Clemens Martin Müller

Physics informed neural networks in radial load flow calculations

A practical implementation of physics informed neural network based load flow calculations in a small artificial test network and the radial IEEE33- and 69-bus test networks

Master's thesis in Energy and the Environment
Supervisor: Olav Bjarte Fosso
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

NTNU
Norwegian University of Science and Technology

Clemens Martin Müller

Physics informed neural networks in radial load flow calculations

A practical implementation of physics informed neural network based load flow calculations in a small artificial test network and the radial IEEE33- and 69-bus test networks

Master's thesis in Energy and the Environment
Supervisor: Olav Bjarte Fosso
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

NTNU
Norwegian University of Science and Technology

Abstract

This thesis investigates how physics informed neural networks can be applied to solve load flow calculations in radial distribution grids. More specifically, tensorflow is used to establish neural networks in python that are trained to solve load flow calculations. The selected approach is tested on an artificial 4 bus test grid and on the radial IEEE33- and 69-bus systems.

An important motivation for this work are the findings described by recent research within physics informed neural network based power system analysis. Physics informed neural networks are capable of giving more accurate predictions than their vanilla counterparts. Moreover, these models have the potential to be more consistent and less dependent upon large amounts of data.

There exists a multitude of different approaches to solve the load flow problem in different power grids. Some approaches are computationally expensive, but yield exact solutions to the problem, others rely on simplifications and operate much faster but are less accurate. Examples of accurate approaches include the Newton Rhapson load flow, Forward Backward Sweep and the Fast Decoupled Load Flow. A frequently used simplified load flow model is DC power flow.

The neural networks used to solve load flows in this thesis are simple, fully connected, feed-forward networks. This implies that the network takes a number of inputs and processes that information by passing it through three hidden layers before passing it on to the output layer. Results from the small test network were promising, both considering average and worst case performance. However, the results from the medium and large test have shown potential for improvement.

Average errors for the medium test network range between 2.26% and 3.36%, with a less than 0.2% error in all voltage magnitude predictions, and an average error of 4.5% to 6.4% in the voltage angle predictions. In the worst case performances, a 10% threshold was found insufficient to predict all state variables for all medium networks. The large networks had an average error of around 50-70%, however magnitude predictions proved to be accurate.

Two important factors with potentially negative impacts on neural network performance were the significant difference between neural network architecture and power grid topology. Of several possible ways to physics inform neural networks, only one approach has ben evaluated in this work. Advantages and disadvantages of different approaches are discussed.

Further work on this topic should evaluate different ways of physics informing neural networks. Two different approaches include utilizing physics informed architectures forcing neural network data-flow closer to the physical power flow, and separating angle and magnitude predictions into different networks. Furthermore, additional performance improvements can be likely be imposed by improving data processing, and by using collocation point training to pre-train neural network weights.

Sammendrag

Denne avhandlingen undersøker hvordan fysikkinformerte nevrale nettverk kan brukes til å løse lastflytberegninger i radielle distribusjonsnett. Mer spesifikt brukes tensorflow til å etablere nevrale nettverk i Python, som trenes opp til å løse lastflytberegninger. Den valgte tilnærmingen er testet på et fiktivt testnett med 4 busser, og på de radielle IEEE33- og 69-buss-systemene.

En viktig motivasjon for dette arbeidet er funnene som er beskrevet i nyere forskning innen fysikkinformert nevralnettbasert kraftsystemanalyse. Fysikkinformerte nevrale nettverk er i stand til å gi mer nøyaktige prediksjoner, mer konsistente resultater og er midre avhengige av store datamengder.

Det finnes en rekke ulike tilnærminger til å løse lastflytproblemet i ulike kraftnett. Noen tilnærminger er beregningskrevende, men gir eksakte løsninger på problemet. Andre tilnærminger baserer seg på forenklinger og er raskere, men mindre nøyaktige. Eksempler på nøyaktige tilnærminger er Newton Rhapson-lastflyt, Forward Backward Sweep og Fast Decoupled Load Flow. En ofte brukt forenklet lastflytmodell er DC power flow.

De nevrale nettverkene som brukes til å løse lastflyt i denne avhandlingen, er enkle, fullt sammenkoblede feed-forward-nettverk. Dette innebærer at nettverket tar imot en rekke inndata og behandler denne informasjonen ved å sende den gjennom tre skjulte lag før den sendes videre til utgangslaget. Resultatene fra det lille, fiktive, testnettverket var lovende, både med tanke på gjennomsnittlig og dårligst mulig ytelse. De mellomstore og store modellene må forbedres før de kan anvendes i praksis.

Den gjennomsnittlige feilen for det mellomstore testnettverket ligger mellom 2.26% og 3.36%, med en feil på mindre enn 0.2% i alle prediksjoner av spenningsmagnitude og en gjennomsnittlig feil på 4.5% til 6.4% i prediksjonene av spenningsvinkel. I de dårligste tilfellene viste det seg at en terskelverdi på 10% ikke var tilstrekkelig til å forutsi alle tilstandsvariabler. De store nettverkene hadde en gjennomsnittlig feil på rundt 50-70%, men prediksjonene av spenningsmagnituder viste seg å være nøyaktige.

To viktige faktorer som potensielt kan ha negativ innvirkning på ytelsen til det nevrale nettverket, var den betydelige forskjellen mellom det nevrale nettverkets arkitektur og kraftnettets topologi, og tilnærmingen til behandling av treningsdata som ble brukt i denne oppgaven. Bare én måte å implementere et fysikalsk informert nevralt nettverk på ble testet i dette arbeidet, selv om det finnes flere andre mulige tilnærminger som kan forbedre ytelsen betydelig. Fordeler og ulemper ved de ulike tilnærmingene diskuteres.

Videre arbeid på dette området bør evaluere ulike måter å fysikkinformere nevrale nettverk på. To ulike tilnærminger omfatter bruk av fysikkinformerte arkitekturer som tvinger dataflyten i nevrale nettverk nærmere den fysiske kraftflyten, og separering av vinkel- og magnitudestprediksjoner i ulike nettverk. I tillegg er det sannsynlig at ytelsen kan forbedres ytterligere ved å forbedre databehandlingen og ved å bruke kollokasjonspunkttrening til å forhåndstrene vektene i nevrale nettverk.

Preface

This thesis is written as a conclusion to my two year masters programme in Energy and the Environment at the Norwegian University of Science and Technology (NTNU). The project is supervised by Olav Bjarte Fosso, and is equivalent to a semesters worth of work (30 ECTS credits).

In contrast to building the thesis upon the previous semesters project work, i choose to explore neural network applications within power systems. As i started consulting the WWW about machine learning and neural networks, i was quickly trapped within this interesting rabbit-hole. Exploring and explaining neural network performance have been motivating and giving throughout this work.

The thesis is written to anyone who is interested in the intersection between the world of neural networks and power system analysis. Moreover, it is written on a level and with a language that should be readily understood by my fellow students.

I would like to thank my supervisor for introducing me to neural networks, and for his continuous support during the project. I could not have finished without my friends who joined med climbing, cycling and skiing. And Oda, this last year would have been so much more demanding without your love and support, thank you!

Clemens Martin Müller

Clemens Martin Müller
Trondheim, June 2023

Contents

Terms and acronyms

Batch size	Amount of test samples predicted before calculation of the loss function.
CustomLoss	loss function developed for use in this thesis.
Epoch	Amount of iterations through the entire training dataset.
FBS	Forward backward sweep, load-flow calculation approach for radial grids.
HPC	High performance computing
Hyperparameters	Parameters defining the structure and method of a neural network.
Karush-Kuhn-Tucker conditions	Optimality conditions in optimization problems involving several variables.
LineFlowLossForAngle	Loss function developed for use in this thesis.
Machine learning	Methods that use data to improve performance on a given set of tasks
Neural networks	Artificial neural networks are computing systems whose behavior is inspired by biological neural networks.
Normalization	The process of scaling input values to NNs to values within [0,1]
Parameters	Variables that can be chosen freely. Depend on bus-type and state variables.
PDE	Partial Differential Equation
Physics informed neural networks	Neural network utilizing physical relationships to improve the networks performance
PRT	Physical regularization term
SMIB	Single Machine Infinite Bus System
SquaredLineFlowLoss	Loss function developed for use in this thesis.
Topology	[Merriam-Webster] *Configuration*, Power grid: Order and arrangement of buses.

List of Figures

List of Tables

1 Introduction

Power flow studies are conducted to analyze the power system steady state [1, p.1]. Using traditional approaches, this problem is solved by using iterative methods such as the *Newton-Rhapson* (NR) method. While these methods provide solutions accurate to a given threshold, they may be impractical for larger systems due to their speed.

Transitioning to a greater share of renewable energy sources depends on the introduction of more volatile energy sources in the power grid [1, p.1]. Distributing this energy in the aging power system is a challenging task with regards to economic feasibility, system reliability and security.

Machine learning techniques can be used to tackle these challenges and promote integration of renewable energy [1, p.1]. This is an area that has been thoroughly researched during the last years. Examples include using machine learning to forecast energy prices and renewable energy production. An important feature of machine learning is its ability to generalize and scale while improving computational efficiency.

However, neural network performance is affected by a very wide range of parameters [2, p.1]. These parameters can be tuned and adjusted until a trained neural network shows its best possible performance. This is common among practitioners, and can often be both time and resource consuming.

Widespread application is held back by its dependency on large datasets and lack of theoretical analysis of lower bound performance [3, p.1]. A way to remedy these issues is to link underlying physical concepts to the design of neural networks. This concept is commonly referred to as physics informed neural networks.

Recent research describes a multitude of different physics informed neural network implementations within the area of power system analysis [3, p.1]. The growing amount of research has lead to a consensus that physics informed neural networks can have practical applications within power systems.

1.1 Objective and problem statement

There remain many unexplored aspects related to neural network based power system analysis. This thesis is dedicated to exploring neural network applications within the domain of load flow calculations in radial distribution grids. It will investigate how neural networks can be applied to solve a regular load flow in three differently sized distribution grids. Simultaneously, different hyperparameters and loss functions will be changed to study their impacts on neural network performance.

The thesis is supervised by Olav Bjarte Fosso, Professor at the Department of Electric Energy. During the work on this thesis, a descriptive problem statement was developed together with a condensed set of contributions found in this thesis.

"*Describe how physics informed loss function based neural networks can be used to calculate load flow solutions for radial distribution grids.*"

Important contributions of this thesis include:

- A practical python implementation of different neural network based load flows.
- A practical implementation of a physics informed loss function in tensorflow.
- A brief literature review of research related to neural network based load flow.

