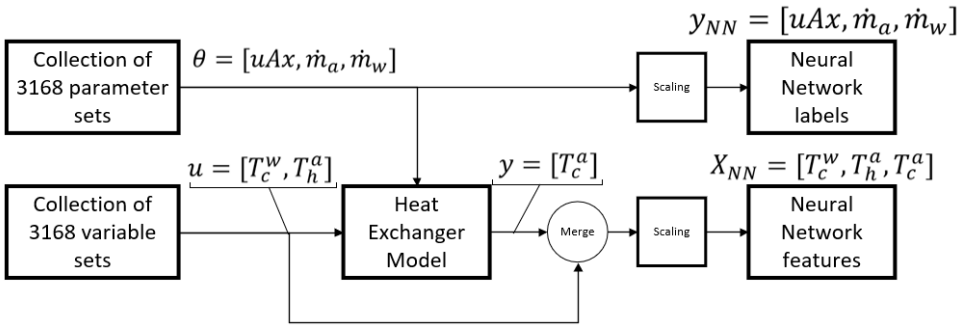



Figure 6.3: Overview of collecting training data to the neural network.

the input data comes in 4-hour batches, each batch is split in eight equally sized sets, making the prepared data set containing 3168 batches of 30-minute data. More than 3168 training sets are needed to make the neural network learn and generalize. Therefore, all 3168 batches will make training data with random model parameters, looping 50 times to create 158400 training sets. In Appendix E4, the Python code is shown for generating the heat exchanger training data.

## 6.3 Generator model training data

As with the heat exchanger, training data must be made for the generator model. The process for making the training data is very similar to the heat exchanger process, with a few differences. These differences are listed below:

- There are six model parameters plus three initial temperatures that are classified as model parameters for the generator model. The range of values for the randomly generated model parameters are shown in Table 6.2.
- The input variables to the mechanistic model are $T_c^a$, $\dot{Q}_r$ and $\dot{Q}_s$.
- The output variables from the generator model are rotor copper temperature $T_{Cu}^r$, stator copper temperature $T_{Cu}^s$, stator iron temperature $T_{Fe}^s$ and hot air temperature $T_h^a$.

- All the model inputs and outputs will be collected as training data. Variables in the training data will later be reduced to match the real SCADA measurements. This is discussed in chapter Chapter 8. For now, all the model inputs and outputs are collected as training features.

Table 6.2: Boundaries for the randomly generated generator model parameter

| Parameter | Minimum | Maximum |
|---|---|---|
| $hA^r_{Cu2a}$ | 200 | 10000 |
| $hA^s_{Cu2Fe}$ | 200 | 10000 |
| $hA^s_{Fe2a}$ | 200 | 10000 |
| $M^s_{Fe}$ | 1500 | 70000 |
| $ML_{const}$ | 0.0 | 1.0 |
| $\dot{m}_a$ | 1 | 50 |
| $T^r_{Cu0}$ | 10 | 70 |
| $T^s_{Fe0}$ | 10 | 40 |
| $T^s_{Cu0}$ | 10 | 70 |

The collected training data will contain seven input features. The generator model is also dynamic, in that the metal temperatures don't change instantly, rather they change over time. The thermal time constants of the generator model might span from several minutes up to hours, depending on the parameter values, and therefore the training data will have 4-hour lengths. However, instead of having a time step of 1 minute, which would result in a total of 240 data points, the time step is reduced to once every third minute. This reduces the number of time steps to 80 steps per batch, which is easier to train [17].

All 396 batches was iterated through 550 times with random parameter values, making a training set of 217800 training sets. Generating this training data took approximately 6 hours of computing power, and the Python code for generating the training data is shown in Appendix E7.

In some situations, the combinations of input values and parameter values generated unrealistic temperatures in the model output. Training sets containing temperatures either above 120 °C or below -10 °C were removed, and the final size of the training set ended up at 212165 sets. The final training data shape ended up being (212165, 80, 7).

## 6.4 Adding noise to signals

In real measurements from SCADA, the sensors that measures temperatures and currents will always have some uncertainty to the measure values. To replicate this uncertainty in the generated training data to the neural networks, some Gaussian noise is added to the training data. The uncertainty of the signal is expressed as the standard deviation of

the signal, and for the PT100 temperature sensors used in the generator, this standard deviation is dependent on temperature. The higher the temperature measurement, the higher the uncertainty is, described in Figure 6.4 [50]. The temperature sensors is assumed to be Class B, with an uncertainty of $0.3 + 0.005 \cdot |T|$, where T is the temperature in Celsius. The maximum expected temperature in the training data is 120 °C, and therefore the maximum uncertainty can be calculated to be 0.9 using values from Figure 6.4. The standard deviation of the added noise is set to 1.0 throughout all the variables and data set. The noise added has a mean value of 0 and is distributed as Gaussian noise.

| Class | Temperature range in °C | | Tolerance value |
|---|---|---|---|
| | Wire-wound (W) | Thin-film (F) | |
| **B** | **-196 ... +600** | **-50 ... +500** | **±(0.30 + 0.0050 \| t \|) 1)** |
| A | -100 ... +450 | -30 ... +300 | ±(0.15 + 0.0020 \| t \|) 1) |
| AA | -50 ... +250 | 0 ... 150 | ±(0.10 + 0.0017 \| t \|) 1) |

1) | t | is the numerical value of the temperature in °C irrespective of the sign.

**Bold: Standard version**

Figure 6.4: A figure showing the uncertainty of the PT100 temperature measurement. Figure taken from [50]

There was a mistake when creating the training data for the generator model, where the noise was never added to the data. Therefore, the neural networks trained on generator training data in Chapter 8 had no added noise. The heat exchanger training data contained the added noise as intended.

## 6.5 Scaling the data for the neural networks

It has been shown that performance of a neural network can be compromised if the input feature values vary greatly in values between different variables, and when the span of possible values is large. The cold air temperature in the generator model will not vary greatly, and will usually keep a temperature of around 10 to 20 degrees C. The rotor and stator power will however vary much more, with the minimum values of around 1000, and maximum values of around 50000. The same is true for the neural network targets (model parameters). For instance iron mass parameter varies between 1500 and 70000 kg. Therefore, before the data is processed by a neural network, all variables are scaled to a value between 0 and 1. The tool used to do this is the Python module Scikit Learn's MinMaxScaler class [51]. Each input feature has a different scale, such that the after scaling, each feature variable will have a span of between 0 and 1.

Scaling a variable effectively ”squishes” all the values contained in a variable such that the minimum value of a variable gets scaled to a 0, and the maximum value of a variable gets scaled to 1.0, with all values in-between following a linear scale. Equation 6.5 shows the formula for scaling an array of values, $x$ to a value between 0 and 1.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{6.5}$$

# 7 Study case 1: Heat Exchanger parameter estimation

This study case will focus on defining a mechanistic thermal heat exchanger model that can be used to predict cold air temperature from hot air temperature and cooling water temperature. In addition, the heat exchanger model parameters will be estimated to fit the heat exchanger at Grunnaais 12 MVA hydroelectric generator. The Python code behind the results in this chapter is shown in Appendix E5 and E6.

The data used for parameter estimation will be based on the prepared data from Chapter 6. The data originally has 396 batches of 4-hour data samples, with time step of 1 second. For this study case, each batch will be split into eight 30-minute length samples with the same time step, effectively making 3168 batches of data.

There will be two different approaches for finding model parameters for the heat exchanger in this Study Case. The first approach will use a Python module called Scipy to minimize the error function, described in Chapter 4.5, by optimizing the model parameters. This approach is listed below.

1. Collect cold water temperature ($T_c^w$) and hot air temperature ($T_h^a$) as model inputs. Also collect the cold air temperatures ($T_c^w$) as this will be used to evaluate the model parameters using the MSPE.
2. For each batch of data, guess an initial model parameter set for the optimization algorithm.
3. Use the Scipy optimization algorithm to minimize the MSPE with respect to the model parameters by calculating the model output with the given set of model parameters. The optimization algorithm adjusts the model parameter such that the output temperature approaches the measured value. This is done for all the 3168 batches of data.
4. Store the optimal parameter predictions for all the batches of data together with the associated MSPE.
5. A weighted average and a normal average are taken over the parameter prediction sets to find unambiguous solutions for the parameters. The weighted average will be based on the inverse of the MSPE associated with a given parameter prediction.

From this, the two resulting model parameter sets will be determined as the optimal parameters found by the optimization algorithm.

The second approach of finding the model parameters will be using neural networks to predict the model parameters based on the $T_c^w$, $T_h^a$ and $T_c^w$ measurements. The approach for doing this is listed below.

1. Load in the heat exchanger training data created in Chapter 6.2.
2. Search for good neural network architectures using uninformed random search through the neural network hyperparameters.
3. Use the best-performing neural network will be trained further on the training data.
4. When the NN training is finished, it will make parameter prediction on all 3168 batches of data. The predicted parameter sets will be stored together with their associated MSPE.
5. A weighted average and a normal average are taken over the parameter prediction sets to find unambiguous solutions for the parameters. The weighted average will be based on the inverse of the MSPE associated with a given parameter prediction. From this, the two resulting model parameter sets will be determined as the optimal parameters found by the neural network.

## 7.1 Heat Exchanger class in Python

There are several interactions that will be required from the mechanistic heat exchanger model. These interactions are calculating the cold air temperature, calculate prediction errors and optimize model parameters. Therefore, a heat exchanger class is made in python, to easily interact with the mechanistic model. This class has the following requirements:

- Change the model parameter values, and store them internally in the class object.
- Calculate the cold air temperature, given the cold water and hot air temperatures as input, using specified parameter values.
- Calculate the mean squared prediction error between estimated and measured cold air temperature.
- Give the class object measured cold air temperature, and be able to calculate model parameters that fits the given measurement best.

The Python code for implementing the heat exchanger class is shown in Figure 7.1.

```python
class HeatExchanger:
  def __init__(self):
    #Inputs:         u = [Tw_c, Ta_h]
    #parameters: theta = [uAx,  mda,  mdw]
    self.chp_w = 4200
    self.chp_a = 1150
    self.A = lambda th: th[0]/(self.chp_w*th[2]) #Constant 1
    self.B = lambda th: th[0]/(self.chp_a*th[1]) #Constant 2
    self.Ta_c = lambda u, th: ((self.A(th) - self.B(th))*u[1] - (self.B(th)*np.exp(self.B(th) - self.A(th)) - self.B(th))*u[0]) / (
    self.A(th) - self.B(th)*np.exp(self.B(th) - self.A(th))) #Cold air temperature calculation

  def give_solution(self, Ta_c_real): #Give measurement data for cold air temp
    self.Ta_c_real = Ta_c_real

  def give_inputs(self, u): #u = [Tw_c, Ta_h]
    self.u = u

  def change_params(self, theta): #theta = [uAx,  mda,  mdw]
    self.theta = theta

  def calc_output(self): #y = Ta_c
    return np.array(self.Ta_c(self.u, self.theta))

  def prediction_error(self, theta): #This is the objective function for the optimizaiton
    self.change_params(theta)
    Ta_c = self.calc_output()
    mpe = np.sum(np.square(np.subtract(Ta_c, self.Ta_c_real)))/len(Ta_c) #Mean squared prediction error
    return mpe

  def find_optimal_parameters(self, u, y_sol, theta_0, theta_bounds): #Optimization with theta = [uAx, mda, mdw]
    self.give_inputs(u)
    self.give_solution(y_sol)
    sol = minimize(self.prediction_error, theta_0, bounds=theta_bounds)
    return sol.x
```

Figure 7.1: Python code for implementing the heat exchanger class

## 7.2 Parameter estimation using the Scipy library

Python has optimization modules for linear, non-linear, constrained, non-constrained, continuous, integer and boolean optimization problems. The heat exchanger is a non-linear optimization problem with boundary constraints. To optimize the three model parameters for the heat exchanger, Scipy's minimize function are used to minimize squared prediction error of the model output. The optimization method is "L-BFGS-B", which is used for non-linear optimization problems with boundary constraints [52].

The approach for finding model with respect to measurements could have taken two approaches. The first is to find the optimal model parameters while looking at the whole data set simultaneously. The second approach, which is used in this project, is to look at data in batches, where the size of these batches are defined in Chapter 6, and find optimal parameters for these batches individually. The optimal parameters may be different between each batch, and therefore, parameter prediction from each batch is stored together with the MSPE.

The specified optimization algorithm requires an initial parameter guess, where the optimization will begin. This initial guess is chosen randomly within the bounds specified in Table 7.1.

Table 7.1: Boundaries for the heat exchanger parameter estimation

| Parameter | Minimum | Maximum |
|---|---|---|
| $uA_x$ | 100 | 100000 |
| $\dot{m}_a$ | 1 | 100 |
| $\dot{m}_w$ | 1 | 100 |

### 7.2.1 Parameter predictions from Scipy

Optimization for the model parameters was done for all 3168 batches of data. The distribution of optimal parameter values and error values are shown in Figure 7.2. As for predicting unambiguous model parameters, a weighted average and a regular average is taken over all the optimal parameter predictions. Since the parameter set with lower error should be weighted more, the inverse of the error is used to weight the parameter predictions. The formula used for doing this is shown in Equation 7.1, where $E_i$ is the error related to the parameter set $x_i$.

$$x_{weighted} = \frac{\sum_{i=1}^{n} \frac{x_i}{E_i}}{\sum_{i=1}^{n} \frac{1}{E_i}} \tag{7.1}$$

The weighted average model parameters for the heat exchanger, together with the mean squared prediction errors are shown in Table 7.2. The table also shows the error values for the normal average parameter values.

Table 7.2: Parameter solutions and their respective errors

| Averaging | $uA_x$ | $\dot{m}_a$ | $\dot{m}_w$ |
|---|---|---|---|
| Normal average | 50317.6 | 50.9 | 51.5 |
| Weighted average | 50409.2 | 29.0 | 43.0 |

## 7.3 Parameter estimation using a recurrent neural network

The objective of the neural network is the same as the optimization algorithm, to minimize the MSPE. However, the neural network is never explicitly told what the prediction errors will be during training. It is only told how close the network was to predicting the intended model parameters. Before the neural network can predict anything, it must be trained. And before a network can be trained, its architecture must be decided. The NN's architecture is defined by the models hyperparameters.

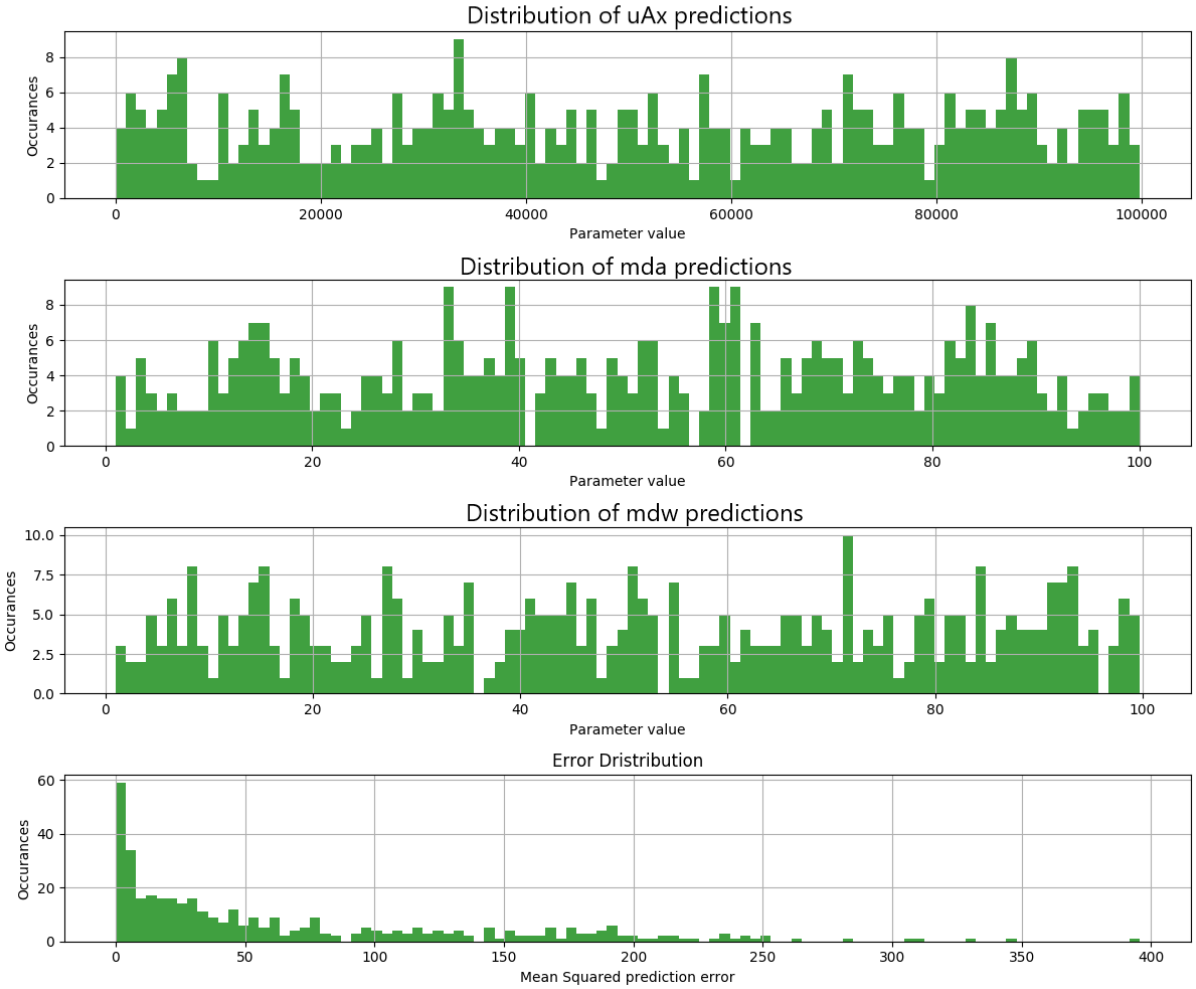



Figure 7.2: Parameter distribution and errors from the optimization on SCADA data.

### 7.3.1 Hyperparameters for the neural network

The process of finding good hyperparameters are known to be computationally demanding [32]. There are currently no clear way of knowing the optimal hyperparameters for a neural network beforehand. In this project, an attempt of filtering out the worst hyperparameters are done by an uninformed random search [53] through the ranges of hyperparameters given in Table 7.3. In total, 20 different neural network architectures was evaluated, which took in total about 16 minutes of computing time. The metric used to order the best performing neural networks was validation accuracy of the network after two epochs of training. The results from the hyperparameter search can be found in Appendix E6, and the best performing hyperparameter settings were the following: Recurrent layers - 1; Recurrent nodes per layer - 105; Dense Layers - 2; Dense nodes per layer - 166; Activation function - Sigmoid; Learning Rate - 0.001; Dropout Rate - 0.306; Recurrent cell type -

GRU.

Table 7.3: Boundaries of the parameter estimation

| Hyperparameter | Value range |
|---|---|
| Number of Recurrent layers | (1, 2) |
| Number of recurrent nodes per layer | (50, 300) |
| Number of Densely connected layers | (1, 2) |
| Number of dense nodes per layer | (50, 300) |
| Activation function of dense layer | (elu, relu, sigmoid) |
| Learning Rate | (0.001, 0.1) |
| Dropout Rate | (0.1, 0.5) |
| Recurrent node type | (LSTM, GRU) |

Finding any clear correlations between the hyperparameters using uninformed random search was difficult with only 20 evaluations. However, there may be signs of correlations between the hyperparameters, shown in Figure 7.3. The correlations are color-coded, with red being a positive correlation, and blue meaning negative correlation. The most noticeable correlations is that training time increases with the increase of any hyperparameter value related to the size of the neural network. One can also observe that accuracy tends to decrease with increasing learning rate.

### 7.3.2 Training the neural network

The chosen neural network is a relatively small neural network, with 62345 trainable parameters, including the different weights and biases in the nodes and LSTM-cells. For this neural network, the EarlyStopping callback, described in Chapter 2.5.4, was set to evaluate validation loss, and stop training if validation loss didn't decrease for 3 epochs.

