Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for ingeniørvitenskap
Institutt for marin teknikk

Simen Karlsen Helgesen

Data-driven modelling and estimation of losses in shipboard electrical power components based on machine learning

Masteroppgave i Marin teknikk
Veileder: Roger Skjetne
Medveileder: Krishna Kumar Nagalingam
Juli 2022

NTNU
Kunnskap for en bedre verden

Simen Karlsen Helgesen

Data-driven modelling and estimation of losses in shipboard electrical power components based on machine learning

Masteroppgave i Marin teknikk
Veileder: Roger Skjetne
Medveileder: Krishna Kumar Nagalingam
Juli 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for ingeniørvitenskap
Institutt for marin teknikk

NTNU
Kunnskap for en bedre verden

Data-driven modelling and estimation of losses in shipboard electrical power components based on machine learning

Simen Karlsen Helgesen

CC-BY 2019/07/18

Abstract

The objective of this paper is to examine the viability of using machine learning to model a shipboard power plant Due to increasingly rigorous regulations made by maritime authorities and national governments on emissions on ships, solutions for emission reductions are needed. There are several solutions to reducing emissions in the industry and academic circles already, though a lot of these solutions are not feasible to use for existing vessels. A solution that can be used without major alterations to existing vessels is to optimise the shipboard power plant. Manually optimizing the power plant is exceedingly difficult, hence there is a need for an intelligent decition support tool. Kongsberg Maritime is therefore developing the EcoAdvisor project, which aims to be this tool. As a part of the EcoAdvisor project, there is a need for a model of the power plant that can calculate the energy losses from each component, such that one has a basis for optimisation. During this thesis, several neural networks has been trained using data from the SKandi Africa vessel. There is a comparison between training a neural network on each component of the power plant individually and training a neural net to model several components simultaneously. The results seems to indicate that modelling several components at once yields the better result.

Sammendrag

Målet med denne oppgaven er å undersøke muligheten for å modellere kraftanlegg om bord på skip ved hjelp av maskinlæring På grunn av stadig strengere forskrifter fra marine autoritetsorganisasjoner og regjeringer verden over er det nødvendig med løsninger som reduserer utslipp fra skipstrafikken. En løsning som kan iverksettes uten å ta store inngrep i skipets design er å optimalisere kraftanlegget om bord på skipet. Det er veldig vanskelig å gjøre denne optimaliseringen for hånd, så det finnes et behov for å få utviklet et verktøy som kan hjelpe til i prosessen. Kongsberg Maritime utvikler derfor verktøyet EcoAdvisor. En av delene i EcoAdvisor er å modellere komponentene i kraftsystemet om bord på et skip, slik at man kan finne hvor det er størst enegitap, og hvor mye energi som går tapt fra kraftverket. Modellen eller modellene skal videre brukes som et grunnlag for å optimere bruk av kraftverket om bord på skip. I denne oppgaven er det laget flere kunstige neurale nettverk som er trent ved hjelp av data fra skipet Skandi Africa. De neurale nettverkene har enten modellert en enkeltkomponent av kraftverket om bord eller modellert flere av de samtidig. Resultatet fra å sammenligne disse neurale nettverkene indikerer at det å modellere flere komponenter samtidig gir det beste sluttresultatet.

Acknowledgment

This thesis would not have been possible without the help from a number of people that have aided me throughout the process. Firstly I want to thank my supervisor Roger Skjetne for helping me define the problem statement, so that I had a goal in mind. Krishna Kumar Nagalingam has helped me navigate the EcoAdvisor tool, and has answered quickly whenever I needed data or specification sheets. Both Bjarne Andre Grimstad and Jakob Anguiers have aided in the work on this thesis by publishing the base code that has been used for this project as open source projects. Namireddy Praveen Reddy has been a great help during the entire revision of this thesis.

S.K.H.

Simen Karlsen Helgesen

Contents

Abstract . . . iii
Sammendrag . . . v
Acknowledgment . . . vii
Contents . . . ix
Figures . . . xi
Tables . . . xiii
Code Listings . . . xv
1 Introduction . . . 1
1.1 Background an motivation . . . 1
1.2 Research questions . . . 3
1.3 Case Study . . . 3
1.4 Thesis outline . . . 5
2 Shipboard Power Plants . . . 7
3 Efficiency Modelling . . . 9
3.1 Static efficiency modelling . . . 9
3.2 Dynamic efficiency modelling . . . 10
3.3 Component Efficiency modelling . . . 10
3.4 System Efficiency modelling . . . 11
4 Modelling using neural networks . . . 13
4.1 Hyper-parameters . . . 13
4.1.1 Depth and breadth . . . 13
4.1.2 Regularization . . . 13
4.1.3 Error . . . 17
4.1.4 Learning rate . . . 19
4.1.5 Optimization algorithm . . . 19
4.1.6 Batch . . . 21
4.1.7 Activation function . . . 21
4.2 Supervised learning algorithms . . . 21
4.2.1 Support Vector Regression (SVR) . . . 21
4.2.2 Deep feedforward (DFF) . . . 22
4.2.3 Long Short Term Memory (LSTM) . . . 23
4.3 Overfitting and underfitting . . . 25
4.3.1 Overfitting . . . 25
4.3.2 Underfitting . . . 25

4.4 Growing and pruning . . . 25
4.4.1 Pruning . . . 25
4.5 Modelling Process . . . 27
4.5.1 Experimental setup . . . 27
4.5.2 Libraries . . . 27
4.5.3 Case Study . . . 28
4.5.4 Width-Depth test . . . 28
4.5.5 Multirun algorithm . . . 29
4.5.6 DFF . . . 33
4.5.7 LSTM . . . 34
4.5.8 Engine . . . 36
4.6 Converter . . . 37
4.6.1 Pre-processing . . . 37
4.6.2 Generator . . . 40
4.6.3 Thruster . . . 41
4.7 Parallel neural network . . . 41
4.7.1 Comparing physics based modelling with machine learning based modelling . . . 42
5 Results and discussion . . . 43
5.1 Engine . . . 43
5.2 Thruster . . . 44
5.3 Generator . . . 45
5.4 Converter . . . 49
5.4.1 Estimated Training cost . . . 49
5.4.2 Results from training . . . 50
5.4.3 Results on test set . . . 51
5.5 Parallel Network . . . 53
6 Conclusion . . . 57
6.1 Recommendation for further work . . . 57
Bibliography . . . 59
A Long short term memory code . . . 65
A.1 LSTM data processor . . . 65
A.2 LSTM model . . . 69
A.3 LSTM main file . . . 79
A.4 LSTM utilities . . . 82
B Deep feedforward code . . . 85

Figures

1.1 Efficiency profile of Skandi Africa main engine 2
1.2 Single line diagram of the power plant of the Skandi Africa Vessel . 4
1.3 Subsection of the system, these components are modelled 5

4.1 Illustration of a fully connected Deep-feedforward neural network with a constant width of 3 and depth of 2 14
4.2 MSE on training and test sets vs size of training set, for data generated from a degree 2 polynomial with Gaussian noise of variance $\sigma^2 = 4$. We fit polynomial models of varying degree to this data. (a) Degree 1. (b) Degree 2. (c) Degree 10. (d) Degree 25. Note that for small training set sizes, the test error of the degree 25 polynomial is higher than that of the degree 2 polynomial, due to overfitting, but this difference vanishes once we have enough data. Note also that the degree 1 polynomial is too simple and has high test error even given large amounts of training data. Figure generated by linregPolyVsN. [18] 18
4.3 Illustration of LSTM neuron, area shaded in red is inside neuron . . 24
4.4 Accuracy of DFF network with constant width of 10 as depth changes 29
4.5 Accuracy of DFF with constant depths of 2,3, and 4 changes by altering width 31
4.6 Constant start width and constant ratio $\frac{width_{firstlayer}}{width_{lastlayer}}$ DFF with changing depth. 32
4.7 Constant depth and constant ratio $\frac{width_{firstlayer}}{width_{lastlayer}}$ DFF with changing start width. 33

5.1 Efficiency of the thruster over time 45
5.2 Efficiency of the generator over time, raw data 47
5.3 Efficiency of the generator over time, calculated using polynomial model 48
5.4 Efficiency of the generator over time, calculated using neural network 49
5.5 Percentage usage of the GPU processing power during the training 50
5.6 MSE value of validation error over the epochs, due to the extreme improvement the MSE-axis has to be shown in log-scale 50

5.7 Linear improvement over the last 200 epochs 51
5.8 Efficiency of the converter over time 52
5.9 Efficiency of the converter over time, shorter time frame 53
5.10 Combined efficiency of the thruster, generator, and converter over time, raw data 54
5.11 Combined efficiency of the thruster, generator, and converter over time, calculated using the polynomial model 55
5.12 Combined efficiency of the thruster, generator, and converter over time, calculated using the neural network model 56

Tables

4.1 Relevant system specifications 27
4.2 Hypothetical dataset showing a potential problem with a normalizing method 35
4.3 Excerpt of the data used for training before it has gone through the pre-processing 38
4.5 Inputs and outputs used 38
4.4 Same part of the training-data set as shown in Table 4.3, after pre-processing 39
4.6 Selection of training set, validation set, and test set 39
4.7 Values for hyperparameters used to obtain the results given in chapter 5 40

5.1 The effects of pruning the model trained on data from engine 43
5.2 The effects of pruning the model trained on data from thruster . . . 44
5.3 The effects of pruning the model trained on data from generator . . 46
5.4 MSE and MAE from test-set 51
5.5 The effects of pruning the model trained on data from all four components 54

Code Listings

Code/LSTM/data_processor.py 65
Code/LSTM/model.py 69
Code/LSTM/run.py 79
Code/LSTM/utils.py 82

Code/Converter/NN.py 85

Chapter 1

Introduction

1.1 Background an motivation

In the recent years, the International Maritime Organization (IMO) has introduced stringent rules and regulation in order to reduce emissions by the maritime industry. The International Maritime Organisation (IMO) said in their submission to the UNFCCC Talanoa Dialogue [1] that their ambition levels were threefold:

Carbon intensity of the ship to decline through implementation of further phases of the energy efficiency design index (EEDI) for new ships: To review with the aim of strengthening the energy efficiency design requirements for ships with the percentage improvement for each phase to be determined for each ship type, as appropriate;

Carbon intensity of international shipping to decline: To reduce CO_2 emissions per transport work, as an average across international shipping, by at least 40% by 2030, pursuing efforts towards 70% by 2050, compared to 2008; and

GHG emissions from international shipping to peak and decline: To peak GHG emissions from international shipping as soon as possible and to reduce the total annual GHG emissions by at least 50% by 2050 compared to 2008 whilst pursuing efforts towards phasing them out as called for in the Vision as a point on a pathway of CO_2 emissions reduction consistent with the Paris Agreement temperature goals.

One of the solutions to reduce fuel consumption is to alter physical properties of a ship. One could for example optimise hull shape, propeller design, or hull coating. While these alterations to a vessel might reduce fuel consumption, they can be intrusive and will impact ship operations. Certain alterations such as changing hull shape to reduce fuel consumption might not be possible without significantly impacting the use and functional profile of the vessel.

An alternate solution is to optimise the shipboard power plant such that there are minimal losses from internal processes. Most components of the shipboard power plant have an efficiency that varies with how close they are to maximum capacity. If one can configure a system in such a way that the components can run at the rates where they are the most efficient, there could be a large reduction in

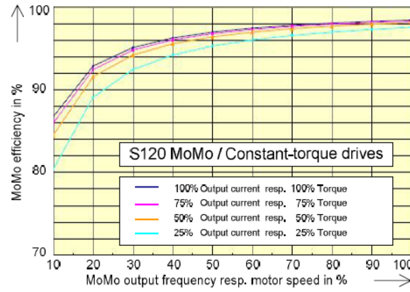


Figure 1.1: Efficiency profile of Skandi Africa main engine

fuel consumption. As an example, Figure 1.1 shows that the main engine aboard the Skandi Africa can be 10% more efficient if run at 100% speed compared to when run at 10% speed.

At present date this optimisation has to be done by the crew onboard, which is difficult due to the lack of visualisation of system states. In addition to the lack of visualisation, the crew has to keep within existing rules and regulations. This optimisation can be done by hand if the crew is properly trained. Though doing so is not feasible as it would take too much time for it to be worthwhile.

To address this issue, Kongsberg Maritime is developing an intelligent decision support tool called EcoAdvisor. The goal of the EcoAdvisor tool is to provide the crew with the means to optimise the shipboard power plant. The tool is still in early development, and access to the existing tools are restricted to a select number of people.

As a part of the EcoAdvisor tool, there is a need for a simple to set up and easy to use model of the power plant aboard. This model is used to find the energy losses in each component, such that one can find the overall efficiency of the shipboard power plant, and which components that are running at a sub-optimal efficiency. The results from this model can then be used to optimise the power plant.

Several methods exists that could be used to model this system. This thesis implements data driven methods. Specifically Neural Network solutions to model the system, as the availability of data seems large enough to employ a machine learning based solution to the modelling problem.

According to [2] a perfect model does not exist, as there is always a drawback to the choices being made in modelling. In the case of traditional mathematical models made from the usage of physical laws and manifacturer data sheets, these faults can lie within the factors omitted from the model for the sake of computability.

1.2 Research questions

This thesis aim to find out a way to model the shipboard electrical power plant aboard a vessel. The computational cost of the modelling should also be taken into consideration so that the resulting models can be used in an industrial application and not just as an academic proof of concept. Finding the balance between computational cost and accuracy is central to the usefulness of the results. As described in **??** the models are to be a part of a larger system, and with the limited [citation required] computational power aboard a vessel computer, along with the need of getting results in real-time the models have a need to be computationally light in order for the larger system to serve its function.

In order to choose the method of modelling that is the most suitable for the purpose, the advantages and disadvantages of each modelling technique needs to be found.

As for the computational power needed for the final models, estimates will need to be calculated. They can be estimated using runtime, but to get the most accurate results the amount of floating point operations needed per time-step of the model has to be calculated.

The aim of this thesis can be summarized as:

1. Why should we use machine-learning based modelling?
2. Is using models based upon machine learning more accurate than using physics based computational models?
3. What kind of accuracy can one expect from modelling a shipboard electrical power plant using a neural network?
4. Can the computational requirements of a neural network be brought down such that a shipboard computer is able to render the results from modelling the power plant in real time?

1.3 Case Study

In order to implement and test a data driven modelling technique, there is a need for data. In order to have access to enough data to develop and test the models, a case study has been chosen. The data used is supplied by Kongsberg Maritime, on a vessel called Skandi Africa.

The Skandi Africa is a vessel owned and operated by DOF Subsea. Through Kognifai, access to the data the vessel collects can be accessed. Due to the need for large quanitities of data being required for some of the techniques used in this thesis, finding a vessel where the amount of data stored is sufficiently large is the main driving factor for the development of data driven models, especially when the data driven method used is based upon machine learning.

The system chosen to gather data from was the vessel Skandi Africa [3]. The Single Line Diagram for the vessel is shown in Figure 1.2, Figure 1.3 is the part of the system that is modelled.

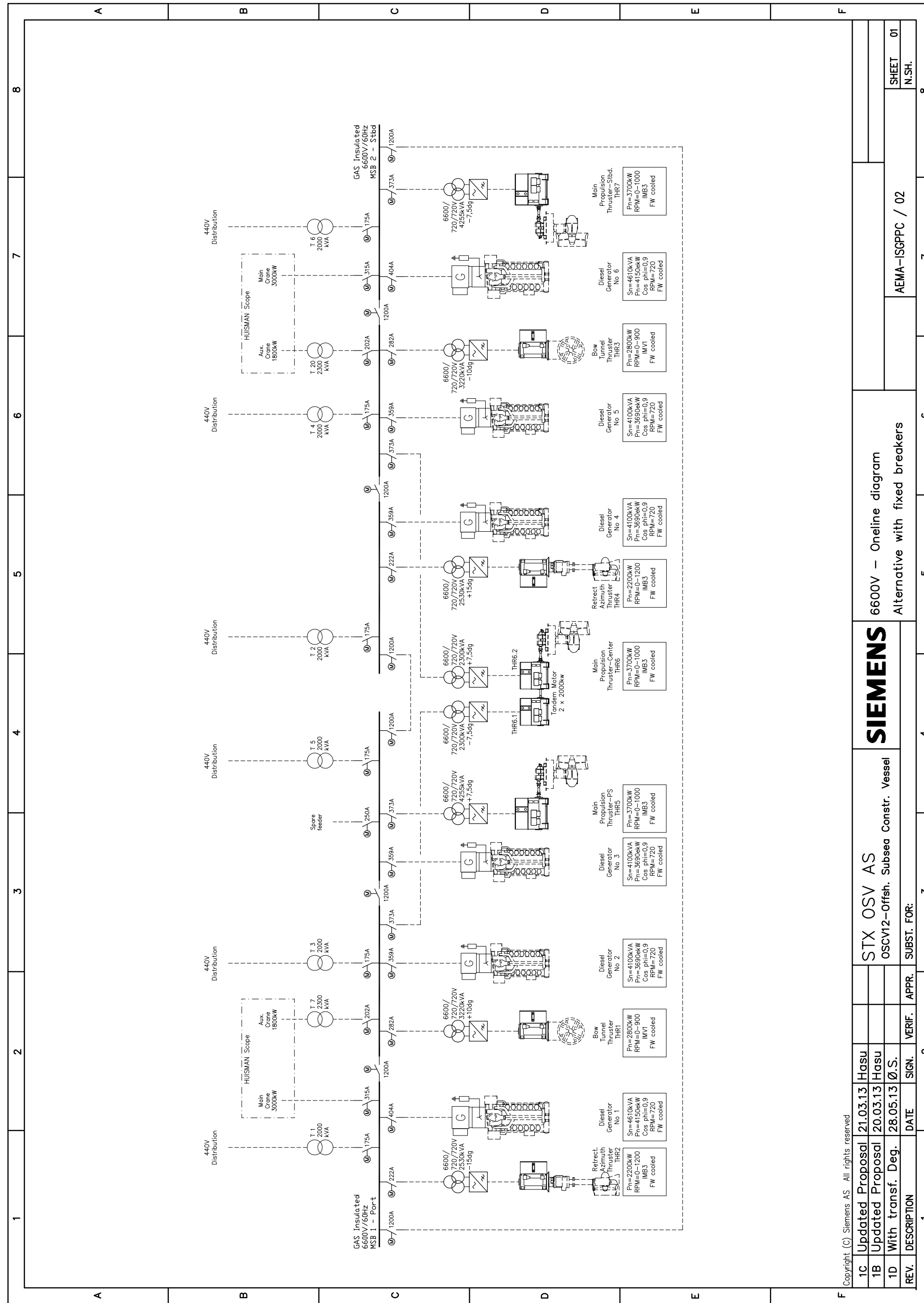


Figure 1.2: Single line diagram of the power plant of the Skandi Africa Vessel

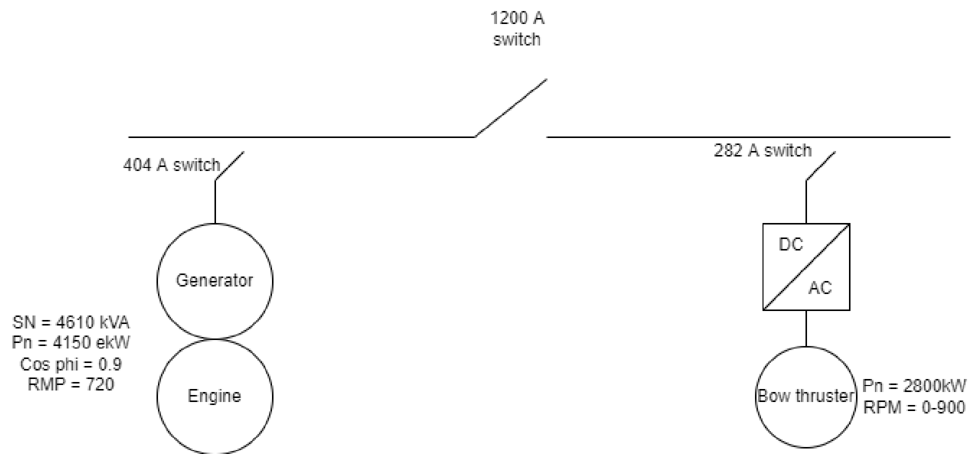


Figure 1.3: Subsection of the system, these components are modelled

1.4 Thesis outline

The first chapter of this thesis explains the motivation behind the research done. The second chapter outlines the design of a shipboard power plant. The third chapter explains the different modelling techniques considered to solve the problem. The fourth chapter describes how the models were created and how the code differs from theory. The fifth chapter shows the results and explains the implications of the results. The sixth chapter concludes the thesis and outlines how the work should continue.

Chapter 2

Shipboard Power Plants

While the traditional way to design a ship has been based on the design spiral created by J. H Evans in 1959, there has been a search for techniques and methods in order to streamline this process [4]. [5] developed a systematic methodology that can reduce the design of the power management system into a solvable problem for a computer. The method is not flawless, as it relies heavily on the traditional model that is in need of methods such as small scale model testing in order to estimate the power requirement for a vessel.

According to [4], designing the configuration of a power plant aboard a vessel is governed by four choices in the early design phase:

- What kind of fuel should the vessel use?
- What type of machinery, where the existing alternatives are:
  - Engine
  - Dual Fuel Engine
  - Genset
  - Battery
  - Fuel Cell
  - Super-capacitor

  A vessel can use more than one alternative
- Should the propulsion type be mechanical or electrical?
- Direct or alternating current in the main distribution grid?

In systems with mechanical propulsion, it is common to have an engine or dual fuel engine. They could be used in conjunction with hybrid systems if there is some form of energy recovery. Electrical propulsion systems, which often utilises podded propulsion [6] are commonly found with genset based systems.

Batteries are increasingly utilized for more than emergency backup energy, as they are a key part of Energy Management Systems. One example of batteries being a key part of the development of such systems is the Blue Power EMS being added to the Kognifai Marketplace [7]. The European Maritime Safety Agency in conjunction with DNV defined six functional roles for batteries in the shipboard

power plant [8]:

1. Being a backup for generators, reducing the need for redundancy in the system
2. Being a buffer, which allows for a phenomenon called peak shaving. This is done to avoid overloading the energy generating systems when a lot of power is needed for a period.
3. Aiding in load optimization, such that the energy generating components at the system can run at peak efficiency point for a larger percentage of the time.
4. Being a source of immediate power in cases where demanding it from other systems could cause system instabilities
5. Harvesting energy renewable sources.
6. Being an emergency backup in case all other systems fail.