Thesis structure

This thesis is structured as follows. Section 2 *Background* introduces important theoretical concepts relevant for this thesis. Section 3 *Literature review* gives a brief introduction to academic work related to Neural Networks and load flow calculations. Section 4 *Approach* describes in detail how the work on this thesis was conducted. Section 5 *Results* describes the convergence performance, average and worst case accuracies of a selection of trained neural networks. Finally, Sections 6 *Discussion* and 7 *Conclusions and further work* discuss the results and provide a conclusion to this work.

1.2 Some formalities

Note on the referencing in this thesis
References in this thesis are given using the IEEE referencing style. In sections 2 Background and section 3 Literature, contents from different references are summarized and adapted to this work. Here, though references are mostly provided in the first sentence of a paragraph, they often also refer to the remaining parts of the paragraph.

Inclusions of parts from the project work
This thesis includes some parts of the authors project work, completed prior to work on this thesis. Where relevant sections from that work are re-used in this thesis, the paragraphs include a reference to [4] at its end. In cases where entire subsections are re-used, the heading includes a reference to [4]. To make used references accessible, the citation style has been adapted to match the thesis, and references from the project work are included in the references for this project.

2 Theoretical background

This section gives an introduction to essential topics in this thesis. The first subsections introduce the concepts of neural network based machine learning and its derivative the physics informed neural network. Later, the last sections introduce power systems and how power system states are calculated using a variety of different approaches.

2.1 Machine learning

All paragraphs without references in section 2.1 as well as sections 2.1.1 and 2.1.2 are all based on [5]. Approximate page numbers are given at the end of each paragraph.

Machine learning aims to adapt computers actions to become more accurate. The topic combines ideas from several academic areas such as neuroscience, biology, statistics, mathematics and physics. *Neural Networks* (NNs) have been developed inspired by the working principles of the human brain. [p.5]

A NNs complexity is usually described in two different types: training complexity and the complexity of application. Since neural network training is not conducted often, more complex and time consuming processes are acceptable. However, it is in many cases a goal to generate fast algorithm with a low computational cost capable of generating accurate solutions. [p.5]

There are different ways to facilitate machine learning, in other words making the machine better at the task it is performing by practicing. A commonly used method is *supervised learning*, utilizing a training set of data together with correct responses. Other learning methods are *unsupervised learning*, *reinforcement learning* and *evolutionary learning*. [p.6]

2.1.1 Neurons

The human brain's nerve cells *neurons* are connected by synapses who transmit electrical signals as the neurons fire. A single neuron can be depicted as a processor conducting a simple task, deciding whether or not to fire. Synapses in the brain that connect neurons firing simultaneously often are strengthened. This concept can be replicated in a computer by modifying the weight of the different neuron connections. [p.11-12]

A simple model of a neuron, developed by McCulloch and Pitts is given in figure 2.1. Input signals in the figure are X_1 trough X_n, and are in the real brain signals coming from other neurons. The different signals all have a strength, modeled by the weights W_1 trough W_n. Once the signals arrive at the neuron, the weighted sum given in equation 2.1 of all the signals is calculated. [p.13-14]

$$h = \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

As the signal strength, h is passed on to the block, f represents the *activation function* (see section 2.1.3). The McCulloch and Pitts neuron utilizes a binary step function in this place. Consequently, their neuron either fires or does not fire, depending on the threshold value.

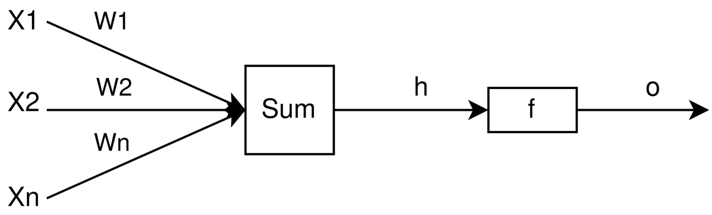

Figure 2.1: *Simple illustration of a neuron, redrawn in draw.io from [5, p.14]*

It is possible to use slightly different activation functions, thereby making the machine more complex. [p.14]

2.1.2 Perceptrons and loss functions

Perceptrons are simply a collection of inputs connected to neurons through a weighted connection, as illustrated by figure 2.2. Inputs to the system are blue, and the outputs are green. The arrows depict the weighted connections. A perceptrons main task is to reproduce a specific target for a given input. To do this, the weights are adjusted in such a way that the perceptron reproduces the correct output pattern. In a perceptron network, all inputs are connected to all neurons giving an output. In other words, the output from one source is distributed to all neurons in the network through paths of different weights. Perceptron and later also NN weights are conveniently stored as a two dimensional matrix. [p.19-20]

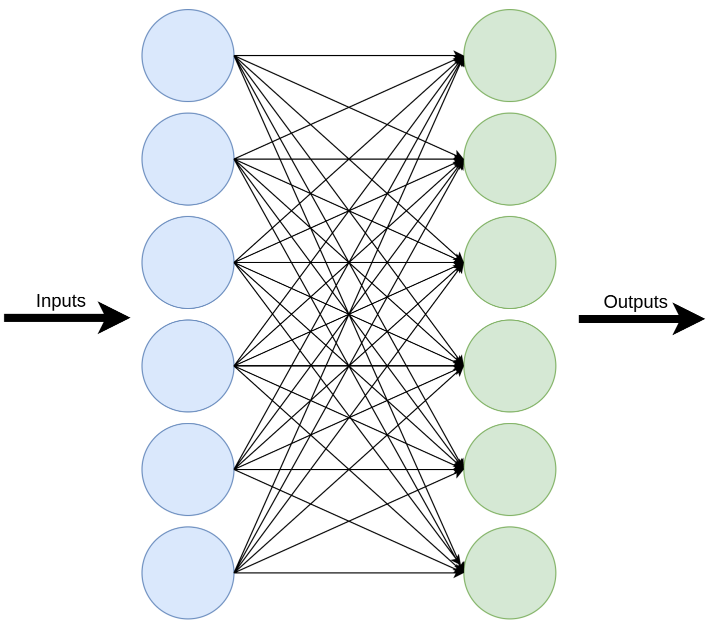

Figure 2.2: *Simple illustration of a perceptron, redrawn and adapted in draw.io from [5, p.20]*

Let $w_{ik} : i \in [1, m]$, be the weights of all input signals to a neuron that triggered incorrectly. These are the weights that should be changed to correct the neuron triggering. The desired direction of change is determined by assigning a loss value to the output. A simple way to do this is to calculate the difference between the target value and actual output value:

$t_k - y_k$. To account for possible negative output values, a positive or negative integer should also be multiplied in to this expression, so that the final $\Delta w_{ik} = \pm(t_k - y_k)$. Note that this method is rarely used in practice, as this simple loss function is outperformed by more complex variants. [p.21]

Reference [6, p.194] explains that different error functions should be considered for different applications. In general, regression problems motivate the use of a sum of squares error function, whereas classification problems can benefit from the use of other error functions. Further, reference [6, p.140] explains that the derivatives of the loss function are used to optimize weights. They are provided to an optimizer, which minimizes the loss value by adjusting network weights. An example of a simple optimizer is the gradient descent method.

To adjust the perceptrons sensitivity to faulty outputs, a learning rate μ is introduced. This parameter affects how fast the network is learning, and may thus also impact the stability. If the learning rate is set too high, the perceptron may become unstable. If it is set too low the system will become a slow learner. Finally, it is also useful to provide each neuron a firing threshold (bias) to be able to adjust a single neurons response independently of all other neurons in the network. [p.21]

Since learning in NNs happens by tuning the networks weights, a way to conduct more computations is to add more layers to the perceptron, as illustrated in figure 2.3. The figure shows an input layer, the usual output layer, and a middle layer which is often referred to as a hidden layer. [p.47]

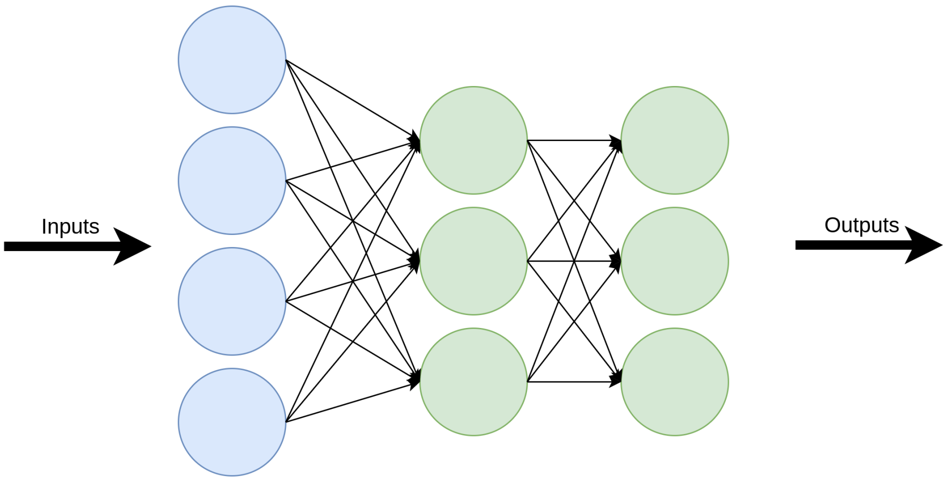

Figure 2.3: *Simple illustration of a multi-layer perceptron, redrawn and adapted in draw.io from [5, p.48]*

2.1.3 Activation functions

Reference [5, p.50-52] explains that *Multi-layer perceptrons* (MLPs) contain several sets of weights since they have multiple layers, all of which have to be adjusted in the learning process to obtain adequate results. To do this in an efficient manner, a form of the gradient descent method is applied. An error function (sometimes: cost function) describing the difference between the actual and desired output of a given neuron is defined, and differentiated with respect to the weights. This requires a differentiable activation function, a sigmoid function given in equation 2.2 is frequently used to this purpose. β represents some positive parameter.

$$g(h) = \frac{1}{1 + exp(-\beta h)} \tag{2.2}$$

The author of [5, p.50-52] further notes that when the direction of weight change is given by the sigmoid function, a separate error function is required to determine the size of the step. This step size is often determined using the square loss function given in equation 2.3.

$$E = \frac{1}{2}\sum_{k=1}^{n}(t_k - y_k)^2 \tag{2.3}$$

According to [7, p.1], another frequently used activation function is the *Rectified Linear Unit* (ReLU) activation function, as given in equation 2.4. This particular loss function enabled faster training for deeper neural networks. It is a robust activation function because it has constant gradients and does not saturate.

$$f(y) = \begin{cases} y : y > 0 \\ 0 : y \leq 0 \end{cases} \tag{2.4}$$

2.1.4 Neural network initialization

Neural network initialization can directly affect training efficiency and convergence performance [8, p.5]. Recent research on this topic has focused on two ways to initialize neural networks: (1) using a pre-training stage and (2) initializing network weights randomly. Pre-training a network requires more training time, and may also lead the network to a sub-optimal local minimum.