The training was initially set to run for 20 epochs, but training was interrupted by the EarlyStopping callback at epoch 15. Training time took in total 4.35 minutes. The training and validation loss are shown in Figure 7.4 (a), while the training and validation accuracy are shown in Figure 7.4 (b). One can observe that the validation losses are always lower than the training losses. The reasons for this behaviour may be coming from any of the three following reasons [54]:

- The validation loss is calculated after training has occurred during an epoch. Due to this, the neural network has been more optimized when validating, than when training.
- Regularization techniques are applied during training, but not during validation. This means that the regularization method dropout is applied when training the neural network, but not used when validating.

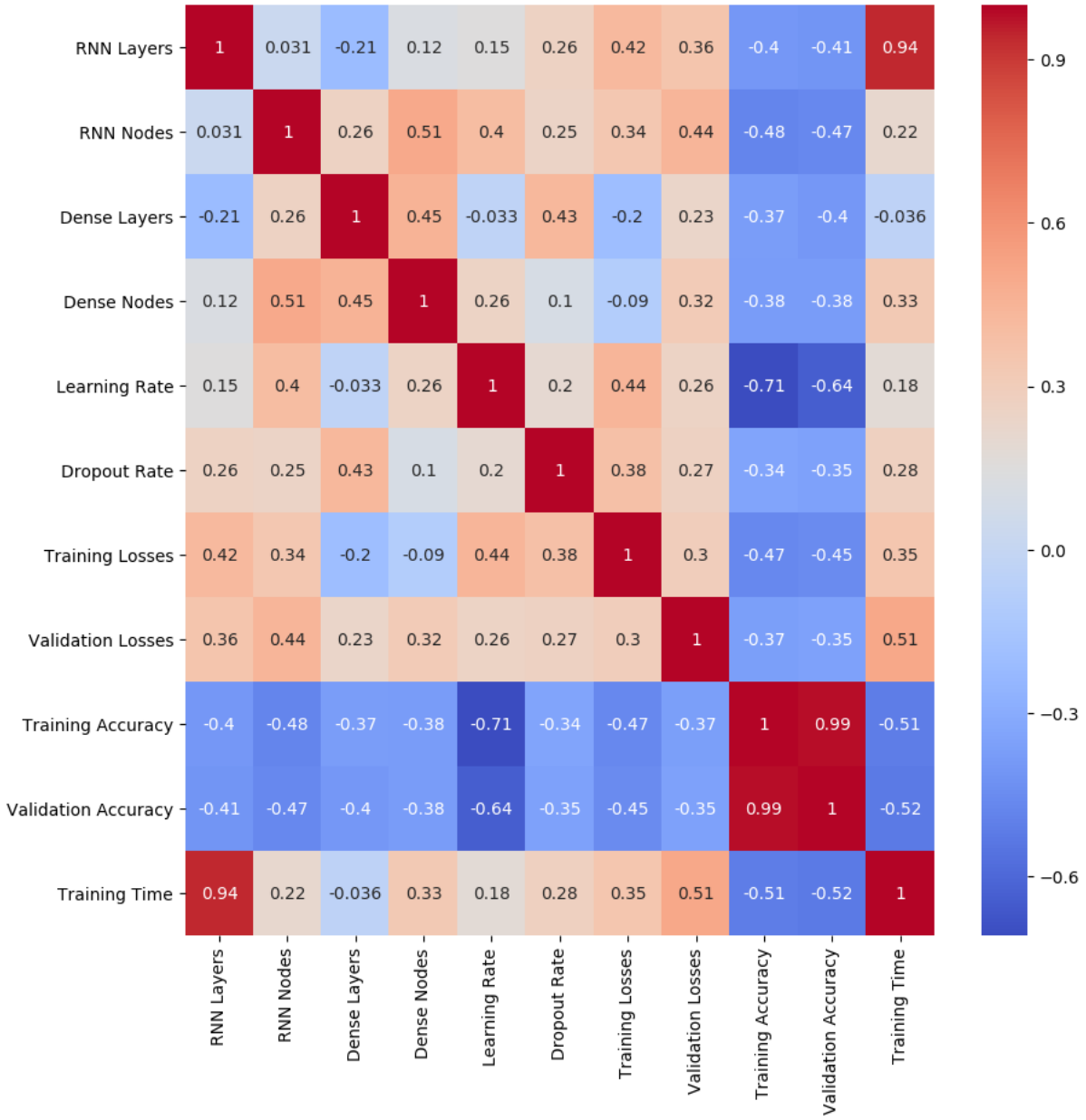



Figure 7.3: The hyperparameter correlation matrix from 20 evaluated hyperparameter sets.

- The validation data may be ”easier” to interpret that the training data. The validation data is in principle chosen randomly from the data set, but there might be a chance that this can occur, although repeated experiments can show that this is not the case in this project.

The behaviour of the accuracies on Figure 7.4 may be partially explained by the same reasons as the losses. The reason that the validation accuracy oscillates from epoch 2 to

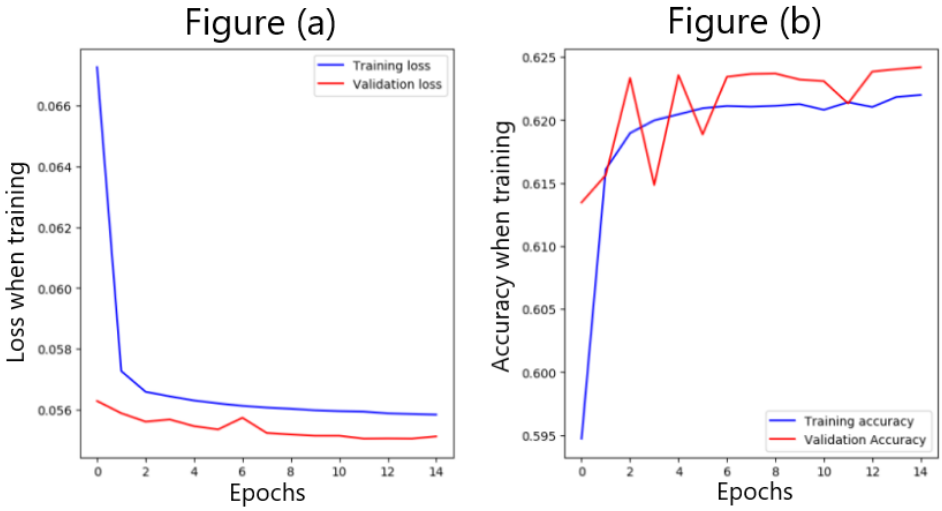



Figure 7.4: Losses and accuracies while training the neural network.

epoch 6 may be because of the regularization applied during training.

### 7.3.3 Neural network parameter predictions

All the 3168 data samples from SCADA was given to the neural network to predict model parameters. From all the data, the neural network parameter predictions are shown as distributions in Figure 7.5, together with the error distribution from the NN predictions. From the parameter predictions, both the weighted and normal average of the parameter values can be calculated as described in the introduction of this study case. The final calculated parameter sets are shown in Table 7.4.

Table 7.4: Average parameter predictions from the neural network

| Averaging | $uA_x$ | $\dot{m}_a$ | $\dot{m}_w$ |
|---|---|---|---|
| Normal average | 65771.9 | 38.5 | 56.9 |
| Weighted average | 65803.1 | 38.5 | 56.9 |

## 7.4 Summary of case study

The metric that is used to evaluate the performance of a parameter set is the mean squared prediction error. Based on this, the different strategies for finding model parameter can be evaluated and ranked. In Table 7.5 the different methods used in this case study list

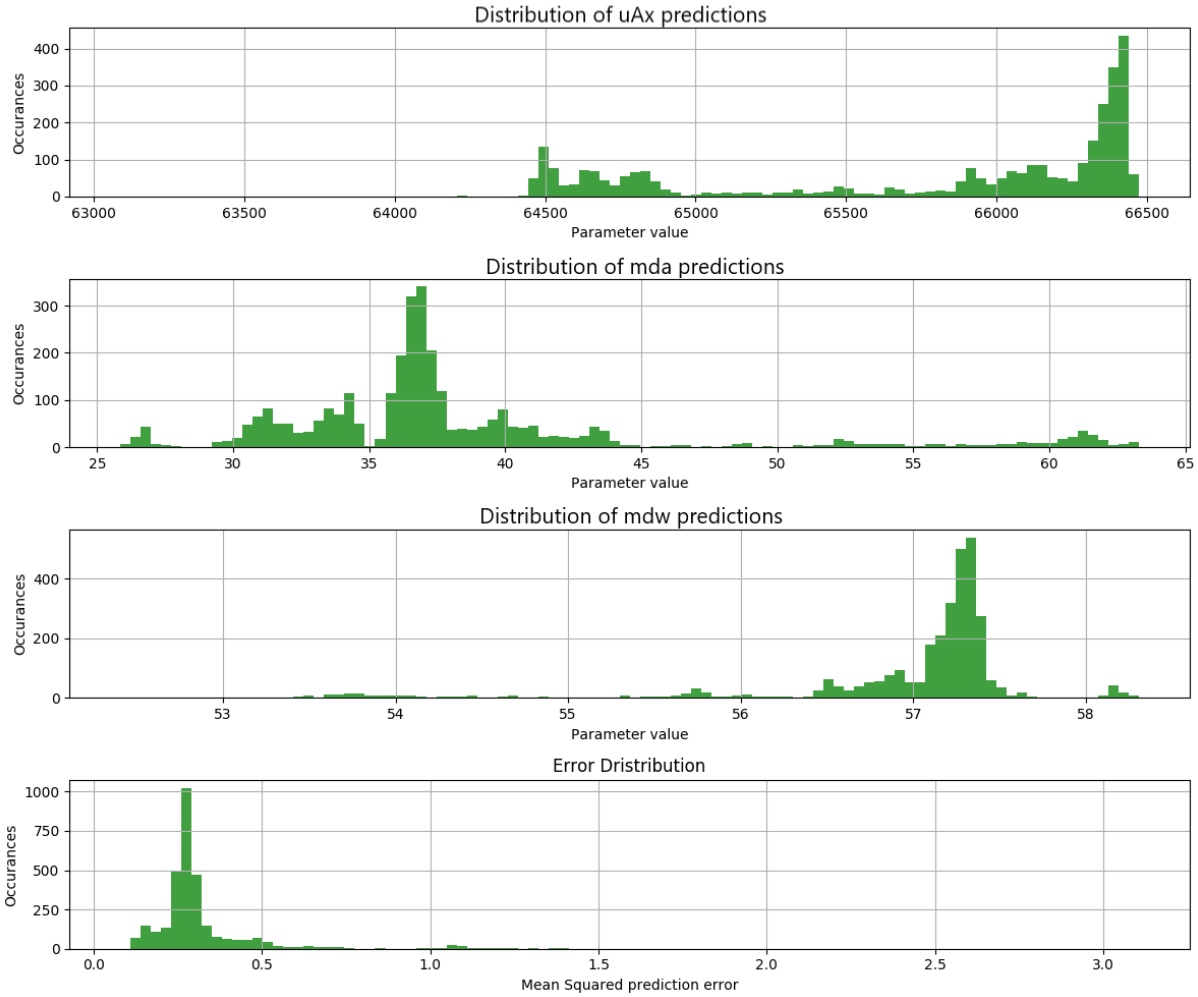



Figure 7.5: Parameter estimation distributions from the neural network on SCADA-data

the MSPE, together with the median error, minimum and maximum errors through the evaluation of all the batches.

Table 7.5: Prediction error values for the predicted parameter sets for the Heat Exchanger

| Averaging | Average error | Median error | Max error | Min error |
|---|---|---|---|---|
| Scipy parameters, normal average | 17.8 | 17.6 | 61.0 | $2.2 \cdot 10^{-3}$ |
| Scipy parameters, weighted average | 3.5 | 0.2 | 45.2 | $6.0 \cdot 10^{-4}$ |
| NN parameters, normal average | 3.33 | 0.19 | 46.59 | $1.4 \cdot 10^{-5}$ |
| NN parameters, weighted average | 3.32 | 0.19 | 46.46 | $5.4 \cdot 10^{-6}$ |

Since this optimization problem only has three parameter values, the error function can be illustrated as a 3-dimensional figure in parameter space. This shape is referred to as

the error shape. In Figure 7.6 any point represents a set of parameter values, and the color represents the MSPE over the whole data set with that given parameter set. All points with a MSPE greater than 5 is filtered out to only show parameter sets which produces a low MSPE. This illustrates that there are a range of good parameter solutions to this problem.

If one wanted to reduce the error shape in space, more data may be required.

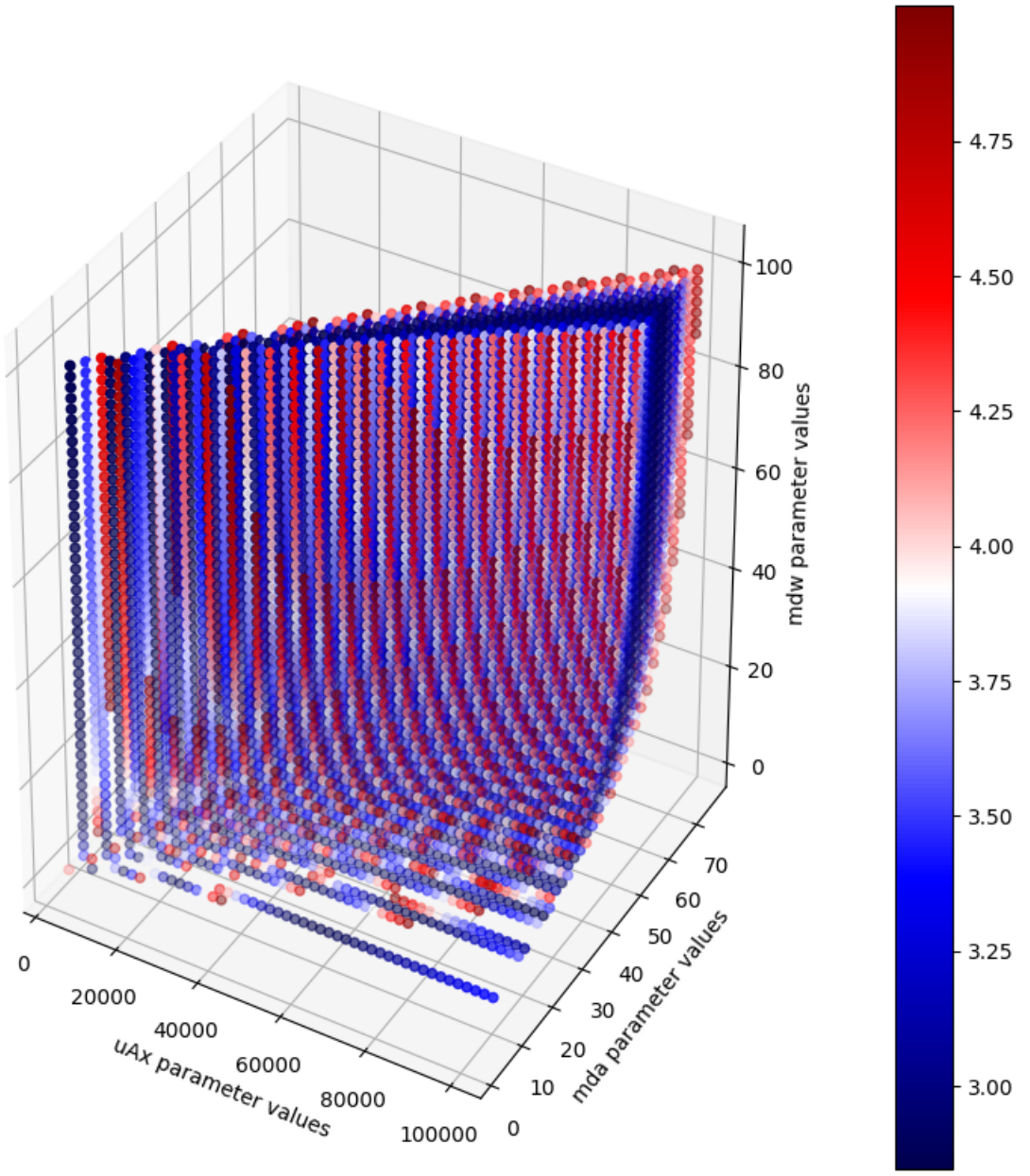



Figure 7.6: The error shape in the heat exchanger parameter space, with MSPE of less than 5.

# 8 Study Case 2: Generator Model Parameter Estimation

This study case will focus on parameter estimation of the thermal generator model developed in Chapter 4.3. Case study 1 in Chapter 7 gave an indication of how the process of parameter estimation proceeded. There are, however, two main differences in this study case, compared to case study 1. The first is that the mechanistic model is described in three first order differential equations instead of one algebraic expression. This means that each time the mechanistic model should predict some temperatures, it needs three initial temperatures before it can solve the equations numerically. These initial temperatures will be a part of the parameter prediction in this case study. The second main difference is that there are nine parameters that must be predicted, instead of the three heat exchanger parameters. In practice, this means that brute force parameter searching will not be a successful strategy, because of the vast available parameter space. In addition to a more difficult mathematical model, solving the three differential equations numerically takes time, and thereby evaluating the MSPE for each parameter prediction. The Python-code for generating the results in this chapter is shown in Appendix E8 and E10.

As in study case 1, parameter estimation will be done thought two different approaches. The first method is to optimize an error function with respect to the model parameters, and the second approach is model parameter prediction using neural networks. A small summary of both the Scipy approach and the neural network approach will be summarized, starting with the Scipy method in the following list.

1. Prepared data is collected from Azure Databricks.
2. A generator model class is made, containing the mathematical model and some useful functions for working with the model.
3. The parameters will be optimized on all 396 batches of data. Initial parameters will be guessed randomly within a boundary and the optimization algorithm should converge to a minimum with respect to MSPE.
4. The predicted parameter distributions should be displayed as histograms, and the final parameter predictions will be calculated using weighted and normal average over all parameter guesses.

5. Performance of the final two parameter predictions will be measured by the MSPE for the total data set.

For the parameter estimation using neural networks, the approach will be as follows.

1. The prepared neural network training data will be loaded in.
2. Neural network hyperparameters will be searched for using the uninformed search method.
3. Three neural networks with good performance will be chosen out, for further training. The three networks isn't necessarily the top three, but rather three well-performing networks which are different in architecture.
4. In addition, two other methods will also be evaluated. The first method is to collect the predictions from the three constructed NN's, and average out their outputs to get a new parameter output. The second method added is to create 20 small neural networks, train them separately, and make them all make parameter predictions. Then average the output of all the 20 networks to get a single parameter prediction.
5. Now that the neural networks are made, the training can begin. The final objective is to take five inputs gathered from the real generator, and make a prediction on the model parameters. What will be tried first is to train neural networks using all outputs and stated from the model, resulting in seven input variables to the RNN. All networks are trained and evaluated, then one of the model outputs will be removed (one of the temperatures not measured) and the process is repeated. The last iteration will remove the last unmeasured variable as input to the RNN, so that the network can train on the same type of data that it will see from measurements.
6. After the networks have all been trained, the networks are evaluated on the training data itself, to see how well it knows the training data. Then the final iteration of the neural network will be delivered data from the real generator, and model parameters will be predicted. There will be 396 parameter predictions, one for each batch, and the resulting parameter prediction will be the weighted and normal average of the predicted parameter distributions.

## 8.1 Generator model class in python

As with the heat exchanger model, the generator model will require several types of interactions, and it will be unpractical to have to set up the equations every time the model will be solved. The generator class will function as the interface between the mathematical model and functions that will be applied to the temperature predictions. The requirements of the generator class is as following:

- When initialized, the generator object will have a specified time step, which describes the length of the time step in each data point given to the generator model such that this doesn't have to be specified with the data itself. One will also have to specify the iron losses and mechanical losses in watt, the rotor copper mass and stator copper mass in kg.
- Model parameter values can be stored in the generator object.
- One batch of input data, stator copper temperature measurement and hot air temperature measurement can be stored in the generator object.
- The model parameters can be estimated using the internally stored variables, some initial parameter guess and parameter bound constraints.
- Should be able to find the MSPE between a measurement and simulation.
- With a given set of model parameters, and some input data and measurements, should be able to estimate only the initial parameter guesses of the states in the model.