Chapter 3

Efficiency Modelling

In this chapter an overview of efficiency models is presented. Efficiency being defined as the ratio between the power going out of a system divided by the power going in to a system $\eta = \frac{P_{out}}{P_{in}}$.

The two main ways to model efficiency in a shipboard power plant are static efficiency modelling and dynamic efficiency modelling [9]. One can further divide both into component and system modelling [9]. When modelling both dynamic efficiency and static efficiency one has to further divide the modelling techniques in such a way that one could take into account for the differences between alternating current (AC) and direct current (DC) architectures [10].

The chapter is concluded with a discussion on the strengths and weaknesses of the main types of efficiency modelling and why using a machine learning based technique has been chosen over using other modelling techniques.

3.1 Static efficiency modelling

Static efficiency modelling is defined by it being an efficiency model of a component or system when it is running at nominal power [11]. Nominal power being the efficiency at nominal loads. A static efficiency model does not take into account the load condition of the component or system being modelled.

When going from a component level to a system level in static efficiency modelling, the efficiency of the system can be calculated by

$$\eta_{AC} = \left(\frac{\sum_j^s \gamma_j \eta_{cd_j} \eta_{i_j} + \sum_k^t P_{ek} \eta_{g_k}}{\sum_j^s \gamma_j \frac{P_{b_j}}{n_{bd_j}} + \sum_k^t \frac{P_{e_k}}{\eta_e f_k}}\right) \cdot \left(\frac{\sum_l^u P_{m_l} + \sum_n^v P_{h_n} + \sum_j^s (1-\gamma_j) P_{b_j} \eta_{bc_j}}{\sum_l^u \frac{P_{m_l}}{\eta_{r_l} \eta_{i_l} \eta_{m_l}} + \sum_n^v \frac{P_{h_n}}{\eta_{i_n}} + \sum_j^s (1-\gamma_j) \frac{P_{b_j}}{\eta_{cc_j} \eta_{i_j}}}\right) \tag{3.1}$$

$$\eta_{\text{FSDC}} = \left(\frac{\sum_j^s \gamma_j P_{b_j} \eta_{cd_j} + \sum_t^k P_{e_k} \eta_{g_k \eta_{T_k}}}{\sum_j^s \gamma_j \frac{P_{b_j}}{\eta_{bd_j}} + \sum_k^t \frac{P_{e_k}}{\eta_{ef_k}}}\right) \cdot \left(\frac{\sum_l^u P_{m_l} + \sum_n^v P_{h_n} + \sum_j^s (1-\gamma_j) P_{b_j} \eta_{bc_j}}{\sum_l^u \frac{P_{m_l}}{\eta_{i_l} \eta_{m_l}} \sum_n^v \frac{P_{h_n}}{\eta_{i_n}} + \sum_j^s (1-\gamma_j) \frac{P_{b_j}}{\eta_{cc_j}}}\right) \tag{3.2}$$

$$\eta_{\text{VSDC}} = \left(\frac{\sum_j^s \gamma_j P_{b_j} \eta_{cd_j} \sum_k^t Pe_k \eta_{g_k} \eta_{T_k}}{\sum_j^s \gamma_j \frac{P_{b_j}}{\eta_{bd_j}} + \sum_k^t \frac{P_{e_k}}{\eta_{ev_k}}}\right) \cdot \left(\frac{\sum_l^u P_{m_l} + \sum_n^v P_{\ n} + \sum_j^s (1-\gamma_j) P_{b_j} \eta_{bc_j}}{\sum_l^u \frac{P_{m_l}}{\eta_{i_j} \eta_{m_l}} + \sum_n^v \frac{P_{h_n}}{\eta_{i_n}} + \sum_j^s (1-\gamma_j) \frac{P_{b_j}}{\eta_{cc_j}}}\right) \tag{3.3}$$

Where P_h is the hotel loads, P_b is the power going in or out of the batteries, P_m are the powers of the propulsion motors, P_e are the powers from the engines, γ is the state of the battery (where 1 indicates the battery is charging, 0 indicates battery discharging), j is the enumeration of the batteries, k is the enumeration of the engines, l is the enumeration of the hotel loads, n is the enumeration of the propulsion motors, FSDC is fixed speed DC, and VSDC is variable speed DC.

3.2 Dynamic efficiency modelling

The main difference between dynamic and static efficiency modelling is that in dynamic efficiency modelling one takes into account how efficiency changes based on operational profile. Most components have a lower efficiency at lower loads [11]. Because the stated efficiency for a system component is typically stated in nominal efficiency, one must take in load dependent data in order to construct a dynamic efficiency model for a given system [12]. Since the dynamic profile for a given component is not typically stated in the data sheet, one must extract these from literature. Fuel combustion engines are the exemption from this, as they typically state their specific fuel oil consumption (SOFC) curve in the data sheet [13].

3.3 Component Efficiency modelling

[13] used a combination of polynomial fitting and rational fitting to model the efficiency of each component in a DC-hybrid power system. Rational fitting being the ratio between polynomials. Rational fitting does have an asymmetrical error profile, as it does result in larger error for negative values of load percentage. However since load percentages are defined as strictly positive, that did not affect the result of the findings in [13].

For a combustion engine, the efficiency can be stated in terms of the SOFC curve which is dependent on both the speed and power delivered, and the specific calorific heat of the fuel used h as such [13]:

$$\eta = \frac{1}{SOFC \cdot h} \tag{3.4}$$

From [14], one can construct a fitted polynomial curve in order to find the dynamic efficiency of a generator. [14] identifies the three main sources of power loss in a generator as copper loss (ohmic losses), iron loss, and mechanical losses. The copper losses are from electric resistance in the generator. The iron losses are

magnetic losses and can be modelled as in Equation3.5. Where B_{max} is the peak magnetic flux density, k_H, k_c, and k_E are constants. Due to the quadratic losses with respect to rotational speed, [14] assumes the mechanical losses stem from linear friction torque.

$$p_{\text{loss, magnetic}} = k_H B_{max}^2 \omega + k_c B_{max}^2 \omega^2 + k_E B_{max}^{1.5} \omega^{1.5} \quad (3.5)$$

The power losses in a power converter can be found by calculating the conduction and the switching losses for each sub-component of the power converter. Using the assumption that the inductor current is ripple-free one can find the switching losses by using [15] :

$$P_{\text{SW, on}} = R_{\text{SW, on}} \cdot I_{\text{S,RMS}}^2 = R_{\text{S,on}} \left(\frac{P_{hv}\sqrt{D}}{V_{hv}} \right)^2 \quad (3.6)$$

With the assumption that transistor capacitance is linear, it is possible to calculate the switching losses by:

$$P_{\text{SW,SW}} = 4\left(\frac{f_{sw}}{2}\right) C_O V_{SM}^2 \quad (3.7)$$

The average conduction loss in a diode over a switching period yields:

$$P_D = V_{fw}\left(\frac{nP_{hv}}{DV_{hv}}\right) + (2-D)R_{D,on}\left(\frac{nP_{hv}}{DV_{hv}}\right)^2 \quad (3.8)$$

Using the energy losses in [15], one can produce a quadratic fit curve to model the dynamic efficiency of the power converter [13].

3.4 System Efficiency modelling

Unlike in static efficiency modelling, where the system efficiency is found by using Equations 3.1, 3.2, and 3.3, in dynamic efficiency modelling typically inserts the model for each component into a simulation software [13]. One then simulates the efficiency by comparing the power going in with the power going out of the system. Where power going in to the system typically is in the form of fuel consumption and battery discharge, and power going out from the system are the system Loads and the power used to charge the batteries. [13] and [16] both found that the method used for load sharing between the combustion engine and the battery will have a significant impact on the overall system efficiency.

Chapter 4

Modelling using neural networks

The purpose of the following chapter are to provide enough mathematical background to the function of a neural network to understand how the network learns. Through understanding how the neural network on a mathematical level, one can use that knowledge in conjunction with knowledge of the system being modelled to find a set of hyper-parameters more quickly than randomly adjusting them until a satisfactory result is found.

4.1 Hyper-parameters

[17] defines hyperparameters as settings one can use to control the behavior of a machine learning algorithm. The hyperparameters are not changed by the algorithm using them.

4.1.1 Depth and breadth

A neural net consists of three types of layers, the input layer, the hidden layer, and the output layer. The hidden layer is then further divided into M layers. The number of layers that the hidden layer consists of is the depth of the neural network. The number of neurons per layer in the hidden layer is the width of the network. Figure 4.1 shows a fully connected Deep-feedforward neural network with a constant width of 3 and a depth of 2.

4.1.2 Regularization

As is explained in section 4.3, one of the main concerns when doing machine learning is overfitting. [17] defines regularization as a modification made to a machine learning algorithm that intends to reduce the test-error but not the training-error.

In this thesis, three different strategies of regularization have been used: L^2, L^1, and the use of a large enough data set.

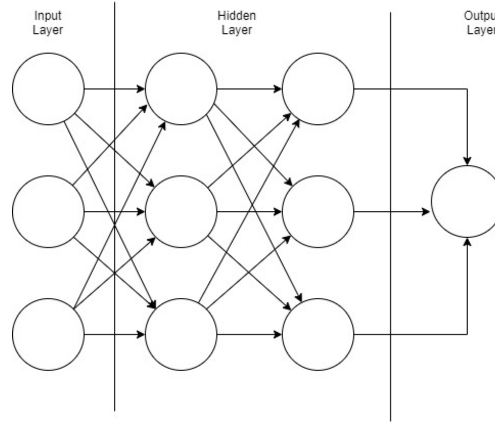


Figure 4.1: Illustration of a fully connected Deep-feedforward neural network with a constant width of 3 and depth of 2

L^2 parameter norm penalty

L^2 regression, also known as ridge regression and Tikhonov regression [17], is one of the more common forms of parameter norm penalties. The strategy revolves around adding a regularization term $\Omega(\boldsymbol{\Theta}) = \frac{1}{2}||\mathbf{w}||_2^2$ to the objective function to drive the weights closer to zero.

By adding the L^2 regularization to an objective function $J(\mathbf{w;X,y})$. You get:

$$\tilde{\mathrm{J}}(\mathbf{w;X,y}) = \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w} + J(\mathbf{w;X,y}) \tag{4.1}$$

Where J is the objective function,$\tilde{J}$ is the regularized objective function,α is a value the programmer sets, w are the weights of the model, X are the input values, and y are the output values.

Which changes the parameter gradient to:

$$\bigtriangledown_{\mathbf{w}}\tilde{\mathbf{J}}(\mathbf{w;X,y}) = \alpha\mathbf{w} + \bigtriangledown_{\mathbf{w}}\mathbf{J}(\mathbf{w;X,y}) \tag{4.2}$$

For each gradient step this update is applied:

$$\mathbf{w} \leftarrow (1-\epsilon\alpha)\mathbf{w} - \epsilon\bigtriangledown_{\mathbf{w}}J(\mathbf{w;X,y}) \tag{4.3}$$

To illustrate how this changes the entire training [17] considers the case of fitting a linear regression model with MSE as the error indicator.

In that case one can approximate the Objective function $\hat{J}$ by:

$$\hat{J}(\boldsymbol{\Theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w}-\mathbf{w}^*)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}^*) \tag{4.4}$$

Where $\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w})$ and $\mathbf{H}$ [1] is the Hessian matrix of J evaluated at $\mathbf{w}^*$ with respect to $\mathbf{w}$. Due to $\mathbf{w}^*$ being defined to be the minimum of J, $\mathbf{H}$ is positive

1

semi definite. One can find the minimum of $\hat{J}$ where the gradient is equal to zero. The gradient of $\hat{J}$ being the Hessian matrix.

Then when L^2 regularization is applied, the minimum of the objective function can be found by solving:

$$\begin{aligned} \alpha\tilde{\mathbf{w}} + \mathbf{H}(\tilde{\mathbf{w}} - \mathbf{w}^*) = 0 \\ (\mathbf{H} + \alpha\mathbf{I})\tilde{\mathbf{w}} = \mathbf{H}\mathbf{w}^* \\ \tilde{w} = (\mathbf{H} + \alpha\mathbf{I})^{-1}\mathbf{H}\mathbf{w}^* \end{aligned} \tag{4.5}$$

Where $\tilde{\mathbf{w}}$ is the location of the minimum and $\mathbf{I}$ is the identity matrix.

To see what happens as α becomes larger, we have to decompose $\mathbf{H}$ into a diagonal matrix $\mathbf{D}$ and an orthonormal basis of egenvectors $\mathbf{E}$ such that $\mathbf{H} = \mathbf{E}\mathbf{D}\mathbf{E}^\top$. Equation 4.5 then becomes:

$$\begin{aligned} \tilde{\mathbf{w}} &= (\mathbf{E}\mathbf{D}\mathbf{E}^\top + \alpha\mathbf{I})^{-1}\mathbf{E}\mathbf{D}\mathbf{E}^\top\mathbf{w}^* \\ &= [\mathbf{E}(\mathbf{D} + \alpha\mathbf{I})\mathbf{E}^\top]^{-1}\mathbf{E}\mathbf{D}\mathbf{E}^\top\mathbf{w}^* \\ &= \mathbf{E}(\mathbf{D} + \alpha\mathbf{I})^{-1}\mathbf{D}\mathbf{E}^\top\mathbf{w}^* \end{aligned} \tag{4.6}$$

The effect of L^2 regularization is to rescale the components of $\mathbf{w}^*$ with the eigenvectors of H by a factor of $\frac{eigenvector_i}{eigenvector_i + \alpha}$. This means that only directions within the N-th dimentional space that contribute significantly to reducing the objective function remains unaffected by the regularization function. The weight decay forces the algorithms to disregard unimportant features it finds of the data it is looking at.

L^1 parameter norm penalty

The L^1 regularization is defined by Equation 4.7, the sum of the absolute values of the parameters.

$$\Omega(\mathbf{\Theta}) = ||\mathbf{w}||_1 = \sum_i |w_i| \tag{4.7}$$

Adding Equation 4.7 to an objective function gives:

$$\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = \alpha||\mathbf{w}||_1 + J(\mathbf{w};\mathbf{X},\mathbf{y}) \tag{4.8}$$

The gradient of Equation 4.8 is:

$$\nabla_{\mathbf{w}}\tilde{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = \alpha \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}}J(\mathbf{w};\mathbf{X},\mathbf{y}) \tag{4.9}$$

Where $\text{sign}(\mathbf{w})$ is element-wise applied.

Unlike L^2 where the gradient is scaled linearly with the regularization contribution, L^1 scales each component of $\mathbf{w}$ with a constant, and the difference in how each component is scaled comes from the $\text{sign}(\mathbf{w}_i)$.

[17] considers the same system as in L^2 regularization to exemplify how this affects the machine learning algorithm. That is a simple linear regression model without bias. The gradient of this objective function is:

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \tag{4.10}$$

Where $\mathbf{H}$ is the same Hessian matrix as for Equation 4.4.

Due to the the sign function in Equation 4.9, finding a clean algebraic solution to the quadratic approximation is not given. To simplify the explanation, the Hessian matrix is therefore assumed to be a strictly positive diagonal matrix. The approximation of the objective function with L^1 regularization is then:

$$\hat{J}(\mathbf{w};\mathbf{X},\mathbf{y}) = J(\mathbf{w}^*;\mathbf{X},\mathbf{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (\mathbf{w}_i - \mathbf{w}_i)^2 + \alpha |w_i| \right] \tag{4.11}$$

Minimizing this approximation gives the solution:

$$w_i = \text{sign}(w_i^*) \max\left(|w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right) \tag{4.12}$$

The effect of this is that if $w_i^* \leq \frac{\alpha}{Hi,i}$, the optimal value of w_i becomes zero. If $w_i > \frac{\alpha}{H_{i,i}}$ the value of w_i shifts by $\frac{\alpha}{H_{i,i}}$.

Therefore when L^1 regularization is applied to a neural net, the neural net becomes far more sparse. Sparse meaning that more parameters have an optimal value of zero. This means that in addition to adding a regularizing effect to the neural net; it can also be useful if one is planning to prune the neural network later.

The regularization effect of big data

In addition to the use of regularization to avoid overfitting, there exists another approach; using vast quantities of data. Given that the data is randomly sampled, the more data provided to the neural net, the better it can learn. Figure 4.2 from [18] shows an example of how the training set size affected a polynomial regression models. For the neural nets in this test done by [18] the original system modelled was a second degree polynomial function. Four models were created:

1. Degree 1 polynomial
2. Degree 2 polynomial
3. Degree 10 polynomial
4. Degree 25 polynomial

In this experiment, one can see that there is a plateau of the test-error. This error is made up of two components; the noise floor and the structural error. The noise floor comes from the inherent uncertainty in the modelled process, and is not possible to reduce to zero. The structural error comes from the algorithms inability to model the system. From a) in Figure 4.2, one can see that both the

training error and the test error for the first degree polynomial model remains high for all sizes of data set. This is because the model is not complex enough to predict the original system. Though for any model that is complex enough to model the system, the test-error will approach the noise floor as the size of the data set approaches infinite. One can also see that the simpler models both start out with a lower error and converges towards the noise floor at a faster pace.

In [19] it is shown that for machine learning applications where the amount of available data is immense it is not always necessary to create complex machine learning techniques to accurately model a system. If one were able to gather data for all possible situations the system one is modelling can be in, the training error would be the same as the test error.

4.1.3 Error

During training, the neural network takes in the input-variables and converts them into a predicted output; it then checks the difference between the predicted output and the actual output. This difference is the error of the network. The two typical ways of calculating the error is using the squared error and the absolute error, shown in Equation 4.13 [20]. Both methods can be used as a performance indicator for a neural network.

When evaluating how the neural network performs one usually measures this by finding the mean error over a set of data; changing Equation 4.13 to Equation 4.14. When doing this the mean absolute error is often changed to be the mean absolute percentage error (MAPE). The mean squared error (MSE) is not changed when taking the mean.

$$L(Y, \hat{f}(X)) = \begin{cases} (Y - \hat{f}(X))^2, \text{squared error} \\ |Y - \hat{f}(X)|, \text{absolute error} \end{cases} \tag{4.13}$$

$$e\bar{r}r = \begin{cases} \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{f}(X)_i)^2, \text{Mean Squared Error} \\ \frac{1}{N}\sum_{i}^{N} = 1\frac{|Y_i - \hat{f}(X)_i|}{Y_i}, \text{Mean Absolute Percentage Error} \end{cases} \tag{4.14}$$

In addition to the method of estimating error, there are three different categories of error. Training error, validation error, and test error.

The training error and validation error are both used to calibrate the neural network such that it can be assumed it has been properly fit to the system it is supposed to model. It is used during training for the network to adjust iself

The test error, also sometimes called the generalization error is a measure of how good the neural network is at modelling the system it is supposed to model. The test error is found by making the neural network predict the outcome from data is has never encountered before.

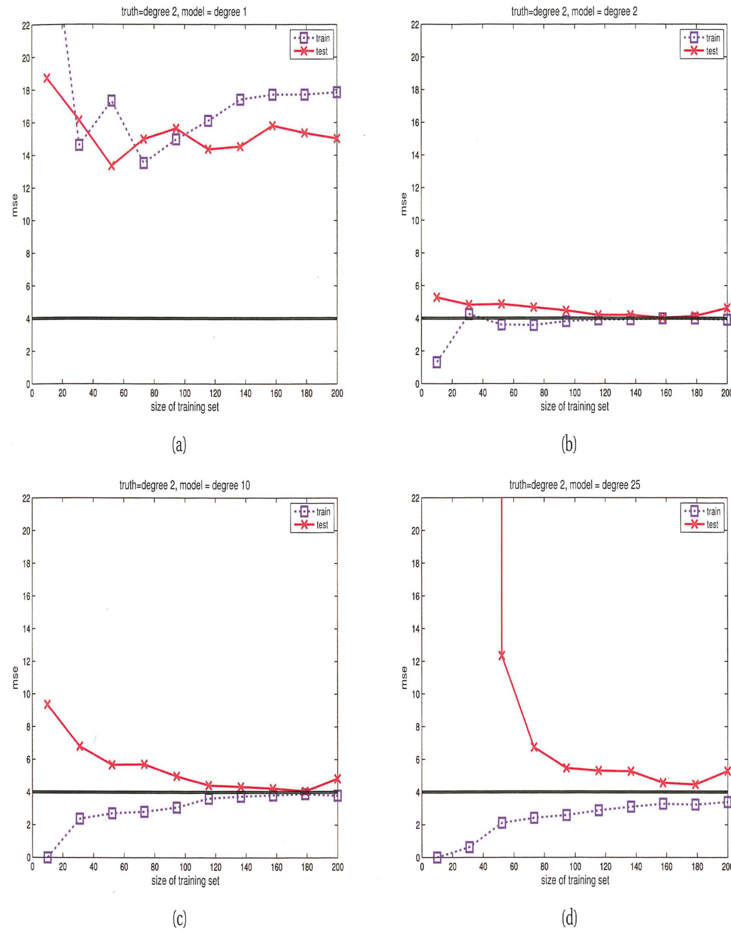


Figure 4.2: MSE on training and test sets vs size of training set, for data generated from a degree 2 polynomial with Gaussian noise of variance $\sigma^2 = 4$. We fit polynomial models of varying degree to this data. (a) Degree 1. (b) Degree 2. (c) Degree 10. (d) Degree 25. Note that for small training set sizes, the test error of the degree 25 polynomial is higher than that of the degree 2 polynomial, due to overfitting, but this difference vanishes once we have enough data. Note also that the degree 1 polynomial is too simple and has high test error even given large amounts of training data. Figure generated by linregPolyVsN. [18]

4.1.4 Learning rate

The method of steepest gradient decent is the basis of how most neural networks adjust their parameters to accurately model the chosen system. The neural network operates as an N-dimentional state-space, where N is the number of neurons in the neural network. Using steepest gradient decent, the neural network tries to move towards a local minima of error. The gradient decent can be described by 4.15. Where ϵ is the learning rate, which functions as the gradient step size [17]. Increasing learning rate will increase the rate at which it goes towards the optimum point, though if set too high it will likely overshoot.

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{4.15}$$

4.1.5 Optimization algorithm

A small numerical change in the value of the learning rate will impact how the neural network learns significantly [17], which means that finding the correct learning rate can be a difficult process. In neural networks with numerous input parameters each input parameter can have vastly differing impacts on the output. There is no consensus on which optimization algorithm is the most effective [21], though for this project the AdaGrad and Adam algorithms gave the best results.