Reference [8, p.5] gives a sensible random initialization strategy for forward propagating networks utilizing ReLU or linear activation functions. This strategy is given by equation 2.5, where w_i represents the input weights into layer i and n represents the number of neurons in layer i.

$$Var[w_i] = \begin{cases} \frac{2}{n_i} & i \neq 1 \\ \frac{1}{n_i} & i = 1 \end{cases} \quad i = 1, ..., L-1 \tag{2.5}$$

The Glorot initializer

Glorot and Bengio [9, p.1, 5] state that deeper NNs designed before 2006 were often not trained successfully. Common for more recent successfully trained deeper NNs was that they were obtained using either new training mechanisms or initialization strategies. The logistic sigmoid function is not suited for use in deep neural networks because of its mean value which in certain cases drives NN layers to saturation.

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \quad , \quad \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \tag{2.6}$$

In their paper, Glorot and Bengio propose initializing NN layer biases as 0, and layer weights W as given in equation 2.6. U represents a uniform distribution, and n the number of neurons in a layer. The subscripts j and $j+1$ are used to indicate the previous and next layer in the NN.

2.1.5 Scaling inputs and outputs

In reference [6, p.296, 298] the authors present pre-processing of data as one of the most important stages in NN development. It is often conveniently carried out on the entire dataset simultaneously. The process is almost always advantageous and can have a significant impact on neural network performance. For regression problems it can be appropriate to apply a similar linear rescaling to the target values. A flowchart illustrating the data-flow for a NN using both pre- and post-processing of data is given in figure 2.4.

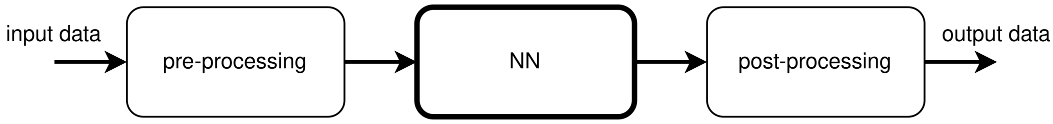

Figure 2.4: *Data flow in neural networks applying both pre- and post-processing of data. Redrawn and adapted in draw.io from [6, p.296]*

In reference [6, p.298] re-scaling of input variables is presented as a common form of pre-processing. This form of pre-processing is particularly important in applications where different inputs have significantly different input magnitudes. This is important because the magnitude of the input does not necessarily reflect the relative importance of the output. Further, reference [6, p.299] comments that input normalization in practice ensures that all input values are close to the order of unity. In such cases it is to be expected that network weights are of similar magnitude. Without re-scaling, certain weights related to significantly different values would have to be adapted.

2.1.6 Machine learning in power system analysis

Following the contents of [10, p.1], it is clear that artificial NNs are capable of learning complex non-linear functions and that they have proven themselves useful as data-based solvers. It is also possible to generate solutions to problems without an algorithmic solution, provided that large amounts of data are available.

Moreover, reference [10, p.1] explains that Power system problems can take many different forms due to its many properties. They can be nonlinear, large-scale, dynamic, discrete etc. Combining these factors make typical power system problems challenging to solve. Due to this complexity, there exist problems within power system problems that cannot be or are very impractical to solve with analytical methods. Provided the availability of large amounts of data, NNs provide an attractive tool to solve such problems.

The authors of [3] emphasize that though NNs demonstrate significant advantages for applications within power system analysis, they cannot remedy all real world problems. An important reason for this is that NN training requires a substantial amount of quality training data, which is often not readily available. Furthermore, they are often considered black box approaches lacking theoretical analysis of its lower bound performance. Moreover, NNs may yield infeasible results if the network is not designed with particular attention to detail. A final reason limiting widespread application of NNs is that they usually are very case specific, thus limiting their ability to produce adequate results on other samples.

2.1.7 Tensorflow

In a book [11, p.8-9] introducing predictive analysis using *Tensorflow*, Karim introduces *Tensorflow* as a framework for machine learning developed by Google. It was made open source in 2015, and has since then been adopted by both industry and academia. The framework carries out calculations using neural networks on multidimensional arrays. Tensorflow utilizes graphs to carry out its computations in such a way that nodes are mathematical operations and edges ensure communication between edges and nodes.

Furthermore, reference [11, p.19-20] mentions that Tensorflow internally represents calculations as a graph. As a consequence, though Tensorflow is used in python, these computations are carried out in a C++ engine, which improves calculation speed significantly. According to Tensorflow documentation [12], later Tensorflow models also support eager execution, where everything is carried out directly in python. Doing this significantly decreases the efficiency of NN training.

2.1.8 A note on neural network topologies

According to reference [6, p.120] a networks diagram determines its corresponding mathematical functions. In simple *feed-forward* networks, all information is processed in one direction through the network, and there exist no feedback loops. This particular way of processing data is commonly referred to as *forward propagation*.

Furthermore, reference [6, p.121] notes that any multi layer network utilizing linear activation functions, there will always exist a corresponding network without any hidden units. Finally, the same reference states that integrating a lot of structure into a network can yield significant advantages.

2.2 Physics informed neural networks

A way to improve NN performance is to include prior knowledge in the network structure or into the pre-processing of data [6, p.295]. Here, prior knowledge refers to additional relevant information that is not provided by the training data.

Recent advances in machine learning techniques and data availability have proven to be valuable developments to several research areas [13, p.1-2]. In many cases, available data is very limited, forcing training of machine learning algorithms on limited datasets often with poor results. These results are based solely on the pattern recognition through traditional machine learning. In many cases there exist vast amounts of knowledge about the topic. Informing the machine learning algorithm of the overlying physical concepts has proven to make algorithms capable of making good generalizations with limited amounts of data.

Using *Physics informed Neural Networks* (PINNs) to model power systems introduces several advantages over traditional NNs [14, p.1]. It requires less training data, may yield smaller NNs, all while demonstrating a high performance. The performance improvement originates from the fact that physics informed NNs directly calculates, without integration, state variables in power systems.

According to Huang et al. [3, p.11] PINNs used as an alternative to traditional power flow solutions face a trade-off between accuracy and efficiency. While system users demand theoretically solid and accurate solutions, there exist use cases that demand real time training. In such applications, a PINN-based power flow solution has no significant advantage over existing linear power flow approximations.

Today there exist four main approaches to PINN design [3, p.1]. (1) Modifying the loss function based on the bounding physical model. (2) Using knowledge about the physical model to initialize NN weights efficiently. (3) Designing NN architecture inspired by the physical system. (4) Using hybrid physics and deep learning models. Advantages over traditional NNs include improved model performance and improved interpretability of the final model, more consistent results, improved generalization capabilities, reduced search space for NN weights thus improving training efficiency and convergence as well as reduced reliance on large dataset.

2.2.1 Using a physics informed loss function

To implement a physics informed loss function, the output variables need to be assigned a physical meaning and adapt the governing equation using output variables to the loss function [3, p.2]. Consequently the general loss function of a PINN could be formulated as given in equation 2.7.

$$L_{PINN} = L(\hat{\vec{y}}, \vec{y}) + \lambda R(\vec{W}, \vec{b}) + \gamma R_{phy}(\vec{X}, \hat{y}) \tag{2.7}$$

In equation 2.7, L represents the regular loss function, R the parametric regularization term applied to the weight and bias matrices, and R_{phy} the *physical regularization term* (PRT), according to the physical relationships. λ and γ are used as regularization hyperparameters. It is the PRT that distinguishes PINNs from its vanilla counterparts [3, p.2]. The following section gives an illustrative example of an implementation of a physics informed loss function for the single machine infinite bus system (SMIB).

Physics informed loss function for the SMIB system

In the SMIB system, the swing equation describing the rotor angles movement as a function of time: $\delta(t)$. This relation is given as the partial differential equation (PDE) in equation 2.8 [3, p.2-3]. ξ represents the inertia constant, κ the damping coefficient, B susceptance of the grid connection while V_g,V_e and P are generator voltage, grid voltage and mechanical power respectively.

$$\xi \frac{\partial^2 \delta}{\partial t^2} + \kappa\omega + BV_gV_e sin(\delta) - P = 0 \tag{2.8}$$

Physics informed networks can approximate equations such as the one given in equation 2.9 [14, p.2]. $u(t, x)$ represents the solution of the equation, $N[u; \lambda]$ represents a nonlinear operator relating the state variables u to system parameters λ. Further, t represents the time, x the system input and $[0, T]$ the time interval of simulation. Note that the domain of Ω can be bounded by prior knowledge.

$$\frac{\partial u}{\partial t} = -N[u, \lambda], x \in \Omega, t \in [0, T] \tag{2.9}$$

Enforcing the physical law governing the dynamic system in the NN is done by defining the PINN as in equation 2.10 [14, p.2]. λ may be both constant or unknown, if they are unknown the goal is to identify the system parameters that satisfy 2.9. Otherwise, if λ is known, the nonlinear operator $N[u, \lambda]$ is simplified to $N[u]$.

$$f(x, t) = \frac{\partial u}{\partial t} + N[u, \lambda] \tag{2.10}$$

Figure 2.5 gives a flowchart of how a PINN processes information. The inputs to the network are given to the left, x and t, which are processed by the network, yielding $u(x, t)$. The flowchart illustrates that the NN and the PINN share the same parameters with different activation functions [14, p.2]. To determine the PINN output of the network, $f(x, t)$ is calculated using the outputs of the neural network as given by equation 2.10. The differentiation required to obtain the PINN output is carried out by means of algorithmic differentiation.