## 8.2 Parameter estimation using Scipy

When finding model parameters for the generator model, the data set prepared in Chapter 6 will be used. The data contains 396 batches of 4 hour long data. For each batch, a parameter estimate will be done using Scipy to minimize the prediction error with respect to the model parameters. As defined in Chapter 4.3, there are four temperature outputs. However, only two of them are measured and collected from the prepared data. Therefore the MSPE will be calculated using the two available temperatures from data, namely stator copper temperature and hot air temperature.

The prediction boundary given to each model parameters are shown in Table 8.1. Parameter estimation for all the 396 batches of data took approximately 9 hours of computing time in Azure Databricks. The distribution of parameter guesses are shown in Figure 8.1.

Seen in Figure 8.1, the parameter guesses was for several of the parameters all over the boundary set boundary in Table 8.1. However, as illustrated in Figure 8.2, the prediction error for most parameter estimations were usually very low. The average MSPE using Scipy was 2.4, when evaluating the MSPE for individual batches of data.

As in case study 1, both the weighted and normal averages are taken over the parameter distributions to obtain an overall parameter prediction. Initial temperatures are not model parameters as they should be unique for each batch. Therefore, when evaluating the total prediction errors of the found optimal parameters, the initial states was computed for each

Table 8.1: Boundaries for the generator parameter estimation

| Parameter | Minimum | Maximum |
|---|---|---|
| $hA^r_{Cu2a}$ | 200 | 10000 |
| $hA^s_{Cu2Fe}$ | 200 | 10000 |
| $hA^s_{Fe2a}$ | 200 | 10000 |
| $M^s_{Fe}$ | 1500 | 70000 |
| $ML_{const}$ | 0.0 | 1.0 |
| $\dot{m}_a$ | 1 | 50 |
| $T^r_{Cu0}$ | 10 | 70 |
| $T^s_{Fe0}$ | 10 | 40 |
| $T^s_{Cu0}$ | 10 | 70 |

batch using Scipy optimize. Instead of minimizing the model parameters with respect to the MSPE, the found model parameters were held constant and the initial temperatures was optimized with respect to MSPE. By this approach, the errors for the total data set was estimated for the model parameters predicted and shown in Table 8.2, together with its associated MSPE.

Table 8.2: Average parameter predictions using Scipy on all the data batches.

| Averaging | $hA^r_{Cu2a}$ | $hA^s_{Cu2Fe}$ | $hA^s_{Fe2a}$ | $M^s_{Fe}$ | $ML_{const}$ | $\dot{m}_a$ | MSPE |
|---|---|---|---|---|---|---|---|
| Normal average | 5145.8 | 5077.0 | 3540.3 | 35654.3 | 0.89 | 6.0 | 0.8977 |
| Weighted average | 4491.8 | 5122.2 | 3269.8 | 36918.2 | 0.91 | 6.0 | 0.7523 |

## 8.3 Search for neural network hyperparameters

Since this problem is more challenging than the heat exchanger case, and since the neural networks may experience different accuracies and losses when training depending on the random initialization of weights and biases, 50 different neural networks are trained for 3 epochs for evaluating the performance of the hyperparameters. The computing time took just over seven hours to complete. The metric that evaluates the neural network performances are the validation accuracy, and the 10 best-performing NN's with their respective hyperparameters are shown in Table 8.3. The neural networks used for parameter estimation is shown with yellow color in the table. Note that the column titles in Table 8.3 are abbreviations, and their meaning is the following: NN num - Neural Network number; Val Acc - Validation Accuracy; Train Acc - Training Accuracy; RNN Lrs - RNN layers; RNN nds - RNN nodes per layer; Dns Lrs - Dense Layers; Dns Nds - Dense Nodes per layer; Act Func - Activation Function; LR - Learning Rate; DR - Dropout. The full code for finding these hyperparameters are shown in Appendix E9.

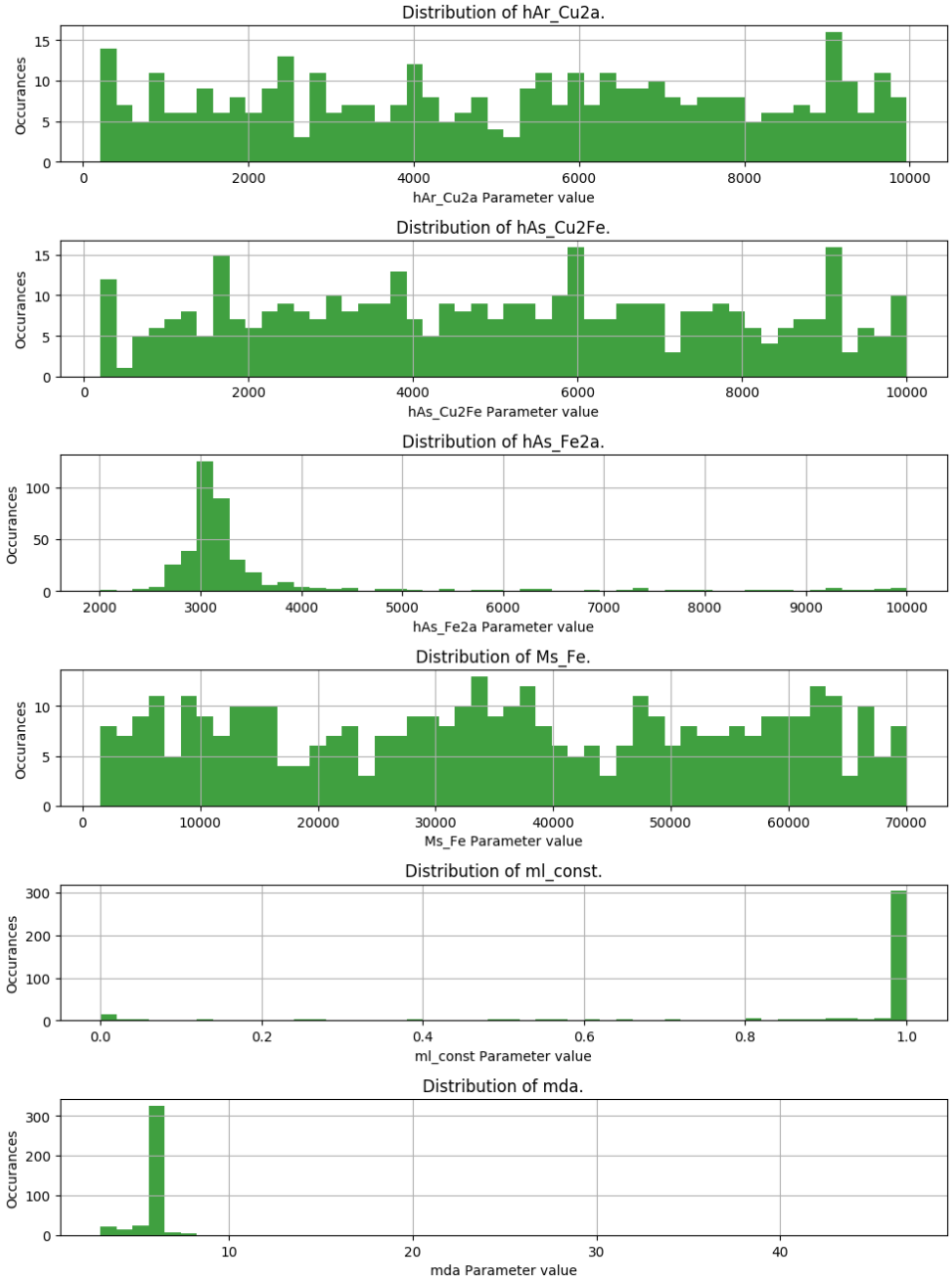



Figure 8.1: Scipy optimization parameter guesses throughout the data set

## 8.4 Defining the neural networks

The final objective is for the neural networks to look at real data from measurements, and predict model parameters based on these measurements. In an ideal case, all model outputs are measured, and can be given to the neural network. However, this is not the

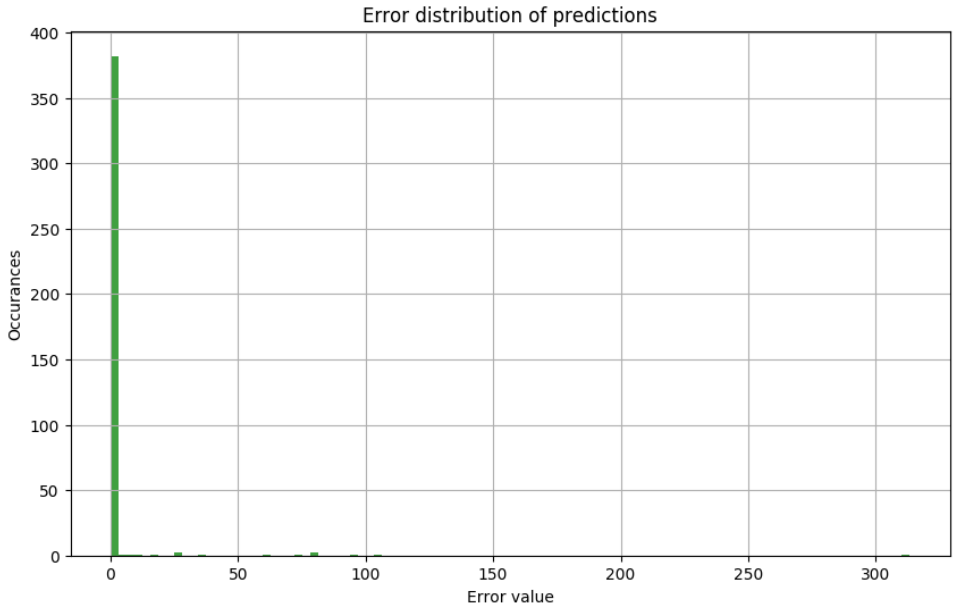



Figure 8.2: All prediction errors while optimizing parameters with Scipy on SCADA-data.

Table 8.3: In order, the ten best performing neural networks while searching for hyperparameters

| NN num | Val Acc | Train Acc | RNN Lrs | RNN Nds | Dns Lrs | Dns Nds | Act Func | LR | DR | RNN Type | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.365 | 0.371 | 2 | 359 | 1 | 314 | elu | 0.012 | 0.448 | GRU | 279 |
| 2 | 0.314 | 0.306 | 2 | 294 | 1 | 107 | elu | 0.044 | 0.164 | LSTM | 248 |
| 3 | 0.193 | 0.314 | 1 | 58 | 1 | 473 | sig | 0.017 | 0.128 | GRU | 109 |
| 4 | 0.171 | 0.140 | 2 | 1201 | 1 | 89 | relu | 0.013 | 0.364 | GRU | 928 |
| 5 | 0.150 | 0.368 | 1 | 374 | 1 | 488 | elu | 0.087 | 0.173 | GRU | 139 |
| 6 | 0.120 | 0.127 | 1 | 377 | 1 | 462 | sig | 0.053 | 0.103 | LSTM | 150 |
| 7 | 0.114 | 0.111 | 2 | 768 | 1 | 76 | sig | 0.080 | 0.322 | LSTM | 464 |
| 8 | 0.113 | 0.111 | 2 | 53 | 2 | 440 | relu | 0.073 | 0.106 | LSTM | 182 |
| 9 | 0.113 | 0.111 | 2 | 467 | 2 | 275 | relu | 0.099 | 0.443 | GRU | 300 |
| 10 | 0.113 | 0.118 | 2 | 982 | 1 | 599 | sig | 0.051 | 0.221 | LSTM | 780 |

case in this project. There are two temperatures that the generator model will predict, which is not measured. However, the neural network will be trained on three different types of data sets to evaluate the impact of having more available data. To be clear, the data sets all come from the same training data, the only difference is the amount of variables present in each set of data. For convenience, and later reference, the three different data sets are named as listed in Table 8.4. The table also shows what they contain as far as input variables to the NN's. The objective for all NNs is always the same: estimate the generator model parameters based on the NN input data. For this project,

Table 8.4: The different Neural network training data sets

| Data set name | Included variables | Removed variables | NN Input shape |
|---|---|---|---|
| DS1 | $T_c^a$, $\dot{Q}_r$, $\dot{Q}_r$, $T_{Cu}^r$, $T_{Fe}^s$, $T_{Cu}^s$, $T_h^a$ | None | (x, 80, 7) |
| DS2 | $T_c^a$, $\dot{Q}_r$, $\dot{Q}_r$, $T_{Cu}^r$, $T_{Cu}^s$, $T_h^a$ | $T_{Fe}^s$ | (x, 80, 6) |
| DS3 | $T_c^a$, $\dot{Q}_r$, $\dot{Q}_r$, $T_{Cu}^s$, $T_h^a$ | $T_{Cu}^r$, $T_{Fe}^s$ | (x, 80, 5) |

five neural network architectures will attempt to find model parameter estimations. Three of these architectures are marked in Table 8.3. The fourth and fifth architectures uses a so-called stacked neural networks to predict model parameters, where the resulting predictions will be an element-wise average of the parameter estimation between the neural networks. This method is illustrated in Figure 8.3, where the number of NNs and number of prediction parameters can be any number in principle. Experiments done by others show that this approach can be a way to reduce prediction errors [26]. The first stacked neural network (SNN) is the ensemble of NN-number 1, 3 and 6 in Table 8.3. The second SNN is the ensemble of 20 NN's with hyperparameters as NN-number 3 in Table 8.3.

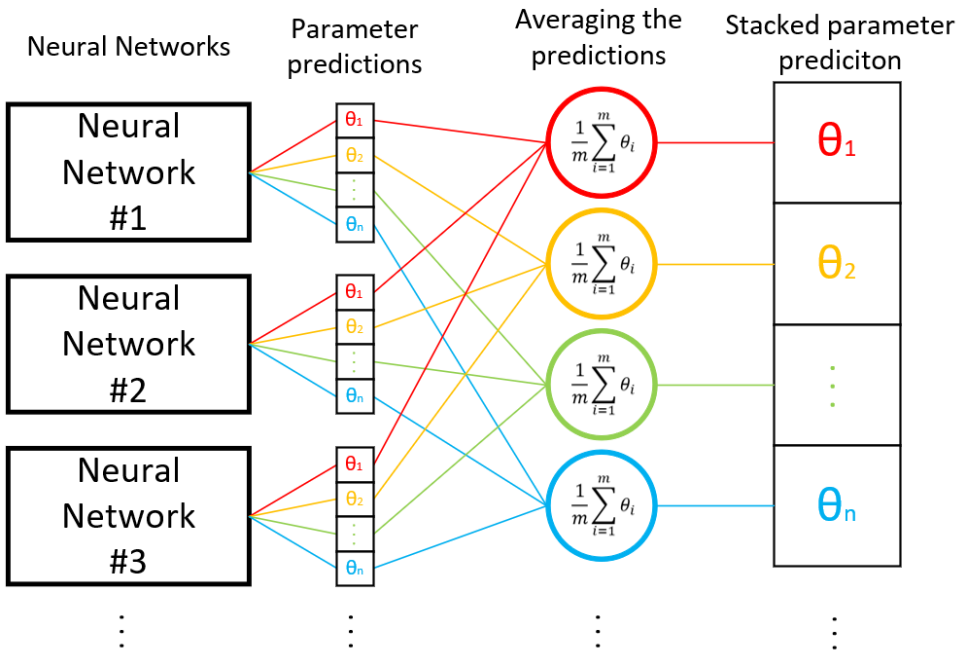



Figure 8.3: Stacked neural network architecture, with the parameter prediction being an element-wise average of the outputs of the neural networks.

| Data Set | Neural Network | Training Accuracy | Validation Accuracy | Training Losses | Validation Losses | NN parameters | Training Time |
|---|---|---|---|---|---|---|---|
| DS1 | NN1 | 0.36 | 0.38 | 0.067 | 0.055 | 175016 | 75 min |
| | NN2 | 0.46 | 0.27 | 0.050 | 0.060 | 12189 | 15 min |
| | NN3 | 0.34 | 0.34 | 0.070 | 0.070 | 585490 | 21 min |
| DS2 | NN1 | 0.36 | 0.37 | 0.053 | 0.048 | 1175016 | 30 min |
| | NN2 | 0.34 | 0.34 | 0.051 | 0.057 | 12015 | 6 min |
| | NN3 | 0.21 | 0.22 | 0.122 | 0.163 | 583982 | 10 min |
| DS3 | NN1 | 0.24 | 0.27 | 0.079 | 0.080 | 1172862 | 60 min |
| | NN2 | 0.29 | 0.26 | 0.075 | 0.070 | 11841 | 10 min |
| | NN3 | 0.20 | 0.16 | 0.102 | 0.130 | 582474 | 21 min |
| | SNN2 | 0.26 | 0.25 | 0.066 | 0.072 | 236820 | 20 min |

### 8.4.1 Training the neural networks

Training and evaluating the neural networks will happen in three separate stages, once for each data sets, DS1, DS2 and DS3. The list below summarizes the training steps for the different neural network structures.

1. DS1: Neural network NN1, NN2 and NN3 was trained on this data set. NN1 was trained without EarlyStopping by accident, and finished in total 60 epochs. NN2 and NN3 used EarlyStopping with a patience of 10 epochs.

2. DS2: Neural network NN1, NN2 and NN3 was trained on this data set. EarlyStopping had a patience of 10 epochs for all the NNs during training.

3. DS3: Neural network NN1, NN2, NN3 and SNN2 was trained on this data set. The EarlyStopping-settings was identical as the previous data set. The SNN2 consists of 20 small neural networks, and each of these networks was trained individually.

The data, training stats and more information about the neural network training can be found in Appendix E10. While training all the neural networks, the losses and accuracies were stored and plotted. The resulting losses and accuracies are shown in Table 8.4.1.

## 8.5 Neural network performance with different data sets

From the training data set there are in total 212165 batches of data that can be used to predict parameter values. Calculating the prediction error for all of the batches, for all five neural network predictions would take a very long time, and instead, 2000 randomly picked batches was selected to evaluate the networks general performance on the data set.

Figure 8.4 displays the MSPE while testing the neural networks on the data sets DS1, DS2 and DS3. One can observe that the amount of data clearly impacts the performance

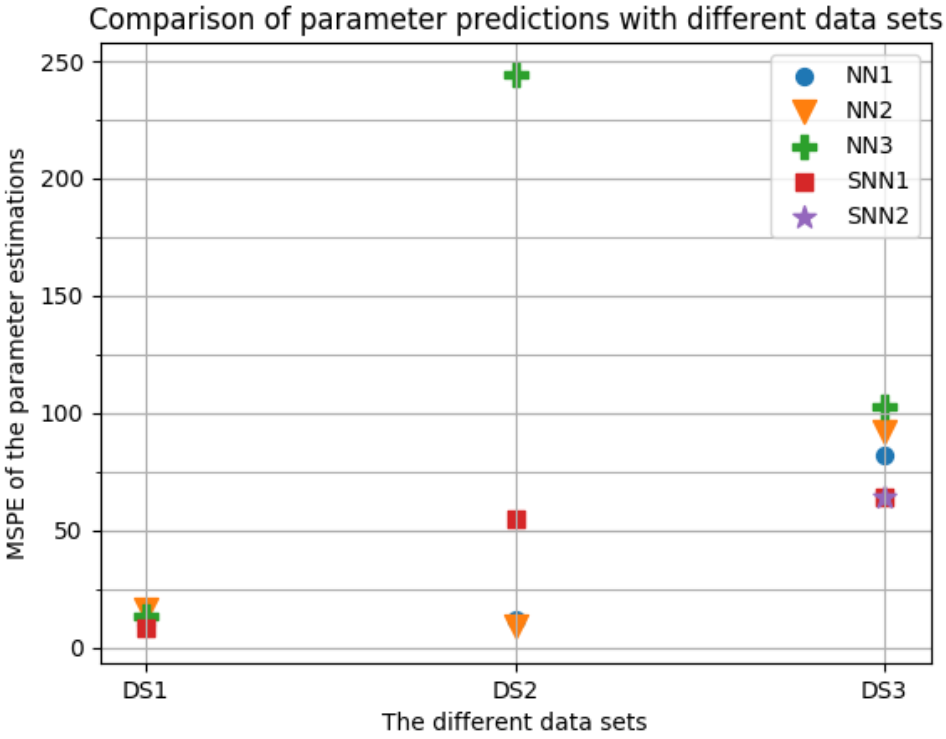



Figure 8.4: The neural network performances on the different data sets.

of the neural networks. Although the performance aren't great in any cases, the performance is significantly reduced by removing the iron core temperature and rotor copper temperature.