Not all inputs and parameters of a neural network will impact the accuracy of the finished product equally. Due to this it can be desirable to not have a learning rate that is global, but rather adaptive and individual for each parameter. AdaGrad does this by creating individual learning rates for all parameters based on the global learning rate. The algorithm then scales all the learning rates proportional to [22]:

$$\text{Scaling factor} = \frac{1}{(\sqrt{\sum_{t=1}^{T}(\text{loss gradient}_t)^2})} \tag{4.16}$$

Where loss gradient_t is how much the loss (error) has changed from time step to time step.

The effect of AdaGrad is that parameters that has a large partial derivative of loss will decrease drastically, while parameters that have not seen large increases in their contribution to the accuracy is scaled more gently.

A potential downside of AdaGrad is that due to the accumulation of historical gradients the algorithm may result in excessive decrease in learning rate. In this project, that problem has been amended by setting a larger value for the initial learning rate.

Adam is can be seen as a variant of the Root Mean Squared Propagation (RMSProp) algorithm with moments [17]. Which is a modification of the AdaGrad algorithm intended to perform better in non-convex state-spaces. As mentioned the AdaGrad algorithm may decrease the learning rate too rapidly due to the accumulation of historical gradients. This can cause issues in state-spaces that has a

Algorithm 1 Pseudocode for the AdaGrad algorithm [17]

Require: Global learning rate ϵ
Require: Initial parameter θ
Require: Small constant ∂, perhaps 10^{-7} for numerical stability
  Initialise gradient accumulation variable $\mathbf{r} = 0$
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$
    Compute gradient: $\mathrm{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i \mathbf{L}((\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
    Accumulate squared gradient: $\mathrm{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$
    Compute update: $\Delta\theta \leftarrow (-\frac{\epsilon}{\partial + \sqrt{r}}) \odot \mathbf{g}$
    Apply update: $\theta \leftarrow \theta + \Delta\theta$
  end while

large enough local minima. The RMSProp algorithm aids with escaping such local minima by adding a exponentially decreasing average to the gradients based on when the gradients were calculated. The exponential decrease makes it such that older gradients are less important.

Adam incorporates the momentum of the change in gradient to calculate a first-order momentum (where the weights are applied expoentially). It then uses this momentum to rescale the learning rate and weights of a parameter [23]. According to [17], there are no clear motivation behind how the Adam algorithm is constructed, though it has been shown to be effective.

Algorithm 2 Pseudocode for the Adam algorithm [17]

Require: Step size ϵ
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in [0,1)
Require: Small constant δ used for numerical stabilization
Require: Initial parameters $\boldsymbol{\Theta}$
  Initialize 1st and 2nd moment variables $\mathbf{s} = 0$, $\mathbf{r} = 0$
  Initialize time step $t = 0$
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$
    Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i \mathbf{L}((\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
    Update biased first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$
    Update biased second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$
    Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
    Compute update: $\boldsymbol{\Delta\Theta} = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$
    Apply update: $\theta \leftarrow \theta + \Delta\theta$
  end while

4.1.6 Batch

Minibatch will affect the outcome in the following ways [17]:

- Larger batches increases how well the algorithm perform on the training- and validation-sets, but will give more complex solutions
- The advantages of having multiple cores on the processor that trains the algorithm is underutilised with small batch sizes
- Memory use scales with batch size, and is often the determining factor for maximum batch size
- If you are using a Graphics Processor Unit to run the training, using batch sizes correlating with powers of two will decrease run-time
- [24] showed that using small batch sizes could offer a regularization effect, they proposed that it might be due to the noise added by using small batches.

4.1.7 Activation function

According to [20] and [18], the deep feedforward neural networks will collapse into linear regression models if one does not include an activation function.

The default recommendation for activation functions is the Rectified Linear Unit (ReLU) [25]. Which is a non-linear function described by Equation 4.17. The ReLU function is a piecewise linear function, which is an advantage because they behave so similarly to linear units [17]. The linearity of ReLU makes it such that the derivative is a constant value, and the second derivative is zero almost everywhere. In comparison to other activation functions that introduce second order effects, the ReLU activation function makes it such that the direction of the gradient is a far more useful metric [17]. The one potential downside with ReLU together with gradient based learning is that datapoints that result in a zero-activation from a neuron will cause the neuron to not learn anything from that datapoint.

$$g(x) = max\{0, z\} \tag{4.17}$$

4.2 Supervised learning algorithms

Three different supervised learning algorithms have been considered to model the shipboard electrical components, they are:

4.2.1 Support Vector Regression (SVR)

Support Vector Regsression (SVR) is a special case of Support Vector Machines (SVM) [].

Although support vector machines are a type of supervised learning algortihm, they are not a type of artificial neural network [].

SVR starts with a linear regression model []:

$$f(x) = x^\top \beta + \beta_0 \tag{4.18}$$

Where β is a unit vector and x are the inputs.
To handle non-linear effects, β is estimated by using:

$$H(\beta, \beta_0) = \sum_{i=1}^{N} V(y_i - f(x_i)) + \frac{\lambda}{2} + ||\beta||^2 \tag{4.19}$$

Where H is the Hessian matrix, λ is a regularization parameter set by the programmer, y are the outputs, and:

$$V_\epsilon(r) = \begin{cases} 0, |r| < \epsilon \\ |r| - \epsilon, |r| \geq \epsilon \end{cases} \tag{4.20}$$

is the error function, where $r = y_i - f(x_i)$. ϵ is a value set by the programmer. If one minimizes H with $\hat{\beta}$and$\hat{\beta_0}$ the solution has the form:

$$\hat{\beta} = \sum_{i=1}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i) x_i \tag{4.21}$$

$$\hat{f}(x) = \sum_{i=1}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i) \langle x, x_i \rangle + \beta_0 \tag{4.22}$$

where $\hat{\alpha}_i^*, \hat{\alpha}_i \geq 0$ and are solutions to:

$$\min_{\hat{\alpha}_i^*, \hat{\alpha}_i} \epsilon \sum_{i=1}^{N} (\hat{\alpha}_i^* + \hat{\alpha}_i) - \sum_{i=1}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i) + \frac{1}{2} \sum_{i,i'}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i)(\hat{\alpha}_{i'}^* - \hat{\alpha}_{i'}) \tag{4.23}$$

Equation 4.23 is constrained by:

$$\begin{gathered} 0 \leq \hat{\alpha}_i^*, \hat{\alpha}_i \\ \sum_{i=1}^{N} (\hat{\alpha}_i^* - \hat{\alpha}_i) = 0 \\ \hat{\alpha}_i^* \hat{\alpha}_i = 0 \end{gathered} \tag{4.24}$$

These constraints makes only a subset of the solution values nonzero, the solutions that lead to a nonzero solution are the support vectors.

4.2.2 Deep feedforward (DFF)

The Deep Feedforward Neural Network (DFF), also called the Multilayer Perceptron is a fairly simple form of neural network, though it performs well in many applications [17]. [26] showed that a DFF can model any suitably smooth function to any desired accuracy if the network is large enough.

The core of the network is the perceptron, which uses a weighted sum of it's inputs run through an activation function with an added bias. If the perceptron is constructed using ReLU as the activation function; the perceptron can be described as the function in Equation 4.25.

$$f(x; W, c, \mathbf{w}, b) = \mathbf{w}^\top \cdot max\{0, \mathbf{W}^\top \mathbf{x} + c\} + b \tag{4.25}$$

Where $f(x; W, c, \mathbf{w}, b)$ is the output from each neuron, $\mathbf{w}$ are the parameters that map from the input to the desired output , $\mathbf{W}$ are the weights of the linear transformation, $\mathbf{x}$ are the inputs to the neuron, $\mathbf{c}$ are the biases, and b is a constant. The values of $\mathbf{w}, \mathbf{W}, \mathbf{c}$, and b are determined by the training.

A more generic formulation of Equation 4.25 is:

$$f(x; W, c, \mathbf{w}, b) = \mathbf{w}^\top \cdot \text{activation function}\{\mathbf{W}^\top \mathbf{x} + c\} + b \tag{4.26}$$

Though as mentioned in subsection 4.1.7 ReLU is the most common activation function.

Both in [27], and in [28] a DFF is initialised as fully connected. This means that the input for a neuron comes from all neurons in the previous layer, and the output from a neuron goes to all neurons in the following layer. A DFF has no feedback connections [20]. A representation of a fully connected DFF is found in Figure 4.1.

4.2.3 Long Short Term Memory (LSTM)

Long short term memory (LSTM) differs from DFF in two main characteristics, it differs in the type of neuron used, and the direction that connection between neurons can take.

In a DFF all neurons in one layer is sends an output to all neurons in the next layer, but there is no information that is sent to a previous layer. LSTM is a form of Recurrent Neural Network (RNN) where there are feedback connections [18].

LSTM networks feature recurrence also inside each neuron in addition to recurrence between neurons. [29] were the first to propose the LSTM architecture, though they had static weights inside the self-loop shown in Figure 4.3. [30] added to the algorithm by dynamically changing the weights based on context.

The internal state, also known as the memory of each neuron is governed by:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right) \tag{4.27}$$

where $f_i^{(t)}$ is the forget gate unit described by Equation 4.28, $g_i^{(t)}$ is the external input gate unit described by Equation 4.29, $x_j^{(t)}$ is the vector with the current input, $h_j^{(t-1)}$ is the vector containing all LSTM neurons, b is the neuron bias, U is the input weights, and W is the recurrent weights. σ is the sigmoid function that is set to give out values between 0 and 1 [18]

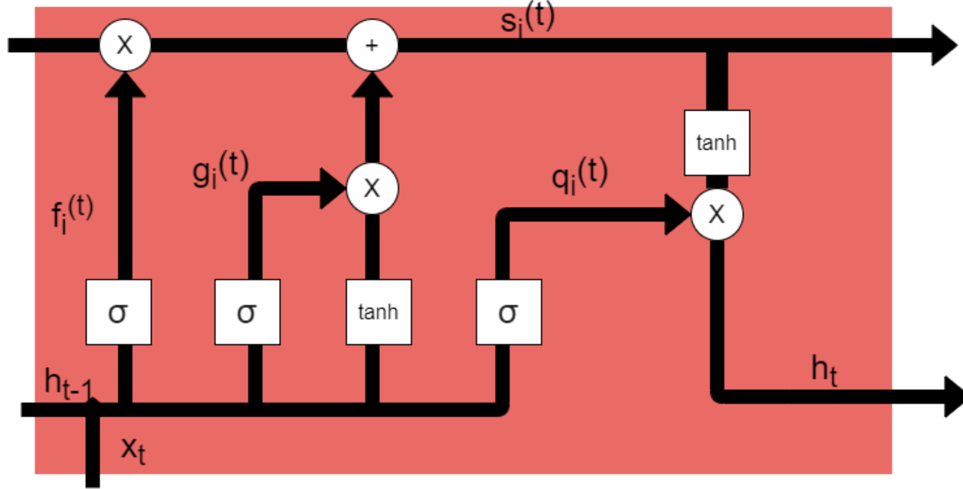


Figure 4.3: Illustration of LSTM neuron, area shaded in red is inside neuron

The forget gate unit, which controls whether the information stored in the neuron should be used for the incoming input or not can be described by:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{4.28}$$

Where $U_{i,j}^f$ are the weights for the inputs, b_i^f are the biases for the forget gate, and $W_{i,j}^f$ are the recurrent weights.

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right) \tag{4.29}$$

The output from the neuron is:

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)} \tag{4.30}$$

The output from the neuron $h_i^{(t)}$ can be suppressed by the output gate described by:

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right) \tag{4.31}$$

Comparing the equations governing an LSTM-neuron to the neuron used in the DFF network, there is a major difference in complexity per neuron.

The first use of LSTM networks were classification purposes [18] though they can also be used for time-series prediction as is the case in this thesis.

4.3 Overfitting and underfitting

4.3.1 Overfitting

Overfitting is a phenomenon that arises when the neural network has a great difference between the training-error and the test-error [17]. This indicates that the neural network is not usable outside the trainingset as it has not learned general rules for how the system being modelled behaves, only how the trainingset functions [18].

There are several analogies that can be used to describe overfitting: One can describe the neural network overfitting as a student preparing for an exam by memorizing the solution manuals for previous exams. The student could do well on the exam given that the exam is similar enough to previous exams, though the student will not know what to if the exam format is slightly changed. Similarly, an overfit neural network will probably do very well on new data that is very similar to the data it is trained on, but will give horrible estimates once it encounters data that is not within the original trainingset.

The most common way to detect overfitting is to observe the difference between training-error and test-error. If one has a very limited dataset, and therefore does not have the opportunity to throw away the test-set if it has been used;

4.3.2 Underfitting

Where overfitting is a problem one can only confirm by checking the difference between test-error and training-error, underfitting can usually be seen from training-error alone [18]. Underfitting is the situation when the training error is much larger than expected [17].

There can be many reasons for underfitting, though one of the common reasons is due to the complexity of the model being too small compared to the system being modelled. In Figure 4.2 a) a second degree polynomial is being modelled by a linear function, this leads to an underfit system, as one can see by the training error being significantly larger than in the other four polynomials modelling the original polynomial.

4.4 Growing and pruning

4.4.1 Pruning

The act of pruning a neural network is to reduce the size of a neural network, while keeping most of the accuracy intact. One of the first published papers written about pruning was [31] who studied the efficacy of pruning a neural network in stead of clipping.

In general, pruning can be divided into six steps [31]:

1. Training an unpruned model

2. Evaluating the unpruned model to find performance
3. Using some form of metric or algorithm to determine what should be removed from the model
4. Removing parts of the model
5. Re-train the model
6. Evaluate the accuracy of the pruned model

The method to decide which parts of the model to remove can be done either randomly or through the use of some decision making algorithm. If the amount of pruning being done to the model is small, the choice of method is not significant [32]. Though if one wants to reduce the size of the model significantly, many methods of intelligent pruning will outperform random choice [33].

When it comes to pruning, there is one question yet to be answered: Is it more or less effective than building an entirely new neural network with a more efficient architecture?

[31] compared reported accuracies of several models trained on the ImageNet database [34] and their results indicated three conclusions:

1. Pruning will significantly improve the ratio of $\frac{\text{accuracy}}{\text{size}}$, and even in some cases increase accuracy
2. In general pruning an inefficient architecture gives a worse accuracy per size than using a more efficient architecture
3. The improvement in $\frac{\text{accuracy}}{\text{size}}$ is proportional to how inefficient the model was before pruning

According to the No Free Lunch Theorem [2] constructing a model that is the theoretically best for all metrics and all use cases. There are always trade-offs for a model. Due to these trade-offs it is important to have a basis for comparison between pruning methods to find what method best suits the needs of a given model.

[31] found that identifying the information needed from existing literature to identify these trade-offs is lacking. The literature has a tendency to:

- Be too vague in describing the experimental setup and by what metrics they measure
- Focus on just a few combinations of dataset and architecture to make any general statements
- For the trade-off curves that are published, too few points of interest are included to find a central tendency.
- Few published articles about pruning techniques make comparisons between their technique and other state-of-the-art methods
- The literature generally does not control for variables that are not directly measured that could have influenced the results

4.5 Modelling Process

4.5.1 Experimental setup

All machine learning algorithms were run on the same computer, with the following components:

Table 4.1: Relevant system specifications

CPU	AMD Ryzen 5 3600X [3.8 GHz]
Cooling unit for CPU	Noctua NH-U14S
GPU	4095MB NVIDIA GeForce RTX 2060 SUPER
PSU	Corsair TX750M, 750W PSU
Motherboard	SUS ROG Strix B450-F GAMING, Socket-AM4
Installed RAM	16 GB
Installed storage	Corsair Force Series MP600 1TB M.2 SSD
OS	Windows 10 Home

All the LSTM-networks are trained using Keras 2.4.3 and TensorFlow 2.4.1, and are using the CPU as the computational hardware. The DFF-networks are trained using PyTorch 1.7.0, where the training has been done using the GPU as the hardware.

4.5.2 Libraries

PyTorch

PyTorch is a machine learning framework available for Linux, Mac, and Windows systems. The supported languages to use PyTorch is Python, C++ and Java [35]. It is compatible with machine learning on a CPU and on a GPU through NVIDIA CUDA [36]. It is an open-source project with an active developer community and comprehensive documentation.

[37] recommended to start using PyTorch when starting with neural network training, as it abstracts away a lot of the operations that will not contribute to the understanding of training and developing a neural network, while keeping the aspects of machine learning that is in the domain of Data Science. [37] has as a rule of thumb that one uses PyTorch when doing research and TensorFlow for production.

The PyTorch library was in this thesis used primarily to produce DFF networks.

TensorFlow

TensorFlow is a machine learning framework developed for internal use within Google, though in 2015 it was released as an open-source tool under the Apache

2.0 licence [28]. The Apache 2.0 [38] allows for the use of TensorFlow for any use without paying royalties.

For this project, TensorFlow was used in conjunction with Keras to create LSTM networks.

Keras

Keras is a tool built on top of TensorFlow 2.0 [39]. The main purpose of Keras is to make the process of writing code to build neural networks much faster, and the resulting code written by someone using Keras usually results in less lines of code written. Keras is in many ways similar to importing a library in Python where the library is written in C. Many of the most used libraries that are used in Python like numpy [40] are written at least partially in C to increase the efficiency when running the code. Keras makes it faster to use the core functionality of TensorFlow, while still keeping the advantages of having TensorFlow as a basis.

4.5.3 Case Study

To see if the neural network approach to model shipboard electrical component was a viable solution, the neural networks had to be designed, trained and tested. To train and test a neural network, data is needed.

The system chosen to gather data from was the vessel Skandi Africa [3]. The Single Line Diagram for the vessel is shown in Figure 1.2, Figure 1.3 is the part of the system that is modelled.

4.5.4 Width-Depth test

Due to the original neural network produced in the project thesis having a rather unconventional shape, a test was done in order to determine if that shape was the best shape given that the other hyper-parameters were fixed. In order to reduce the number of hours needed with human intervention in the test subsection 4.5.5 was created. subsection 4.5.5 keeps all the hyper parameters that are not width and depth constant for one run of the algorithm. This test could have been repeated with more nested loops, each loop corresponding to a hyper-parameter. A test of this nature would grow exponentially with both number of different hyper-parameters and amount of steps within each hyper-parameter. There are in total ten categories of hyper-parameters that can be adjusted by the DFF-code. This means that if one wants to test ten options per hyper-parameter one would have to train 10^{10} neural networks. If one could get the average time for each neural network training to take 5 seconds, the exploration would take over 1500 years.

1. Type of optimizer
2. Constant values needed by optimizer algorithm
3. Type of activation function
4. Width of each layer

5. Depth of network
6. Learning rate
7. Batch size
8. Number of epochs
9. Type of regularizer
10. Value of constant for regularizer function

4.5.5 Multirun algorithm

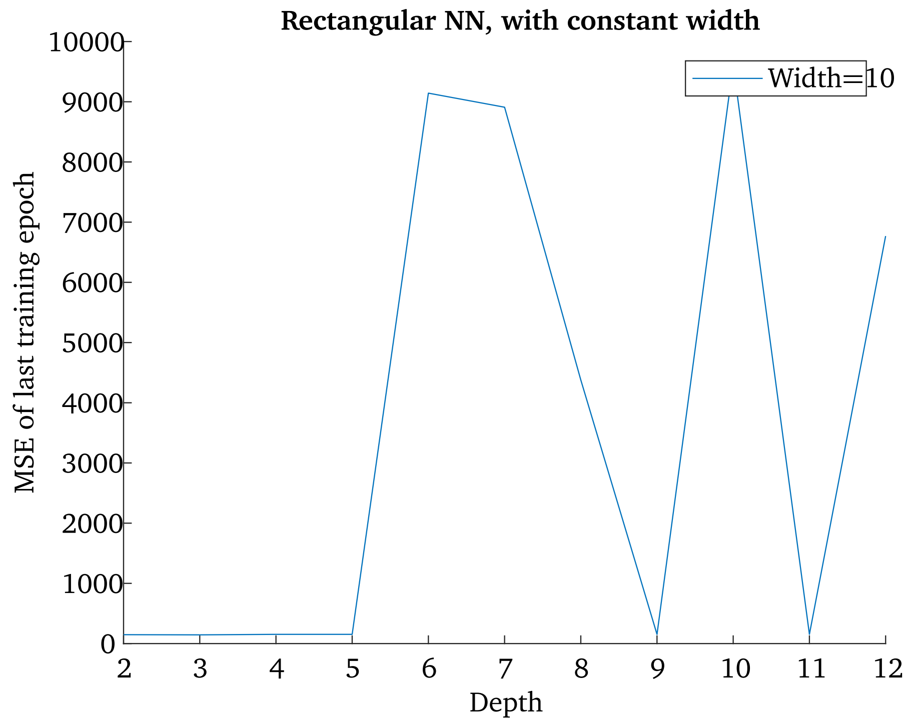


Figure 4.4: Accuracy of DFF network with constant width of 10 as depth changes

Algorithm 3 Pseudocode for the multirun script

Require: Learning rate ϵ
Require: Number of epochs E
Require: Shape of neural network, rectangular or conical
Require: Maximum allowable depth $Depth_{max}$
Require: Maximum allowable width $Width_{max}$
  Make an array Depthlist, enumerating from 2 to $Depth_{max}$
  Make an array Widthlist, that increases with 10 for each step. Starting at 10 and ending at $Width_{max}$
  if Shape of neural network is rectangular **then**
    for every Depth in Depthlist **do**
      for every Width in Widthlist **do**
        Create shape of neural network, by creating array that is Depth long and all values are Width
        Create log file for the final result
        Create log file for console output in case there are problems
        Create log file for how long it took to train this neural network
        Run script to train the neural network
      end for
    end for
  end if
  if Shape of neural network is conical **then**
    for every Depth in Depthlist **do**
      for every Width in Widthlist **do**
Require: Ratio between final and first hidden layer R
        Calculate the width change per layer Δ_{width} by ceil$\frac{\text{Width of first layer}}{\text{Depth}\cdot R}$
        Create shape of neural network, by creating array that is Depth long and all values are 0, called hidden_layers
        Fill the first entry of hidden_layers with starting Width
        for each entry in hidden_layers **do**
          Give each entry the value previous entry - Δ_{width}
          Create log file for the final result
          Create log file for console output in case there are problems
          Create log file for how long it took to train this neural network
          Run script to train the neural network
        end for
      end for
    end for
  end if

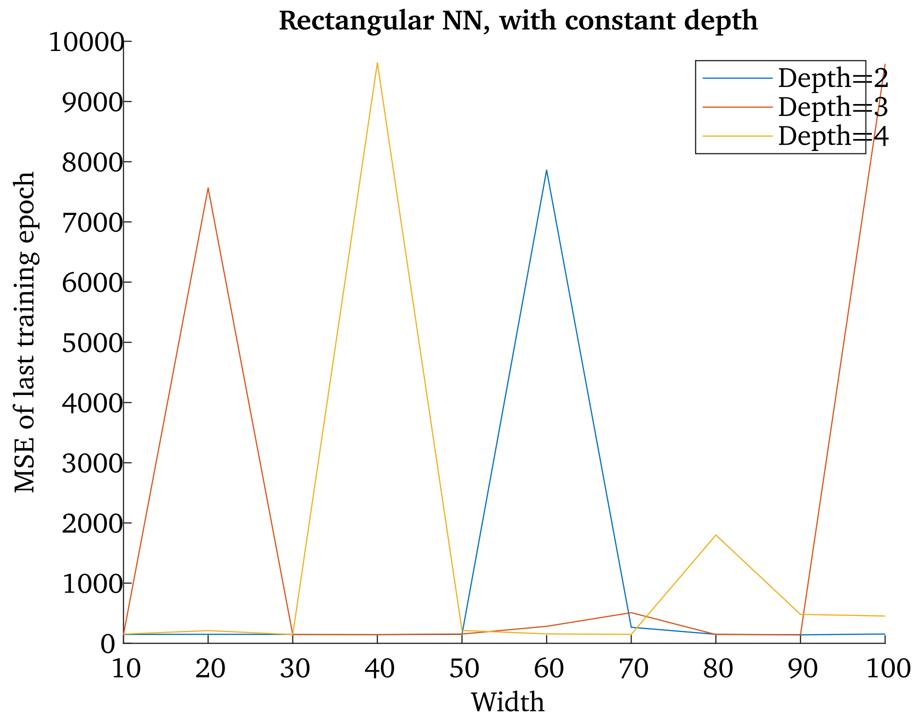


Figure 4.5: Accuracy of DFF with constant depths of 2,3, and 4 changes by altering width

While Figure 4.4 and Figure 4.5 checks for how changing the width or depth for the DFF network if the width of each layer is the same, the DFF networks in Figure 4.6 and Figure 4.7 have decreasing width. That means that each layer contains fewer neurons than the preceding layer. The ratio between the width of the first and last hidden layer is 30, each step rounding up.