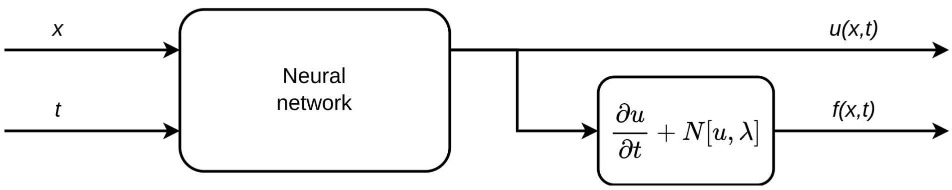

Figure 2.5: *Flowchart of a physics informed neural network. Redrawn and adapted in draw.io from [14, p.2]*

Combining equation 2.8 and 2.10 yields equation 2.11, which is the PINN formulation of the SMIB system [3, p.3].

$$f(t, P) := \xi \frac{\partial^2 \delta}{\partial t^2} + \kappa\omega + BV_g V_e sin(\delta) - P \tag{2.11}$$

The parameters of shared NN are adjusted to optimal values by minimizing the loss function MSE as given in equation 2.14 [14, p.2-3]. Its two constituents MSE_u and MSE_f denote the mean squared error referred to N_u training data and N_f collocation points. In utilizing this implementation, MSE_u ensures that the boundary conditions of the independent input variables x are within their boundaries. MSE_f penalizes outputs incompliant with the physical model.

$$MSE_u = \frac{1}{N_u} \sum_{i}^{N_u} |u(t_u^i, x_u^i) - u^i|^2 \tag{2.12}$$

$$MSE_f = \frac{1}{N_f} \sum_{i}^{N_f} |f(t_f^i, x_f^i)|^2 \tag{2.13}$$

$$MSE = MSE_u + MSE_f \tag{2.14}$$

2.2.2 Using physics informed initialization

As the learning procedure of NNs is a highly nonlinear process, it is sensitive to the selection of its initial solution [3, p.3]. Proper selection of initial weights can improve convergence speed and help prevent the model from converging in a local minima. Initializing the system from a physics informed basis can yield similar results to transfer learning by improving training efficiency. This approach is particularly useful where the cost of data acquisition is high. A possible way to implement physics informed learning is to pre-train the NN using synthetic data before fine tuning the model on real world data.

2.2.3 Using physics informed design of architecture

Physics informed design of NN architecture strives to replicate the data-flow in the physical problem solution [3, p.3-4]. The aim of encouraging and discouraging certain connections is to improve NN performance on a wider range of samples. A way to do this is to impose physical meaning to the outputs of selected hidden layers, before applying specific connection layers to encode certain physical relations.

Other methods include utilizing some sort of physical meaning to given system weights while keeping them fixed during the entire training procedure [3, p.3-4]. This approach allows a more robust training procedure and improved interpretability. Further approaches include the concept of *Graph Neural Networks* (GNNs). These network types can be especially beneficial since power grids can be represented as graphs. For this reason GNNs may be considered a kind of topology aware neural networks.

2.2.4 Using hybrid physics-neural-network models

A final way to utilize NNs for problem is to use NNs to solve parts of the problem, while the remaining parts are solved using the underlying physical principles [3, p.4]. This approach can be particularly useful in cases where parts of the physical solution are particularly cumbersome to derive. In such cases, NNs can be used to bridge the gap making the most of concrete problem background and the function approximation ability of a NN.

2.2.5 Collocation points

Collocation points are different from regular training points as they do not posses a pre-calculated solution [15, p.3]. Instead of basing the cost function on the resulting prediction error, collocation errors are only based on the underlying physical principles. In a DC-OPF case, collocation errors are calculated as discrepancies within the *Karush-Kuhn-Tucker* (KKT) conditions.

2.2.6 Extreme learning machines

The learning speed of feedforward NNs is a bottleneck to their performance [16, p.1-2]. Earlier research has shown that a single layer feed forward network with N hidden neurons and randomly chosen input weights and layer biases can exactly learn N observations. A single layer NN contains two sets of weights. When the first is initialized randomly this leaves only a single set of weights to adjust. This concept is commonly referred to as an *Extreme Learning Machine* (ELM).

An *Extreme Learning Machine* (ELM) is trained using the minimum norm least squares solution $\hat{\beta}$ of the equation system given in equation 2.15 [16, p.4]. $\mathbf{H}$ represents the hidden layer output matrix, ie. the outputs provided by the hidden neurons transformed by random weights. $\mathbf{T}$ represents the vector containing the desired solutions. The minimum solution is given by equation 2.16, where $\mathbf{H}^{\dagger}$ is the Moore-Penrose generalized inverse of $\mathbf{H}$.

$$\mathbf{H}\beta = \mathbf{T} \tag{2.15}$$

$$\hat{\beta} = \mathbf{H}^{\dagger}\mathbf{T} \tag{2.16}$$

Important features related to the ELM include its extreme training speed achieved by matrix calculations of output weights [16, p.11]. It has improved generalization capability compared to gradient descent learning schemes. The ELM approach is less prone to converging within local minima. Finally, an ELM can be used with non-differentiable activation functions.

A significant limitation to the ELM is that it cannot utilize more than one hidden layer[16, p.11]. Despite this, its ability to approximate any continuous function makes it applicable to a wide range of applications.

By splitting the master problem into smaller subproblems, it is possible to use several connected ELMs to solve more complex problems [17, p.1]. Compared to a regular NN, this uses less memory.

2.3 The Idun cluster

Idun is a *High Performance Computing* (HPC) cluster, whose intention is to serve as a platform for rapid testing and prototyping of HPC software and for research within energy efficient computing [18]. It consist of 73 nodes and has access to two storage arrays. Students and employees associated to NTNU may apply for access to Idun's resources for their work.

2.4 Power systems

Sections 2.4 and 2.5 briefly describe how the power system is evolving, typical characteristics of distribution and transmission grids as well as how the power system state is calculated using different load flow approaches.

2.4.1 On the energy transition

An environmental friendly electrical power supply is an important aspect of sustainable development [19, p.14-15]. Currently, the power sector is responsible for a major fraction of global CO_2 emissions. Studying charts describing the energy sources in today's energy mix, it is evident that the world relies heavily on fossil resources for electricity production. When considering global total energy use, the fossil share is even greater. [4]

Transitioning the power system away from fossil energy sources will demand more electrical energy [19, p.18]. Simultaneously, human population and living standards are increasing which demands even more energy. Thus the demand for electrical energy is likely to increase. As a consequence, the stress on the current power grid will increase. [4]

The future importance of renewable energy resources will require a significant reconfiguration of the current power system [20, p.1-2]. It is expected that the power grid will become smarter by containing flexible consumption and intelligent power system control systems. Furthermore, it is expected that Distributed Energy Resources (DERs) will become an important part of the future, greener, energy systems. [4]

Reference [20, p.2] also notes that DERs often are located closer to the load. In practice this means that more power generation will be established in the distribution grid, on the customer side of the power meter.

Finally, the authors of reference [20, p.2] comment that distribution systems are sensitive to DER generation capacities. It is the distribution system capacity which determines what generation capacities are suitable for specific grids. This is because DERs can affect distribution system voltages.

2.4.2 Transmission and distribution grid characteristics [4]

Due to significant differences between the transmission and the distribution system, transmission system load flow algorithms cannot be directly applied to the distribution system [21, p.1]. Important differences between the systems is the network topology and the R/X ratio in the system. A full list of significant differences is given in table 2.1.

Table 2.1: *Comparison of transmission and distribution system characteristics [21, p.1]*

Description	Transmission systems	Distribution systems
Network structure	Meshed	Radial
R/X ratio	near zero	high
Symmetricity	balanced	Unbalanced or balanced
Load buses	mostly PQ	both PQ and composite
Generator buses	Mostly PV	PV, PQ and composite
Voltage angle	Not negligible	negligible

2.4.3 Parameters and state variables

Each power system bus i is associated with four variables, the voltage magnitude and angle $v_i\angle\delta_i$, the net real power injection P_i and the net reactive power injection Q_i [22, p.345]. For each bus, two of these variables are given as inputs while the other two are calculated using a power flow algorithm. Which variables are dependent on the inputs is specified by the bus type as shown in table 2.2.

In addition to load data, information about transmission lines is also required to conduct load flow calculations [22, p.346]. This is commonly done by constructing the $\mathbf{Y_{bus}}$ matrix

Table 2.2: *Overview of bus types in load flow calculations [22, p.345-346]*

	Slack bus	**Generator bus**	**Load bus**
Common reference	Swing bus	PQ bus	PV bus
Input variables/Parameters	$v_i \angle \delta_i$	P_i, v_i	P_i, Q_i
Dependent variables/state variables	P_i, Q_i	Q_i, δ_i	$V_i \angle \delta_i$

from the line and transformer data of the system in question. In the $\mathbf{Y_{bus}}$ matrix, all diagonal elements Y_{ii} are the sum of admittances connected to bus i, whereas the off diagonal elements Y_{ij} are the negative sum of all admittances connecting buses i and j.

2.4.4 Grid topologies [4]

The power grid topology describes the order of connection of different nodes in the power grid [23] [24]. It is possible to store this data in different ways, for example using a branch bus incidence matrix or as a list of buses that are connected.

Distribution system operators frequently change the states of network switches to optimize voltage profiles, to alleviate faults or to minimize system losses [24]. Consequently utilities are not always aware of the exact network configuration at all times. When the network topology is not known, it can be estimated using topology detection or topology identification. When the line infrastructure, impedances and switch placements are known, identifying the exact system topology is simpler, and topology detection is used. When the system infrastructure is completely unknown, topology identification technique can be applied to estimate the system topology.

2.5 Load flow calculations in power systems

Power flow equations are the basic equations describing the relationship between active P and reactive Q power injections and the complex voltages ($v_i \angle \delta_i$) in the power flow model at steady state balanced three phase conditions [22, p.345] [25, p.1]. Since this model is nonlinear, power system analysis is often associated with a significant computational load.

This computational burden can be reduced by applying linear power flow models [25, p.1]. Such models are frequently used in the solution of optimal power flow problems as it allows an efficient solution with guaranteed convergence. A frequently used linear power flow model is the DC power flow model which focuses on P and δ while neglecting Q and v.

The non-linear power flow equations may be formulated as given in equation 2.17 [22, p.350]. Where G_{ij} and B_{ij} are the polar components of the complex admittance conductance and susceptance respectively.

$$\begin{aligned} P_i &= v_i \sum_{j=1}^{N} v_j [G_{ij} sin(\delta_i - \delta_j) + B_{ij} cos(\delta_i - \delta_j)] \\ Q_i &= v_i \sum_{j=1}^{N} v_j [G_{ij} cos(\delta_i - \delta_j) - B_{ij} sin(\delta_i - \delta_j)] \end{aligned} \tag{2.17}$$

2.5.1 Newton-Rhapson method

The contents of this section are based on reference [22, p.353-356].

Considering the power flow equations analogous to the nonlinear function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, the $\mathbf{x}$ vector contains all bus angles and voltage magnitudes, the $\mathbf{y}$ vector contains all power injections, and $\mathbf{f}(\mathbf{x})$ represents all the power flow equations. Further, an additional matrix $\mathbf{J}$ containing all sensitivities of the $\mathbf{y}$ vector with respect to the $\mathbf{x}$ vector is calculated.