### 8.5.1 Parameter estimation with currently measured data points

While the training data gave an overview of how the neural networks could guess model parameters from a mathematical model, the SCADA-data is not taken from a model, rather a real synchronous generator. The currently measured data points at Grunnaai is the equivalent to data set DS3, and therefore the NNs trained on DS3 will make predictions on the real measurements.

From the generator data set, the neural network predicted a set of parameters for each batch of data. The predicted parameter sets are shown in Table 8.5.

From all parameter estimations, the MSPE was calculated, and this is shown in Figure 8.5. Read from left, the first columns are the average MSPE throughout the 396 predictions

Table 8.5: Average parameter predictions from the neural networks on all the data batches.

| Neural Network | $hA^r_{Cu2a}$ | $hA^s_{Cu2Fe}$ | $hA^s_{Fe2a}$ | $M^s_{Fe}$ | $ML_{const}$ | $\dot{m}_a$ | MSPE |
|---|---|---|---|---|---|---|---|
| NN1 - NA | 6599.9 | 5991.6 | 6747.1 | 49823.7 | 0.66 | 20.8 | 185.4 |
| NN1 - WA | 6504.4 | 6013.9 | 6792.3 | 49887.8 | 0.66 | 20.3 | 183.2 |
| NN2 - NA | 4938.6 | 4650.0 | 5669.2 | 27420.9 | 0.56 | 17.9 | 162.9 |
| NN2 - WA | 4904.1 | 4660.5 | 5649.0 | 27298.7 | 0.57 | 17.9 | 162.4 |
| NN3 - NA | 3118.1 | 5144.1 | 6851.9 | 39927.0 | 0.56 | 31.8 | 218.5 |
| NN3 - WA | 3115.5 | 5146.4 | 6851.2 | 39926.6 | 0.56 | 32.0 | 219.2 |
| SNN1 - NA | 4885.5 | 5261.9 | 6422.7 | 39057.2 | 0.59 | 23.5 | 191.8 |
| SNN1 - WA | 4876.1 | 5286.0 | 6431.7 | 39075.3 | 0.59 | 23.6 | 192.3 |
| SNN2 - NA | 4131.1 | 4837.1 | 4931.5 | 35575.8 | 0.57 | 14.9 | 129.1 |
| SNN2 - WA | 4121.0 | 4859.3 | 4940.3 | 35626.1 | 0.57 | 14.8 | 129.1 |

for all the NNs, both when the parameters were averaged, and when they were averaged with inverse error as weight. The second column are the median error, the third is the maximum error occurred during any predictions throughout the data set, while the last columns are the minimum error that occurred.

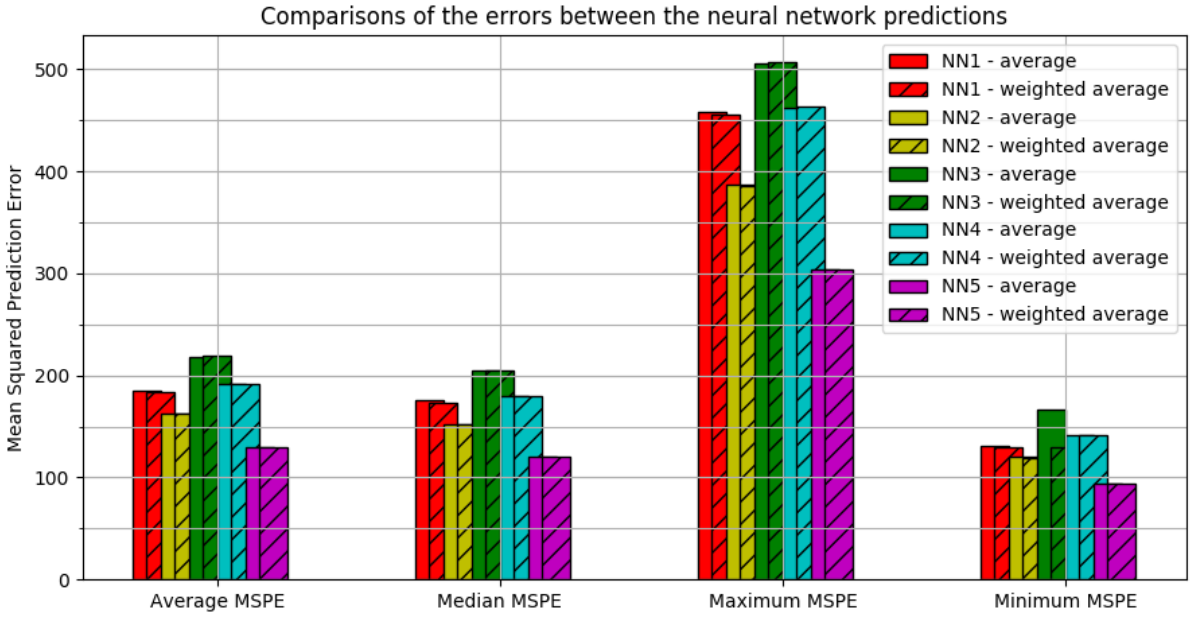



Figure 8.5: The neural network performances on the different data sets.

## 8.6 Summary of the parameter predictions

From the Scipy optimization method, two sets of model parameter values were calculated as optimal sets. One set was from the element-wise average of all predicted parameter sets, and the other set was the weighted average. From the neural network, ten model parameter sets were predicted, two for each neural network, a normal average and a weighted average. For the neural network parameter estimations, only temperature predictions from weighted average parameter sets will be shown in this chapter, because they were usually better performing.

Using the two average parameter predictions from Scipy, the resulting generator model prediction can be observed in Figure 8.6. The figure shows a 4-hour batch of temperature predictions from the generator model, using both parameter sets found from Scipy.

Estimation of the generator temperatures, based on the neural network parameter predictions is shown in Figure 8.7. The input data is the same as used in Figure 8.6. In that way, the differences between the two prediction can be more clearly observed.

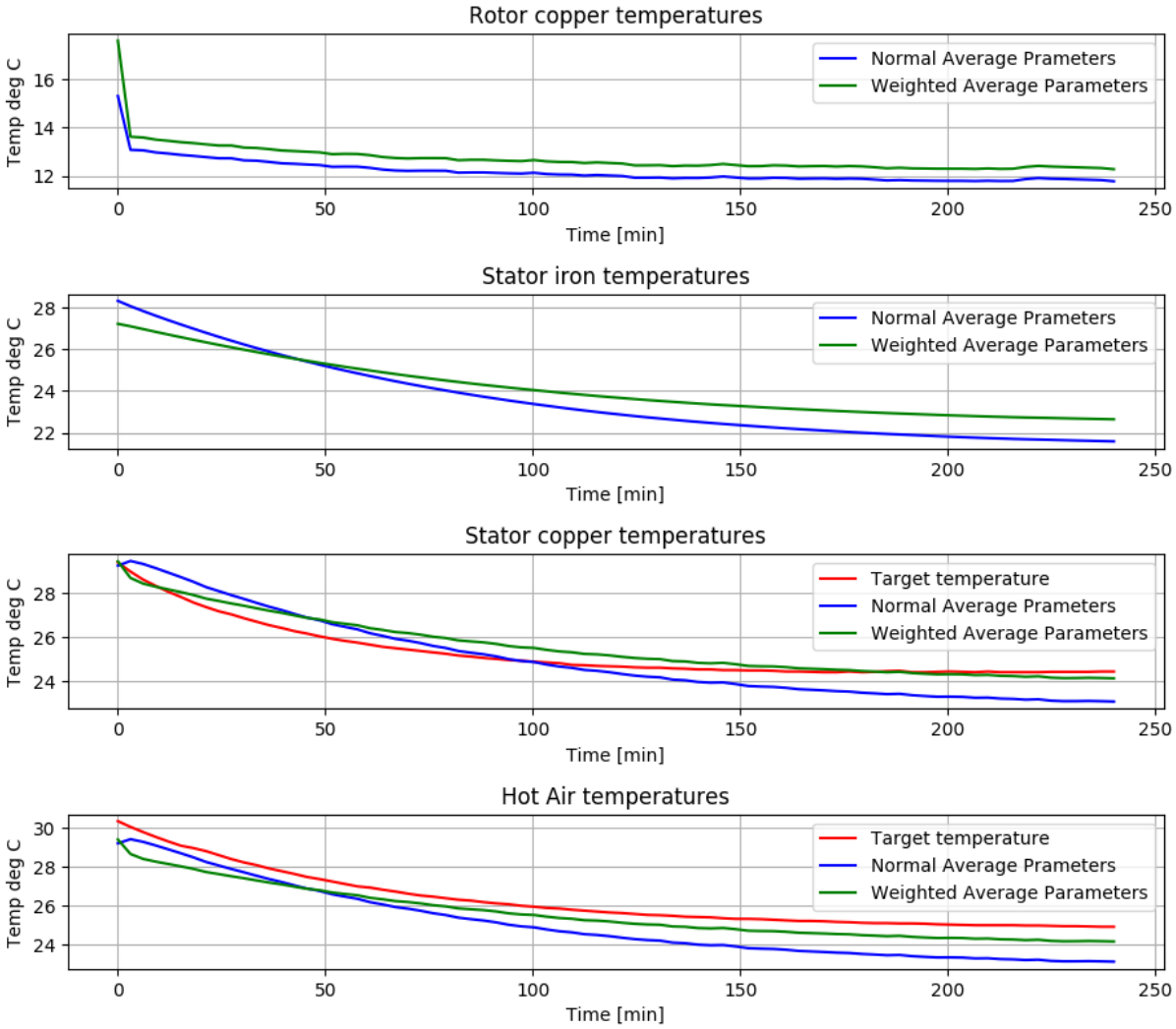



Figure 8.6: Temperature predictions from the generator with parameters found from the Scipy optimization.

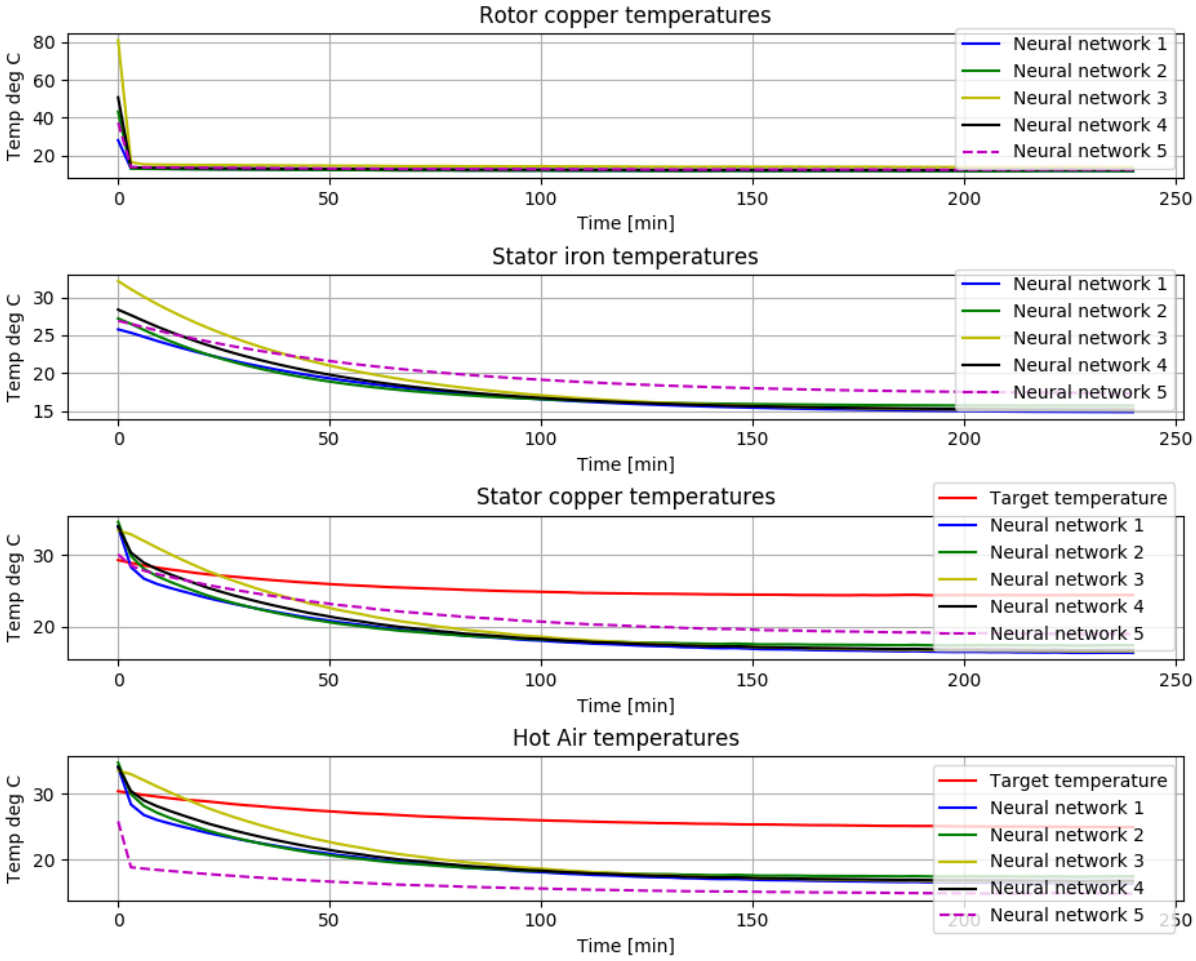



Figure 8.7: Temperature prediction in the generator from parameter estimation using the neural networks.

# 9 Discussion

- The predicted parameters for both the heat exchanger and the generator model isn't necessarily realistic. One could have set stricter boundaries on the optimization, but optimally, more and better measurements should help filtering out many unrealistic parameter sets.
- The neural network had a slight advantage in that it had already been used to most of the input data, namely cold water temperature $T_c^w$ and hot air temperature $T_h^a$, which the training data is based on.
- Initially, the thought was to have the generator model and the heat exchanger model in the same model. This was however difficult as there was missing a few temperatures that could have been useful, such as hot water temp and mass flow rate of cooling air and water.
- The boundary constraints may have been too strict, or completely wrong.
- When finding neural networks from hyperparameter search, the Parento graph could have been drawn from both losses and both accuracies to filter out strictly worse hyperparameters.
- I did accidentally forget to add measurement noise to the training data for the generator model.
- When searching for hyperparameters, the search should have been done once for each data set DS1, DS2 and DS3 because these are in principle different problems that the NNs has to solve.
- In this project, the rotor and stator resistances are assumed constant. This is just an approximation, as these resistance values should change with temperature.
- More focus should have been on training the RNN with just 5 feature variables, Since this is what is actually measured/obtained from SCADA.
- Where am I on the curve in Figure of data vs accuracy?

There is a lot of literature around parameter estimation using both deterministic and probabilistic methods. But in regards to mechanical model parameter estimation using recurrent neural network, the literature has been more scarce. Some articles like[55] has done similar studies, but with a Hopkins Neural Network, which this project may

have benefited from using. The parameter estimation was attempted with basic-level neural network technology. This chapter will be discussing three main aspects of this report, namely: Data acquisition and preparation; Modeling of the thermal system; The parameter estimation methods.

## 9.1 Data acquisition and preparation

Data acquired in this project are from SCADA-data from Grunnaais 12 MVA synchronous machine. SCADA-data is sub-optimal for this type of project because of several reasons, listed below.

1. The data points has uneven distributions between the data points, which has to be corrected by interpolating the acquired data points. Imagine that two data points are separated two hours in time. There is no way of knowing if there was a linear change or a sudden change between the two data points. This may have an impact if the variable affects the model significantly.
2. Because SCADA is only logging variables with a change larger than a specified dead-band, the acquired data may be inaccurate somewhat. It would have been advantageous to get the raw data, and work with unfiltered data, or apply a filter that gives even time steps.
3. SCADA doesn't contain all the measured data. In the power plant, numerous measurements are taken, and only some are sent to SCADA. This includes measurements such as hot cooling water temperature and the generator hall air temperature. These variables would allow for more accurate models to be developed, and parameters to be better predicted.

### 9.1.1 Estimating missing variables

In Chapter 6.1 the rotor and stator copper losses was estimated based on the resistances from the heat run test and the respective currents. These currents had to be estimated because the correct currents was not accessible in SCADA. The estimation of the magnetization and armature currents may be inaccurate, and this is one layer of uncertainty that could have been avoided with access to all the variables from Skagerak.

There were several other variables that would have been advantageous to have as input to the generator model. This includes: mass flow of air through the generator; mass flow of water through the heat exchanger; generator hall temperature. Because these variables was inaccessible or not existed as measurements, the mechanistic model had to be simplified. One simplification is that the heat exchanger was separated from the generator

model because of the high uncertainty of cold air temperature prediction from the heat exchanger model. Since the cold air temperature is measured, this could have been another variable for determining the model performance using MSPE, but was instead separated into two models. Had hot water temperature been a part of the accessible measurements, the heat exchanger might have been much more reliably modeled.

## 9.2 Thermal modeling and simulations

The lumped parameter thermal network is fast and easy to compute. In literature such as [6] & [8], much more elegant and accurate LPTN models are made, compared to this project. The fact that the model in this project is simple, may also be responsible for some of the inaccuracies predicted with the best sets of model parameters.

The mechanistic thermal generator model was developed in python, and solved in Scipy. There are several other alternatives for thermal modeling, both with graphical and programmable interfaces such as Julia, Matlab/Simulink or Modelica. Modelica has a working API with Python, and could have decreased simulation time, while making it easier to set up the equations for the mechanistic models.

The developed model initially intended to include several other components in the LPTN model. The main reason for simplifying the model was because of inaccessible variables and time. SCADA provided for most of the variables needed, but did not include all variables for a more sophisticated model. There is several other variables that are not measured in Grunnaai today, and more measurements will definitively lead to better models and parameter estimations. Some addition that were excluded from the final mechanistic thermal generator model are listed below.

- Separating the currently named stator iron with a stator core and stator body component.
- Including a rotor iron component, where both the rotor copper and rotor iron gets cooled down by the cooling air, in addition to heat conduction between them.
- Including a separate temperature for the air gap between rotor and stator, indicating that the cooling air has been heated some before cooling down the stator.
- Include radiation losses between the stator iron body and the environment.
- have a more representative and accurate heat transfer from the generator bearings into the thermal system.
- Instead of having one lumped capacitance with one heat conductance in the metal systems, having a number of distributed lumped thermal elements. The cooling air

heats up as it travels through the generator, and this could have been modeled with several lumped thermal systems.

## 9.3 Results from parameter estimation

Two study cases were presented in this project for estimating model parameters using two different methods. The first method was to minimize the mean squared prediction error with respect to model parameters as a constrained optimization problem in Scipy. The second method was to use neural networks to predict model parameters with the model inputs and model outputs as inputs to the networks. In study case 1 in Chapter 7, Figure 7.6, an illustration of the error function was displayed as a function of the three model parameters. The 3-D shape shown in the figure will be referred to as the heat exchanger error shape. It shows that although good parameter estimations were found for the heat exchanger model, there were widely different parameter solutions with similar low prediction error. The error shape could have been reduced with more available data, and with the hot water temperature as an additional variable to compute MSPE from. In the heat exchanger case, there were only three parameters, and the error shape then becomes a three dimensional shape. For the thermal generator model, there are six unknown model parameters plus three initial state values that needs to be predicted, and such an error shape cannot be illustrated graphically. However, the generator model is likely to have a more complex error shape, which may consist of ”pockets” of local minimum regions, which means that optimization functions can easily get stuck in these pockets. With more measurements such as the stator iron and rotor copper temperature, the error shape can be drastically reduced in parameter space, and better model parameters can be easier to find. This is especially important if further work will focus on the development of more complex models.

Results from both the constrained optimization method, and the neural network method was obtained in Chapters 7 and 8, and the results will be discussed for both study cases.

### 9.3.1 Discussion of results from Study Case 1

Both the neural network and the Scipy optimization method showed good model parameter predictions. The parameter prediction distribution for all the 396 batches of generator data shows that the Scipy optimization has a much wider range of parameter guesses compared to the neural network. However, when averaging the parameter predictions on each case, the results showed that the final parameter predictions was precise in both cases. This case study showed that the neural network is capable of predicting model parameters based on a time-series of input values. There are no guarantee that the predicted

parameters are realistic. With the hot water temperature, the MSPE would have had to adapt the model parameters with more constraints, leading to more realistic values.