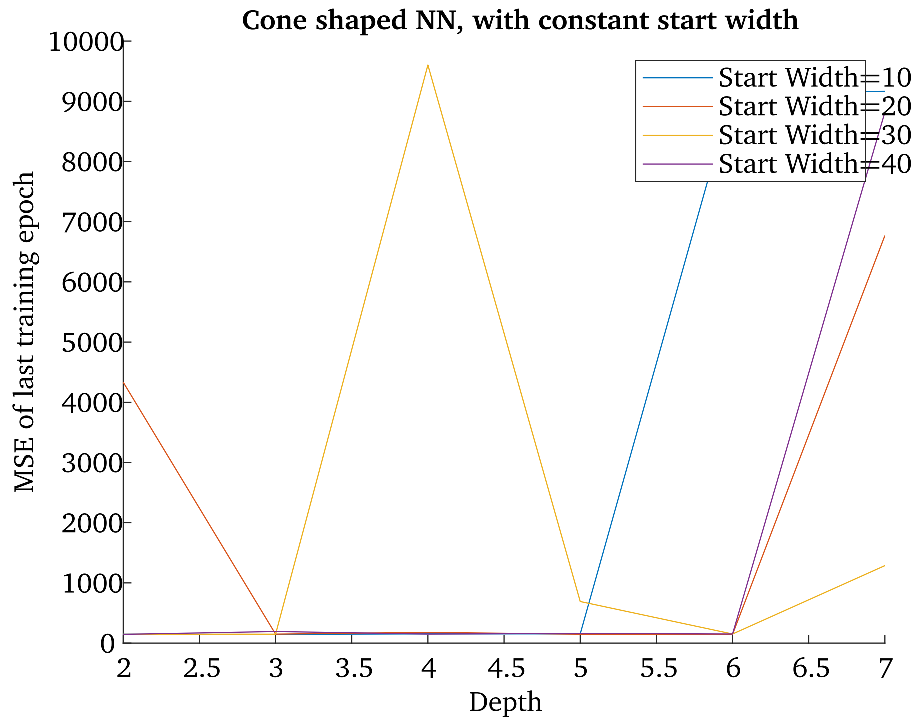


Figure 4.6: Constant start width and constant ratio $\frac{width_{firstlayer}}{width_{lastlayer}}$ DFF with changing depth.

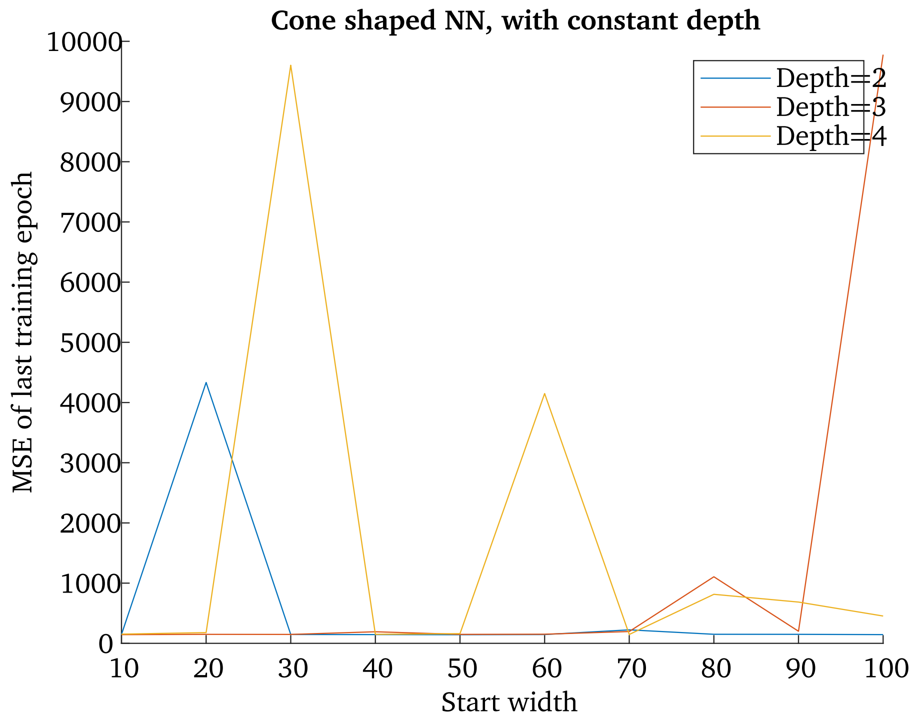


Figure 4.7: Constant depth and constant ratio $\frac{width_{firstlayer}}{width_{lastlayer}}$ DFF with changing start width.

The computational cost of doing the width-depth search is about 390 hours of running time, at the same loads as given in subsection 5.4.1.

Figure 4.4, Figure 4.5, and Figure 4.6 all show one general trend. The most accurate results came from the smallest networks. Some combinations of width and depth of the neural network yielded neural networks with a large MSE, though almost all neural networks with either very large width or depth ended up with accuracies far above the smaller neural networks.

The one exception to this trend is the cone shaped neural network with a depth of two. This network only improved as the width of the first layer increased.

As expected, the results from the brute force tuning of these two hyper parameters took a huge amount of computational power, yet yielded accuracies worse than the human tuned neural network. If brute force tuning were to be applied to all hyper-parameters in the same way, tuning a model for a shipboard power plant would take longer than the expected lifetime of the vessel.

4.5.6 DFF

The base code for the DFF-networks comes from the course content in TTK28 - Modelling with Neural Networks [41]. The code consists of:

- The __init__ function, which initialises the neural net.

- Then forward function, which defines how to pass information between the layers of the hidden layer
- get_num_parameters which is used to check the number of parameters in the neural net, which is useful information to get when tuning the neural net
- The train function which defines how the training loop of the neural net should happen
- the main() function, which takes in all the above functions so that the code can be executed on command

4.5.7 LSTM

The base code for the LSTM-based networks is a modified version of a framework made by Jakob Aungiers, which published his code under the GNU Affero General Public License v3.0 at [42]. There are several reasons this code was chosen as a base for the LSTM in the thesis:

- The code is very readable, making alterations and additions easier
- The base code already had support for LSTM, DFF, and dropout layers
- The base code is set up such that all hyper-parameters can be altered from a single .json file
- The code was published under a licence which allows the usage and alteration of the code without the need to pay royalties to the creator

The code from Jakob Aungiers have been altered, as there were some missing components that were not needed for his application, though they were needed for the more complex systems modeled in this thesis.

The first change in the code is in the data processor. The original solution is a computationally simple implementation described by 4. This solution worked for the intended use case, though when it encounters cases such as:

Algorithm 4 Pseudocode for the normalize windows algorithm

```
Require: Window_data WD
  Create an empty array Normalized_data ND
  for each window in WD do
    Create an empty array normalized_window NW
    for Each column in window do
      Divide all values in column by the value in the first entry of that column
      Put the divided values into the correct column in NW
    end for
    Reshape NW to have the same shape as original window
  end for
  return ND
```

1. The first datapoint in the window has a zero-value

Table 4.2: Hypothetical dataset showing a potential problem with a normalizing method

	Input 1	Input 2	**Input 1 Normalized**	Input 2 Normalized
Datapoint 1	**0.1**	1	**1**	1
Datapoint 2	**1**	0.1	**10**	0.1
Datapoint 3	**2**	0.5	**20**	0.5

2. The first datapoint in the window has a significantly smaller value than the rest of the window

The first issue would result in a division by zero error. Avoiding a division by zero error can be done in several ways, though all of them would add computational complexity to the function, thus defeating the purpose of having this simplification. The second issue is not as common and does not lead to a crash, though it can defeat the entire purpose of normalization. One normalizes the data into an LSTM because the architecture works much better with values between 0 and 1[20]. Table 4.2 shows how the old normalizer would deal with a potential data-window that can arise by normalizing the data with the previous method; if the first data-point is significantly smaller than the following values for an input, the rest of the values for that input will be significantly larger than 1.

The While there are several ways to add to the original solution to remove the issues, the solution that ended up being chosen was to take advantage of sklearn.prepocessor.MinMaxScaler() [43] with the built in normalization function.

The second alteration was to add the option of L^1 and L^2 regularization to the network In TensorFlow L^1 and L^2 regularization can be added as one of three functions; L^1 regularization, L^2 regularization, or using L^1_L^2. While using L^1_L^2 is functionally the same as adding both a L^1 and a L^2 regularizer, adding two of them to the same layer results in a fatal error for the compiler. The model builder function found in section A.2 therefore had to add some simple logical statements that will make sure that only a single regularizer is used per layer.

The third change was to add an option to normalize the entire dataset at the same time, instead of normalizing per batch. Normalizing the entire dataset at once, instead of a batch by batch is done for time saving reasons. While the sklearn.preprocessing.MinMaxScaler() [43] used in the per batch normalization (see normalise_windows(self, window_data, single_window=False) in section A.1) is an efficient method, the number of times it has to run for datasets as large as used in this thesis makes the process take an unnecessary amount of time. The normalize_dataset(self, filename) in section A.1 takes advantage of the pandas library [44] to normalize the entire dataset in less than 3 seconds when run by the CPU listed in Table 4.1.

The fourth change was made to accommodate for the datasets that are spread out over several files. Instead of manually merging these files that can result in human error, there is a function that can handle the operation. The function also seems to get around the issue where opening a very large .csv file with excel can

cause some of the data to be lost.

Pruning

The pruning method implemented in this thesis comes from the tensorflow_model_optimization [45] library, more specifically it is mostly based upon using the prune_low_magnitude(*args, **kwargs) function within the library. While as stated in subsection 4.4.1 random choice pruning does outperform not doing any pruning if the metric one is evaluating a neural network by is $\frac{\text{accuracy}}{\text{size}}$, having a metric by which to choose what to prune is better than random choice. The prune_low_magnitude(*args, **kwargs) function uses the magnitude of the output from a neuron to decide what to prune away, pruning away the neurons that have consistently low outputs. The function takes in the wanted final sparsity as an input.

The pruning process can be summarized as:

1. Initialization:
   - Find the parameters it needs for the process in the configs object passed to the function
   - Define a name for the pruned model
   - Loading the dataset
   - Calculate the number of datapoints in the dataset
2. Create a new model by copying the old one, and altering it so it can be pruned [2]
3. Re-train the pruned model by using the TensorFlow fit() function
4. Remove the pruning variables added in step 2
5. Save the pruned and re-trained model
6. Open the pruned model and use the post-training quantization technique [46] to create a TF_lite model, which is even smaller than the pruned model

Step 6 is not strictly necessary, though the post-training quantization does reduce model size significantly. Due to the post-training quantization not being strictly necessary to the pruning process, the code does also contain a small_model function that does step 1-5 of the pruning process.

4.5.8 Engine

Two neural networks approaches were attempted to construct a model for the main engine aboard the Skandi Africa. The first approach was to use a growing approach , which was used to determine how many different types of inputs were needed to properly model the engine. To test the growing method, a DFF network was used.

The sensors from the Skandi Africa vessel that concerned the main engine were divided into four categories:

[2]The pruning function used needs to add a number of variables to the model, so that it can determine what to prune

1. Engine Speed, Speed of turbocharger, Exhaust Temperature After Turbine, Exhaust Temperature Before Turbine, the desired temperature of the PID controller for the engine
2. Charge Air Pressure, Fuel Oil Pressure
3. Temperature PID Control, ME Fuel Oil Press Before Filter
4. Exhaust temperature from each cylinder in the engine

The network started out by just using set 1, then it trained itself again using set 1 and set 2, for each run adding the next set of inputs. The desired output to find was the Active Power going to the Generator.

In order to reduce the need for human intervention, 3 was made, that kept the other hyper-parameters constant for each trial. After each finished trial, the results were analyzed to see which set of inputs performed the best, and in which ways the hyper-parameters could be adjusted to get a better result.

The DFF neural network would eventually be abandoned, after it seemed like the method plateaued at a mean average percentage error of 10%, which was too high of an error to be satisfactory. After failing to get any satisfying results with a DFF, an LSTM model was created instead.

The LSTM-network uses all the same inputs as the DFF network did, except the desired output is also an input for this network. When the network has finished training, it gets two subsequent samples from each input variables and tries to predict the next value for each input category. The loss of the network is the mean squared sum of how wrong it was for each prediction.

The data for the engine was normalized before starting the training set using the normalize_dataset(self, filename) in section A.1, as it saved time during the training process.

4.6 Converter

The converter is the only component that is modelled through the use of a DFF-network

4.6.1 Pre-processing

The data aquired from Kongberg Maritime for the converter was not pre-processed. Since the data is raw data, and comes from four different sensors with varying polling rates, the raw data takes the form as seen in Table 4.3. The data is unusable for the DFF architecture, so it has to be pre-processed before the training can be done.

While doing a forward hold [3] could have been done, it resulted in strange behaviour from the neural network. Therefore the data was processed by doing a linear interpolation between data points for each column of input data, such

[3] A forward hold would be to repeat the same value for a sensor reading until a new different reading is met

Table 4.3: Excerpt of the data used for training before it has gone through the pre-processing

Africa.404_XI_11016	Africa.571_TT_114	Africa.571_TT_124	Africa.871_CB_TR1_KW
	30.3998661		
166.980072			
		23.0329304	
			166
	30.4124737		
		23.1964645	
149.034424			
			224
	30.5385456		
209.195099			
		23.1084061	
			205
		23.0958271	
	30.4502945		
122.030472			
			198

that the issues with forward hold would not arise; while having a value for all the input and output parameters for all data points. In Table 4.4 one can see an excerpt of how the interpolated data looks like.

Choosing training, validation and test sets

Once the data set has been pre-processed, it went through four steps:

1. The input and output data was chosen according to Table 4.5
2. It was divided into training and test sets, with the test set being 10% of the total data
3. 10% of the training set was pseudorandomly extracted to form the validation set
4. The training and test sets were used to train and tune the neural net

Table 4.5: Inputs and outputs used

Input/Output	Tag	Desctiption
Input	Africa.571_TT_124	Temperature in transformer room
Input	Africa.571_TT_114	Temperature in thruster room
Input	Africa.871_CB_TR1_KW	Energy going into the transformer
Output	Africa.404_XI_11016	Energy consumed by the thruster motor

Table 4.4: Same part of the training-data set as shown in Table 4.3, after pre-processing

Africa.404_XI_11016	Africa.571_TT_114	Africa.571_TT_124	Africa.871_CB_TR1_KW
168.6037	30.42508	23.10212	200.4
167.7919	30.39987	23.07906	191.8
166.9801	30.40239	23.05599	183.2
163.9891	30.40491	23.03293	174.6
160.9982	30.40743	23.07381	166
158.0072	30.40995	23.1147	177.6
155.0163	30.41247	23.15558	189.2
152.0254	30.43769	23.19646	200.8
149.0344	30.4629	23.18179	212.4
164.0746	30.48812	23.16711	224
179.1148	30.51333	23.15244	220.2
194.1549	30.53855	23.13776	216.4
209.1951	30.5209	23.12308	212.6
191.7622	30.50325	23.10841	208.8
174.3292	30.48559	23.10212	205
156.8963	30.46794	23.09583	203.25
139.4634	30.45029	23.09223	201.5
122.0305	30.45282	23.08864	199.75
138.9791	30.45534	23.08504	198

The final distribution of the size of each set is shown in Table 4.6. The validation set is pseudorandomly extracted from the test set using a static seed. It is randomly extracted because it is resampled every epoch, and having a static validation set would lead to an overfitting to the subsection of the data that is extracted as validation set. The reason it is chosen using a static seed is to give the neural net the same basis for training each time, such that the difference in results is due to different values in the hyper-parameters and not due to a better selection of validation set.

Having the same training- and validation sets for the entire tuning process could lead to a situation where the hyper-parameters are tuned to the point of overfitting. To see whether this is the case or not, the neural net has been tested on the test-set, consisting of data the neural net has never seen before.

Table 4.6: Selection of training set, validation set, and test set

Set	Typical size in proportion to all the available data
Traning-set	81%
Validation-set	9%
Test-set	10%

Table 4.7: Values for hyperparameters used to obtain the results given in chapter 5

Hyper Parameter	Value/Function used
Activation Function	ReLU
Optimisation Algorithm	Adaptive Gradients
Learning rate (Lr)	0.7
Learning rate decay	$0.001 \times Lr \times (\frac{1}{n_{epochs}})$
Number of epochs n_{epochs}	12000
L2 regularisation	0.015
Width	25
Depth	500
Batch size	131072

Variation of hyper-parameters

Varying the learning rate of an algorithm is analogous to varying the proportional term in a PID-regulator. Having a learning rate that is too high will lead to the network ovencompensating for errors, and may never find the optimum configuration. A learning rate that is too low will take a long time to find the best configuration. When using the optimisation function used in this neural net; the adaptive gradient optimisation function, one can set the learning rate to be higher than if one were to use other types of optimisation function. This is due to the learning rate decay, which will decrease the learning rate based upon how well the network has improved.

Having a high number of epochs will give the neural net greater time to correct itself to find the best solution. It is usually limited by computational cost, though having too many epochs can lead to a situation where the neural network is overfitted for the training data.

When deciding the width of the neural net, one must take account for the complexity of the system that is going to be modelled. The complexity of the modelled system is proportional to the width of the network. The depth will influence the flexibility of the neural network.

4.6.2 Generator

The generator is modelled using an LSTM network, with the same performance indicator as the engine. As the other LSTM networks, the data is normalized before running the training algorithm. The generator uses the following inputs:

1. The voltage in the generator
2. Generator frequency
3. Generator active power Active power [kW]
4. Generator Power Factor
5. Generator Reactive power [kVAr]

6. Power produced over the last day
7. A series of numbers used for calculating the power produced over the last day
8. A series of numbers used for calculating the power produced over the last trip

4.6.3 Thruster

The thruster is modelled using an LSTM network, with the same performance indicator as the engine.

The inputs to the network is:

1. RPM of the thruster motor
2. The motor power
3. Current going through the motor
4. The temperature in the thruster room
5. The temprature in the converter room
6. The amount of power consumed

To test the degradation of the performance over time, the network was trained on a set of data from 10.09.2020 to 21.09.2020, and further tested on a set of data from 21.10.2020 to 02.11.2020.

4.7 Parallel neural network

To see if the best solution is one large neural net or several smaller neural networks, the parallel test was created. For the parallel-test, the data from all four pre-existing neural networks was combined into one .csv file. An LSTM neural network was then created and trained on the data-set.

The process of creating an tuning the parallel network took the following steps:

1. Take datasets taken at the same time from all four components
2. Combine the datasets into one large network
3. Set up the configuration file
4. Change the configuration

Step three and four had to be repeated quite a few times, because due to the complexity of the system the training crashed a few times. While training all of the LSTM networks, the loss of the network started slow and slowly got larger as the network saw more data. For each new epoch the loss at the end would be smaller. Though for the parallel network the loss would grow to infinity when it was between 40% and 60% done with the first epoch, giving out an MSE of NaN. This issue was amended by two alterations to the network:

- Increasing both the depth and the width of the network

- Adding L^2 regularization to each layer, and increasing the value of the regularization for each failed run

4.7.1 Comparing physics based modelling with machine learning based modelling

While both static and dynamic modelling can model a system with high fidelity while being computationally effective, they usually do not account for all factors that can be present in the system. [47] describes how the efficiency of a battery will degrade with age. [48] used data driven methods to predict how propeller and hull cleaning could affect the performance of a vessel.

Chapter 5

Results and discussion

In the following chapter the training results from the neural networks outlined in chapter 4 are presented. This includes their shape, architecture, and the error they had on the test-set. Then, the results from each neural network are described and presented. If applicable, the neural networks will be compared to either the other neural networks in the thesis or literature pertaining to the same type of component.

5.1 Engine

The first layer is the LSTM input layer, which can be divided into two sublayers. The first sublayer consists of 16 neurons, which takes in the 8 input values from two sequential time steps. The second sublayer consists of 200 LSTM neurons. It is a return layer which means that the feedback mechanisms in the neural net can send impulses back to this layer.

The second layer is a dropout layer, with a droput rate of 20%.

The third layer is an LSTM layer with 100 neurons, it is also a return sequence layer.

The fourth layer is an LSTM layer with 100 neurons, it is not return sequence layer.

The fifth layer is a dropout layer, with a droput rate of 25%.

The sixth and final layer is a dense layer, the same type of layer as a DFF network uses, with 8 neurons, a ReLU activation function and an L2-regularization value of 0.001.