The *Newton Rhapson* (NR) algorithm can be summarized as follows: Start with a set of input variables $\mathbf{x}$, compute the difference between the actual bus powers and the bus powers calculated using $\mathbf{f}(\mathbf{x})$ as shown in equation 2.18. Note that i is used to specify iteration number in the remainder of this algorithm.

$$\Delta\mathbf{y}(\mathbf{i}) = \mathbf{y} - \mathbf{f}(\mathbf{x}) \tag{2.18}$$

Once the Jacobian matrix is calculated, the sensitivities are used to calculate step change in $\mathbf{x}$ to approach the solution. This can be done using Gaussian elimination or back substitution to solve the equation 2.19 for $\Delta\mathbf{x}$.

$$\mathbf{J}\boldsymbol{\Delta}\mathbf{x} = \boldsymbol{\Delta}\mathbf{y}(\mathbf{i}) \tag{2.19}$$

Finally, $\mathbf{x}$ is updated to prepare for the next iteration as given in equation 2.20.

$$\mathbf{x}(\mathbf{i+1}) = \mathbf{x}(\mathbf{i}) + \boldsymbol{\Delta}\mathbf{x}(\mathbf{i}) \tag{2.20}$$

This iterative procedure is continued until convergence is obtained or a pre-specified number of iterations has been conducted. If generator limits are exceeded within this iterative procedure, the voltage controlled generator bus is changed to a load bus (PQ). This PQ bus now has its P and Q injections set to its max values. Consequently, an iteration where generator limits are exceeded will make the generating voltage a dependent variable, and change the order of the mismatch equations.

2.5.2 Forward Backward Sweep [4]

Forward Backward Sweep (FBS) is a method to obtain a load flow solution well suited for radial and weakly meshed networks [26, p.696]. It is an efficient algorithm as it does not require a composite system model and it is modular as it is possible to update the variables in a single node. [27] The algorithm has proven to give accurate results for given test networks, and is is carried out though the following steps:

1. Establish a system model and select an applicable load model
2. Make the assumption that all bus voltages are 1pu, then calculate the loads at all buses.
3. Knowing the loads: calculate the power flows in the backward direction.
4. Knowing the flows: recalculate the voltages in the forward direction and update the loads using the new voltages.
5. Repeat the power flow and voltage calculations as described until the system has converged.

Since its infancy, the FBS algorithm has continuously been developed to suit the evolving distribution system [28].

When applying the FBS method to weakly meshed systems, the load flow problem is solved by breaking up the links that mesh the system together [26, p.696]. Then the interconnections are accounted for injecting currents at the two end nodes, thus completing the solution of the meshed system [29, p.753].

A weakness of the FBS method can be a slow convergence rate compared to a centralized approach using a gradient descent scheme (NR). This is especially valid for heavily loaded systems where a centralized approach may converge with a faster processing time and in fewer iterations [26, p.696].

FBS and PyDSAL

PyDSAL is a distribution system analysis library based on a FBS load flow solution to determine the power system state [23]. Similarly to the NR solution approach where voltage sensitivities are calculated during the solution, the FBS solution algorithm in PyDSAL calculates voltage sensitivities with reference to the slack bus.

2.5.3 DC power flow

There exists a strong coupling between real power P and the voltage angle δ, as well as between reactive power injections Q and voltage magnitudes $|v|$ [30, p.1]. This power flow model uses the strong connection between P and δ to provide satisfactory approximations of P and δ in power systems [25, p.2]. By assuming $v \approx 1$, ignoring branch resistance, all shunt elements and neglecting reactive power completely the power flow problem is reduced to a linear relationship. x_{ij} represents the branch reactance of branch ij. This power flow model is frequently used in power system planning and market clearing.

Using the aforementioned simplifications, the real power balance equation becomes as given in equation 2.21 [22, p.372-373]. Where $\mathbf{B}$ represents the imaginary part of the $\mathbf{Y_{bus}}$ matrix, and the vector $\mathbf{P}$ contains all real power injections in the system. Furthermore, the power flow on lines in the power system can be expressed by equation 2.22. x_{ij} represents the reactance on the line in question.

$$-\mathbf{B}\delta = \mathbf{P} \tag{2.21}$$

$$P_{ij} = \frac{\delta_i - \delta_j}{x_{ij}} \tag{2.22}$$

2.5.4 Fast Decoupled Load Flow

In teaching NR load flow it is common to refer to the different sensitivities present in the jacobian by different suffixes [22, p.372]. In total there are four values related through sensitivities, these are input parameters (P and Q), and the output voltage magnitudes and angles. Expanding equation 2.19 using the more detailed notation from [22] yields equation 2.23.

$$\begin{bmatrix} \mathbf{J_1} & \mathbf{J_2} \\ \mathbf{J_3} & \mathbf{J_4} \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \Delta\mathbf{V} \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{P} \\ \Delta\mathbf{Q} \end{bmatrix} \tag{2.23}$$

In this equation $\mathbf{J_1}$ contains the sensitivities $\frac{\partial P}{\partial \delta}$, $\mathbf{J_2}$ the sensitivities $\frac{\partial P}{\partial V}$ etc. In a *Fast Decoupled Load Flow* (FDLF), $\mathbf{J_2}$ and $\mathbf{J_3}$ ($\frac{\partial P}{\partial V}$ and $\frac{\partial Q}{\partial \delta}$) are neglected. This yields a set of equations which makes it possible to solve a load flow quicker than equation 2.19, yet maintaining the accuracy from a traditional NR solution [22, p.372]. FDLF is more efficient than a NR solution algorithm because the jacobian elements do not have to be recalculated in every iteration.

$$\begin{aligned} \mathbf{J_1}\boldsymbol{\Delta}\delta(\mathbf{i}) &= \boldsymbol{\Delta}\mathbf{P}(\mathbf{i}) \\ \mathbf{J_4}\boldsymbol{\Delta}\mathbf{V}(\mathbf{i+1}) &= \boldsymbol{\Delta}\mathbf{Q}(\mathbf{i+1}) \end{aligned} \tag{2.24}$$

The FDLF iteration equation is given in equation 2.24 [22, p.372]. The numbers in the parentheses indicate the iteration number. The second line in the equation uses index $i+1$ to clarify that the lines should be calculated and state variables updated one by one. It is important to acknowledge that FDLF is not a way to calculate $\Delta\delta$ and $\Delta\mathbf{V}$ separately, but rather that FDLF obtains approximately the same solution as NR using separate calculations [31]. Further it is worth noting that the two calculations can be conducted in either sequence. The two different variants are referred to as primal and dual FDLF.

2.5.5 Probabilistic Power Flow

During power system operation and in power grid expansion planning it is important to know the system load for the range of possible loads [32, p.1]. Due to reasons such as measurement inaccuracies, forecast errors or outages power system loads are not known precisely for all operating scenarios. This is a challenging problem because traditional ways of solving it requires as many solutions as there are load combinations.

Probabilistic Power Flow (PPF) is able to address a number of the operational challenges occurring in the power system due to increasing amounts of renewables in the power system [33, p.1]. The method is traditionally based on a Monte Carlo Simulation, which associates it with a significant computational burden.

Using a Monte Carlo Simulation, PPF is solved by randomly generating variables before calculating the load flow [33, p.1]. To ensure an accurate description of random variables in the power system, a large number of samples need to be included in the process. Consequently it is the need for repeated load flow calculations that makes PPF a tedious process not well suited for practical applications.

2.5.6 Calculating branch power flows

Using the π-equivalent model of a transmission line, the power flows on transmission lines can be calculated using equation 2.25 [31, p.63-64]. I_{ij} and I_{ji} represent the complex currents flowing in the line, using the suffixes to describe the direction. y refers to an element in the $\mathbf{y_{bus}}$ matrix, $\vec{V}$ represent the complex node voltages and y^{sh} is the shunt admittance.

$$\begin{aligned} I_{ij} &= y_{ij}(\vec{V_i} - \vec{V_j}) + y_{ij}^{sh}\vec{V_i} \\ I_{ji} &= y_{ji}(\vec{V_j} - \vec{V_i}) + y_{ji}^{sh}\vec{V_j} \end{aligned} \tag{2.25}$$

Power transfer distribution factors

Power Transfer Distribution Factors (PTDF) are important for tasks such as power system security analyses, planning and redistribution [34, p.1]. They describe how real power flows in the power grid behave due to an injection of power at a node in the power system. PTDFs are defined by equation 2.26. ΔF_{ij} denotes the change in real power flow on the line between nodes i and j. b_{ij} represents susceptance of the same line, and X is an element of $\mathbf{X}$, the pseudo inverse of the nodal susceptance matrix. Suffixes i and j represent the nodes connected by the line in question, s and r are nodes with increasing and decreasing power injections respectively.

$$\begin{aligned} PTDF_{(i,j),(s,r)} :=& \frac{\Delta F_{ij}}{\Delta P} \\ =& b_{ij}(X_{is} - X_{ir} - X_{js} + X_{jr}) \end{aligned} \tag{2.26}$$

2.6 Optimal power flow studies

The objective of *Optimal Power Flow* (OPF) calculations is to obtain a power flow solution for a given grid, load demand and objective function [22, p.390]. it is possible to solve this problem through a full AC OPF calculation, or to approximate a solution using a DC OPF calculation. In a case where no elements in the power system (line or generator capacities) are overloaded, the OPF problem is identical to an economic dispatch problem.

Economic dispatch determines an output of power generating units that minimizes total operating costs while meeting total load demand [22, p.376]. This point is found at the point where all power generating units operate at the same incremental operating cost. When all power generating units operate at this point, moving production to another unit would increase its cost and thus move the system away from its optimum.

Based on the load flow equations and system limitations (voltage and flow constraints) it is possible to develop a set of equations that must be fulfilled for optimality to occur [35, p.2]. These equations are commonly referred to as the (KKT) conditions.

3 Literature review

This section is included in the thesis to give a brief review of literature using different neural network based approaches on studies relevant to power flow calculations.

3.1 Neural network based power flow as an alternative to DC power flow

In reference [1, p.1] the authors point out that the industry relies on the linear DC power flow approximation to perform fast load flow calculations. While the method is adequately fast and robust, it can yield inaccurate flows on some lines in the system. This challenge is likely to increase in magnitude as greater amounts of renewable energy is introduced into the power system. To combat these challenges, they propose using a NN based model to obtain load flow results both fast and accurately. The results of the study are then used to evaluate its performance particularly compared to the non iterative DC power flow.

Artificial test data for training, validation and testing was generated in 10 000 samples by varying both initial voltage and demand at each node within $\pm$10% of the respective systems base values [1, p.2-3]. The network was fed with normalized voltage magnitudes and sums of active and reactive power injections at all buses. Output data was selected to be voltage magnitude and active power flow on each branch.

The selected network had five hidden layers, utilizing the *leaky ReLU* function for activation [1, p.3]. Weights were adjusted by means of stochastic gradient descent, based on the errors calculated using the mean square error of the prediction and exact value.