### 9.3.2 Discussion of results from Study Case 2

When comparing the the temperature predictions from parameter estimation, based on the Scipy optimization approach, the temperature estimation is much better than with the neural network predictions. The average MSPE of the Scipy parameter estimation for the weighted average parameters are 0.7523, which is a really good temperature estimation for almost all batches of SCADA-data. The best neural network parameter estimation had a MSPE of 129.1, which is much worse than the optimization algorithm. There may be several causes of this, some are listed below.

- The training data was not representative to real data. When the neural network tried predicting model parameters from training data, the MSPE was in the region of 15, shown in Appendix E11.
- The neural network architectures chosen for this problem might have been unfit for this objective. There are many types of architectures which were not tried in this project.

Other types of neural network types, such as Hopfield neural networks may have performed better on this task. It has been demonstrated that this type of neural network are able to predict model parameters with great accuracy [55].

### 9.3.3 Method of generating the training data

All of the training data was created using measured and estimated variables obtained from SCADA. These input variables were used for several iterations with random model parameters to generate a variety for different model outputs. There are two weaknesses to this approach. The first is that the initial state temperatures $T^r_{Cu0}$, $T^s_{Fe0}$ and $T^s_{Cu0}$ was set randomly together with the generator model parameters for each training set. With this approach, the most likely outcome is that the output temperatures will start at temperatures which is far from their steady-state values, resulting in a step-change behaviour at the start of every training set. This is a problem, because the neural network learns that thet data should normally have this sudden change in temperatures at the start of every input data set. When real data is shown, where measurements has been at steady-state for hours, the neural network expects a change in temperature, and wrongly predicts the model parameters and initial parameters. The second weakness is that the same input data will be used over and over again when generating the training data. Although the model outputs become different each training set because of different parameter values, the input values have only 396 unique sets of data.

In this project, four hours were chosen as a standard batch-size for the neural network, with a time step of 3 minutes between each data point. More experimentation should be done on this part, with both more and fewer data points for each batch, and shorter and longer sampling times.

Another point in making the training data more representative is to remove the sudden changes in the training set. An easy and quick way of doing this is to simulate the generator model one hour extra for each generated training set, and remove the first hour, such that the temperatures have the time to settle down to steady state, before the training data is "recorded".

The last point of making better training data is to not be confined to only use real measured data as inputs to the model, when generating training data. In addition to the use of measured variable data as inputs, artificially generated input values could be used. With this artificially generated input, all types of input changes could be explored, and it might be that the neural network learns the dynamics of the system much better with for instance step changes and ramp changes in input variables.

From Figure 5.1, the importance of much training data is illustrated. There is no way of knowing where on this graph this project is located, based on the amount of generated training data. It might be that the accuracy is already saturated, or that more data will significantly benefit the accuracy. More experimentation is needed to be able to indicate this.

### 9.3.4 The consistency of parameter predictions

Compared to the Scipy-optimization method, the parameter distributions for the neural networks had much narrower prediction distributions, indicating that the neural networks were consistent in the parameter predictions compared to the Scipy method. Although the neural networks were performing much worse than the Scipy optimization method, Scipy didn't have the same consistent parameter estimations for all the batches of data, and instead came up with widely different solutions every time. The parameter distributions is shown in Appendix E11.

The error shape is unknown in size and shape because it is a six-dimensional shape for the generator model. However, Figure 9.1 attempts to show that when only considering three model parameters at a time, two error shapes can be illustrated. The parameter spaces in the two figures in Figure 9.1 are chosen deliberately. Figure (a) chose the parameters which had the most narrow parameter distributions, and Figure (b) shows the more uncertain parameter distributions. Figure 9.1 (a) shows the error shape considering the parameter space $hA_s^{Fe2a}$, $ml_const$ and $\dot{m}_a$. This has a more defined error shape compared to Figure 9.1 (b), which is the error shape of $hA_r^{Cu2a}$, $hA_s^{Cu2Fe}$ and $M_s^{Fe}$ parameter space. My main hypothesis for this is that there is no measurements on the variables that are

directly affected by these parameters. For instance, rotor temperature can indicate the heat conductivity from rotor copper to the air, stator iron temperature can indicate both the stator iron mass and its heat conductivity to the cooling air.

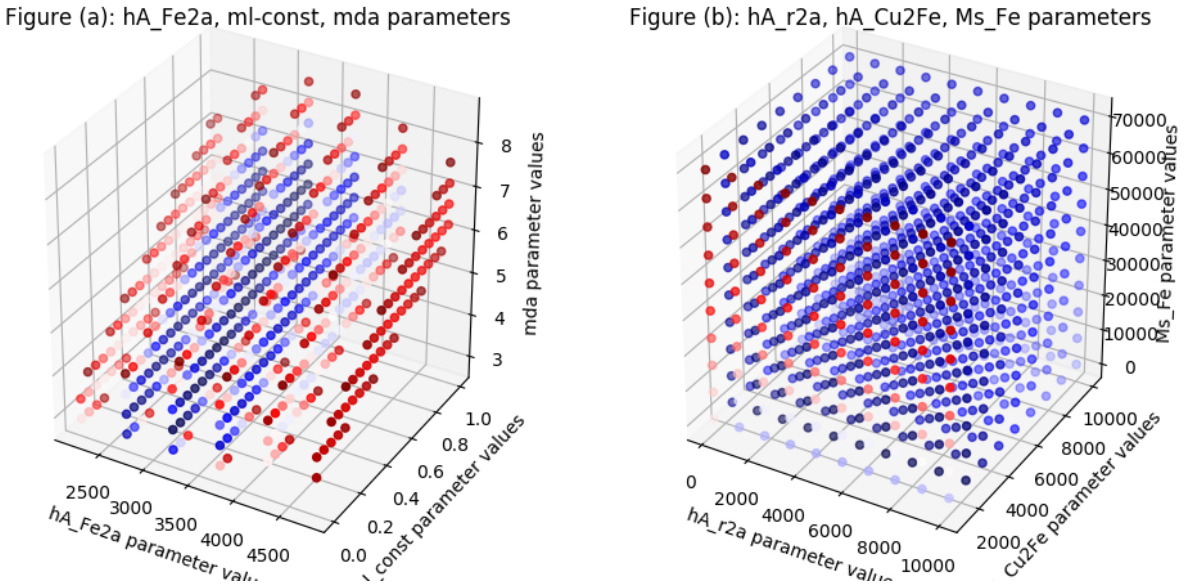



Figure 9.1: Illustration of two error shapes in two different three dimensional parameter spaces.

# 10 Conclusion and further work

The main goal of this project was to assess how model parameters can be predicted in thermal mechanistic models using machine learning. By working towards this goal, basic machine learning theory and thermal physics principles has been introduced. SCADA-data from Grunnaai has been collected and processed such that it can be used for both machine learning and simulation with the thermal mechanistic models. These models has been introduced and parameter estimation has been done in two separate study cases. Study Case 1 was parameter estimation for the thermal heat exchanger model, and Study Case 2 was parameter estimation for the thermal generator model. The findings in this project will be presented in Chapter 10.1, and thoughts of further work is discussed in Chapter 10.2.

## 10.1 Conclusion

This project contained Study Case 1 and Study Case 2 to evaluate the defined heat exchanger and generator thermal models respectively. In both cases, the results from parameter estimation using an optimization algorithm to minimize the error function was that model parameters obtained this way had a wide range of possible parameter solutions with respect to the error function. This indicates that the optimization algorithm found many parameter sets with a low prediction error. The result also showed that the final predicted parameter estimations using this method performed with low prediction error compared to real data.

When using neural networks for parameter estimations, results from Study Case 1 shows that the neural network predicted parameter values consistently with a low prediction error. When evaluating neural network hyperparameters, there were no significant neural network architecture that had any noticeable advantage in this study case, with a basis of 20 neural network architectures evaluated.

In Study Case 2, five different neural network architectures was attempted for predicting the generator model parameters. The best performing architecture consisted of 20 stacked neural networks, which was trained for 2 epochs each. The resulting parameter prediction from this method used the weighted average values of all 20 parameter predictions, with respect to the prediction error. Although this was the best performing neural

network architecture, out of the five presented in this report, the neural network had poor performing parameter predictions. The hypothesis for these results are discussed in Chapter 9. However, when prediction model parameters, the model parameter predictions were usually in a narrow range of values, indicating that the neural network trained with unrepresentative training data. The neural network may have been wrong, but it was consistently wrong.

## 10.2 Further work

Even though the parameter estimations from the neural networks resulted in poor temperature predictions, one should not yet abandon the idea of neural networks predicting model parameters. Results shows that the neural network has consistency in predicting model parameters. With better training data together with more temperature measurements data and a more precise mechanistic thermal model of the generator, the neural network might be able to predict model parameters with much lower prediction error. In the long run, if this approach shows better results, it might be possible to make an algorithm that can generalize to predict generator model parameters for a wide spectrum of generator types.

# Bibliography


[1] *kraftverk – Store norske leksikon*. [Online]. Available: https://snl.no/kraftverk (visited on 11/05/2020).

[2] *Kraftproduksjon - NVE*. [Online]. Available: https://www.nve.no/energiforsyning/kraftproduksjon/?ref=mainmenu%7B%5C&%7Dfbclid=IwAR16OuPV9i94HQd-DxczOGknqbNJU3MDmf7ZsuuEhhflk87S-u%7B%5C_%7DHcYwOVRw (visited on 11/05/2020).

[3] F. Arnesen, *Vannkraft - NVE*, 2015. [Online]. Available: https://www.nve.no/energiforsyning/kraftproduksjon/vannkraft/?ref=mainmenu%20https://www.nve.no/energiforsyning/vannkraft/ (visited on 11/05/2020).

[4] *Historie / Vindkraft i Norge / Vindkraft / Vindportalen - informasjonssiden om vindkraft / Hjem - Vindinfo*. [Online]. Available: https://www.vindportalen.no/Vindportalen-informasjonssiden-om-vindkraft/Vindkraft/Vindkraft-i-Norge/Kart-over-vindkraftprosjekter-i-Norge%20http://www.vindportalen.no/Vindportalen-informasjonssiden-om-vindkraft/Vindkraft/Vindkraft-i-Norge/Historie (visited on 11/05/2020).

[5] P. March, *Flexible Operation of Hydropower Plants*, January. ResearchGate, 2017, pp. 1–76, ISBN: 3002011185.

[6] C. Mejuto, 'Improved lumped parameter thermal modelling of synchronous generators', 2010. [Online]. Available: https://www.era.lib.ed.ac.uk/handle/1842/4612.

[7] G. Traxler-Samek, R. Zickermann and A. Schwery, 'Cooling airflow, losses, and temperatures in large air-cooled synchronous machines', *IEEE Transactions on Industrial Electronics*, vol. 57, no. 1, pp. 172–180, 2010, ISSN: 02780046. DOI: 10.1109/TIE.2009.2031191.

[8] T. Øyvang, J. K. Nøland, R. Sharma, G. J. Hegglid and B. Lie, 'Enhanced Power Capability of Generator Units for Increased Operational Security Using NMPC', *IEEE Transactions on Power Systems*, vol. 35, no. 2, pp. 1562–1571, 2020, ISSN: 15580679. DOI: 10.1109/TPWRS.2019.2944673.

[9] *Power plants - Skagerak Kraft*. [Online]. Available: https://www.skagerakkraft.no/kraftverk%7B%5C_%7D2/ (visited on 11/05/2020).

[10] *Grunnåi - Skagerak Kraft*. [Online]. Available: https://www.skagerakkraft.no/grunnai/category2311.html (visited on 11/05/2020).

[11] *Digital Twin | Siemens*. [Online]. Available: https://www.plm.automation.siemens.com/global/en/our-story/glossary/digital-twin/24465 (visited on 11/05/2020).

[12] T. Welte and J. Foros, *MonitorX - Final report*. 2019, ISBN: 9788243610644.

[13] J. Pyrhönen, T. Jokinen and V. Hrabovcová, *Design of Rotating Electrical Machines*. 2008, pp. 1–512, ISBN: 9780470695166. DOI: 10.1002/9780470740095.

[14] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew and I. Mordatch, 'Emergent tool use from multi-agent interaction', 2019. [Online]. Available: https://openai.com/blog/emergent-tool-use/.

[15] *NC-series - Azure Virtual Machines - Azure Virtual Machines | Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series (visited on 14/05/2020).

[16] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2014.

[17] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017, ISBN: 9781491962299.

[18] *Identifying and eliminating bugs in learned predictive models | DeepMind*. [Online]. Available: https://deepmind.com/blog/article/robust-and-verified-ai (visited on 11/05/2020).

[19] P. Lantz, *Machine Learning with R - Second Edition*. Packt Publishin, 2015.

[20] P. Kordík, 'Machine Learning for Recommender systems — Part 1 ( algorithms , evaluation and cold start )', vol. 1, pp. 1–17, 2018. [Online]. Available: https://medium.com/recombee-blog/machine-learning-for-recommender-systems-part-1-algorithms-evaluation-and-cold-start-6f696683d0ed%20https://getpocket.com/a/read/2212052095.

[21] *AlphaGo Zero: Starting from scratch | DeepMind*. [Online]. Available: https://deepmind.com/blog/article/alphago-zero-starting-scratch (visited on 11/05/2020).

[22] *OpenAI Progress*. [Online]. Available: https://openai.com/progress/ (visited on 11/05/2020).

[23] J. Chen, *Neural Network Definition*, 2019. [Online]. Available: https://www.investopedia.com/terms/n/neuralnetwork.asp (visited on 12/05/2020).

[24] D. Gupta, *Activation Functions | Fundamentals Of Deep Learning*. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/ (visited on 11/05/2020).

[25] C. François, *Keras: The python deep learning library*, 2015. [Online]. Available: https://keras.io/ (visited on 11/05/2020).

[26] M. Mohammadi and S. Das, 'SNN: Stacked Neural Networks', 2016. arXiv: 1605.08512. [Online]. Available: http://arxiv.org/abs/1605.08512.

[27] ASIMOV, *THE NEURAL NETWORK ZOO THE ASIMOV INSTITUTE*, 2017. [Online]. Available: https://www.asimovinstitute.org/neural-network-zoo/%20https://www.asimovinstitute.org/neural-network-zoo/%7B%5C%%7D0Ahttp://www.asimovinstitute.org/neural-network-zoo/ (visited on 11/05/2020).

[28] R. T. Chen, Y. Rubanova, J. Bettencourt and D. Duvenaud, 'Ordinary Differential Equations', *Neural Ordinary Differential Equations*, 2019, ISSN: 20385757. DOI: 10.1007/978-3-662-55774-7_3. arXiv: 1806.07366v5.

[29] Y. Wang, M. Huang, L. Zhao and X. Zhu, 'Attention-based LSTM for Aspect-level Sentiment Classification', pp. 606–615,

[30] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber, 'Lstm: A search space odyssey', *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[31] J. Jordan, 'Setting the learning rate of your neural network.', *Data Science*, no. March 2018, pp. 1–30, 2018. [Online]. Available: https://www.jeremyjordan.me/nn-learning-rate/.

[32] T. Domhan, J. Springenberg and F. Hutter, *Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves*, 2015. [Online]. Available: https://www.aaai.org/ocs/index.php/IJCAI/IJCAI15/paper/view/11468/11222.

[33] 'Second Law of Thermodynamics', in, 2017, pp. 181–200. DOI: 10.1142/9789813224339_0007. [Online]. Available: https://www.grc.nasa.gov/WWW/K-12/airplane/thermo2.html.

[34] A. T. Olson and K. A. Shelstad, 'Convective heat transfer.', 1987, ISSN: 00179310. DOI: 10.1016/0017-9310(85)90062-6. [Online]. Available: https://www.engineeringtoolbox.com/convective-heat-transfer-d%7B%5C_%7D430.html.

[35] P. Koštial, I. Špička, Z. Jančíková, J. David, J. Valícek, M. Harnicarova and V. Rusnák, 'Lumped capacitance model in thermal analysis of solid materials', *Journal of Physics: Conference Series*, vol. 588, p. 012 006, Feb. 2015. DOI: 10.1088/1742-6596/588/1/012006.

[36] *Time Constant | Definition of Time Constant by Merriam-Webster*. [Online]. Available: https://www.merriam-webster.com/dictionary/time%20constant (visited on 11/05/2020).

[37] B. Lie, *Modeling of dynamic systems*. 2019. DOI: 10.1016/0967-0661(95)90022-5.

[38] *Email from ingunn granstrøm, title: Sensor datablader*, Apr. 2020.

[39] PROCESSPARAMETERS, *What is a PT100 sensor working principle? Sensors UK manufactured*, 2019. [Online]. Available: https://www.processparameters.co.uk/pt100-sensor-working-principle/ (visited on 12/05/2020).

[40] Chris Woodford, *How do heat exchangers work? - Explain that Stuff*, 2018. [Online]. Available: https://www.explainthatstuff.com/how-heat-exchangers-work.html (visited on 11/05/2020).

[41] *Specific Heat of some common Substances*, 2015. [Online]. Available: https://www.engineeringtoolbox.com/specific-heat-capacity-d%7B%5C_%7D391.html.

[42] *Methods of mathematical modelling*. 1995, pp. 49–131, ISBN: 9783319230412. DOI: 10.1007/978-3-0348-7318-5_2.

[43] S. Luo, *Optimization: Loss Function Under the Hood (Part I)*. [Online]. Available: https://towardsdatascience.com/optimization-of-supervised-learning-loss-function-under-the-hood-df1791391c82 (visited on 11/05/2020).

[44] B. Rohrer, *Choosing an error function*. [Online]. Available: https://e2eml.school/how%7B%5C_%7Dmodeling%7B%5C_%7Dworks%7B%5C_%7D3.html (visited on 11/05/2020).

[45] *SCADA - Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/SCADA (visited on 11/05/2020).

[46] *What is an Avro file?*, 2018. [Online]. Available: https://fileinfo.com/extension/avro%20https://fileinfo.com/extension/sbsar%7B%5C%%7D0Ahttps://fileinfo.com/extension/dicom (visited on 11/05/2020).

[47] *scipy.interpolate.PchipInterpolator — SciPy v1.4.1 Reference Guide*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.PchipInterpolator.html%7B%5C#%7Dscipy.interpolate.PchipInterpolator (visited on 11/05/2020).

[48] *pandas.Series.interpolate — pandas 1.0.3 documentation*. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.interpolate.html (visited on 11/05/2020).

[49] *pandas.DataFrame.resample — pandas 1.0.3 documentation*. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.resample.html (visited on 11/05/2020).

[50] *Simplex Stator Winding RTDs | RTD PT100 & Bifilar Wound Slot Resistance Thermometer*. [Online]. Available: https://www.technocontrols.com/simplex-stator-winding.html (visited on 11/05/2020).

[51] *sklearn.preprocessing.MinMaxScaler — scikit-learn 0.22.2 documentation*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html (visited on 11/05/2020).

[52] R. H. Byrd, P. Lu, J. Nocedal and C. Zhu, 'A Limited Memory Algorithm for Bound Constrained Optimization', *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, 1995, ISSN: 1064-8275. DOI: 10.1137/0916069.

[53] 'Uninformed Search', [Online]. Available: https://cs.lmu.edu/%7B~%7Dray/notes/usearch/.

[54] A. Rosebrock, *Why is my validation loss lower than my training loss? - PyImageSearch*. [Online]. Available: https://www.pyimagesearch.com/2019/10/14/why-is-my-validation-loss-lower-than-my-training-loss/ (visited on 11/05/2020).

[55] H. Alonso, T. Mendonça and P. Rocha, 'Hopfield neural networks for on-line parameter estimation', *Neural Networks*, vol. 22, no. 4, pp. 450–462, 2009, ISSN: 08936080. DOI: 10.1016/j.neunet.2009.01.015. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2009.01.015.

# Appendix A

# Task Description of the Masters Thesis

This appendix includes the main goal and objectives for this thesis.