Table 5.1: The effects of pruning the model trained on data from engine

	Model	**Pruned model**
Size of file	4051 KB	1032 KB
Reported MSE	0.0012	3.5922e-06
Time taken	6 minutes 59.9 seconds	36 minutes 42.2 seconds

The performance of the DFF-network for the engine can be explained by multiple factors. One reason could be that the data-set was simply not large enough. Though the LSTM-network trained on the same data-set had much better performance, therefore the size of the data set seems to not be the reason it did not perform as expected. Based upon the results from training the neural networks, it seems that using a DFF-network to model an engine yields worse results than using the LSTM-architecture.

When pruning a neural network, the expected result is a smaller network with worse accuracy. In this case the accuracy rose after pruning. This indicates that the original network has been over fit to certain parts of the data-set. As mentioned in subsection 4.4.1, pruning is more effective for networks where the original network is large and inefficient. The experimental results from this network seems to indicate that the size of the original network was large enough for it to negatively impact performance. The reported accuracy of the neural network is higher than what was achieved in [10], which indicates that the neural network some factor that was not covered by the dynamic model, which still had an impact on the result.

The generator could not be compared to the polynomial model from the Skandi Africa data, as power was not a recorded data when the data was gathered.

5.2 Thruster

The first layer is the LSTM input layer, which can be divided into two sublayers. The first sublayer consists of 18 neurons, which takes in the 6 input values from three sequential time steps. The second sublayer consists of 100 LSTM neurons. It is a return layer which means that the feedback mechanisms in the neural net can send impulses back to this layer.

The second layer is a dropout layer, with a droput rate of 30%.

The third layer is an LSTM layer with 75 neurons, it is also a return sequence layer.

The fourth layer is an LSTM layer with 25 neurons, it is not return sequence layer.

The fifth layer is a dropout layer, with a droput rate of 25%.

The sixth and final layer is a dense layer, the same type of layer as a DFF network uses, with 5 neurons, a ReLU activation function and an L2-regularization value of 0.0012.

Table 5.2: The effects of pruning the model trained on data from thruster

	Model	**Pruned model**
Size of file	292 KB	105 KB
Reported MSE	1.9324e-04	1.9101e-05
Time taken	10 minutes 50.4 seconds	22 minutes 43.7 seconds

This network was trained with three different data sets, the first data set was from 10.09.2020 to 21.09.2020, the second set of data was from 21.10.2020 to 02.11.2020. The difference in performance between the two data sets was insignificant. The network had significantly better accuracy when trained on a data-set which was constructed by combining the two data-sets.

The lack of difference in performance between the two data-sets can be caused by several factors. The two time periods were not far enough apart for the effects described in [48] to take effect, therefore one would not expect there to be a significant change in the behavior of the thruster. The data-set was also large enough, and spanned a long enough time period for the regularizing effect of big data to compensate for minute differences in performance from day to day.

The thruster model was tested up against the polynomial fit model from [13]. Using the polynomial:

$$\eta = f(power) = f(x) = \frac{-0.02949x^2 + 100.2849x + 11.7351}{x + 2.0215} \tag{5.1}$$

Where x is the power output from the thruster. Measured as a percentage of maximum output. The data was normalized with the same script used to normalize the LSTM networks, using a window size equal to the entire data-set.

The MSE on the polynomial model on the data from Skandi Africa was 0.002.

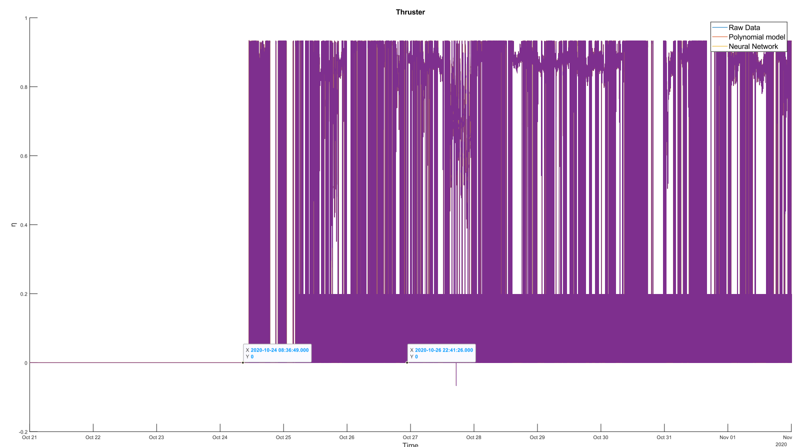


Figure 5.1: Efficiency of the thruster over time

Due to the data from the thruster changing rapidly

5.3 Generator

The first layer is the LSTM input layer, which can be divided into two sublayers. The first sublayer consists of 57 neurons, which takes in the 19 input values from

three sequential time steps. The second sublayer consists of 300 LSTM neurons. It is a return layer which means that the feedback mechanisms in the neural net can send impulses back to this layer.

The second layer is a dropout layer, with a droput rate of 30%.

The third layer is an LSTM layer with 150 neurons, it is also a return sequence layer.

The fourth layer is an LSTM layer with 75 neurons, it is not return sequence layer.

The fifth layer is a dropout layer, with a droput rate of 25%.

The sixth and final layer is a dense layer, the same type of layer as a DFF network uses, with 10 neurons, a ReLU activation function and an L2-regularization value of 0.001.

Table 5.3: The effects of pruning the model trained on data from generator

	Model	**Pruned model**
Size of file	8529 KB	22774 KB
Reported MSE	0.0033	1.2813e-05
Time taken	9 minutes 50.7 seconds	20 minutes 47.8 seconds

The generator model was tested up against the polynomial fit model from [13]. Using the polynomial from an electric generator:

$$\eta = f(power) = f(x) = \frac{-0.026024x^2 + 97.0165x + 5.3057}{x + 0.76677} \tag{5.2}$$

Where x is the power output from the generator. Measured as a percentage of maximum output. The data was normalized with the same script used to normalize the LSTM networks, using a window size equal to the entire data-set. The model was then tested on the training set, and efficiency was calculated by dividing the power measured going out of the generator compared to the power measured going in.

The MSE of the polynomial model on the data from Skandi Africa was 0.0013.

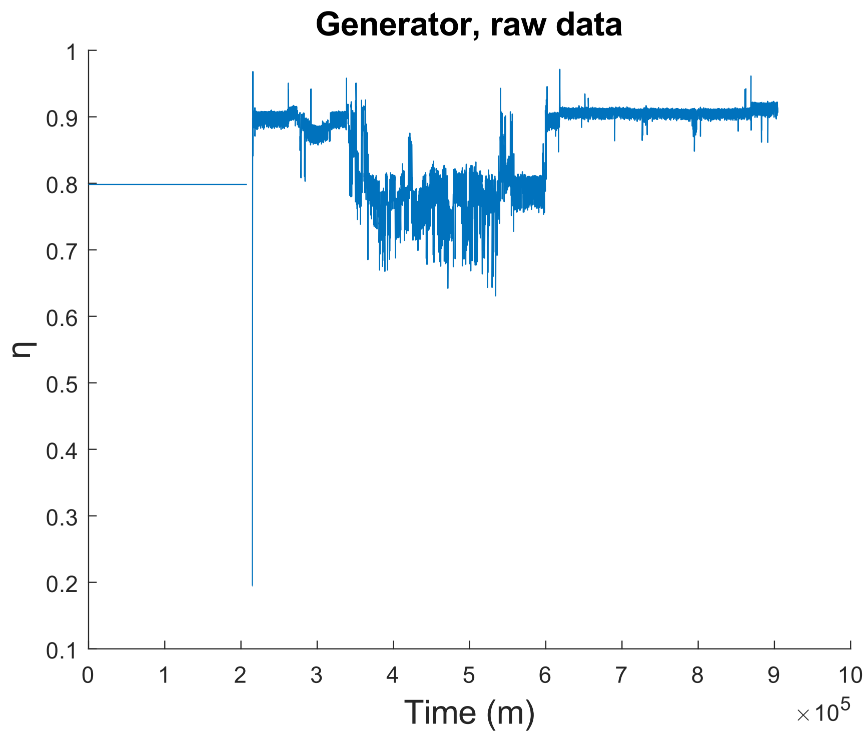


Figure 5.2: Efficiency of the generator over time, raw data

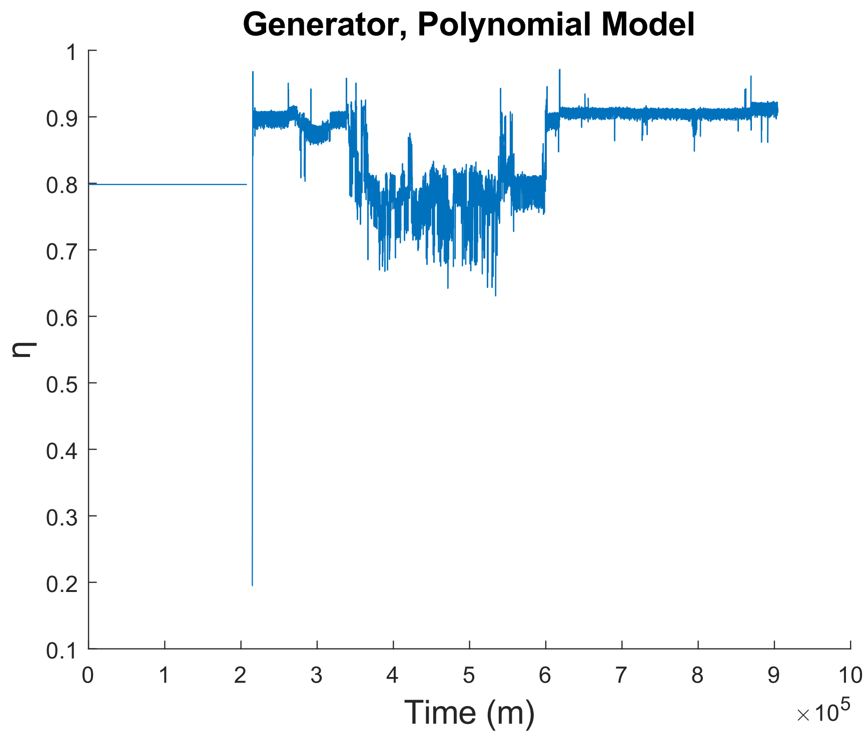


Figure 5.3: Efficiency of the generator over time, calculated using polynomial model

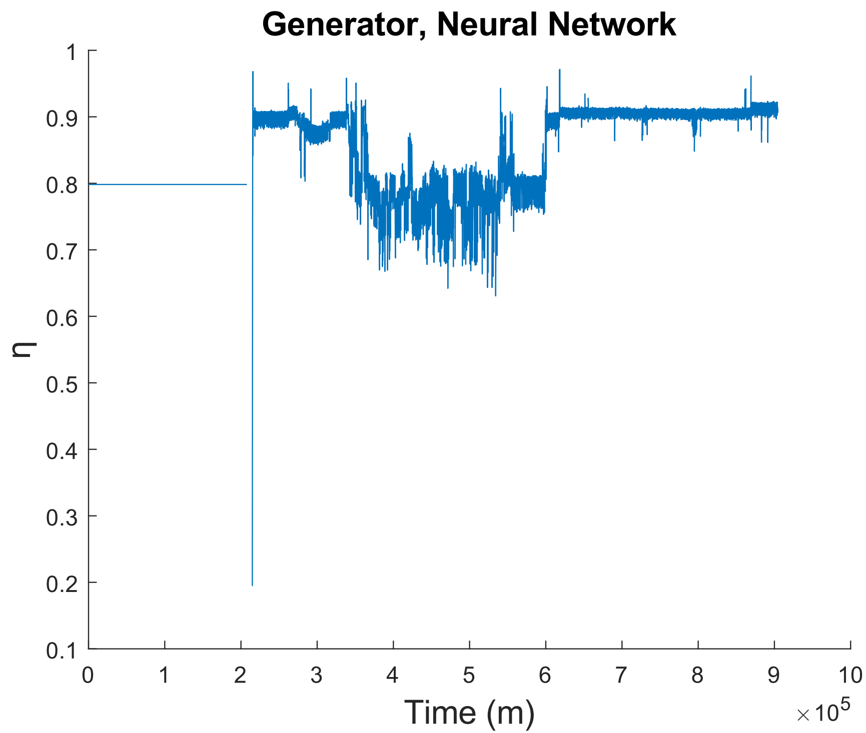


Figure 5.4: Efficiency of the generator over time, calculated using neural network

5.4 Converter

5.4.1 Estimated Training cost

Training the neural network took 2405 seconds. It was trained exclusively using the Graphics Processor Unit, which was a NVIDIA RTX 2060 Super [49].

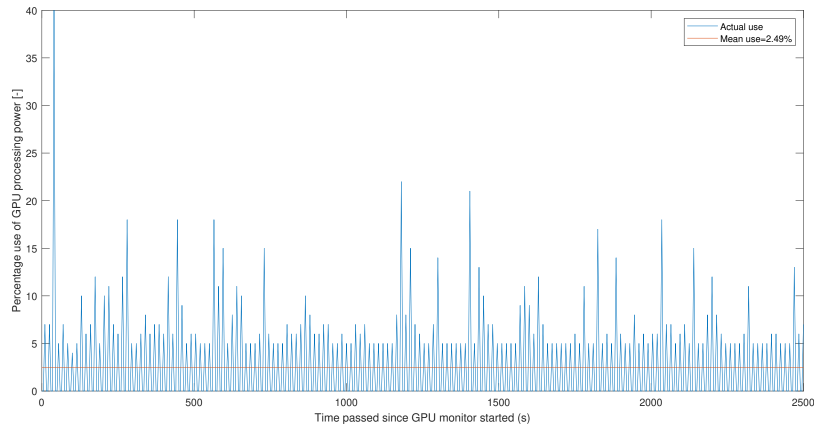


Figure 5.5: Percentage usage of the GPU processing power during the training

5.4.2 Results from training

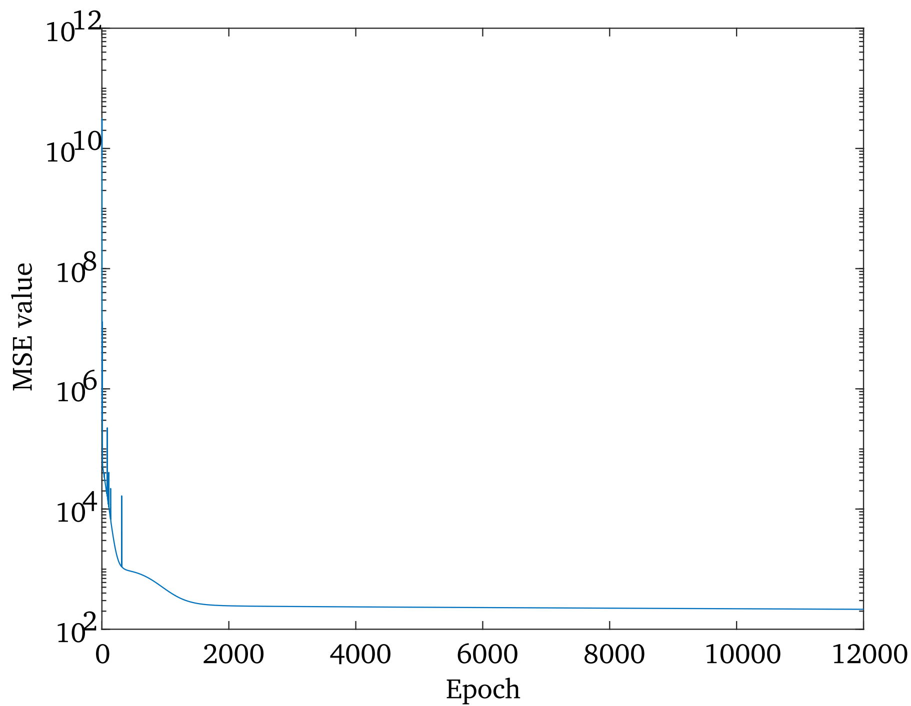


Figure 5.6: MSE value of validation error over the epochs, due to the extreme improvement the MSE-axis has to be shown in log-scale

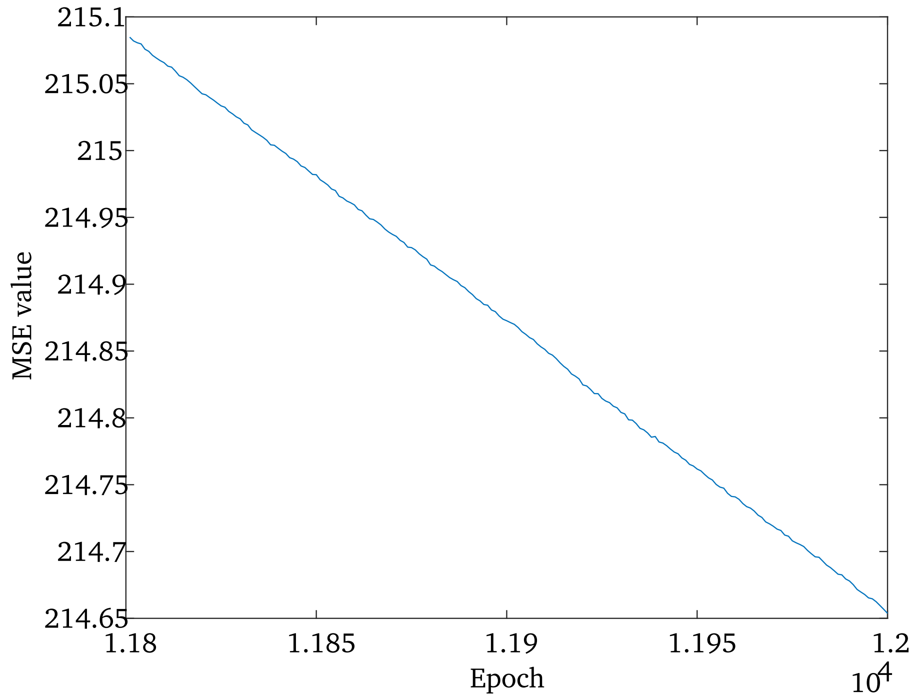


Figure 5.7: Linear improvement over the last 200 epochs

The converter model was tested up against the polynomial fit model from [13]. Using the polynomial from a buck converter:

$$\eta = f(power) = f(x) = \frac{-0.07178x^2 + 99.1532x + 14.1382}{x + 0.18274} \tag{5.3}$$

Where x is the power output from the converter. Measured as a percentage of maximum output. The data was normalized with the same script used to normalize the LSTM networks, using a window size equal to the entire data-set. The model was then tested on the training set, and efficiency was calculated by dividing the power measured going out of the converter compared to the power measured going in.

The MSE of the polynomial model on the data from Skandi Africa was 0.071.

5.4.3 Results on test set

The results from the test set can be found in Table 5.4.

Table 5.4: MSE and MAE from test-set

Mean Squared Error	Mean Absolute Error
212.3276	7.51024

A MAE of 7.5 equates to about 5% mean average percentage error over the test-set. A 5% mean average percentage error means that for each value that the network produces, the real answer is on average ± 5% of the answer given.

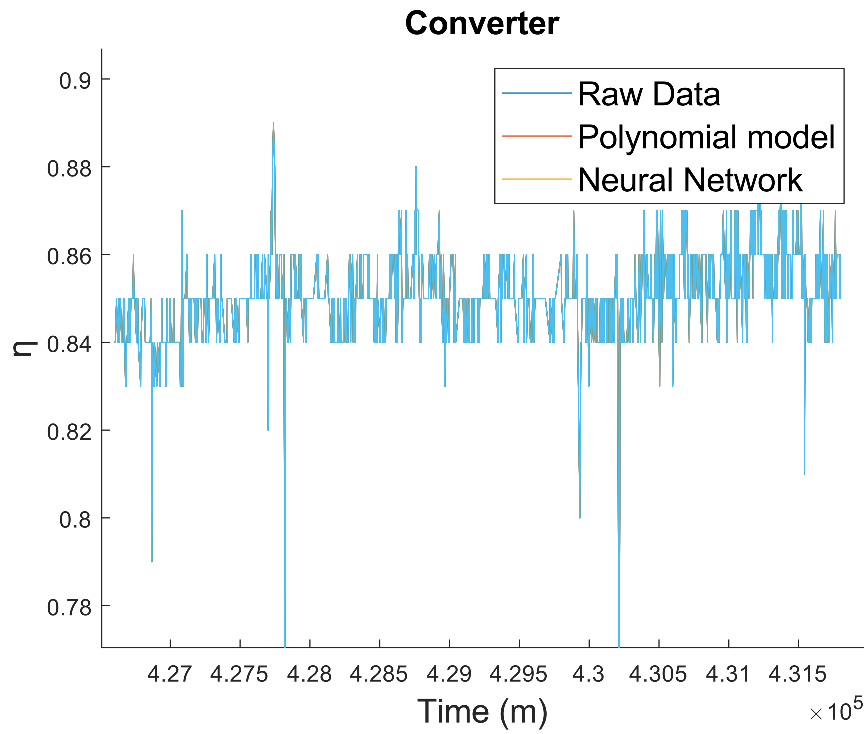


Figure 5.8: Efficiency of the converter over time

..

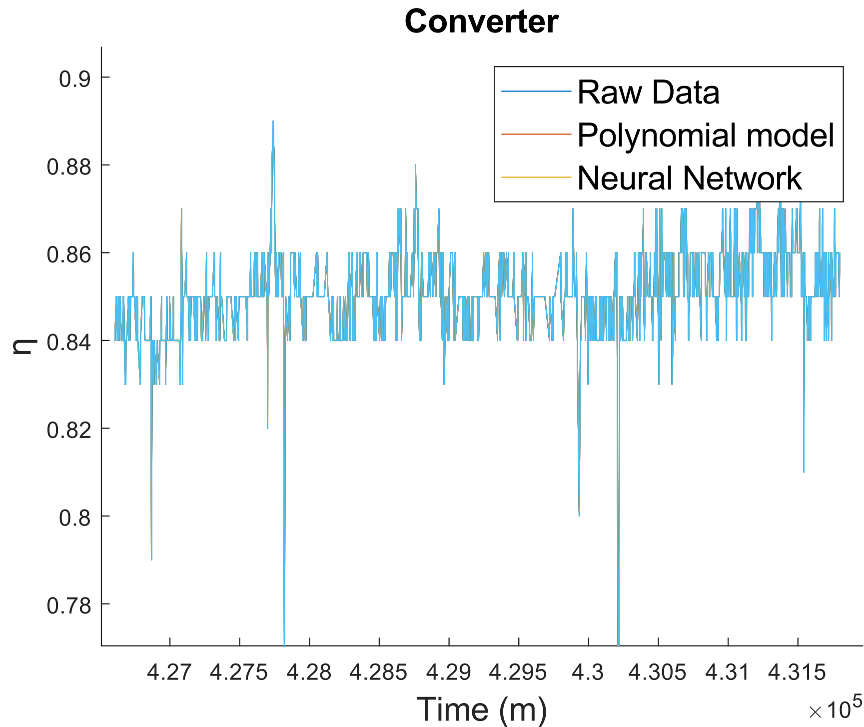


Figure 5.9: Efficiency of the converter over time, shorter time frame

Due to the converter data having a large data set, along with both models created having a small error, only a portion of the data is being plotted. Plotting the entire data-set leads to plots which are hard to read and interpret. There is a small difference in accuracy between the two models.