Using the mean absolute error and percentage relative difference to evaluate the models performance on IEEE 9, 24, 39, 57 and 118 bus test systems, the authors note a performance improvement over DC power flow [1, p.3-4]. Voltage magnitudes are predicted with an accuracy of 0.5%, while the line flows are closer to actual flows in the network.

Though the results are promising as the NN model outperforms a DC power flow model, the authors comment that the narrow range of operating voltages makes the NN model easier to train [1, p.4].

3.2 Neural networks for fast calculations of probabilistic power flow

In reference [8, p.1-2] it is stated that deeper neural networks have the ability to extract more complex relationships than shallow networks. Despite this, deeper neural networks can be considered insufficient. Therefore the authors propose to explicitly include the physical model in the training process, with the ultimate goal to speed up PPF calculations.

Furthermore, the authors in reference [8, p.2] present three improvements to the neural network implementation. (1) A composite loss/objective function based upon the branch flow equations should be used. (2) The training process should be simplified in a similar manner as FDLF, by removing the gradients related to system voltages. (3) An adequate mix of activation functions should be used together with an improved network initialization technique.

The neural network input and output vectors are selected as nodal power injections and complex voltages respectively [8, p.3]. The output vector is selected as it is a complete description of the power system. Since line flow calculations are dependent on the current system state, errors from the state calculation may propagate into the line flow calculations. To avoid this, the authors augment the loss function for network training, with inaccuracies in line flow calculations. The augmented loss function is given in equation 3.1.

In equation 3.1, $\mathbf{J}(\mathbf{P_{out}}, \hat{\mathbf{P}}_{\mathbf{out}})$ represents the normalized mean square error in the branch power flows, and $\mathbf{J}(\mathbf{Q_{out}}, \hat{\mathbf{Q}}_{\mathbf{out}})$ represents the normalized mean square error in reactive power flow [8, p.3]. This way of implementing the power flow model to the model will improve its learning speed.

$$loss_{new} = loss + \mathbf{J}(\mathbf{P_{out}}, \hat{\mathbf{P}}_{\mathbf{out}}) + \mathbf{J}(\mathbf{Q_{out}}, \hat{\mathbf{Q}}_{\mathbf{out}}) \tag{3.1}$$

Guiding the voltage angle calculations of the NN is more important than guiding the voltage magnitudes to their correct values [8, p.4]. This is due to the fact that voltage magnitudes fluctuate much less, usually within $\pm$5%, whereas voltage angles may fluctuate much more sometimes greater than $\pm$30 deg. Furthermore, the phase angle characteristic is more complex than that of the voltage magnitude, thus the angle relationships are more challenging to learn. In addition to this, voltage guidance in the learning process is more computationally intensive than angle guidance. Consequently, to improve both training efficiency and performance it is reasonable to remove voltage magnitude guidance from NN training.

A final addition to the training process should be removing the guidance of reactive power on the phase angles [8, p.5]. Due to the fact that transmission grids generally have a much greater inductance than resistance, as formulated in equation 3.2. In normal cases phase angle differences between two buses is often small, which makes equation 3.3 true.

$$|B_{ij}| \gg |G_{ij}| \tag{3.2}$$

$$sin(\Theta_{ij}) < cos(\Theta_{ij}) \tag{3.3}$$

$$\frac{\partial P_{ij}}{\partial \Theta_i} > \frac{\partial Q_{ij}}{\partial \Theta_i} \tag{3.4}$$

Reactive branch power flows have a lower impact on the training process than active power flow, therefore the reactive branch powers influence on the power angle should be ignored [8, p.5]. Following equations 3.2 and 3.3, equation 3.4 is true. In addition, the errors in reactive power flow estimates are generally lower than those of active power flow estimates because voltage magnitudes are easier learned. Furthermore, the standard deviation of active branch flow is generally lower than reactive power flow since active load demand usually is greater.

The ReLU function is used as activation function for all layers except for the output layer [8, p.5-6]. As the ReLU function does not allow for negative outputs, the output layer utilizes a linear activation function. Finally, the authors use a special NN weight initialization technique initializing all weights using a zero-mean Gaussian distribution. All biases are initialized as zero.

Results show that modifying the loss function and using the proposed initialization method yields significant performance benefits [8, p.8]. These measures are capable of speeding up the network convergence and can reduce the risk of over-fitting the voltage magnitudes. Comparing further simplifications, removing guidance for voltage magnitudes and angles in the training process, maintains the performance benefits gained by modifying the loss function and initializing network weights with the proposed method. In the small test network (modified IEEE 30 bus) there was a a small performance loss, whereas the larger networks (modified IEEE 118/661 bus networks) experienced a noticeable improvement.

3.3 Load flow applications for neural networks

In 2010, reference [36, p.1] proposed using NNs to solve the basic load flow problem, to solve the load flow problem with active and reactive power limits, to determine a starting point for load flows with ill conditioned matrices and to establish static external equivalent circuits. Since these process are frequently used live in network control applications, NNs could be used to provide a solution through relatively simple calculations. Due to the scope of this thesis, only their approach to the basic load flow problem is included.

Using a single NN to estimate all variables for a system produces unsatisfactory results [36, p.2]. They can be improved by utilizing two NNs per bus, one to estimate voltage magnitude and one to estimate the voltage angle. Consequently their NN model consisted of an input layer, a hidden layer, and the output layer.

Inputs to the neural networks are given as a vector X containing the values G_d^i and B_d^i as described in equation 3.5 [36, p.3]. G_d^i and B_d^i are vectors based on conductance and susceptance matrices and system loading. G_d^{norm} and B_d^{NORM} represent the diagonal elements of conductance and susceptance matrices, normalized using the largest base case element. P_L, Q_L, and Pg represent the load and generator powers normalized using their base case values. CV_p and CV_q are utilized to add information about system outages. Given an outage in branch km, CV_p and CV_q contain the outaged branches conductance and susceptance. The two values are also used to compensate for shunt and generator outages.

$$\begin{aligned} G_d &= V_g^2 G_d^{NORM} - P_L^{NORM} + Pg_{pv}^{NORM} + CV_p \\ B_d &= -V_g^2 B_d^{NORM} - Q_L^{NORM} + CV_q \end{aligned} \tag{3.5}$$

Training and simulation data was limited to load ranges of $[0.75, 1.25]$ and $[0.73, 1.27]$ respectively, and applied simultaneously for all buses [36, p.3-4]. The number of neurons in the hidden layer was determined through evaluation of NN performance. For training and validation the set of networks was trained for the IEEE 30 bus using 1149 samples and verified using 1212 samples. Finally, the IEEE 57 bus was also simulated by a set of networks trained on 743 samples, and simulated on 864 samples. The reported maximum percentage magnitude and angle deviations where 0.12% and 0.036% respectively.

3.4 Using neural networks to emulate the optimal power flow algorithm

Baker [37, p.1] proposes using neural networks instead of computationally expensive matrix inversion and factorization within power system optimization algorithms. As feasibility cannot be ensured using an iterative NN solution, a subset of the proposed solution is forwarded to a power flow solver to ensure the feasibility of the final solution. Advantages over prior approaches includes few restrictions on training data, no direct prediction of a solution (as a regular solver is emulated), each iteration utilizes all system information which traditional solvers do not and deep networks are not required.

Iterations are conducted using a three layer neural network with inputs and outputs $\mathbf{x^k}$, containing vectors of complex voltages and complex generator powers [37, p.2]. Inputs where normalized to restrain all values within $[0, 1]$. Test data was generated using MATPOWER Interior Point Solver using a convergence tolerance of 10^{-4}. For each load bus, 500 different loading scenarios where generated based on a $\pm 40\%$ uniform distribution around the base load. Only test data with a valid load flow solution was included in the dataset.

Results show that the NN approach is capable of delivering feasible AC-OPF solutios with errors below 3% in similar times as a DC-OPF calculation [37, p.2-3]. Both the average and worst error of the NN approach showed significant improvements over the traditional DC-OPF approaches.

3.5 Physics informed neural networks for power system state and parameter estimation

Pagnier and Chertkov [38, p.1] provide two reasons that NN based power system analysis approaches were not trusted by network operators. Firstly, purely NN based approaches lacked a physical foundation, and secondly due to data scarcity in certain operating environment. To tackle these changes, the authors have developed a physics informed method to reconstruct physical parameters such as line admittances within power flow models. Their proposed method outperforms pure NN approaches that do not consider power flow physics.

Previously, parameters in the power flow equations were updated systematically while delays and uncertainties were accounted for in state and parameter estimation [38, p.1]. Generally, this did not induce any problems in the power system as it in many cases was overbuilt. Due to the widespread integration of renewable energy sources, this fact is changing and the power system is being pushed closer to its limits.

3.6 Physics informed NNs to minimize worst case DC-OPF violations

The authors in reference [15, p.1] propose using a PINN to conduct a DC-OPF calculation with guarantees related to constraint violations, maximum deviation between predicted and optimal decision variables and a maximum sub-optimality of the entire input domain. PINNs are especially relevant to this topic since regular NNs require large training data sets of good quality. Considering that the model should be able to predict abnormalities with comparable accuracies, traditional NN models can be challenging to generate.

Optimal generation outputs are calculated based on the reactive power demand as input [15, p.2-3]. The physics informed element is implemented into the NN by modifying the loss function based KKT condition discrepancies. As this evaluation requires estimates of dual variables, a separate neural network with equal inputs is used to calculate the required dual values. during network training, weights are adjusted according to the mean absolute error.

In addition to modifying the cost function, the network is trained to operate similar to the underlying physical principles by using a set of collocation points [15, p.3]. Collocation points are different from regular data points due to the fact that the prediction error is only based on the discrepancy compared to the KKT condition. For the collocation points, the errors in dispatch or dual value prediction are considered zero.

PINN performance was evaluated based on 3 test networks each consisting of 39, 118 and 162 buses. Active power demand was varied in the interval $[60, 100]\%$ of nominal, and a total of 100,000 samples were generated [15, p.4]. 20% were used for regular training, 50% reserved for collocation points and the remaining 30% to evaluate model performance.

Worst case guarantees effectively set an upper tolerable limit to constraint violations [15, p.4]. These can be implemented by reformulating the NN into a *Mixed Integer Linear Programming* (MILP) Problem it is possible to calculate the maximum deviation that the NN model will predict for any input.

The two different NNs used for predictions utilized slightly different architectures [15, p.4-5]. To predict optimal active power dispatch a network with three hidden layers using 20 neurons in each was used. To predict the dual values a network with three layers each consisting of 30 neurons.

The proposed PINN has a satisfactory generalization ability. During the simulation procedure, it was observed that PINN performance was sensitive to changes in hyperparameters [15, p.4].

3.6.1 Further adaptations to AC-OPF

In [35] the concept from [15] is developed from PINN based DC-OPF to AC-OPF. [35] has a similar objective, providing an accurate solution to the AC-OPF problem swiftly with guarantees of solution accuracy.