# FMH606 Master's Thesis

**Title:** Thermal and Electrical modelling of Grunnåi hydro generator through machine learning

**USN Supervisors:** Main supervisor: Thomas Øyvang, co-supervisors: Ola Marius Lysaker and Bernt Lie

**External Partners:** Ingunn Granstrøm (Skagerak Kraft)

**Task background:**

Skagerak Energi has recently put into place a new 12 MVA synchronous generator at Grunnåi (Seljord). The intention is that this hydropower plant is used as an asset for research in some periods during the years of operation. Moreover, Skagerak Kraft wants to develop a digital twin of the machine for optimal utilization and flexible operation. There is a large amount of sensor data soon available from the machine through cloud-based storage. Complex problems involving a large amount of data and many variables makes Machine Learning (ML) an interesting approach.

This thesis work is an investigation on if/how machine learning can be used to improve thermal and electrical models of the hydroelectric generator. The project should explore the possibility of a hybrid-model (both mechanistic and data-driven). This implies considering the mechanistic models and combine them with the available sensor data to possibly give an estimate on model deviations and errors.

**Task objectives:**

- Do a survey on state of the art machine learning techniques for developing time-dependent models.
- Describe temperature measurements and sensor types used in Grunnåi hydro power plant.
- Explore the thermal and electrical physics of a hydroelectric generator. The architecture and controller methods for the machine should also be understood.
- Prepare mechanistic models for the thermal and electrical parts of Grunnåi's 12 MVA generator. These models will be used to generate training data for the neural network models.
- Explore the different architectures for the neural networks when making the electrical and thermal data-driven models. Hyperparameters must also be considered.
- Use the resources provided by Skagerak Energi to extract data from Azure Databricks, where operational data of Grunnåi's 12 MVA generator is stored. This data shall be used to train data-driven thermal and electrical models.
- Propose a data-driven model of the electrical control system of the hydroelectric generator, based on machine learning technology.
- Propose a data-driven thermal model of the generator based on machine learning technology.
- Document the results and findings in the master's thesis report, and possibly in a suitable conference/journal paper.

**Supervision:**

Generally, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

**Signatures**:

Student (write clearly in all capitalized letters):

EMIL MELFALD

Supervisor (date and signature):

31.01.202

Thomas Ølyreng

Student (date and signature):

31.01.2020

Emil Melfald

# Appendix B

# Minutes of meeting, second project meeting 28.04.2020

In this minutes of meeting, the task description was formally changed to not include the electrical model in this thesis.

Universitetet i Sørøst-Norge Porsgrunn 28.04.2020
Fakultet for teknologi, naturvitenskap og maritime fag

Til: Bernt Lie, Ola Marius Lysaker, Thomas Øyvang, Ingunn Granstrøm

**MØTEREFERAT TIL PROSJEKTMØTE NR 2**

**Prosjekttittel: Thermal model parameter estimation of a hydro generator using machine learning**

**Hovedveileder:** Thomas Øyvang

**Biveiledere:** Bernt Lie, Ola Marius Lysaker

**Sted: Skype-møte**
**Tid og varighet:** 28.04.2020 kl. 12:00 – 13:00 (1 time)

Sak 06/2020 Innledningssaker

1. Godkjenning av møteinnkalling – Godkjent med kommentar: Ta med tittel på rapporten og navn på veiledere.
2. Godkjenning av saksliste - Godkjent
3. Saker til Eventuelt – 1 sak: Oppgavebeskrivelsen

Sak 07/2020 Fremdrift så langt

1. Rapporten skal prøveinnleveres til Thomas 30.04.2020
2. Maskinlæring er ferdiggjort for prosjektet

Sak 08/2020 Resultater fra maskinlæring

1. Noen fremgangsmåter og resultater er vist i separat presentasjon.

Sak 05/2020 Eventuelt

1. Oppgavebeskrivelse: Oppgavebeskrivelsen ble skrevet ved prosjektstart, og på grunn av tid og data så vil følgende punkter endres/fjernes fra målene til rapporten:

- Explore the thermal ~~and electrical~~ physics of a hydroelectric generator. The architecture and controller methods for the machine should also be understood.
- Use the resources provided by Skagerak Energi to extract data from Azure Databricks, where operational data of Grunnåi's 12 MVA generator is stored. This data shall be used to train data-driven thermal ~~and electrical~~ models.
- ~~X~~ Propose a data-driven model of the electrical control system of the hydroelectric generator, based on machine learning technology.

Møteleder og referent:
Emil Melfald

# Appendix C

# Heat Exchanger model analysis in Python

This appendix contains the heat exchanger model in python code, together with some step responses and sensitivity analysis of the model.

databricks5.1 - Heat Exchanger Analysis

# Heat Exchanger model analysis

In this notebook, a small verification will be done for the heat exchanger model to verify that the model inputs, outputs and parameters work as intended. There will be two parts of this notebook, where part 1 is the input step responses and part 2 is a small sensitivity analysis of the model parameters.

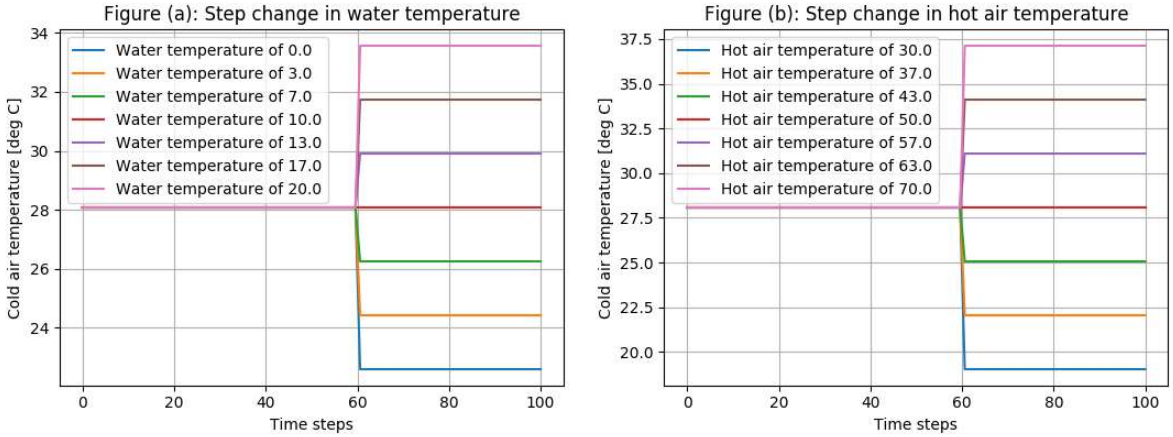
```
import numpy as np
import scipy as sp
from scipy.optimize import minimize, fsolve
from math import exp
import datetime
import seaborn
import pandas as pd
import matplotlib.pyplot as plt


def load_dataframe(path):
  df_load = spark.read.csv(path, header=True)
  df_load = df_load.toPandas() #Converts spark.dataframe to a pandas
dataframe
  df_load['Time'] = pd.to_datetime(df_load['Time']) #Converts the timestamp
from string to a datetime
  df_load = df_load.set_index('Time').sort_index().reset_index()

  try:
    df_load.set_index(['Batches', 'Time'], inplace=True) #Sets the batches
and timestamps as index
  except:
    df_load.set_index('Time', inplace=True) #Sets the timestamps as index

  df_load = df_load.astype(np.float16) #Converts all values from string to
float.
  #df_load.sort_index(inplace=True) #Sorts the dataframe before returning
it.
  return df_load

def save_dataframe(df, path):
  df = df.reset_index() #Spark dataframes has no index, so if index is not
reset, then information is lost (?)
  df = spark.createDataFrame(df)
  df.write.csv(path, header=True) #Writes all data to a .csv, including
headers.
```

```
class HeatExchanger:
  def __init__(self):
    #Inputs:         u = [Tw_c, Ta_h]
    #parameters: theta = [uAx,  mda,  mdw]
    self.chp_w = 4200
    self.chp_a = 1150
    self.A = lambda th: th[0]/(self.chp_w*th[2]) #Constant 1
    self.B = lambda th: th[0]/(self.chp_a*th[1]) #Constant 2
    self.Ta_c = lambda u, th: ((self.A(th) - self.B(th))*u[1] -
(self.B(th)*np.exp(self.B(th) - self.A(th)) - self.B(th))*u[0]) / (
    self.A(th) - self.B(th)*np.exp(self.B(th) - self.A(th))) #Cold air
temperature calculation

  def give_solution(self, Ta_c_real): #Give measurement data for cold air
temp
    self.Ta_c_real = Ta_c_real

  def give_inputs(self, u): #u = [Tw_c, Ta_h]
    self.u = u

  def change_params(self, theta): #theta = [uAx,  mda,  mdw]
    self.theta = theta

  def calc_output(self): #y = Ta_c
    return np.array(self.Ta_c(self.u, self.theta))

  def prediction_error(self, theta): #This is the objective function for the
optimizaiton
    self.change_params(theta)
    Ta_c = self.calc_output()
    mpe = np.sum(np.square(np.subtract(Ta_c, self.Ta_c_real)))/len(Ta_c)
#Mean squared prediction error
    return mpe

  def find_optimal_parameters(self, u, y_sol, theta_0, theta_bounds):
#Optimization with theta = [uAx, mda, mdw]
    self.give_inputs(u)
    self.give_solution(y_sol)
    sol = minimize(self.prediction_error, theta_0, bounds=theta_bounds)
    return sol.x
```

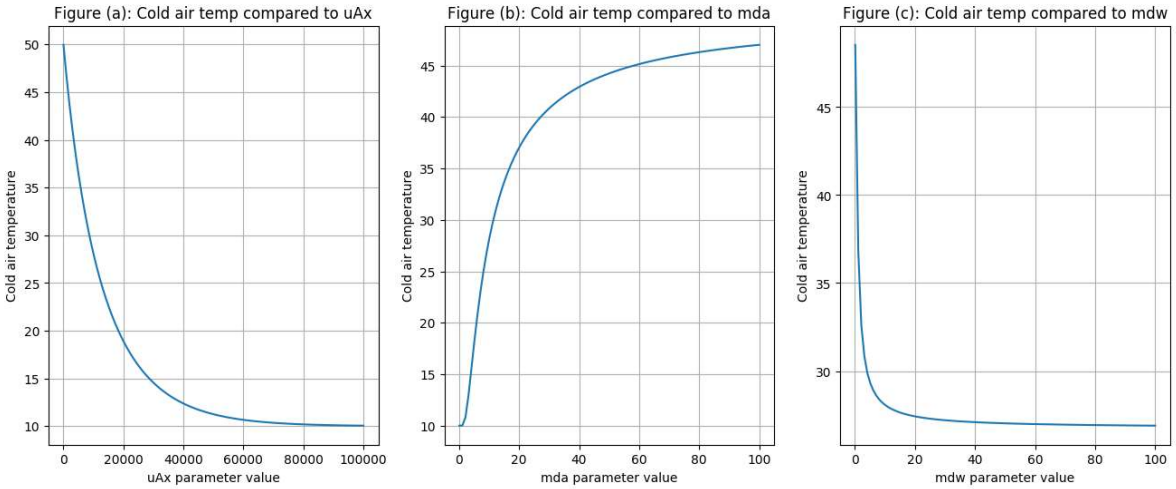
# Part 1: Input responses

```
#Some model parameters:
theta = [10000, 10, 10]
HE1 = HeatExchanger()
HE1.change_params(theta)

#Define the steps:
Ta_h_steps = np.linspace(-20, 20, 7)
Tw_c_steps = np.linspace(-10, 10, 7)

Ta_h_original = 50
Tw_c_original = 10

n = 100 #Number of steps to plot
n_step = 40 #Where the step change will occure

t = np.linspace(0, n, n)
fig, ax = plt.subplots(1,2)

for Ta_h_step, Tw_c_step in zip(Ta_h_steps, Tw_c_steps):
  #Calculate for water step change:
  Tw_c1 = np.ones(n-n_step)*Tw_c_original
  Tw_c2 = np.ones(n_step)*(Tw_c_original + Tw_c_step)
  Tw_c = np.append(Tw_c1, Tw_c2)

  Ta_h = np.ones_like(Tw_c)*Ta_h_original
  HE1.give_inputs(np.array([Tw_c, Ta_h]))
  Ta_c_w = HE1.calc_output()

  #Calculate for air step change:
  Ta_h1 = np.ones(n-n_step)*Ta_h_original
  Ta_h2 = np.ones(n_step)*(Ta_h_original + Ta_h_step)
  Ta_h = np.append(Ta_h1, Ta_h2)

  Tw_c = np.ones_like(Ta_h)*Tw_c_original
  HE1.give_inputs(np.array([Tw_c, Ta_h]))
  Ta_c_a = HE1.calc_output()

  ax[0].plot(t, Ta_c_w, label='Water temperature of
{}'.format(round(Tw_c_original + Tw_c_step),2))
  ax[0].set_xlabel('Time steps')
  ax[0].set_ylabel('Cold air temperature [deg C]')
  ax[0].set_title('Figure (a): Step change in water temperature')
  ax[0].legend(loc=2)
  ax[0].grid()

  ax[1].plot(t, Ta_c_a, label='Hot air temperature of
{}'.format(round(Ta_h_original + Ta_h_step), 2))
  ax[1].set_xlabel('Time steps')
  ax[1].set_ylabel('Cold air temperature [deg C]')
  ax[1].set_title('Figure (b): Step change in hot air temperature')
```

```
  ax[1].legend(loc=2)
  ax[1].grid()
fig.set_figwidth(13)
fig.set_figheight(4.2)
display(fig)
```




# Sensitivity analysis on model parameters



# Appendix D

# Generator model analysis in Python

This appendix contains the generator thermal model in python code, together with some step responses and sensitivity analysis of the model.

databricks8 - Generator model analysis

# Generator model

The generator model will in first iteration be simple. It is a lumped parameter model with:

states x = (Tr_Cu, Ts_Fe, 'Ts_Cu')

outputs y = (Tr_Cu, Ts_Fe, Ts_Cu, Ta_h)

inputs = (Ta_c, Qd_r, Qd_s)

parameters theta = (hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Cr_Cu, Cs_Fe, Cs_Cu, ml_const, mda, Tr_Cu0, Ts_Fe0, Ts_Cu0)

constants = (chp_a, chp_Cu, chp_Fe)

This notebook wil make the generator model class, and do analysis to make sure that the dynamics works as intended.

1) Create a generator class model.

2) Analyze responses for step changes in inputs

3) Do sensitivity analysis on the model parameters

```
from scipy.integrate import solve_ivp

import numpy as np
import scipy as sp
from scipy.optimize import minimize, fsolve
from math import exp
import datetime
import seaborn
import pandas as pd
import matplotlib.pyplot as plt

#import sympy
#from sympy import solve
#from sympy import Symbol

def load_dataframe(path):
  df_load = spark.read.csv(path, header=True)
  df_load = df_load.toPandas() #Converts spark.dataframe to a pandas
dataframe
  df_load['Time'] = pd.to_datetime(df_load['Time']) #Converts the timestamp
from string to a datetime
  df_load = df_load.set_index('Time').sort_index().reset_index()

  try:
    df_load.set_index(['Batches', 'Time'], inplace=True) #Sets the batches
and timestamps as index
  except:
    df_load.set_index('Time', inplace=True) #Sets the timestamps as index

  df_load = df_load.astype(np.float16) #Converts all values from string to
float.
  #df_load.sort_index(inplace=True) #Sorts the dataframe before returning
it.
  return df_load

def save_dataframe(df, path):
  df = df.reset_index() #Spark dataframes has no index, so if index is not
reset, then information is lost (?)
  df = spark.createDataFrame(df)
  df.write.csv(path, header=True) #Writes all data to a .csv, including
headers.
```