5.5 Parallel Network

The first layer is the LSTM input layer, which can be divided into two sublayers. The first sublayer consists of 138 neurons, which takes in the 49 input values from two sequential time steps. The second sublayer consists of 600 LSTM neurons. It is a return layer which means that the feedback mechanisms in the neural net can send impulses back to this layer.

The second layer is a dropout layer, with a droput rate of 20%.

The third layer is an LSTM layer with 100 neurons, it is also a return sequence layer.

The fourth layer is an LSTM layer with 100 neurons, it is not return sequence layer.

The fifth layer is a dropout layer, with a droput rate of 25%.

The sixth and final layer is a dense layer, the same type of layer as a DFF network uses, with 8 neurons, a ReLU activation function and an L2-regularization value of 0.001.

Table 5.5: The effects of pruning the model trained on data from all four components

	Model	**Pruned model**
Size of file	73.5 MB	24.5 MB
Reported MSE	0.0283	1.85E-06
Time taken	1 hour 10 minutes 23.1 seconds	2 hours 25 minutes 12.5 seconds

While the error for the original network where the data from all components were used as input is worse than the individual components, the error is smaller than if one calculates the total error from all individual components using propagation of uncertainty.

As expected, the largest neural network gained the most from the pruning process.

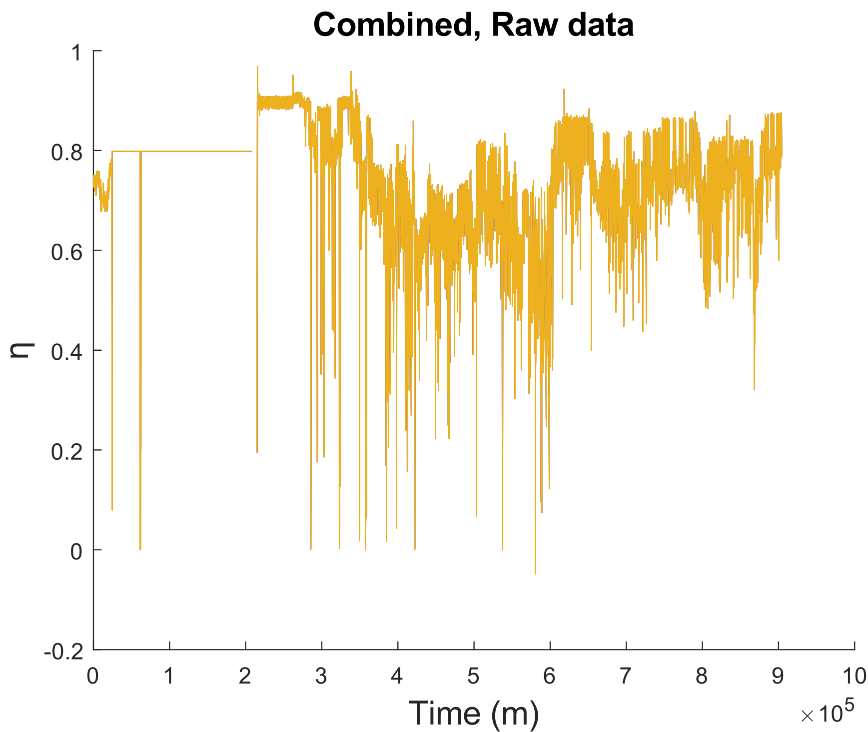


Figure 5.10: Combined efficiency of the thruster, generator, and converter over time, raw data

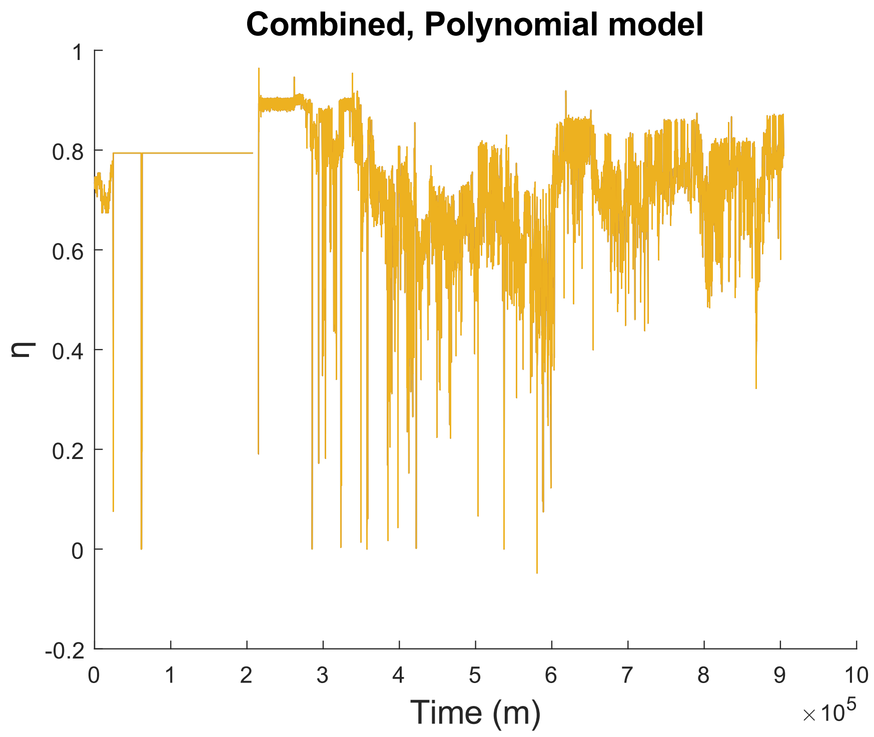


Figure 5.11: Combined efficiency of the thruster, generator, and converter over time, calculated using the polynomial model

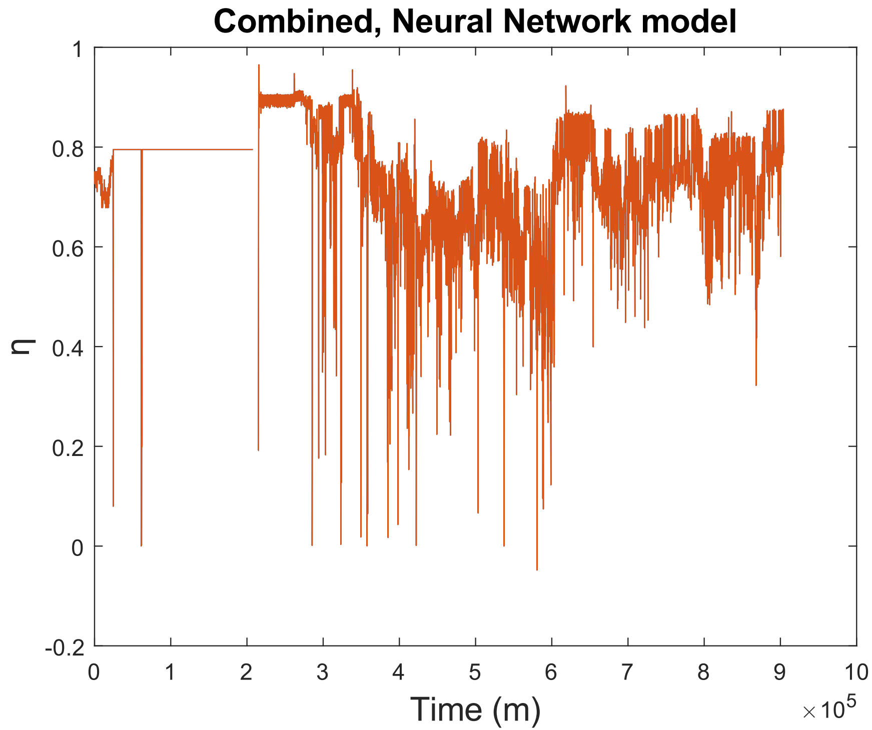


Figure 5.12: Combined efficiency of the thruster, generator, and converter over time, calculated using the neural network model

As mentioned in section 5.1, the model of the engine does not calculate the efficiency of the engine, as the data for power going in and power going out is not available. The data from the engine is part of the inputs to the combined neural network, though the output from the engine has not been taken into the formation of the graph.

For the combined model, when a component has been turned off, it is treated as having 100% efficiency. This is to avoid the system assuming 0% efficiency if a component is not being actively used.

Chapter 6

Conclusion

This thesis has investigated the viability of using artificial neural networks for modelling shipboard power components. The thesis aim was to find out if artificial neural networks could have a competitive advantage over physics based modelling or other data-driven modelling techniques.

Due to the nature of a machine learning based artificial neural network, using a case study in order to get access to data was necessary. Skandi Africa was the vessel that provided the data. Skandi Africa provided enough data for the regularizing effect of big data to have an effect on the accuracy of the models.

The artificial neural networks seem to be more accurate than the investigated physics based modelling techniques, although with a larger demand for computational power.

6.1 Recommendation for further work

This thesis is the first step in the process of using data driven modelling using machine learning to model shipboard electrical components. The area with the most potential for improving the data-driven modelling approach described in this thesis would be to improve the data gathering process. At the moment it is a process that needs substantial human intervention, which is an inefficiency in the system. The program should be expanded to include a module that gathers data directly from a server such that it can be used for training or retraining of the neural networks. The code can be used for retraining of existing neural networks, and has been used to retrain neural networks after the pruning process.

In addition to the data gathering process being improved, more data is also needed. Even though no degradation of performance for the neural networks were found, this might be because the data gathered all was from within a time scope of less than a year. There might be factors in the systems being modelled that do not show in such a short time-frame. At the time of writing this thesis, the components modelled has not aged enough for a performance degradation to be found by the neural networks.

In this thesis, four components of the larger system has been modelled. They have been modelled both as separate models and as a combined system. The results indicate that the better solution has been to model the system as a whole is the best solution, though that is not necessarily true for a system that encompasses all shipboard electrical components aboard a large vessel. Therefore the library of neural networks has to be expanded to include all components aboard a vessel, both as separate models and as a combined model to verify if the findings in this thesis can be generalized. While the models made do function on the computer they were trained on, they have not been tested to work on the hardware that is aboard the ship in the case study. Because the purpose of constructing the model has been for them to be used at the hardware aboard a ship in order to aid with reducing fuel use, they hardware about a ship has to be able to run the models. To be able to do this one would have to follow the process of:

1. Identify how much computing power is available on a vessel on a case-by-case basis
2. Prune the neural networks down to a size where they can reasonably be run
3. Do a Hardware-In-The-Loop test
4. If Hardware-In-The-Loop test is successful, make it compatible with the rest of the software

Identifying how much computing power is available to each model is crucial because if the models are too large, it will not be possible to run the models in real time. If the models are not able to be run in real time, they will not be useful in decision making aboard a vessel. The results in this thesis suggests that the pruning process resulted in a better performing neural network, though that might not hold true if the pruning process is aggressive enough. If the neural networks have to be pruned to the point where the accuracy falls below the acceptable threshold, one might have to consider finding an architecture that is more efficient from the start, which might not be any of the types of neural network used in this thesis.

In addition to the need for computing power, the hardware also needs to be able to run either the .h5 files or the tf-lite files that the program produces as the file to represent the neural networks.

If the Hardware-In-The-Loop test is successful, the models needs to be incorporated into the decision making software which is being developed by Kongsberg Maritime. Per now the models do not give out an estimate of energy loss as an estimate, although finding the energy loss is just a matter of subtracting the energy out from the energy coming in to a component. A piece of software that extracts the necessary data from the sensors aboard, feeds it to the neural networks, and converts the output from the neural networks into the desired format for the rest of the decision making algorithm needs to be made.

Bibliography

[1] I. M. Organization, *Tanaloa dialogue*. [Online]. Available: https://unfccc.int/process-and-meetings/the-paris-agreement/the-paris-agreement/2018-talanoa-dialogue-platform.

[2] D. H. Wolpert and W. G. Macready, 'No free lunch theorems for optimization.,' *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION,*, vol. 1, no. 22, pp. 69–82, 1997.

[3] *Skandi africa (offshore supply ship) registered in bahamas - vessel details, current position and voyage information - imo 9687459, mmsi 311000369, call sign c6bu7*. [Online]. Available: https://www.marinetraffic.com/en/ais/details/ships/shipid:2725642/mmsi:311000369/imo:9687459/vessel:SKANDI%5C%5FAFRICA.

[4] A. Papanikolaou, *Ship design - methodologies of preliminary design*. Springer, 2016.

[5] M.-I. Roh and K.-Y. Lee, *Computational ship design*. Springer Nature Singapore, 2018.

[6] O. Veneri, F. Migliardini, C. Capasso and P. Corbo, 'Overview of electric propulsion and generation architectures for naval applications,' in *2012 Electrical Systems for Aircraft, Railway and Ship Propulsion*, Oct. 2012, pp. 1–6. DOI: 10.1109/ESARS.2012.6387448.

[7] K. Group, *Kongsberg digital adds recogni as a new partner to the kognifai marketplace*, Jul. 2021. [Online]. Available: https://www.kongsberg.com/digital/resources/news-archive/2021/Kongsberg-Digital-adds-Recogni-as-a-new-partner-to-the-Kognifai-Marketplace/.

[8] H. Helgesen, S. Henningsgård and A. A. Langli, *Study on electrical energy storage for ships, report no.: 2019-0217, rev. 04*, Jan. 2021. [Online]. Available: http://www.emsa.europa.eu/publications/item/3895-study-on-electrical-energy-storage-for-ships.html.

[9] P. Ghimire, M. Zadeh, E. Pedersen and J. Thorstensen, 'Dynamic efficiency modeling of a marine dc hybrid power system,' in *2021 IEEE Applied Power Electronics Conference and Exposition (APEC)*, 2021, pp. 855–862. DOI: 10.1109/APEC42165.2021.9487343.

[10] P. Ghimire, M. Zadeh, J. Thorstensen and E. Pedersen, 'Data-driven efficiency modeling and analysis of all-electric ship powertrain: A comparison of power system architectures,' *IEEE Transactions on Transportation Electrification*, vol. 8, no. 2, pp. 1930–1943, 2022. DOI: 10.1109/TTE.2021.3123886.

[11] P. Ghimire, M. Zadeh, J. Thorstensen and E. Pedersen, 'Data-driven efficiency modeling and analysis of all-electric ship powertrain: A comparison of power system architectures,' *IEEE Transactions on Transportation Electrification*, vol. 8, no. 2, pp. 1930–1943, 2022. DOI: 10.1109/TTE.2021.3123886.

[12] *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society, Vienna, Austria, November 10-13, 2013*, IEEE, 2013. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6683943.

[13] P. Ghimire, M. Zadeh, E. Pedersen and J. Thorstensen, 'Dynamic efficiency modeling of a marine dc hybrid power system,' in *2021 IEEE Applied Power Electronics Conference and Exposition (APEC)*, 2021, pp. 855–862. DOI: 10.1109/APEC42165.2021.9487343.

[14] *Models and Methods for Efficiency Estimation of a Marine Electric Power Grid*, vol. Volume 7A: Ocean Engineering, International Conference on Offshore Mechanics and Arctic Engineering, V07AT06A039, Jun. 2017. DOI: 10.1115/OMAE2017-61625. eprint: https://asmedigitalcollection.asme.org/OMAE/proceedings-pdf/OMAE2017/57731/V07AT06A039/2533890/v07at06a039-omae2017-61625.pdf. [Online]. Available: https://doi.org/10.1115/OMAE2017-61625.

[15] B. Zahedi, L. E. Norum and K. B. Ludvigsen, 'Optimized efficiency of all-electric ships by dc hybrid power systems,' *Journal of Power Sources*, vol. 255, pp. 341–354, 2014, ISSN: 0378-7753. DOI: https://doi.org/10.1016/j.jpowsour.2014.01.031. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378775314000469.

[16] E. Skjong, T. A. Johansen, M. Molinas and A. J. Sørensen, 'Approaches to economic energy management in diesel–electric marine vessels,' *IEEE Transactions on Transportation Electrification*, vol. 3, no. 1, pp. 22–35, 2017. DOI: 10.1109/TTE.2017.2648178.

[17] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*. The MIT Press, 2017.

[18] K. P. Murphy, *Machine learning: a probabilistic perspective*. The MIT Press, 2012.

[19] A. Halevy, P. Norvig and F. Pereira, 'The unreasonable effectiveness of data,' *Intelligent Systems, IEEE*, vol. 24, pp. 8–12, May 2009. DOI: 10.1109/MIS.2009.36.

[20] T. Hastie, J. Friedman and R. Tisbshirani, *The Elements of statistical learning: data mining, inference, and prediction*. Springer, 2017.

[21] T. Schaul, I. Antonoglou and D. Silver, *Unit tests for stochastic optimization*, Feb. 2014. [Online]. Available: https://arxiv.org/abs/1312.6055.

[22] J. Duchi, E. Hazan and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization**, 2011. [Online]. Available: https://jmlr.csail.mit.edu/papers/volume12/duchi11a/duchi11a.pdf.

[23] D. P. Kingma and J. Baa, *Adam: A method for stochastic optimization*. [Online]. Available: https://arxiv.org/pdf/1412.6980.pdf%5C%20%5C%22%5C%20entire%5C%20document.

[24] D. Wilson and T. R. Martinez, 'The general inefficiency of batch training for gradient descent learning,' *Neural Networks*, vol. 16, no. 10, pp. 1429–1451, 2003, ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(03)00138-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608003001382.

[25] X. Glorot, A. Bordes and Y. Bengio, *Deep sparse rectifier neural networks*, Jun. 2011. [Online]. Available: http://proceedings.mlr.press/v15/glorot11a.

[26] K. Hornik, 'Approximation capabilities of multilayer feedforward networks,' *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991, ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089360809190009T.

[27] *Pytorch installation page*. [Online]. Available: https://pytorch.org/.

[28] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[29] S. Hochreiter and J. Schmidhuber, 'Long short-term memory,' *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[30] F. Gers, J. Schmidhuber and F. Cummins, 'Learning to forget: Continual prediction with lstm,' *Neural computation*, vol. 12, pp. 2451–71, Oct. 2000. DOI: 10.1162/089976600300015015.

[31] D. W. Blalock, J. G. Ortiz, J. Frankle and J. Guttag, 'What is the state of neural network pruning?' *ArXiv*, vol. abs/2003.03033, 2020.

[32] A. S. Morcos, H. Yu, M. Paganini and Y. Tian, *One ticket to win them all: Generalizing lottery ticket initializations across datasets and optimizers*, 2019. arXiv: 1906.02773 [stat.ML].

[33] T. Gale, E. Elsen and S. Hooker, *The state of sparsity in deep neural networks*, 2019. arXiv: 1902.09574 [cs.LG].

[34] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, 'Imagenet: A large-scale hierarchical image database,' in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[35] *Pytorch installation page*. [Online]. Available: https://pytorch.org/get-started/locally/.

[36] *Cuda zone*, May 2021. [Online]. Available: https://developer.nvidia.com/cuda-zone.

[37] B. Grimstad, *Building your first neural network*, lectures in TTK28 - modelling with neural networks.

[38] T. A. S. Foundation, The Apache 2.0 Licence. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0.

[39] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[40] C. R. Harris, K. J. Millman, S. J. der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Rio, M. Wiebe, P. Peterson, P. Gerard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke and T. E. Oliphant, 'Array programming with NumPy,' *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[41] B. Grimstad, *Bgrimstad/ttk28-courseware*. [Online]. Available: https://github.com/bgrimstad/TTK28-Courseware.

[42] J. Anguiers, *Lstm-neural-network-for-time-series-prediction*, https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction, 2019.

[43] *Sklearn.preprocessing.minmaxscaler*, Documentation. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[44] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[45] *Pruning in keras example: Tensorflow model optimization*, Guide on how to use the tensorflow_model_optimization library. [Online]. Available: https://www.tensorflow.org/model%5C%5Foptimization/guide/pruning/pruning%5C%5Fwith%5C%5Fkeras.

[46] *Post-training quantization*, Documentation for the post-training quantization technique in TensorFlow. [Online]. Available: https://www.tensorflow.org/lite/performance/post%5C%5Ftraining%5C%5Fquantization.

[47] S. Atalay, M. Sheikh, A. Mariani, Y. Merla, E. Bower and W. D. Widanage, 'Theory of battery ageing in a lithium-ion battery: Capacity fade, nonlinear ageing and lifetime prediction,' *Journal of Power Sources*, vol. 478, p. 229 026, 2020, ISSN: 0378-7753. DOI: https://doi.org/10.1016/j.jpowsour.2020.229026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378775320313239.

[48] P. Gupta, *Ship performance monitoring using in-service measurements and big data analysis methods*, Jan. 2022. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2988194.

[49] *Nvidia geforce rtx 2060 super*. [Online]. Available: https://www.nvidia.com/nb-no/geforce/graphics-cards/rtx-2060-super/.

[50] S. K. Helgesen, *Simenkh/datadrivenmodelling*, Jul. 2021. [Online]. Available: https://github.com/SimenKH/DataDrivenModelling.