Given an AC-OPF problem, the objective is to predict a vector of active and reactive power generation setpoints given an input vector containing active and reactive power demands [35, p.1-3]. Information is processed by three independent neural networks, one to predict optimal complex generator outputs, one for voltage setpoints and finally one network to predict dual values.

Again, the cost function is modified to include the discrepancy in the (AC) KKT conditions [35, p.3]. This discrepancy is later used to train the network on a set of collocation points without specified solutions. During collocation point training, the mean squared error in dispatch, voltage and dual value predictions are neglected.

In addition to evaluate PINN performance on an unseen dataset, the authors have made an effort to formulate worst case guarantees of the model performance [35, p.4]. A MILP problem was formulated to calculate power generation constraints. Estimating the worst case error in line flow constraint estimates is more complex due to the convex and quadratic nature of the AC-PF equations. Consequently, a Mixed Integer Quadratic Constrained Quadratic Programming approach was used to generate the worst case guarantees.

The final algorithm was tested on a 39, 118 and 162 bus cases. Active and reactive power demands were assumed independently of each other, and specified to be within $[60, 100]\%$ of their nominal value[35, p.4]. A total of 10 000 test samples were generated, of which 50% were reserved for collocation training, 20% used for regular training, and 30% reserved for verification. Training was conducted in 1000 epochs with the data separated into 200 batches.

Post implementation, the authors note that the PINN requires a considerably more training time than a traditional NN [35, p.5]. However, it is important not to forget that PINNs do not require prepared solutions for all datasets used in training, which reduces other aspects of time spent generating PINNs.

Results show lower percentage deviations for PINN performance when compared to simple NNs [35, p.5-6]. Errors seem to increase with network size, but are below 2.5% for all predictions. Further the authors note that PINNs provide lower worst case errors compared to NNs, and that further 15-30% improvements are achievable by hyperparameter tuning. Finally, the authors note that model tuning to reduce worst case errors reduces the average prediction accuracy.

3.7 Using a stacked ELM

In [17, p.1] the authors propose using a *Stacked Extreme Learning Machine* (SELM) to solve the OPF problem. An increasing amount of renewables introduced into the power system requires OPF calculations to be solved in real time. While there exist computationally cheaper models obtained using simplifications, they may not be adequately accurate. Other methods to some extent using heuristics to improve model accuracy have proven robust. However, all methods still require several iterations within the solver.

Data driven approaches have gained an increasing amount of popularity in recent years, in particular NN based approaches [17, p.1]. An important disadvantage when using NNs is that their performance is dependent on a number of hyperparameters. These need to be adjusted manually, and there exists no efficient algorithm to guide this process. Using a SELM can reduce training while simultaneously reducing the amount of hyperparameters that need to be tuned.

Each layer in the SELM, ELM is a single layer feed forward network [17, p.1-2]. Input layer weights are generated randomly and output weights are calculated through a set of matrix computations. This process improves training speed and reduces the amount of hyperparameters to be tuned. A SELM splits an otherwise large NN into several smaller networks thus utilizing less memory and improved feature extraction capability.

An OPF problem may be regarded a nonlinear projection taking complex power demands as input and outputting complex power generation and voltages at all nodes in addition to line power flows [17, p.3-4]. As the learning capacity of a SELM is limited for problems with complex relationships, the learning process is divided into three stages:

1. The first ELM learns the relationship between complex nodal power demands and network branch flows.
2. A second ELM learns the relationship between complex power flows and complex voltages.
3. Finally, the third ELM maps complex power injections, voltages and flows to the final dispatch.

The accuracy in each single layer does not have to be sufficiently high because stages two and three also function as error correction stages.

Including all proposed improvements to the original SELM, the authors report promising results [17, p.7]. The proposed model yields close to 99% accuracy for all values except for the voltage angle and reactive dispatch, whose accuracies are close to 95%. Despite the high accuracy, the training time is very low, at 83.23s on the IEEE39 bus test system. Similar results were achieved using IEEE 57, 118 bus and on the Polish 2383 bus systems. [p.8] Finally, the authors comment that the proposed method requires few adjustments and should be simple to apply to other systems.

4 Approach

This section gives a description of how the NNs for prediction were developed. The first subsection describes the reasoning supporting practical choices, and the following parts describe in chronological order how the code repository [39] was developed.

Table 4.1: *Names, sizes and Nicknames of the test networks used in this thesis*

Test network	Number of buses	Nickname
Small test network	4	Small
IEEE 33 bus	33	Medium
IEEE 69 bus	69	Large

In the remainder of this thesis the different test networks are referenced using the nicknames in table 4.1. The provided nicknames are used to reference both the test network and its related NN to solve the load flow of the test network, which reference is meant should be interpreted by context.

4.1 Deciding on practicalities

Building a python library of classes and functions to build NNs from scratch requires a wide variety of decisions. This section describes the practical decisions governing the development.

4.1.1 Selecting the machine learning package in python

To generate the neural network used in this thesis, an existing machine learning package was used for model creation in this thesis. Since the choice was made early in this work, the two candidates considered were Google/Alphabets Tensorflow package and the *scikit-learn* package [40]. Tensorflow was considered as it was used in [14], and scikit as it was proposed by the thesis supervisor. Finally, Tensorflow was selected as it natively supports automatic differentiation, which was important for the PINN implementation in [14].

4.1.2 Selecting test networks

Due to limited previous experience with neural networks, a the small test network from figure 4.1 was chosen for the initial implementation in order of improving the understanding of the underlying algorithms. A non radial network was chosen because it allows for slight topological changes, and since synthetic test data was to be generate using the NR algorithm.

To verify the flexibility of the neural networks, the same approach was applied to the IEEE 33 and 69 bus systems. An important reason to use other test networks is to demonstrate that PINNs are capable of providing accurate results without excessive tuning of hyperparameters. Since the thesis supervisor provided input data for the IEEE 33 and 69 bus on a compatible data format, they where selected for use in this thesis.

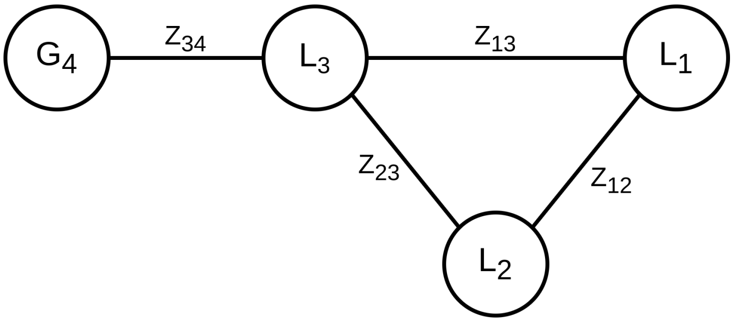

Figure 4.1: *Single line diagram of the network used for the initial tests. Circles represent buses, drawn in draw.io*

The IEEE 33 and 69 bus systems are illustrated in figures 4.2 and 4.3. In this thesis, both systems have been treated as radial distribution systems. As a consequence, all switchable branches and additional generating units (other than slack node at bus 1) have been neglected.

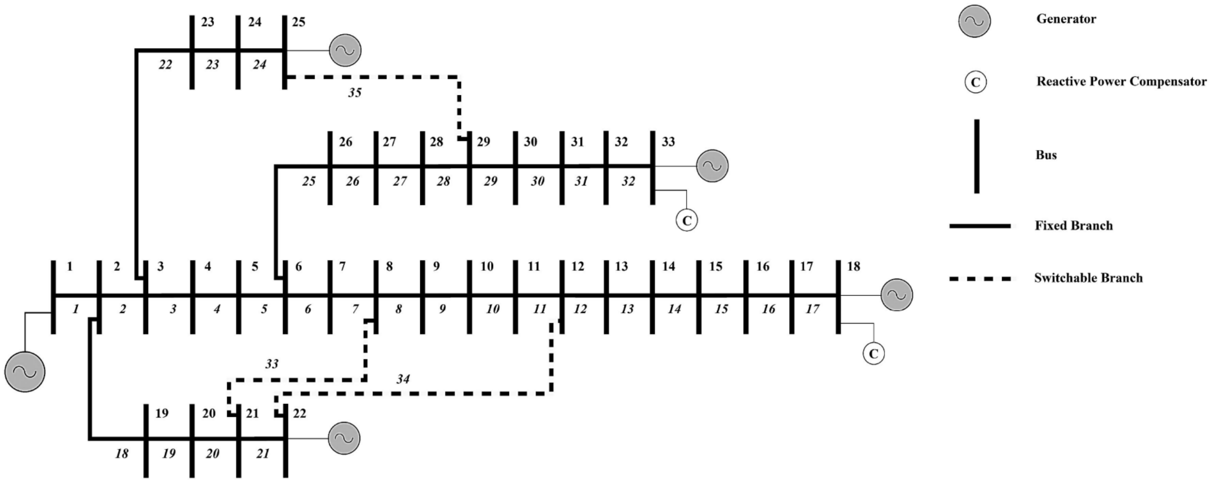

Figure 4.2: *The IEEE 33 bus test network [41]*

4.1.3 Selecting a load flow solution algorithm

The load flow algorithm used to generate the test data for the small network is a standard NR solver that does not handle generator overloads or voltage deviations. These limitations were not considered to deteriorate the quality of the results of this thesis as it focuses on the application and pattern recognition abilities of neural networks. Furthermore, the work on this thesis was intended for use on distribution grids which are usually supplied by a single source with limited voltage regulation capabilities.

To reduce the workload related to this thesis, a NR solver was selected to solve load flow cases. The particular solver used was developed by the author as part of a specialization course (TET4575) during the fall in 2022. Further arguments considered in favor of using a NR solver were its robustness when facing meshed grid configurations.

Due the the scope of this thesis, studying distribution grids, all test networks test network are supplied by a single slack bus without the presence of any other generator buses.