# 1: The generator class model

```
#!!!!!!!!LATEST MODEL!!!!!!!!!!

#The generator model:
#Inputs: u_in = [Ta_c, Qd_r, Qd_s]
#Outputs u_out = [Tr_Cu, Ts_Fe, Ts_Cu, Ta_h]
#Parameters theta = [hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Ms_Fe, ml_const, mda,
Tr_Cu0, Ts_Fe0, Ts_Cu0] - 9 parameters (instead of 11)
#Estimated parameters: c = [Qd_Fe, Qd_mech, Mr_Cu, Ms_Cu, mda]

#From the heat run test, estimated Qd_Fe = 37888, estimated Qd_mech = 53696
#From the heat exchanger optimization, mda = 30.43 kg/s
#From Generator Data sheet, the different masses are:
#Mr_Cu = 369 kg copper
#Ms_Cu = 1552 kg copper
#Ms_Fe = ???

#

class GeneratorModel:
  def __init__(self, dt, c):
    #-------------------------------------------------------------------------
------------------------------
    #Configure the constants
    self.chp_a = 1150 # Specific heat capacity of air [J/kg]
    self.chp_Cu = 385 # Specific heat capacity of copper [J/kg]
    self.chp_Fe = 444 # Specific heat capacity of iron [J/kg]

    #Estimated parameters:
    self.Qd_Fe = c[0]
    self.Qd_mech = c[1]
    self.Cr_Cu = c[2]*self.chp_Cu
    self.Cs_Cu = c[3]*self.chp_Cu

    self.dt = dt #Length timesteps taken throughout the data [s] -- May need
to convert this to a smaller number for calculations, and return the same
timestep later.
    #-------------------------------------------------------------------------
------------------------------
    #Configure the equations
    #States: x = [Tr_Cu, Ts_Fe, Ts_Cu]
    #Outputs: y = [Tr_Cu, Ts_Fe, Ts_Cu, Ta_h] - 4 outputs (2 can be
verified)
    #Inputs: u = [Ta_c, Qd_r, Qd_s] - 3 inputs
    #Params: theta = [hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Ms_Fe, ml_const, mda,
Tr_Cu0, Ts_Fe0, Ts_Cu0] - 9 parameters (instead of 11)

    #Generator air equations:
    self.Ta_h = lambda x, u, theta: (self.Qd_r2a(x, u, theta) +
self.Qd_s2a(x, u, theta) + self.Qd_f(x, u, theta) +
u[0]*theta[5]*self.chp_a) / (theta[5]*self.chp_a)
```

```
    self.Qd_r2a = lambda x, u, theta: theta[0]*(x[0]-u[0])
    self.Qd_s2a = lambda x, u, theta: theta[2]*(x[1]-u[0])
    self.Qd_f = lambda x, u, theta: theta[4]*self.Qd_mech #The heat flow
from bearings to the cooling air

    #Generator Rotor copper equation:
    self.dTr_Cu_dt = lambda x, u, theta: (u[1] - self.Qd_r2a(x, u, theta)) /
self.Cr_Cu

    #Generator Stator iron equation:
    self.dTs_Fe_dt = lambda x, u, theta: (self.Qd_Fe + self.Qd_s_Cu2Fe(x, u,
theta) - self.Qd_s2a(x, u, theta)) / (theta[3]*self.chp_Fe)
    self.Qd_s_Cu2Fe = lambda x, u, theta: theta[1]*(x[2] - x[1])

    #Generator Stator copper equations:
    self.dTs_Cu_dt = lambda x, u, theta: (u[2] - self.Qd_s_Cu2Fe(x, u,
theta)) / self.Cs_Cu

  def change_parameters(self, theta): #theta in this case are: theta =
[hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Ms_Fe, ml_const, Tr_Cu0, Ts_Fe0, Ts_Cu0]
    self.theta = theta.copy()

  def give_inputs(self, u):
    self.u = u

  def give_solution(self, y_sol): # y_sol = [Ts_Cu, Ta_h] measurements
    self.y_sol = y_sol


  def sys_ODEs(self, t, x): #Collects all the differential equations to a
signle method that can be handled by the ODE solver from scipy
    u = self.u_current #Uses this timestep input values
    theta = self.theta

    dTr_Cu_dt = self.dTr_Cu_dt(x, u, theta)
    dTs_Cu_dt = self.dTs_Cu_dt(x, u, theta)
    dTs_Fe_dt = self.dTs_Fe_dt(x, u, theta)
    return dTr_Cu_dt, dTs_Fe_dt, dTs_Cu_dt


  def calc_one_ts(self, u_ts, theta):
    #Given initial values, input variables for this given timestep and
parameter values, this function solves the ODE's and returns the solved
value array
    x0 = theta[-3:].copy()
    t_span = (0, self.dt)
    self.u_current = u_ts
    self.change_parameters(theta)
    sol = solve_ivp(self.sys_ODEs, t_span, x0)
    x1 = sol.y[:, -1] #Gets the last calculated values in the timespan
    Ta_h = self.Ta_h(x1, u_ts, theta)
```

```
    y = np.append(x1, Ta_h)
    return y


  def calc_batch_output(self, u, theta):
    y =[]
    self.change_parameters(theta)
    theta_calc = self.theta.copy()
    x0 = theta_calc[-3:]

    Ta_h = self.Ta_h(x0, u[0], theta_calc)
    y0 = np.append(x0, Ta_h)
    y.append(y0)

    for u_in in u:
      theta_calc[-3:] = x0
      y_sol = self.calc_one_ts(u_in, theta_calc)
      x0 = y_sol[0:-1]
      y.append(y_sol)
    y = np.array(y)
    return y[:-1]


  def find_optimal_parameters(self, u, y_sol, theta0, theta_bounds):
    #y_sol = [Tr_Cu, Ta_h]
    self.give_inputs(u)
    self.give_solution(y_sol)
    sol = minimize(self.squared_error, theta0, bounds=theta_bounds)
    return sol.x


  def squared_error(self, theta_opt): #This function is to be minimized.
    y_est = self.calc_batch_output(self.u, theta_opt).transpose() #Outputs:
y = [Tr_Cu, Ts_Fe, Ts_Cu, Ta_h]
    error = (np.square(y_est[2] - self.y_sol[0]) + np.square(y_est[3] -
self.y_sol[1]))/len(y_est[0])
    return np.sum(error)


  def find_optimal_parameters2(self, data, theta0, theta_bounds): #Now u is
the dataframe with inputs, still divided into batches. The same goes for
y_sol.
    self.data = data
    sol = minimize(self.squared_batched_error, theta0, bounds=theta_bounds)
    return sol.x


  def squared_batched_error(self, theta):
    n = len(set(self.data.reset_index()['Batches']))
    batches = ['Batch {}'.format(i) for i in range(1, 10)]
    error = []
```

```
    for batch in batches:
      #Inputs
      Qd_r = self.data.loc[[batch], :]['Rotor Copper Losses [W]'].values
      Qd_s = self.data.loc[[batch], :]['Stator Copper Losses [W]'].values
      Ta_c = self.data.loc[[batch], :]['Kaldluft'].values
      #Outputs solution
      Tr_Cu = self.data.loc[[batch], :]['Avg Stator Temp [deg C]'].values
      Ta_h = self.data.loc[[batch], :]['Varmluft'].values


      u = np.array([Qd_r, Qd_s, Ta_c]).transpose()
      y_est = self.calc_batch_output(u, theta).transpose() #Outputs:
y.transpose() = [Tr_Cu, Ts_Fe, Ts_Cu, Ta_h]
      error.append(np.sum(np.square(y_est[0] - Tr_Cu) + np.square(y_est[3] -
Ta_h)))
    return sum(error)
```

```
class GeneratorModel:
  def __init__(self, dt, Qd_Fe, Qd_mech):
    #-------------------------------------------------------------------------
------------------------------
    #Configure the constants
    self.chp_a = 1150 # Specific heat capacity of air [J/kg]
    self.chp_Cu = 385 # Specific heat capacity of copper [J/kg]
    self.chp_Fe = 444 # Specific heat capacity of iron [J/kg]
    self.Qd_Fe = Qd_Fe
    self.Qd_mech = Qd_mech
    self.dt = dt #Length timesteps taken throughout the data [s] -- May need
to convert this to a smaller number for calculations, and return the same
timestep later.
    #-------------------------------------------------------------------------
------------------------------
    #Configure the equations
    #States: x = [Tr_Cu, Ts_Fe, Ts_Cu]
    #Outputs: y = [Tr_Cu, Ts_Fe, Ts_Cu, Ta_h] - 4 outputs (2 can be
verified)
    #Inputs: u = [Ta_c, Qd_r, Qd_s] - 3 inputs
    #Params: theta = [hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Cr_Cu, Cs_Fe, Cs_Cu,
ml_const, mda, Tr_Cu0, Ts_Fe0, Ts_Cu0] - 11 parameters

    #Generator air equations:
    self.Ta_h = lambda x, u, theta: (self.Qd_r2a(x, u, theta) +
self.Qd_s2a(x, u, theta) + self.Qd_f(x, u, theta) +
u[0]*theta[7]*self.chp_a) / (theta[7]*self.chp_a)
    self.Qd_r2a = lambda x, u, theta: theta[0]*(x[0]-u[0])
    self.Qd_s2a = lambda x, u, theta: theta[2]*(x[1]-u[0])
    self.Qd_f = lambda x, u, theta: theta[6]*self.Qd_mech #The heat flow
from bearings to the cooling air

    #Generator Rotor copper equation:
    self.dTr_Cu_dt = lambda x, u, theta: (u[1] - self.Qd_r2a(x, u, theta)) /
(theta[3])

    #Generator Stator iron equation:
    self.dTs_Fe_dt = lambda x, u, theta: (self.Qd_Fe + self.Qd_s_Cu2Fe(x, u,
theta) - self.Qd_s2a(x, u, theta)) / theta[4]
    self.Qd_s_Cu2Fe = lambda x, u, theta: theta[1]*(x[2] - x[1])

    #Generator Stator copper equations:
    self.dTs_Cu_dt = lambda x, u, theta: (u[2] - self.Qd_s_Cu2Fe(x, u,
theta)) / (theta[5])

  def change_parameters(self, theta): #theta in this case are: theta =
[hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Cr_Cu, Cs_Fe, Cs_Cu, ml_const, mda, Tr_Cu0,
Ts_Fe0, Ts_Cu0]
    self.theta = theta.copy()
```

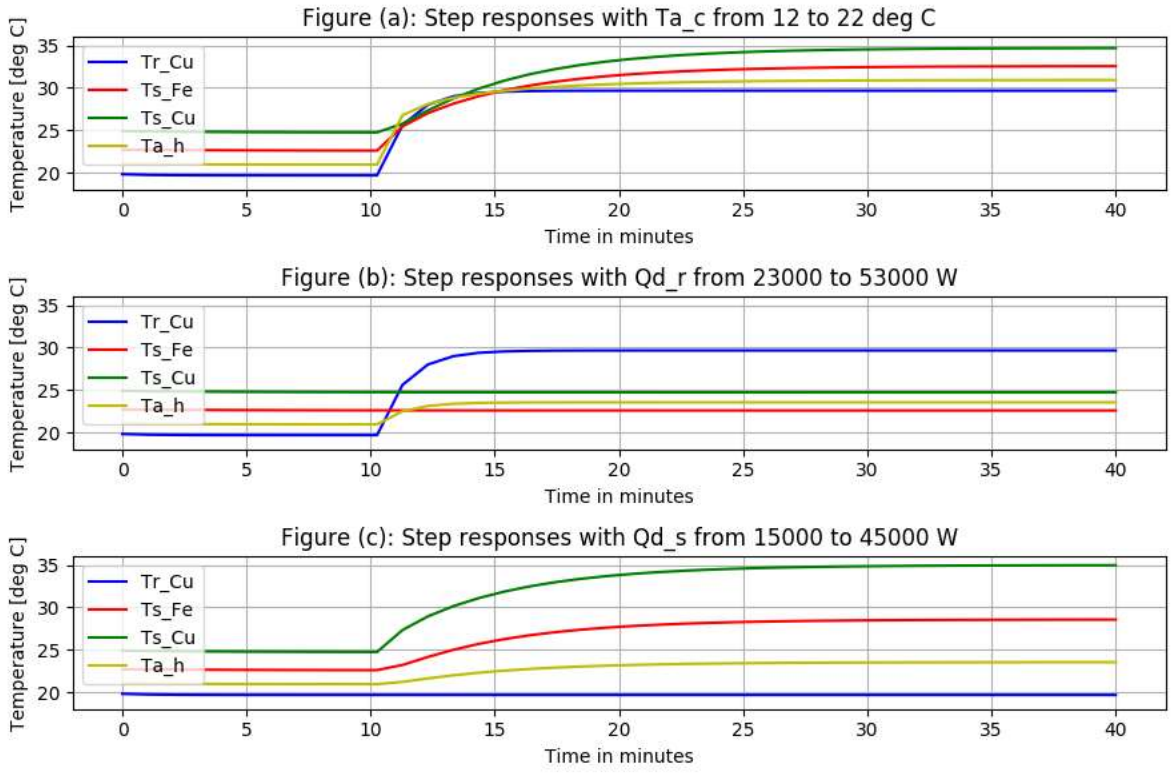
```
  def give_inputs(self, u):
    self.u = u

  def give_solution(self, y_sol): # y_sol = [Ts_Cu, Ta_h] measurements
    self.y_sol = y_sol


  def sys_ODEs(self, t, x): #Collects all the differential equations to a
signle method that can be handled by the ODE solver from scipy
    u = self.u_current #Uses this timestep input values
    theta = self.theta

    dTr_Cu_dt = self.dTr_Cu_dt(x, u, theta)
    dTs_Cu_dt = self.dTs_Cu_dt(x, u, theta)
    dTs_Fe_dt = self.dTs_Fe_dt(x, u, theta)
    return dTr_Cu_dt, dTs_Fe_dt, dTs_Cu_dt


  def calc_one_ts(self, u_ts, theta):
    #Given initial values, input variables for this given timestep and
parameter values, this function solves the ODE's and returns the solved
value array
    x0 = theta[-3:].copy()
    t_span = (0, self.dt)
    self.u_current = u_ts
    self.change_parameters(theta)
    sol = solve_ivp(self.sys_ODEs, t_span, x0)
    x1 = sol.y[:, -1] #Gets the last calculated values in the timespan
    Ta_h = self.Ta_h(x1, u_ts, theta)
    y = np.append(x1, Ta_h)
    return y


  def calc_batch_output(self, u, theta):
    y =[]
    self.change_parameters(theta)
    theta_calc = self.theta.copy()
    x0 = theta_calc[-3:]

    Ta_h = self.Ta_h(x0, u[0], theta_calc)
    y0 = np.append(x0, Ta_h)
    y.append(y0)

    for u_in in u:
      theta_calc[-3:] = x0
      y_sol = self.calc_one_ts(u_in, theta_calc)
      x0 = y_sol[0:-1]
      y.append(y_sol)
    y = np.array(y)
    return y[:-1]
```

```
  def find_optimal_parameters(self, u, y_sol, theta0, theta_bounds):
    self.u = u
    self.y_sol = y_sol
    sol = minimize(self.least_squared_error, theta0, bounds=theta_bounds)
    return sol.x


  def least_squared_error(self, theta_opt):
    y_est = calc_batch_output(self.u, theta_opt).transpose() #Outputs: y =
[Tr_Cu, Ts_Fe, Ts_Cu, Ta_h]
    error = np.square(y_est[0] - self.y_sol[0]) + np.square(y_est[3] -
self.y_sol[1])
    return error



data =
load_dataframe("/mnt/output/emil_data/prepared_data/collected_data.csv")
```

# 2: Input step responses

```
#theta =     [hAr_Cu2a, hAs_Cu2Fe, hAs_Fe2a, Cr_Cu, Cs_Fe, Cs_Cu, ml_const,
mda, Tr_Cu0, Ts_Fe0, Ts_Cu0]
theta_test = [3000,        7000,     5000,  200000, 700000, 500000, 0.5,
10,   19.8,   22.7,   24.9] # These parameter values are the basis of
analysis.
n_start = 10
n_response = 30

Ta_c_test = np.ones(n_start)*12     # Input value constant
Qd_r_test = np.ones(n_start)*23000  # Input value constant
Qd_s_test = np.ones(n_start)*15000  # Input value constant
Ta_c_test2 = np.ones(n_response)*12     # Input value constant
Qd_r_test2 = np.ones(n_response)*23000  # Input value constant
Qd_s_test2 = np.ones(n_response)*15000  # Input value constant

Qd_Fe = 37888
Qd_mech = 53696
GM1 = GeneratorModel(60, Qd_Fe, Qd_mech)
GM1.change_parameters(theta_test)

#Base Inputs:
Ta_c_base = np.append(Ta_c_test, Ta_c_test2)
Qd_r_base = np.append(Qd_r_test, Qd_r_test2)
Qd_s_base = np.append(Qd_s_test, Qd_s_test2)
u_base = np.array([Ta_c_base, Qd_r_base, Qd_s_base],
dtype=np.float32).transpose()
y_base = GM1.calc_batch_output(u_base, theta_test).transpose()

#Step in Ta_c
Ta_c_test_1 = np.append(Ta_c_test, Ta_c_test2+10)
u_test_1 = np.array([Ta_c_test_1, Qd_r_base, Qd_s_base],
dtype=np.float32).transpose()
y_test_1 = GM1.calc_batch_output(u_test_1, theta_test).transpose()

#Step in Qd_r
Qd_r_test_2 = np.append(Qd_r_test, Qd_r_test2+30000)
u_test_2 = np.array([Ta_c_base, Qd_r_test_2, Qd_s_base],
dtype=np.float32).transpose()
y_test_2 = GM1.calc_batch_output(u_test_2, theta_test).transpose()

#Step in Qd_s
Qd_s_test_3 = np.append(Qd_s_test, Qd_s_test2+30000)
u_test_3 = np.array([Ta_c_base, Qd_r_base, Qd_s_test_3],
dtype=np.float32).transpose()
y_test_3 = GM1.calc_batch_output(u_test_3, theta_test).transpose()


#Plotting:
n = n_start + n_response
t = np.linspace(0, n, n)
```

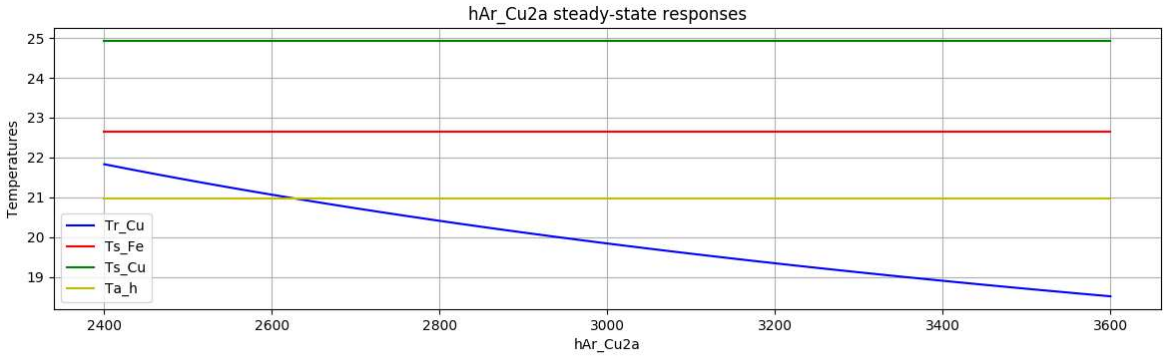
```
y = [y_test_1, y_test_2, y_test_3]

titles = ['Figure (a): Step responses with Ta_c from 12 to 22 deg C',
'Figure (b): Step responses with Qd_r from 23000 to 53000 W', 'Figure (c):
Step responses with Qd_s from 15000 to 45000 W']
fig, ax = plt.subplots(3,1)
for i, y_sim in enumerate(y):
  ax[i].set_title(titles[i])
  ax[i].plot(t, y_sim[0], 'b', label='Tr_Cu')
  ax[i].plot(t, y_sim[1], 'r', label='Ts_Fe')
  ax[i].plot(t, y_sim[2], 'g', label='Ts_Cu')
  ax[i].plot(t, y_sim[3], 'y', label='Ta_h')
  ax[i].legend(loc='upper left')
  ax[i].set_xlabel('Time in minutes')
  ax[i].set_ylabel('Temperature [deg C]')
  ax[i].set_ylim((18, 36))
  ax[i].grid()

fig.set_figheight(6)
fig.set_figwidth(9)
fig.tight_layout()
display(fig)
```

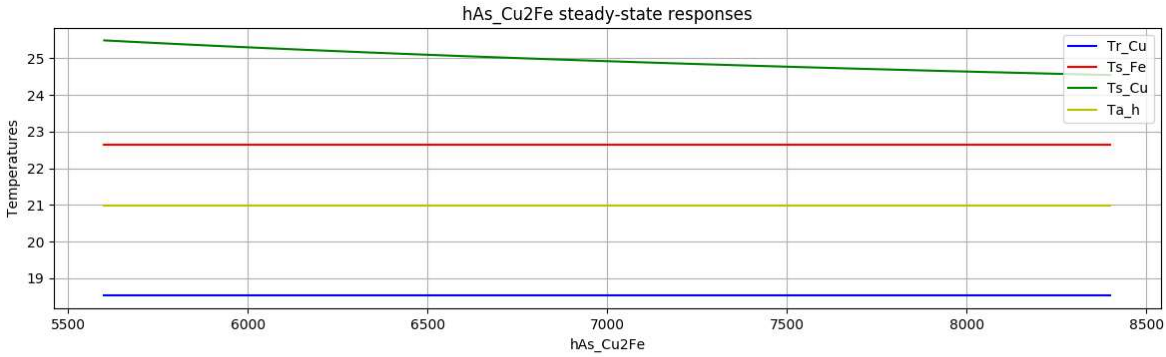


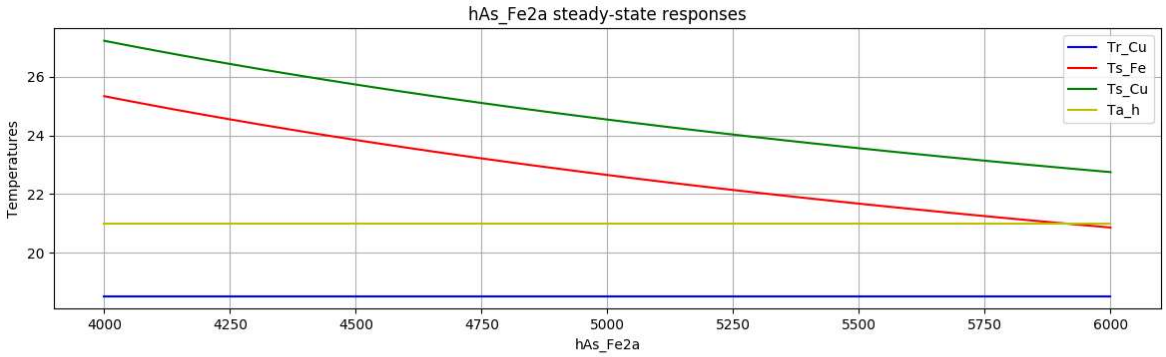

# Parameter Sensitivity analysis

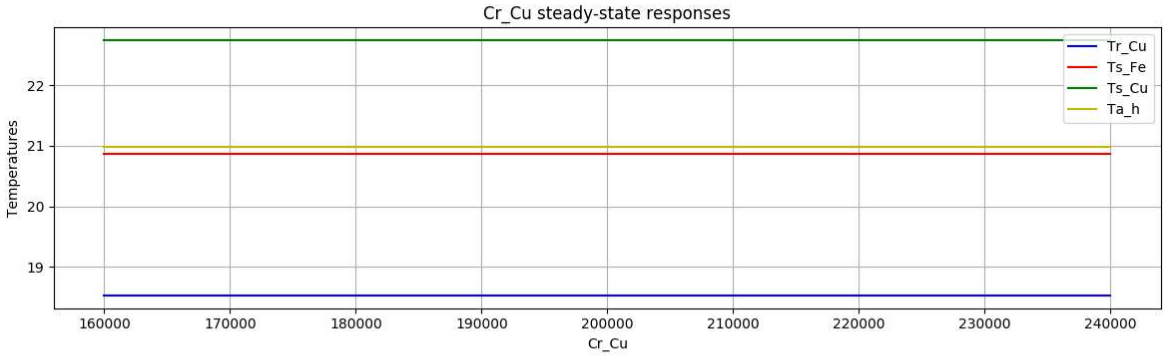
This section looks at what happens to the steady state values of the outputs when changing one parameter value at a time. The changes will be from -20 % to +20 % from "base" value. The inputs will be constant at all times, and the input values are the u_base from the step response section