Appendix A

Long short term memory code

The code can also be found at [50]

A.1 LSTM data processor

```
import math
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import itertools

class DataLoader():
   """A class for loading and transforming data for the lstm model
      ↪ """

   def __init__(self, filename, split, cols):
      dataframe = pd.read_csv(filename)
      i_split = int(len(dataframe) * split)
      self.data_train = dataframe.get(cols).values[:i_split]
      self.data_test = dataframe.get(cols).values[i_split:]
      self.len_train = len(self.data_train)
      self.len_test = len(self.data_test)
      self.len_train_windows = None

   def get_test_data(self, seq_len, normalise):
      '''
      Create x, y test data windows
      Warning: batch method, not generative, make sure you have
         ↪ enough memory to
      load data, otherwise reduce size of the training split.
      '''
```

```
        data_windows = []
        for i in range(self.len_test - seq_len):
            data_windows.append(self.data_test[i:i+seq_len])

        data_windows = np.array(data_windows).astype(float)
        data_windows = self.normalise_windows(data_windows,
            ↪ single_window=False) if normalise else data_windows

        x = data_windows[:, :-1]
        y = data_windows[:, -1, [0]]
        return x,y

    def get_train_data(self, seq_len, normalise):
        '''
        Create x, y train data windows
        Warning: batch method, not generative, make sure you have
            ↪ enough memory to
        load data, otherwise use generate_training_window() method.
        '''
        data_x = []
        data_y = []
        for i in range(self.len_train - seq_len):
            x, y = self._next_window(i, seq_len, normalise)
            data_x.append(x)
            data_y.append(y)
        return np.array(data_x), np.array(data_y)

    def generate_train_batch(self, seq_len, batch_size, normalise):
        '''Yield a generator of training data from filename on given
            ↪ list of cols split for train/test'''
        i = 0
        while i < (self.len_train - seq_len):
            x_batch = []
            y_batch = []
            for b in range(batch_size):
                if i >= (self.len_train - seq_len):
                    # stop-condition for a smaller final batch if data
                        ↪ doesn't divide evenly
                    yield np.array(x_batch), np.array(y_batch)
                    i = 0
                x, y = self._next_window(i, seq_len, normalise)
                x_batch.append(x)
                y_batch.append(y)
                i += 1
```

```
            yield np.array(x_batch), np.array(y_batch)

    def _next_window(self, i, seq_len, normalise):
        '''Generates the next data window from the given index
            ↪ location i'''
        window = self.data_train[i:i+seq_len]
        window = self.normalise_windows(window, single_window=True)[0]
            ↪  if normalise else window
        x = window[:-1]
        y = window[-1, [0]]
        return x, y

    def normalise_windows(self, window_data, single_window=False):
        '''Normalise window with a base value of zero'''

        normalised_data = []
        scaler=MinMaxScaler()
        window_data = [window_data] if single_window else window_data
        for window in window_data:
            normalised_window = []
            for col_i in range(window.shape[1]):
                try:
                    #normalised_col = [((float(p) / float(window[0, col_i
                        ↪ ])) - 1) for p in window[:, col_i]]
                    #print(normalised_col)
                    #print(type(normalised_col))

                    temp=[]
                    for p in window[:,col_i]:
                        temp.append(p)
                    temp=pd.DataFrame(temp,columns=['col'])
                    temp=scaler.fit_transform(temp)
                    normalised_col = temp.tolist()
                    normalised_col=list(itertools.chain.from_iterable(
                        ↪ normalised_col))
                    for item in normalised_col:
                        item=float(item)
                    #print(normalised_col)
                    #print(type(normalised_col2))
                    #print("heheheh")

                except ZeroDivisionError:
                    normalised_col = [((float(p) / (float(window[0, col_i
                        ↪ ])+0.01)) - 1) for p in window[:, col_i]]
```

```
                normalised_window.append(normalised_col)
            normalised_window = np.array(normalised_window).T # reshape
                ↪  and transpose array back into original
                ↪ multidimensional format
            normalised_data.append(normalised_window)
        return np.array(normalised_data)
    def normalize_dataset(self, filename):
        '''Normalize the entire dataset, instead of doing it window by
            ↪  window'''
        df = pd.read_csv(file,low_memory=False)
        for col in df:
            try:
                df[col]=df[col].to_numpy(dtype=np.float64)
                largestvalue=df[col].max()
                df[col]=df[col]/largestvalue
            except ZeroDivisionError:
                #if the largest value in a column is 0, then there is no
                    ↪  point to waste computing power
                print(col)
                df=df.drop(columns=col)
            except ValueError:
                #support for learning with data that are not numbers is
                    ↪ not yet supported"
                print(col)
                df=df.drop(columns=col)
        df.to_csv(filename)


def dataset_joiner(path,desired_filename):
    '''
    For joining together several files in the dataset into one larger
        ↪  file, if the dataset is spread out over several files
    Files must not be in the data folder in this project
    '''
    illegal_path=os.path.dirname(os.path.dirname(os.path.abspath(
        ↪ __file__)))+r'data\''
␣␣␣␣if␣path==illegal_path:
␣␣␣␣␣␣␣␣print("Please␣put␣the␣files␣you␣want␣to␣join␣in␣another␣
    ↪ folder␣to␣avoid␣messy␣results")
␣␣␣␣␣␣␣␣return
␣␣␣␣count=0
␣␣␣␣for␣file␣in␣os.listdir(path):
```

```
        if count==0:
            temp=path+file
            bigdf=pd.read_csv(temp,low_memory=False)
        else:
            temp=path+file
            df=pd.read_csv(temp,low_memory=False)
            bigdf.append(df)
        count+=1
    save=illegal_path+desired_filename+'.csv'
    bigdf.to_csv(save)
    print("Joined")
```

A.2 LSTM model

```
import os
import math
import numpy as np
import datetime as dt
from numpy import newaxis
from core.utils import Timer
from keras.layers import Dense, Activation, Dropout, LSTM
from keras.models import Sequential, load_model
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.regularizers import l1,l2,l1_l2
import tensorflow as tf
import tempfile
import tensorflow_model_optimization as tfmot
from tensorflow.keras.models import load_model
import sys
class Model():
      """A class for an building and inferencing an lstm model"""

      def __init__(self):
            self.model = Sequential()

      def load_model(self, filepath):
            print('[Model] Loading model from file %s' % filepath)
            self.model = load_model(filepath)

      def build_model(self, configs):
            timer = Timer()
            timer.start()
```

```
        for layer in configs['model']['layers']:
            neurons = layer['neurons'] if 'neurons' in layer
                ↪ else None
            dropout_rate = layer['rate'] if 'rate' in layer
                ↪ else None
            activation = layer['activation'] if 'activation'
                ↪ in layer else None
            return_seq = layer['return_seq'] if 'return_seq'
                ↪ in layer else None
            input_timesteps = layer['input_timesteps'] if '
                ↪ input_timesteps' in layer else None
            input_dim = layer['input_dim'] if 'input_dim' in
                ↪ layer else None
            L1 = layer['L1'] if 'L1' in layer else None
            L2 = layer['L2'] if 'L1' in layer else None
            reg=None
            print("boop")
            if ((L1!=None) and (L2!=None)):
                reg=l1_l2(l1=L1,l2=L2)
            elif (L1 != None):
                reg=l1(l1=L1)
            elif (L2 != None):
                reg=l2(l2=L2)

            if layer['type'] == 'dense':
                self.model.add(Dense(neurons, activation=
                    ↪ activation,kernel_regularizer=reg))
            if layer['type'] == 'lstm':
                self.model.add(LSTM(neurons, input_shape=(
                    ↪ input_timesteps, input_dim),
                    ↪ return_sequences=return_seq,
                    ↪ kernel_regularizer=reg))
            if layer['type'] == 'dropout':
                self.model.add(Dropout(dropout_rate))

        self.model.compile(loss=configs['model']['loss'],
            ↪ optimizer=configs['model']['optimizer'])

        print('[Model]␣Model␣Compiled')
        timer.stop()


    def train(self, x, y, epochs, batch_size, save_dir):
        configs = json.load(open('config.json', 'r'))
```

```
        timer = Timer()
        timer.start()
        print('[Model]␣Training␣Started')
        print('[Model]␣%s␣epochs,␣%s␣batch␣size' % (epochs,
            ↪ batch_size))
        if 'trained_model_name' in configs['data']:
            save_fname = os.path.join(save_dir,configs['data'
                ↪ ]['trained_model_name'] ,'trained_at_%s-e%s
                ↪ .h5' % (dt.datetime.now().strftime('%d%m%Y
                ↪ -%H%M%S'), str(epochs)))
        else:
            save_fname = os.path.join(save_dir, '%s-e%s.h5' %
                ↪ (dt.datetime.now().strftime('%d%m%Y-%H%M%S'
                ↪ ), str(epochs)))

        callbacks = [
            EarlyStopping(monitor='val_loss', patience=2),
            ModelCheckpoint(filepath=save_fname, monitor='
                ↪ val_loss', save_best_only=True)
        ]
        self.model.fit(
            x,
            y,
            epochs=epochs,
            batch_size=batch_size,
            callbacks=callbacks
        )
        self.model.save(save_fname)

        print('[Model]␣Training␣Completed.␣Model␣saved␣as␣%s' %
            ↪ save_fname)
        timer.stop()

    def train_generator(self, data_gen, epochs, batch_size,
        ↪ steps_per_epoch, save_dir):
        timer = Timer()
        timer.start()
        print('[Model]␣Training␣Started')
        print('[Model]␣%s␣epochs,␣%s␣batch␣size,␣%s␣batches␣per␣
            ↪ epoch' % (epochs, batch_size, steps_per_epoch))

        save_fname = os.path.join(save_dir, '%s-e%s.h5' % (dt.
            ↪ datetime.now().strftime('%d%m%Y-%H%M%S'), str(
```

```
            ↪ epochs)))
        callbacks = [
            ModelCheckpoint(filepath=save_fname, monitor='loss
                ↪ ', save_best_only=True)
        ]
        self.model.fit_generator(
            data_gen,
            steps_per_epoch=steps_per_epoch,
            epochs=epochs,
            callbacks=callbacks,
            workers=1
        )

        print('[Model]␣Training␣Completed.␣Model␣saved␣as␣%s' %
            ↪ save_fname)
        timer.stop()

    def predict_point_by_point(self, data):
        #Predict each timestep given the last sequence of true
            ↪ data, in effect only predicting 1 step ahead each
            ↪  time
        print('[Model]␣Predicting␣Point-by-Point...')
        predicted = self.model.predict(data)
        predicted = np.reshape(predicted, (predicted.size,))
        return predicted

    def predict_sequences_multiple(self, data, window_size,
        ↪ prediction_len):
        #Predict sequence of 50 steps before shifting prediction
            ↪  run forward by 50 steps
        print('[Model]␣Predicting␣Sequences␣Multiple...')
        prediction_seqs = []
        for i in range(int(len(data)/prediction_len)):
            curr_frame = data[i*prediction_len]
            predicted = []
            for j in range(prediction_len):
                predicted.append(self.model.predict(
                    ↪ curr_frame[newaxis,:,:])[0,0])
                curr_frame = curr_frame[1:]
                curr_frame = np.insert(curr_frame, [
                    ↪ window_size-2], predicted[-1], axis
                    ↪ =0)
            prediction_seqs.append(predicted)
        return prediction_seqs
```

```
    def predict_sequence_full(self, data, window_size):
        #Shift the window by 1 new prediction each time, re-run
            ↪ predictions on new window
        print('[Model]␣Predicting␣Sequences␣Full...')
        curr_frame = data[0]
        predicted = []
        for i in range(len(data)):
            predicted.append(self.model.predict(curr_frame[
                ↪ newaxis,:,:])[0,0])
            curr_frame = curr_frame[1:]
            curr_frame = np.insert(curr_frame, [window_size
                ↪ -2], predicted[-1], axis=0)
        return predicted


def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.
      ↪ ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)

def sparsity_pruning(configs,data,model,save_dir):
    prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude


    save_fname = os.path.join(save_dir, '%s-e%s.h5' % (dt.datetime
        ↪ .now().strftime('%d%m%Y-%H%M%S'), "pruned"))

    # Compute end step to finish pruning after 2 epochs.
    batch_size = configs["training"]["batch_size"]
    epochs = configs["training"]["epochs"]
    validation_split = configs["training"]["validation_split"]
    x, y = data.get_train_data(
    seq_len=configs['data']['sequence_length'],
    normalise=configs['data']['normalise']
  )
    num_data_points = x.shape[0] * (1 - validation_split)
```

```
    end_step = np.ceil(num_data_points / batch_size).astype(np.
        ↪ int32) * epochs

    # Define model for pruning.
    pruning_params = {
  'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(
      ↪ initial_sparsity=configs["pruning_parameters"]["
      ↪ initial_sparsity"],
                                                            final_sparsity=
                                                                ↪ configs["
                                                                ↪ pruning_parameters
                                                                ↪ "]["
                                                                ↪ final_sparsity
                                                                ↪ "],
                                                            begin_step=0,
                                                            end_step=end_step)
    }
    model_for_pruning = prune_low_magnitude(model, **
        ↪ pruning_params)

    # `prune_low_magnitude` requires a recompile.
    model_for_pruning.compile(optimizer=configs["model"]["
        ↪ optimizer"],
          loss=tf.keras.losses.SparseCategoricalCrossentropy(
              ↪ from_logits=True),
          metrics=['accuracy'])

    model_for_pruning.summary()
    logdir = tempfile.mkdtemp()

    callbacks = [
            tfmot.sparsity.keras.UpdatePruningStep(),
            tfmot.sparsity.keras.PruningSummaries(log_dir=logdir),
    ]

    model_for_pruning.fit(x, y,
              batch_size=batch_size, epochs=epochs, validation_split
                  ↪ =validation_split,
              callbacks=callbacks)
    try:
            model_for_pruning.save(save_fname)
    except:
            model_for_pruning.save(save_dir)
```

```
    return save_fname


def small_model(configs,data,model,save_dir):
    #if not os.path.exists(configs['model']['save_dir']): os.
        ↪ makedirs(configs['model']['save_dir'])

    #fname=sparsity_pruning(configs,data,model,save_dir)
    #model=load_model(fname)
    prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude


    save_fname = os.path.join(save_dir, '%s-e%s.h5' % (dt.datetime
        ↪ .now().strftime('%d%m%Y-%H%M%S'), "pruned"))

    # Compute end step to finish pruning after 2 epochs.
    batch_size = configs["training"]["batch_size"]
    epochs = configs["training"]["epochs"]
    validation_split = configs["training"]["validation_split"]
    x, y = data.get_train_data(
    seq_len=configs['data']['sequence_length'],
    normalise=configs['data']['normalise']
  )
    num_data_points = x.shape[0] * (1 - validation_split)
    end_step = np.ceil(num_data_points / batch_size).astype(np.
        ↪ int32) * epochs

    # Define model for pruning.
    pruning_params = {
   'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(
      ↪ initial_sparsity=configs["pruning_parameters"]["
      ↪ initial_sparsity"],
                                                  final_sparsity=
                                                      ↪ configs["
                                                      ↪ pruning_parameters
                                                      ↪ "]["
                                                      ↪ final_sparsity
                                                      ↪ "],
                                                  begin_step=0,
                                                  end_step=end_step)
    }
    model_for_pruning = prune_low_magnitude(model, **
        ↪ pruning_params)
```

```
    # ‘prune_low_magnitude‘ requires a recompile.
    model_for_pruning.compile(optimizer=configs["model"]["
        ↪ optimizer"],
         loss=tf.keras.losses.SparseCategoricalCrossentropy(
             ↪ from_logits=True),
         metrics=['accuracy'])

    model_for_pruning.summary()
    logdir = tempfile.mkdtemp()

    callbacks = [
          tfmot.sparsity.keras.UpdatePruningStep(),
          tfmot.sparsity.keras.PruningSummaries(log_dir=logdir),
    ]

    model_for_pruning.fit(x, y,
            batch_size=batch_size, epochs=epochs, validation_split
                ↪ =validation_split,
            callbacks=callbacks)
    model_for_export = tfmot.sparsity.keras.strip_pruning(
        ↪ model_for_pruning)

    _, pruned_keras_file = tempfile.mkstemp('.h5')
    tf.keras.models.save_model(model_for_export, pruned_keras_file
        ↪ , include_optimizer=False)
    print('Saved␣pruned␣Keras␣model␣to:', pruned_keras_file)

    converter = tf.lite.TFLiteConverter.from_keras_model(
        ↪ model_for_export)
    pruned_tflite_model = converter.convert()

    _, pruned_tflite_file = tempfile.mkstemp('.tflite')
    with open(pruned_tflite_file, 'wb') as f:
          f.write(pruned_tflite_model)

    print('Saved␣pruned␣TFLite␣model␣to:', pruned_tflite_file)
    return pruned_tflite_file

def very_small_model(configs,data,model,save_dir):
    old=sys.stdout
    sys.stdout=open("pruning-engine.txt",'w')
    timer = Timer()
    timer.start()
    prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude
```

```
    save_fname = os.path.join(save_dir, '%s-e%s.h5' % (dt.datetime
        ↪ .now().strftime('%d%m%Y-%H%M%S'), "pruned"))

    # Compute end step to finish pruning after 2 epochs.
    batch_size = configs["training"]["batch_size"]
    epochs = configs["training"]["epochs"]
    validation_split = configs["training"]["validation_split"]
    x, y = data.get_train_data(
    seq_len=configs['data']['sequence_length'],
    normalise=configs['data']['normalise']
)
    num_data_points = x.shape[0] * (1 - validation_split)
    end_step = np.ceil(num_data_points / batch_size).astype(np.
        ↪ int32) * epochs

    # Define model for pruning.
    pruning_params = {
  'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(
      ↪ initial_sparsity=configs["pruning_parameters"]["
      ↪ initial_sparsity"],
                                                    final_sparsity=
                                                        ↪ configs["
                                                        ↪ pruning_parameters
                                                        ↪ "]["
                                                        ↪ final_sparsity
                                                        ↪ "],
                                                    begin_step=0,
                                                    end_step=end_step)
    }
    model_for_pruning = prune_low_magnitude(model, **
        ↪ pruning_params)

    # 'prune_low_magnitude' requires a recompile.
    model_for_pruning.compile(optimizer=configs["model"]["
        ↪ optimizer"],
         loss=tf.keras.losses.SparseCategoricalCrossentropy(
             ↪ from_logits=True),
         metrics=['accuracy'])

    model_for_pruning.summary()
    logdir = tempfile.mkdtemp()
```

```
callbacks = [
      tfmot.sparsity.keras.UpdatePruningStep(),
      tfmot.sparsity.keras.PruningSummaries(log_dir=logdir),
]

model_for_pruning.fit(x, y,
        batch_size=batch_size, epochs=epochs, validation_split
            ↪ =validation_split,
        callbacks=callbacks)
model_for_export = tfmot.sparsity.keras.strip_pruning(
    ↪ model_for_pruning)

_, pruned_keras_file = tempfile.mkstemp('.h5')
tf.keras.models.save_model(model_for_export, pruned_keras_file
    ↪ , include_optimizer=False)
print('Saved␣pruned␣Keras␣model␣to:', pruned_keras_file)

converter = tf.lite.TFLiteConverter.from_keras_model(
    ↪ model_for_export)
pruned_tflite_model = converter.convert()

_, pruned_tflite_file = tempfile.mkstemp('.tflite')
with open(pruned_tflite_file, 'wb') as f:
      f.write(pruned_tflite_model)

print('Saved␣pruned␣TFLite␣model␣to:', pruned_tflite_file)
converter = tf.lite.TFLiteConverter.from_keras_model(
    ↪ model_for_export)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_and_pruned_tflite_model = converter.convert()

_, quantized_and_pruned_tflite_file = tempfile.mkstemp('.
    ↪ tflite')

with open(quantized_and_pruned_tflite_file, 'wb') as f:
      f.write(quantized_and_pruned_tflite_model)

print('Saved␣quantized␣and␣pruned␣TFLite␣model␣to:',
    ↪ quantized_and_pruned_tflite_file)

#print("Size of gzipped baseline Keras model: %.2f bytes" % (
    ↪ get_gzipped_model_size(keras_file)))
print("Size␣of␣gzipped␣pruned␣and␣quantized␣TFlite␣model:␣%.2f
    ↪ ␣bytes" % (get_gzipped_model_size(
```

```
        ↪ quantized_and_pruned_tflite_file)))
    timer.stop()
    sys.stdout.close()
    sys.stdout=old
```

A.3 LSTM main file

```
__author__ = "Jakob␣Aungiers"
__copyright__ = "Jakob␣Aungiers␣2018"
__version__ = "2.0.0"
__license__ = "MIT"

import os
import json
import time
import math
import matplotlib.pyplot as plt
from core.data_processor import DataLoader
from core.model import Model
import core.model as MDL
from tensorflow.keras.models import load_model
import sys
def plot_results(predicted_data, true_data):
   fig = plt.figure(facecolor='white')
   ax = fig.add_subplot(111)
   ax.plot(true_data, label='True␣Data')
   plt.plot(predicted_data, label='Prediction')
   plt.legend()
   plt.show()


def plot_results_multiple(predicted_data, true_data, prediction_len)
    ↪ :
   fig = plt.figure(facecolor='white')
   ax = fig.add_subplot(111)
   ax.plot(true_data, label='True␣Data')
      # Pad the list of predictions to shift it in the graph to it's
          ↪  correct start
   for i, data in enumerate(predicted_data):
      padding = [None for p in range(i * prediction_len)]
      plt.plot(padding + data, label='Prediction')
      plt.legend()
   plt.show()
```

```
def main():
    old=sys.stdout
    sys.stdout=open("engine.txt",'w')

    configs = json.load(open('config.json', 'r'))
    if not os.path.exists(configs['model']['save_dir']): os.makedirs(
        ↪ configs['model']['save_dir'])

    data = DataLoader(
        os.path.join('data', configs['data']['filename']),
        configs['data']['train_test_split'],
        configs['data']['columns']
    )

    model = Model()
    model.build_model(configs)
    x, y = data.get_train_data(
        seq_len=configs['data']['sequence_length'],
        normalise=configs['data']['normalise']
    )

    '''
        # in-memory training
        model.train(
                x,
                y,
                epochs = configs['training']['epochs'],
                batch_size = configs['training']['batch_size'],
                save_dir = configs['model']['save_dir']
        )
        '''
    # out-of memory generative training
    steps_per_epoch = math.ceil((data.len_train - configs['data']['
        ↪ sequence_length']) / configs['training']['batch_size'])

    model.train_generator(
        data_gen=data.generate_train_batch(
            seq_len=configs['data']['sequence_length'],
            batch_size=configs['training']['batch_size'],
            normalise=configs['data']['normalise']
        ),
        epochs=configs['training']['epochs'],
```

```
        batch_size=configs['training']['batch_size'],
        steps_per_epoch=steps_per_epoch,
        save_dir=configs['model']['save_dir']
    )

    x_test, y_test = data.get_test_data(
        seq_len=configs['data']['sequence_length'],
        normalise=configs['data']['normalise']
    )




    sys.stdout.close()
    sys.stdout=old

    #predictions = model.predict_sequences_multiple(x_test, configs['
        ↪ data']['sequence_length'], configs['data']['
        ↪ sequence_length'])
    #predictions = model.predict_sequence_full(x_test, configs['data
        ↪ ']['sequence_length'])
    #predictions = model.predict_point_by_point(x_test)
    #print(predictions)

    #plot_results_multiple(predictions, y_test, configs['data']['
        ↪ sequence_length'])
    # plot_results(predictions, y_test)


#if __name__ == '__main__':
#main()

def TestModel(model_filepath):
    configs = json.load(open('config.json', 'r'))
    if not os.path.exists(configs['model']['save_dir']): os.makedirs(
        ↪ configs['model']['save_dir'])
    model=Model()
    model.load_model(model_filepath)
    data = DataLoader(
        os.path.join('data', configs['data']['filename']),
        configs['data']['train_test_split'],
        configs['data']['columns']
    )
    x_test, y_test = data.get_test_data(
```

```
        seq_len=configs['data']['sequence_length'],
        normalise=configs['data']['normalise']
    )
    predictions = model.predict_sequences_multiple(x_test, configs['
        ↪ data']['sequence_length'], configs['data']['
        ↪ sequence_length'])
    predictions = model.predict_sequence_full(x_test, configs['data'
        ↪ ]['sequence_length'])
    predictions = model.predict_point_by_point(x_test)
    print(predictions)

main()


#TestModel(r"C:\Users\simen\source\repos\LSTM-Neural-Network-for-
    ↪ Time-Series-Prediction\saved_models\07072021-120542-e6.h5")

configs = json.load(open('config.json', 'r'))
data = DataLoader(
        os.path.join('data', configs['data']['filename']),
        configs['data']['train_test_split'],
        configs['data']['columns']
    )
x, y = data.get_train_data(
        seq_len=configs['data']['sequence_length'],
        normalise=configs['data']['normalise']
    )

save_dir="C:/Users/simen/source/repos/LSTM-Neural-Network-for-Time-
    ↪ Series-Prediction/pruned_models/"

model_1=load_model(r"C:\Users\simen\source\repos\LSTM-Neural-Network
    ↪ -for-Time-Series-Prediction\saved_models\07072021-170114-e4.
    ↪ h5")
MDL.very_small_model(configs,data,model_1,save_dir)
#open("results.txt",'w')
#MDL.very_small_model(configs,data,model,save_dir)
```

A.4 LSTM utilities

```
import datetime as dt

class Timer():
```

```
    def __init__(self):
        self.start_dt = None

    def start(self):
        self.start_dt = dt.datetime.now()

    def stop(self):
        end_dt = dt.datetime.now()
        print('Time␣taken:␣%s' % (end_dt - self.start_dt))
```

Appendix B

Deep feedforward code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov 18 15:38:53 2020

@author: simen
"""

# -*- coding: utf-8 -*-
"""
Created on Wed Sep 30 11:15:37 2020
@author: simen
base code taken from https://github.com/bgrimstad/TTK28-Courseware/
    ↪ blob/master/model/flow_model.ipynb
"""
import matplotlib.pyplot as plt
import pandas as pd
import torch
from torch.utils.data import DataLoader
from math import sqrt
import sys
import xlsxwriter
import numpy as np
import GPUtil as GPU
from threading import Thread
import time
import math
class Monitor(Thread):
   def __init__(self, delay):
      super(Monitor, self).__init__()
      self.stopped = False
      self.delay = delay # Time between calls to GPUtil
```

```
        self.start()

    def run(self):
        while not self.stopped:
            GPU.showUtilization()
            time.sleep(self.delay)

    def stop(self):
        self.stopped = True
class Net(torch.nn.Module):
    """
    PyTorch offers several ways to construct neural networks.
    Here we choose to implement the network as a Module class.
    This gives us full control over the construction and clarifies
        ↪ our intentions.
    """

    def __init__(self, layers):
        """
        Constructor of neural network
        :param layers: list of layer widths. Note that len(layers) =
            ↪ network depth + 1 since we incl. the input layer.
        """
        super().__init__()

        self.device = 'cuda' #if torch.cuda.is_available() else 'cpu'

        assert len(layers) >= 2, "At␣least␣two␣layers␣are␣required␣(
            ↪ incl.␣input␣and␣output␣layer)"
        self.layers = layers

        # Fully connected linear layers
        linear_layers = []

        for i in range(len(self.layers) - 1):
            n_in = self.layers[i]
            n_out = self.layers[i+1]
            layer = torch.nn.Linear(n_in, n_out)

            # Initialize weights and biases
            a = 1 if i == 0 else 2
            layer.weight.data = torch.randn((n_out, n_in)) * sqrt(a /
                ↪ n_in)
            layer.bias.data = torch.zeros(n_out)
```

```
            # Add to list
            linear_layers.append(layer)

        # Modules/layers must be registered to enable saving of model
        self.linear_layers = torch.nn.ModuleList(linear_layers)

        # Non-linearity (e.g. ReLU, ELU, or SELU)
        self.act = torch.nn.ReLU(inplace=False)

    def forward(self, input):
        """
        Forward pass to evaluate network for input values
        :param input: tensor assumed to be of size (batch_size,
           ↪ n_inputs)
        :return: output tensor
        """
        x = input
        for l in self.linear_layers[:-1]:
            x = l(x)
            x = self.act(x)

        output_layer = self.linear_layers[-1]
        return output_layer(x)

    def get_num_parameters(self):
        return sum(p.numel() for p in self.parameters())

    def save(self, path: str):
        """
        Save model state
        :param path: Path to save model state
        :return: None
        """
        torch.save({
            'model_state_dict': self.state_dict(),
        }, path)

    def load(self, path: str):
        """
        Load model state from file
        :param path: Path to saved model state
        :return: None
        """
```

```
      checkpoint = torch.load(path, map_location=torch.device("cuda"
          ↪ if torch.cuda.is_available() else "cpu"))
      self.load_state_dict(checkpoint['model_state_dict'])

def train(
      net: torch.nn.Module,
      train_loader: DataLoader,
      val_loader: DataLoader,
      n_epochs: int,
      lr: float,
      l2_reg: float, MSElist, gpu
) -> torch.nn.Module:
   """
   Train model using mini-batch SGD
   After each epoch, we evaluate the model on validation data
   :param net: initialized neural network
   :param train_loader: DataLoader containing training set
   :param n_epochs: number of epochs to train
   :param lr: learning rate (default: 0.001)
   :param l2_reg: L2 regularization factor (default: 0)
   :return: torch.nn.Module: trained model.
   """

   # Define loss and optimizer
   criterion = torch.nn.MSELoss(reduction='sum')
   optimizer = torch.optim.Adagrad(net.parameters(), lr=lr, lr_decay
       ↪ =0.001*lr*(1/n_epochs))

   # Train Network
   for epoch in range(n_epochs):
      for inputs, labels in train_loader:
         # Zero the parameter gradients (from last iteration)
         optimizer.zero_grad()

         # Forward propagation
         outputs = net(inputs)

         # Compute cost function
         batch_mse = criterion(outputs, labels)

         reg_loss = 0
         for param in net.parameters():
            reg_loss += param.pow(2).sum()
```

```
            cost = batch_mse + l2_reg * reg_loss

            # Backward propagation to compute gradient
            cost.backward()

            # Update parameters using gradient
            optimizer.step()
            #print("GPU RAM Free: {0:.0f}MB | Used: {1:.0f}MB | Util
                ↪ {2:3.0f}% | Total {3:.0f}MB".format(gpu.memoryFree,
                ↪ gpu.memoryUsed, gpu.memoryUtil*100, gpu.memoryTotal)
                ↪ )

        # Evaluate model on validation data
        mse_val = 0
        for inputs, labels in val_loader:
            mse_val += torch.sum(torch.pow(labels - net(inputs), 2)).
                ↪ item()
        mse_val /= len(val_loader.dataset)
        MSElist.append(mse_val)
        #print(f'Epoch: {epoch + 1}: Val MSE: {mse_val}')

    return net

def main():

    old=sys.stdout
    sys.stdout=open('console-log5.txt','w')
    monitor = Monitor(5)
    GPUs = GPU.getGPUs()
    gpu = GPUs[0]
    t0=time.time()


    random_seed =13371337 # This seed is also used in the pandas
        ↪ sample() method below
    torch.manual_seed(random_seed)
    df_unfixed = pd.read_csv(r"C:\Users\simen\OneDrive\Skrivebord\
        ↪ Prosjektoppgave\Test2.csv", index_col=0)
    for col in df_unfixed:
        df_unfixed[col] = pd.to_numeric(df_unfixed[col], errors='
            ↪ coerce')

    df=df_unfixed.interpolate(method='linear',limit_direction='
        ↪ forward')
```

```
df=df.dropna(axis=0)
print("Data␣finished␣interpolating")
print('Sizze␣of␣dataset', df.shape)
# Test set (this is the period for which we must estimate QTOT)
test_set = df.iloc[838860:1048570]

# Make a copy of the dataset and remove the test data
train_val_set = df.copy().drop(test_set.index)

# Sample validation data without replacement (10%)
val_set = train_val_set.sample(frac=0.1, replace=False,
    ↪ random_state=random_seed)

# The remaining data is used for training (90%)
train_set = train_val_set.copy().drop(val_set.index)

# Check that the numbers add up
n_points = len(train_set) + len(val_set) + len(test_set)
print(f'{len(df)}␣=␣{len(train_set)}␣+␣{len(val_set)}␣+␣{len(
    ↪ test_set)}␣=␣{n_points}')
INPUT_COLS = [ 'Africa.571_TT_124', 'Africa.871_CB_TR1_KW','
    ↪ Africa.571_TT_114'] #'Africa.571_TT_114','Africa.871
    ↪ _XI_10151', 'Africa.871_XI_10207','Africa.871_XI_10259','
    ↪ Africa.871_XI_10302','Africa.871_XI_10303','Africa.871
    ↪ _XI_10304','Africa.871_XI_10306','Africa.871_XI_10312','
    ↪ Africa.871_XI_10315','Africa.871_XI_10363','Africa.871
    ↪ _XI_10409']
OUTPUT_COLS = ['Africa.404_XI_11016']

# Get input and output tensors and convert them to torch tensors
x_train = torch.from_numpy(train_set[INPUT_COLS].values).to(torch
    ↪ .float)
y_train = torch.from_numpy(train_set[OUTPUT_COLS].values).to(
    ↪ torch.float)

x_val = torch.from_numpy(val_set[INPUT_COLS].values).to(torch.
    ↪ float)
y_val = torch.from_numpy(val_set[OUTPUT_COLS].values).to(torch.
    ↪ float)

# Create dataset loaders
# Here we specify the batch size and if the data should be
    ↪ shuffled
train_dataset = torch.utils.data.TensorDataset(x_train, y_train)
```

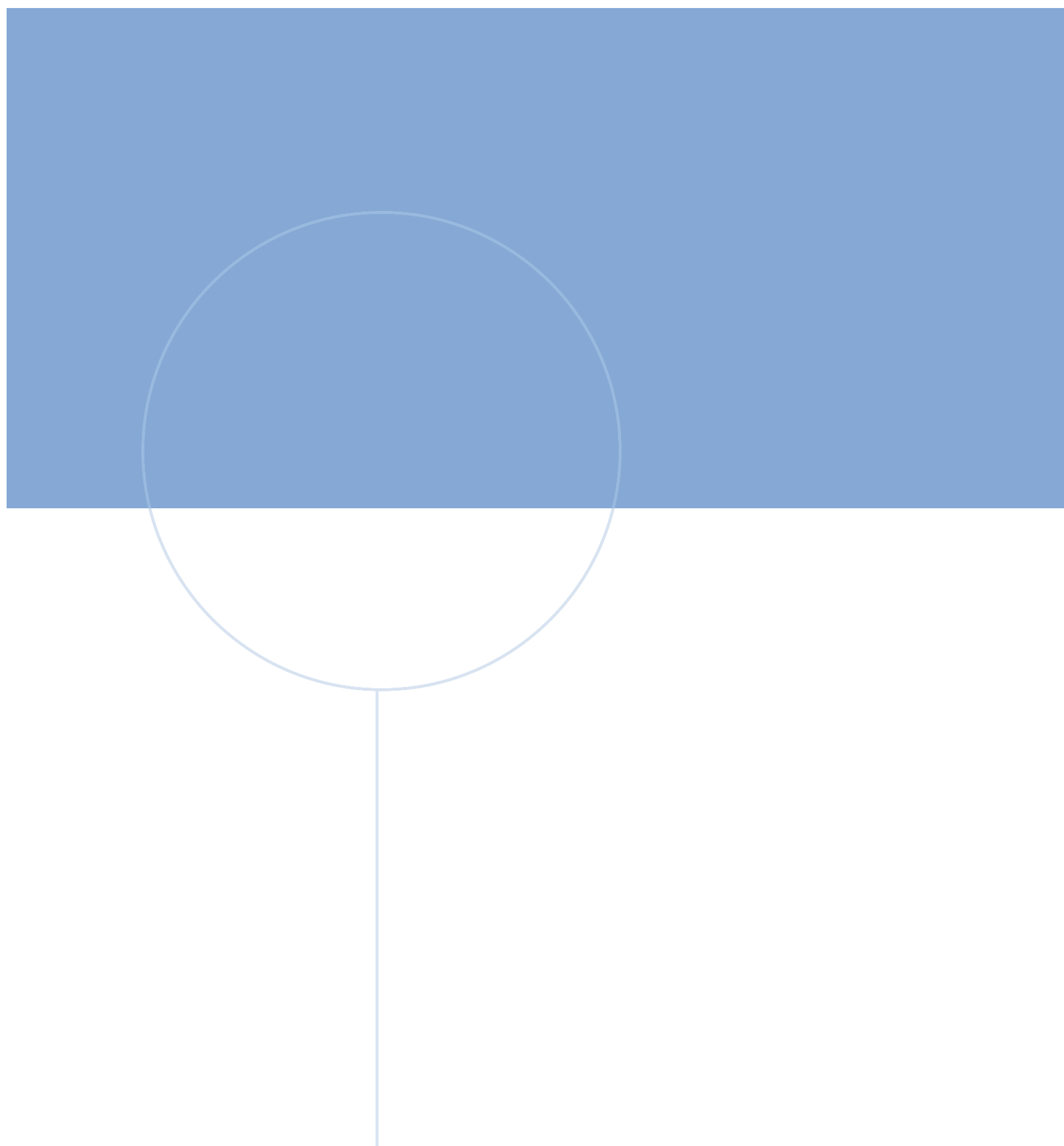
```
train_loader = torch.utils.data.DataLoader(train_dataset,
    ↪ batch_size=131072, shuffle=True)

val_dataset = torch.utils.data.TensorDataset(x_val, y_val)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=
    ↪ len(val_set), shuffle=False)

layers = [len(INPUT_COLS), 500,25, len(OUTPUT_COLS)]
net = Net(layers)

print(f'Layers:␣{layers}')
print(f'Number␣of␣model␣parameters:␣{net.get_num_parameters()}')

n_epochs = 12000
lr = 0.7
l2_reg = 0.015 # 10


MSElist=[]

net = train(net, train_loader, val_loader, n_epochs, lr, l2_reg,
    ↪ MSElist,gpu)

with open('MSEovertime.txt','w') as f:
   sys.stdout=f
   epoke=1
   for element in MSElist:
      print('Epoch:',epoke, '␣␣␣␣␣Val␣MSE:␣' ,element, '\n␣\n')
      epoke+=1


workbook = xlsxwriter.Workbook('MSE.xlsx')
worksheet = workbook.add_worksheet()
row=0
for element in MSElist:
   worksheet.write(row,0,element)
   row+=1
workbook.close()
monitor.stop()
t1=time.time()-t0
sys.stdout=open('timespent.txt','w')
print('Time␣for␣entire␣script␣to␣run:', t1)
sys.stdout=old
# Get input and output as torch tensors
```

```
    x_test = torch.from_numpy(test_set[INPUT_COLS].values).to(torch.
        ↪ float)



    y_test = torch.from_numpy(test_set[OUTPUT_COLS].values).to(torch.
        ↪ float)

    # Make prediction
    pred_test = net(x_test)

    # Compute MSE, MAE and MAPE on test data
    print('Error␣on␣test␣data')

    mse_test = torch.mean(torch.pow(pred_test - y_test, 2))
    print(f'MSE:␣{mse_test.item()}')

    mae_test = torch.mean(torch.abs(pred_test - y_test))
    print(f'MAE:␣{mae_test.item()}')

    mape_test = 100*torch.mean(torch.abs(torch.div(pred_test - y_test
        ↪ , y_test)))
    print(f'MAPE:␣{mape_test.item()}␣%')


def multi_run(mode):
    #constants
    epochs=12000
    learning_rate=0.7
    #rectangular mode
    if (mode==1):
      modus="Rectangular"
      hidden_layers=[]
      max_depth=20
      max_width=100
      depthlist=range(2,max_depth)
      widthlist=np.arange(10,max_width+10,10).tolist()
      for depth in depthlist:
         for width in widthlist:
            hidden_layers=[width]*depth
            result_filename="results_using_"+modus+"
                ↪ neuralnet_with_depth_"+str(depth)+"_and␣width_"+
                ↪ str(width)+".xlsx"
```

```
            console_log_filename="console_log_using_"+modus+"neuralnet_with_depth_"+str(depth)+"_and width_"+str(width)+".txt"
            time_spent_filename="time_spent_using_"+modus+"neuralnet_with_depth_"+str(depth)+"_and width_"+str(width)+".txt"
            main(hidden_layers,epochs,learning_rate,result_filename,console_log_filename,time_spent_filename)


if (mode==2):
    modus="Cone"
    hidden_layers=[]
    max_depth=100
    max_width=500
    depthlist=range(2,max_depth)
    widthlist=np.arange(10,max_width+10,10).tolist()
    ratio_between_width_of_first_and_last_layer=30
    for depth in depthlist:
        for width in widthlist:
            first_layer=width
            layer_width_change=math.floor((first_layer/ratio_between_width_of_first_and_last_layer)/depth)
            hidden_layers=[None]*depth

            for i in range(len(hidden_layers)):
                if (i==0):
                    hidden_layers[i]=first_layer
                else:
                    hidden_layers[i]=hidden_layers[i-1]-layer_width_change
            result_filename="results_using_"+modus+"_neuralnet_with_depth_"+str(depth)+"start_width="+str(hidden_layers[0])+"_last_layer_"+str(hidden_layers[-1])+".xlsx"
            console_log_filename="console_log_using_"+modus+"_neuralnet_with_depth_"+str(depth)+"start_width="+str(hidden_layers[0])+"_last_layer_"+str(hidden_layers[-1])+".txt"
            time_spent_filename="time_spent_using_"+modus+"_neuralnet_with_depth_"+str(depth)+"start_width="+str(hidden_layers[0])+"_last_layer_"+str(hidden_layers[-1])+".txt"
```

```
            main(hidden_layers,epochs,learning_rate,result_filename,
             ↪ console_log_filename,time_spent_filename)


def GrowingApproach():
    output=['Africa.871_XI_10207']
    inputs=[['Africa.601_XI_10114','Africa.601_UA_10176','Africa.601
        ↪ _TI_10199','Africa.601_TI_10200','Africa.601_TI_10179','
        ↪ Africa.601_TI_10179','Africa.601_TI_10178'],['Africa.601
        ↪ _XI_10114','Africa.601_UA_10176','Africa.601_TI_10199','
        ↪ Africa.601_TI_10200','Africa.601_TI_10179','Africa.601
        ↪ _TI_10179','Africa.601_TI_10178','Africa.601_PI_10177','
        ↪ Africa.601_PT_10163'],['Africa.601_XI_10114','Africa.601
        ↪ _UA_10176','Africa.601_TI_10199','Africa.601_TI_10200','
        ↪ Africa.601_TI_10179','Africa.601_TI_10179','Africa.601
        ↪ _TI_10178','Africa.601_PI_10177','Africa.601_PT_10163','
        ↪ Africa.601_PI_10170','601_TI_10172'],['Africa.601_XI_10114
        ↪ ','Africa.601_UA_10176','Africa.601_TI_10199','Africa.601
        ↪ _TI_10200','Africa.601_TI_10179','Africa.601_TI_10179','
        ↪ Africa.601_TI_10178','Africa.601_PI_10177','Africa.601
        ↪ _PT_10163','Africa.601_PI_10170','601_TI_10172','Africa
        ↪ .601_TT_10189','Africa.601_TT_10188','Africa.601_TT_10187'
        ↪ ,'Africa.601_TT_10186','Africa.601_TT_10185','Africa.601
        ↪ _TT_10184','Africa.601_TT_10183','Africa.601_TT_10182','
        ↪ Africa.601_TT_10181']]

    #inputs=[['Africa.601_XI_10114'],['Africa.601_XI_10114','Africa
        ↪ .601_UA_10176','Africa.601_TI_10199','Africa.601_TI_10200
        ↪ '],['Africa.601_XI_10114','Africa.601_UA_10176','Africa
        ↪ .601_TI_10199','Africa.601_TI_10200','Africa.601_TI_10179
        ↪ ','Africa.601_TI_10179','Africa.601_TI_10178'],['Africa
        ↪ .601_XI_10114','Africa.601_UA_10176','Africa.601_TI_10199
        ↪ ','Africa.601_TI_10200','Africa.601_TI_10179','Africa.601
        ↪ _TI_10179','Africa.601_TI_10178','Africa.601_PI_10177','
        ↪ Africa.601_PT_10163'],['Africa.601_XI_10114','Africa.601
        ↪ _UA_10176','Africa.601_TI_10199','Africa.601_TI_10200','
        ↪ Africa.601_TI_10179','Africa.601_TI_10179','Africa.601
        ↪ _TI_10178','Africa.601_PI_10177','Africa.601_PT_10163','
        ↪ Africa.601_PI_10170','601_TI_10172'],['Africa.601_XI_10114
        ↪ ','Africa.601_UA_10176','Africa.601_TI_10199','Africa.601
        ↪ _TI_10200','Africa.601_TI_10179','Africa.601_TI_10179','
        ↪ Africa.601_TI_10178','Africa.601_PI_10177','Africa.601
        ↪ _PT_10163','Africa.601_PI_10170','601_TI_10172','Africa
        ↪ .601_TT_10189','Africa.601_TT_10188','Africa.601_TT_10187
```

```
    ↪ ','Africa.601_TT_10186','Africa.601_TT_10185','Africa.601
    ↪ _TT_10184','Africa.601_TT_10183','Africa.601_TT_10182','
    ↪ Africa.601_TT_10181']]
teller=0
for inputset in inputs:
    # if inputset != inputs[-1]:
    # continue
    epochs=6000
    learning_rate=0.7
    hidden_layers=[500,250,125,75,10]
    result_filename="CONEresults_using_input_set_"+str(teller)+".
        ↪ xlsx"
    console_log_filename="CONEconsole_log_using_input_set"+str(
        ↪ teller)+".txt"
    time_spent_filename="CONEtime_spent_using_input_set"+str(
        ↪ teller)+".txt"
    main(hidden_layers,epochs,learning_rate,result_filename,
        ↪ console_log_filename,time_spent_filename,output,
        ↪ inputset)
    teller+=1
```



NTNU
Kunnskap for en bedre verden