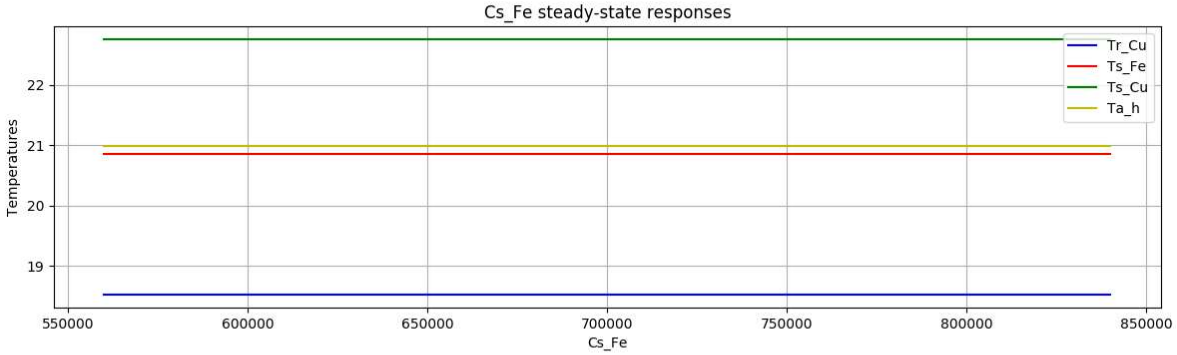
```
titles = ['hAr_Cu2a steady-state responses', 'hAs_Cu2Fe steady-state
responses', 'hAs_Fe2a steady-state responses', 'Cr_Cu steady-state
responses', 'Cs_Fe steady-state responses', 'Cs_Cu steady-state responses',
          'ml_const steady-state responses', 'mda steady-state responses',
'Tr_Cu0 steady-state responses', 'Ts_Fe0 steady-state responses', 'Ts_Cu0
steady-state responses']
theta_names = ['hAr_Cu2a', 'hAs_Cu2Fe', 'hAs_Fe2a', 'Cr_Cu', 'Cs_Fe',
'Cs_Cu', 'ml_const', 'mda', 'Tr_Cu0', 'Ts_Fe0', 'Ts_Cu0']
names = ['Tr_Cu', 'Ts_Fe', 'Ts_Cu', 'Ta_h']
colors = ['b', 'r', 'g', 'y']

theta_test = [3000, 7000, 5000, 200000, 700000, 500000, 0.5, 10, 19.8, 22.7,
24.9] # These parameter values are the basis of analysis.

#Parameter ranges
param_array = np.linspace(0.8, 1.2, 41)
theta_ranges = []
for theta in theta_test:
  theta_ranges.append(theta*param_array)

fig, ax = plt.subplots(len(theta_test), 1)
for i, theta_range in enumerate(theta_ranges):
  theta_params = theta_test #Resets parameter values
  y_last = []
  for theta_var in theta_range:
    theta_params[i] = theta_var
    y = GM1.calc_batch_output(u_base, theta_params)
    y_last.append(y[-1])
  y_last = np.array(y_last).transpose()

  for j, y in enumerate(y_last):
    ax[i].plot(theta_range, y, colors[j], label=names[j])

  ax[i].set_xlabel(theta_names[i])
  ax[i].set_ylabel('Temperatures')
  ax[i].set_title(titles[i])
  ax[i].grid()
  ax[i].legend()

fig.set_figheight(40)
fig.set_figwidth(12)
fig.tight_layout()
display(fig)
```

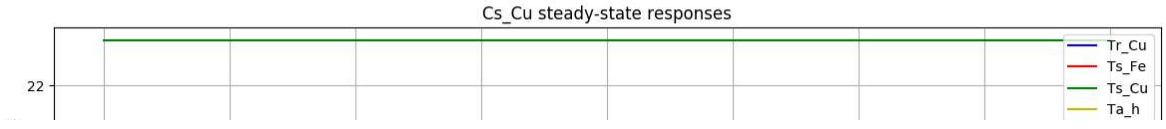

hAr_Cu2a steady-state responses
Temperatures
hAr_Cu2a
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h

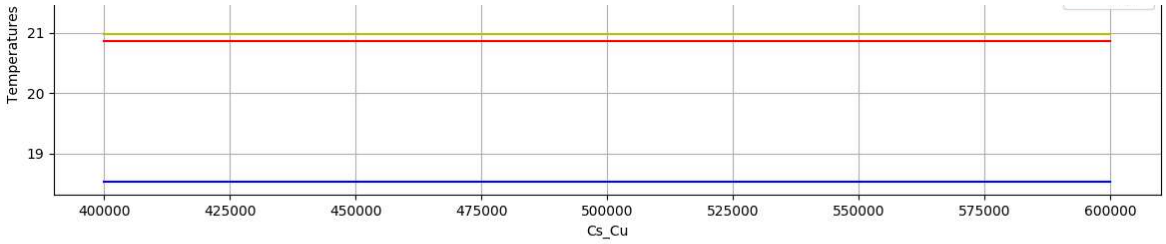

hAs_Cu2Fe steady-state responses
Temperatures
hAs_Cu2Fe
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h

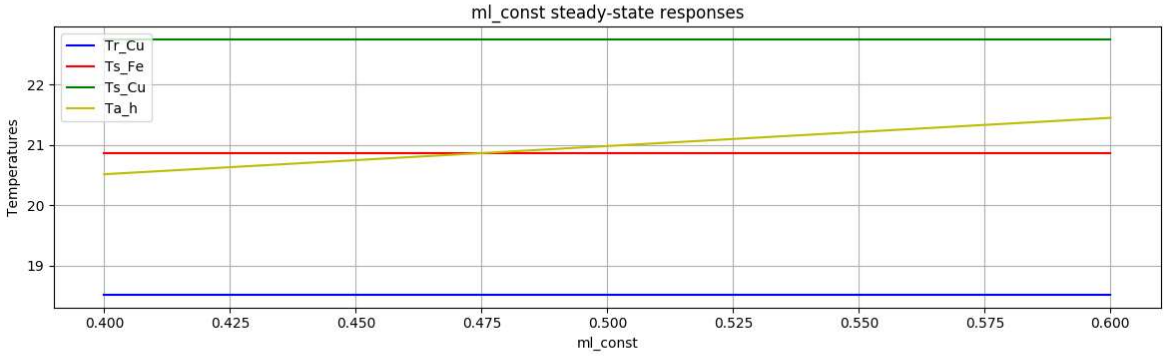

hAs_Fe2a steady-state responses
Temperatures
hAs_Fe2a
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h

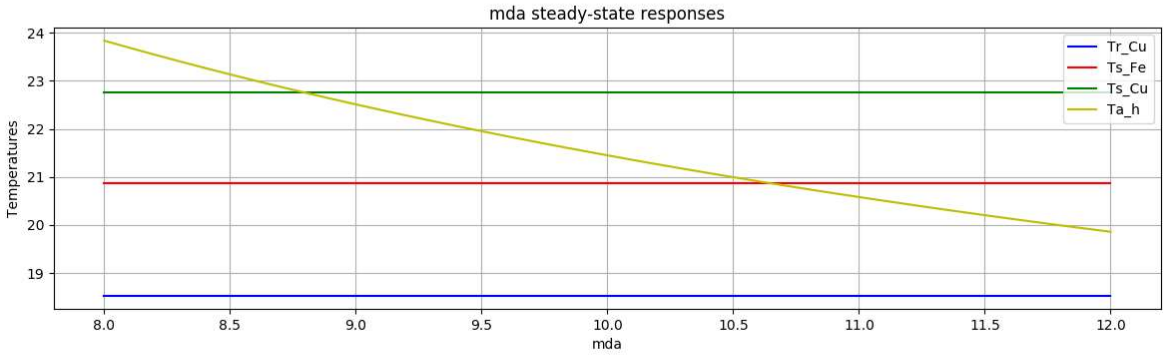

Cr_Cu steady-state responses
Temperatures
Cr_Cu
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h

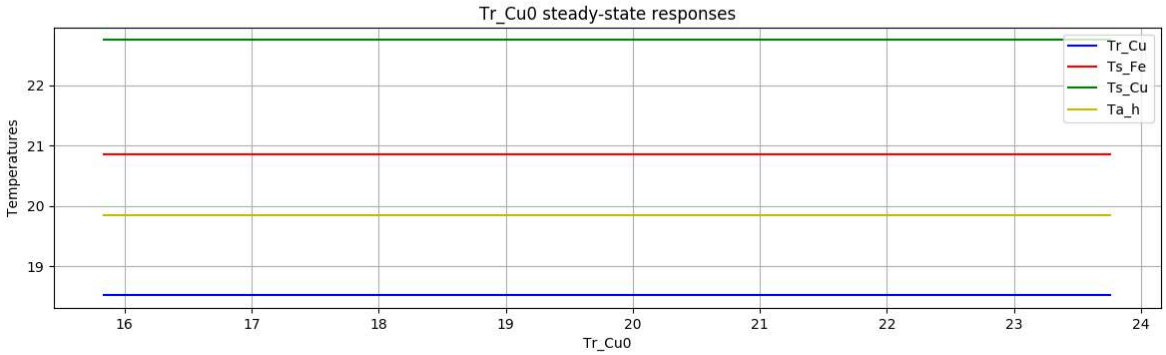

Cs_Fe steady-state responses
Temperatures
Cs_Fe
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h

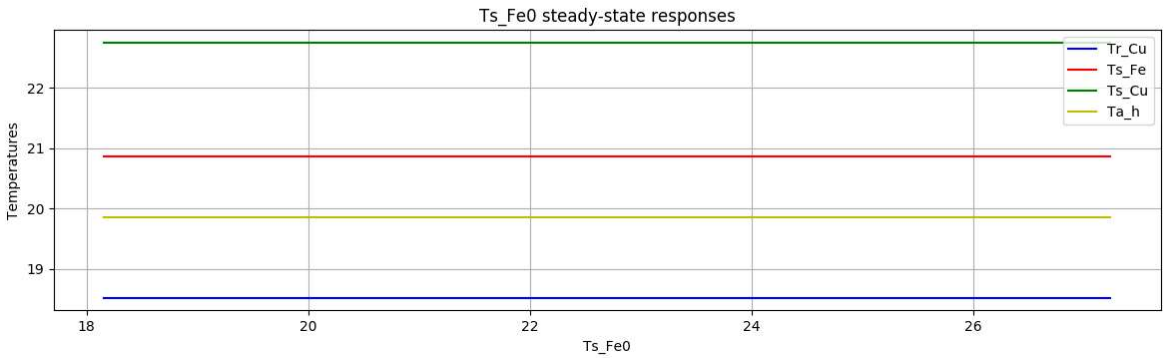

Cs_Cu steady-state responses
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h

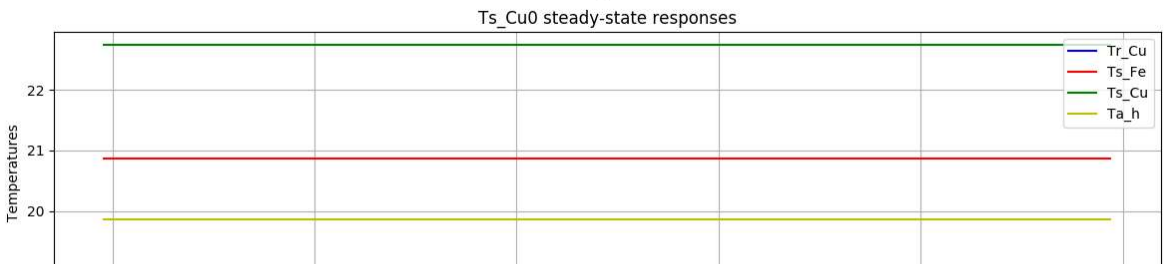

Temperatures
Cs_Cu

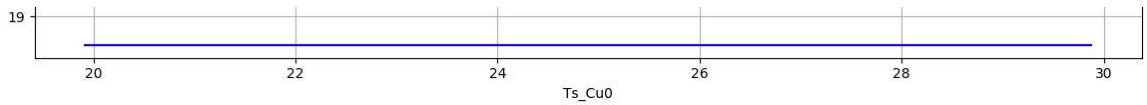

ml_const steady-state responses
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Temperatures
ml_const


mda steady-state responses
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Temperatures
mda


Tr_Cu0 steady-state responses
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Temperatures
Tr_Cu0

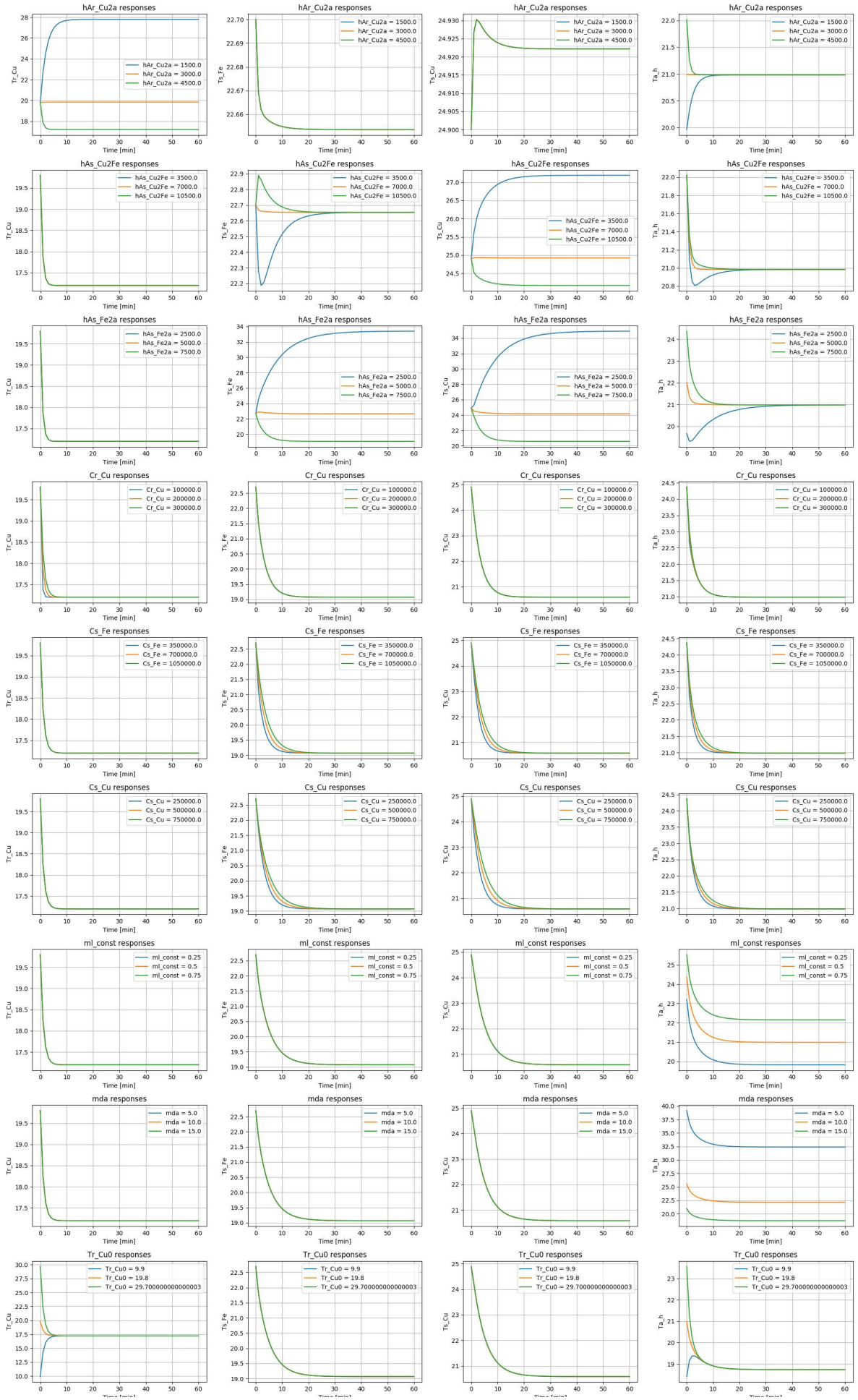

Ts_Fe0 steady-state responses
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Temperatures
Ts_Fe0

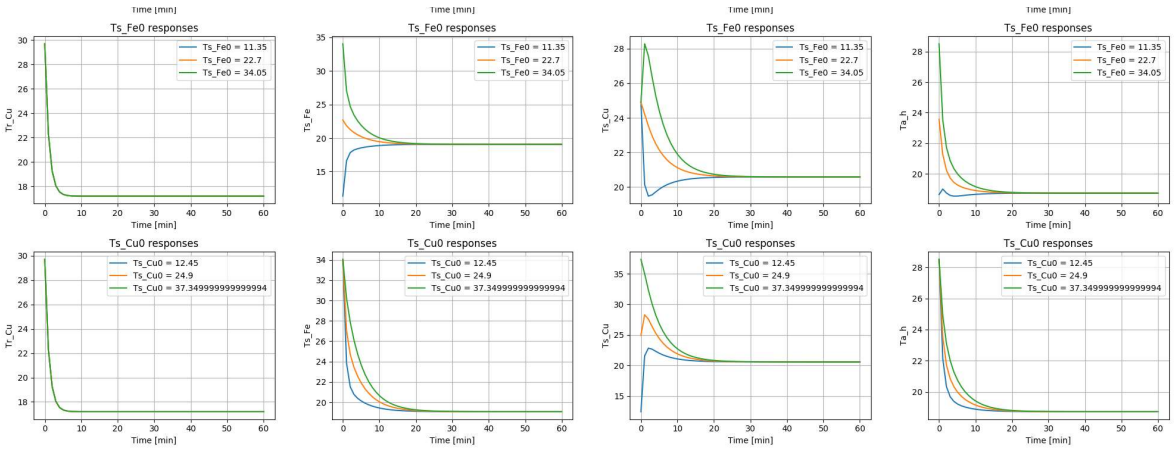

Ts_Cu0 steady-state responses
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Temperatures




```
len(u_base)
```

```
Out[202]: 60
```


hAr_Cu2a responses
hAr_Cu2a = 1500.0
hAr_Cu2a = 3000.0
hAr_Cu2a = 4500.0
hAs_Cu2Fe responses
hAs_Cu2Fe = 3500.0
hAs_Cu2Fe = 7000.0
hAs_Cu2Fe = 10500.0
hAs_Fe2a responses
hAs_Fe2a = 2500.0
hAs_Fe2a = 5000.0
hAs_Fe2a = 7500.0
Cr_Cu responses
Cr_Cu = 100000.0
Cr_Cu = 200000.0
Cr_Cu = 300000.0
Cs_Fe responses
Cs_Fe = 350000.0
Cs_Fe = 700000.0
Cs_Fe = 1050000.0
Cs_Cu responses
Cs_Cu = 250000.0
Cs_Cu = 500000.0
Cs_Cu = 750000.0
ml_const responses
ml_const = 0.25
ml_const = 0.5
ml_const = 0.75
mda responses
mda = 5.0
mda = 10.0
mda = 15.0
Tr_Cu0 responses
Tr_Cu0 = 9.9
Tr_Cu0 = 19.8
Tr_Cu0 = 29.700000000000003
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Time [min]


Ts_Fe0 responses
Ts_Fe0 = 11.35
Ts_Fe0 = 22.7
Ts_Fe0 = 34.05
Tr_Cu
Ts_Fe
Ts_Cu
Ta_h
Time [min]
Ts_Cu0 responses
Ts_Cu0 = 12.45
Ts_Cu0 = 24.9
Ts_Cu0 = 37.349999999999994

# Appendix E

# Electronic Appendices

The python code done in this project takes up a lot of pages, and are therefore included as electronic appendices. This is still work that has been done as a part of this thesis, and are therefore included.

| Appendix number | Description |
|---|---|
| Appendix E1 | SCADA-data import from Azure Datalake |
| Appendix E2 | Preparation of Data from Azure |
| Appendix E3 | Collecting inputs to models |
| Appendix E4 | Heat Exchanger training data |
| Appendix E5 | Heat Exchanger Scipy Optimization |
| Appendix E6 | Heat Exchanger Neural Network |
| Appendix E7 | Generator Training Data |
| Appendix E8 | Generator Model Scipy Optimization |
| Appendix E9 | Generator Model NN Hyperparameters |
| Appendix E10 | Generator Model NN Training |
| Appendix E11 | Generator Model Result Analysis |