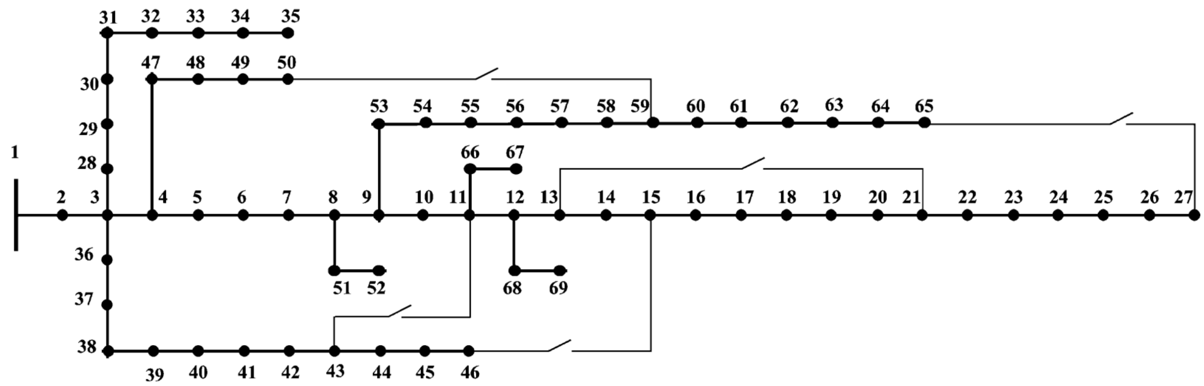

Figure 4.3: *The IEEE 69 bus test network [42]*

While, traditional distribution grids are operated radially, this thesis considers the possibility of slightly meshed topologies in distribution grids. An important reason to include meshed topologies in the analysis is that it will not present an additional computational cost once the neural network is trained.

As the NR solver was slow solving larger networks, an adapted version of PyDSAL used in the project work was used to solve load-flows for the medium and large test networks. This saved both time and computational resources. The required code to perform a load-flow calculation with PyDSAL is available through [39].

4.1.4 Generating synthetic test data for the small test network

To generate a dataset for inputs to the NN model, simulations were conducted on the test network to establish feasible ranges for load variations. Each load was increased until the NR solver did not converge, while keeping the other loads at their normal levels. The critical load level for each load was noted down, and random numbers were generated using *np.random.uniform(low=, high=, size=,)*.

This process was repeated for all three load buses for both active (P) and reactive (Q) power. Note that the active and reactive power demands at each node were predicted independently of each other, and that slack bus data is not included in the neural network. Finally, the data was stored in large two dimensional *np.arrays*, on the format [PPPQQQ] and [VVV$\delta\delta\delta$].

4.1.5 Generating synthetic test data for the medium and large test networks

A similar approach to the one presented in [1, p.2] was used to generate test data for the medium and large test systems. The authors of reference [1] generate an artificial dataset by varying nodal load demand within the interval $[0.9, 1.1]$ of nominal load demand. This approach has been adopted in this thesis, and is used as a baseline. A dataset varying nodal load demand within $[0.8, 1.2]$ was also generated for the purpose of performance comparison. Dataset generation was based on *np.random.Generator.uniform*.

In order to make the results simpler to compare, the amount of randomly generated data samples was kept constant at 60 000 data samples, of which 20% were reserved for testing.

The filename of the file used to generate the large datasets was *gen_dataset_general.py* in [39]. Similarly to the small NN data, the medium and large training data was stored using the input format [PPP...QQQ...], and the output format [VVV...$\delta\delta\delta$...].

Note that the system loading and system parameters were all used as provided in the system description files found on github. There is no guarantee that these are the default values as described by IEEE. However, since the objective of this work is mostly a proof of concept, actual values of system loading are of lesser importance.

4.1.6 Testing different hyperparameters impact on NN performance

To determine the impacts different hyperparameters have on NN performance, they were changed one by one. Once results for a wider range of hyperparameter changes was present, promising results were combined to see if further improvements could be obtained.

4.1.7 Use of computational resources

Training of the small and medium NNs was primarily carried out using a Dell XPS 9360, (private laptop, 16gb ram, intel i7 7500U). This was done for ease of implementation and bug fixing in the code repository. Training of some medium and all large test network were carried out on the HPC Idun exclusively, as the memory on the XPS 9360 was insufficient. To save time adapting the library of functions all simulations were executed on a single processing core on Idun.

4.2 Developing the initial neural network model

This subsection describes in chronological order how the different parts of the code in repository [39] was developed. Relevant parts also include the reasoning behind the selection of hyperparameters used in the different NN models.

The first implementation of a neural network was written using a function based approach in python, using Tensorflow to operate and train the NN model. A functioning neural network can be found in the aforementioned repository at commit 63d2697. This code was used as a baseline for further developments, and its performance is described in closer detail in subsection 5.1.

4.2.1 Selection of hyperparameters

All hyperparameters used in NN training are given in tables 5.1, 5.5 and 5.18. For testing and evaluation of the neural network a dataset containing 60,000 samples was used, 20% of the dataset was reserved for performance evaluation of the neural network. The NN architecture is given as number of neurons from from inputs to outputs, as illustrated in figure 4.4. Since the generated dataset was purely synthetic, it was considered unnecessary to randomly shuffle the order of test samples. Model weights were initialized using tensorflow's default initializer: *tf.keras.initializers.GlorotUniform*

In an attempt to improve performance, inputs were scaled down using a factor of two, thus ensuring that all input values were approximately within the interval [0, 1]. As the outputs of the NN are given using a per unit representation, they are in most cases also in the interval [0, 1]. Considering that a standard MSE error function was used to evaluate the results, outputs were scaled up by a factor of 10. In doing this, the goal was to keep the error function from diminishing when approaching the objective value. Note that the selection of the up-scaling factor was entirely based upon intuition.

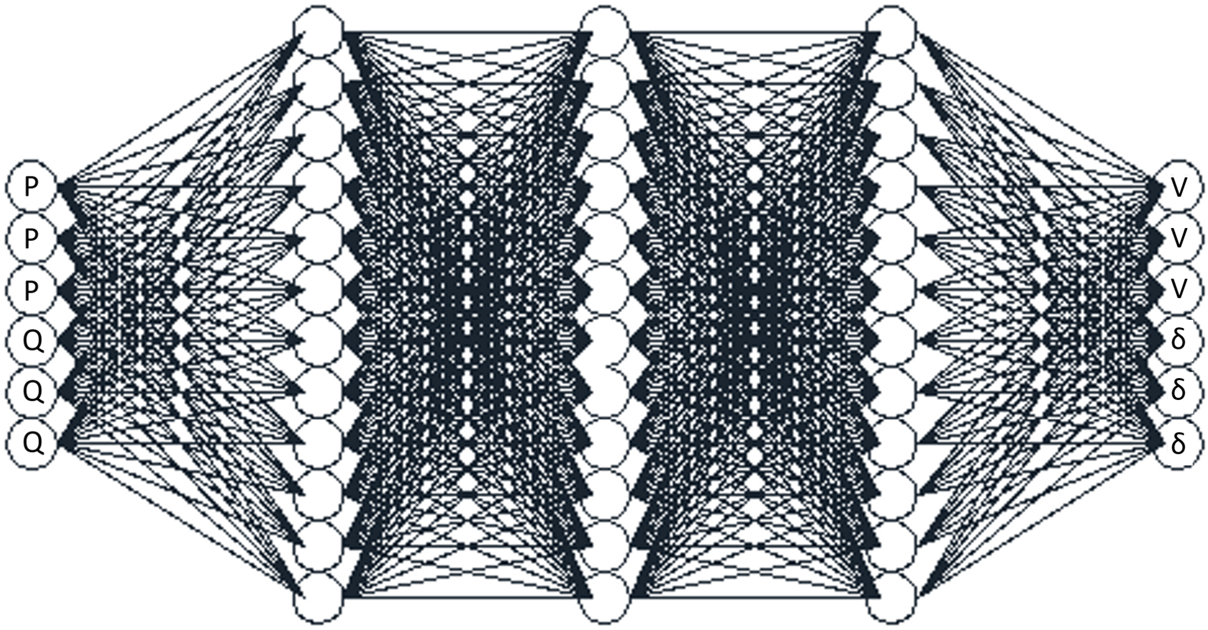

Figure 4.4: *Illustration of the small neural network architecture with inputs on the left and outputs on the right. Generated with and adapted from the eiffel2 python package.*

For later training procedures with the medium and large test networks, some of the hyperparameters were changed in an attempt to improve performance. Efforts were made to keep most parameters as similar as possible between different training sessions. For sessions evaluating the impact of a reduced learning rate, the number of total training epochs was increased to allow for the same amount of total learning. Furthermore, the number of training epochs was increased with increasing NN size.

Due to limited experience with neural networks, and because the ReLU activation function is used in literature, it was selected for use with all NNs trained in this thesis.

4.2.2 Evaluating neural network performance

The small neural network performance was evaluated by predicting the output values of unseen test samples, and comparing it to the known NR solution. For the small test network all comparisons were conducted on test data using evolutions of the functions in [39] at commit 63d2697. The functions at this commit take a list of NN models, in this case the same NN model at different checkpoints. For each element in this list, the test dataset is used to predict the outputs the test dataset. These predictions are then compared to the known outputs, before average accuracies and worst prediction accuracies were calculated.

Later the *generate_performance_data_dict* method, described in section 4.3.2 was generated. This method was used to analyze performance of the small PINN and all medium and large NNs.

4.3 Generalizing the neural network model using an object oriented approach

To improve control over NN hyperparameters throughout a series of simulations on both the same or between different test networks, the code used to produce the results in subsection 5.1 was adapted to an object oriented programming environment.

In this implementation, a python class *NeuralNetwork* was created. It's attributes are described in table 4.2. The attributes were added to *NeuralNetwork* to improve consistency in hyperparameters for different simulations and NN models.

Table 4.2: *Overview of the attributes to the Python class NeuralNetwork*

Attribute name	**Intended datatype**	**Explanation**
l_rate	float	Learning rate
epochs	int	Amount of training epochs
batch_size	int	Batch size during training
l_data	np.array	Input training data
l_sol	np.array	Solution to training data
t_data	np.array	Test/verification data
t_sol	np.array	Solution to test data
model_sol	np.array	NN predictions of *t_data*
norm_input	int	input normalizer
norm_output	int	output normalizer
tf_model	Tensorflow model	Tensorflow model
loss_fn	tf.keras loss function	Loss function used by Tensorflow
initializer	tf.keras function	Tensorflow weight initializer
avg_model_predict_time	float	Single prediction time
abs_percentage_pred_errors	np.array	Average accuracies of all outputs
architecture	python list	Topology of *tf_model*
performance_dict	python dictionary	Dictionary of performance data

A *NeuralNetwork* object is able to utilize a selection of class methods, most of which are thoroughly documented in the file *NN_objects.py* in [39]. The bulk of its methods adapt a desired data format to the Tensorflow package. Important methods include *init_data*, *init_nn_model_dynamic* and *model_pred*. These methods load training data, initialize and compile a tensorflow model from a topology list and calculate the models predictions. The NN is trained using *tf.model.fit*.

4.3.1 Folder structure in [39]

The code in [39] is split between different folders. The files containing the NR solver, its dependencies and all *.xls* system descriptions are located in the folder *LF_3bus*. Relevant files from PyDSAL are located in the PyDSAL folder. Finally, the project folder contains the main file: *NN_function.py*, and the file to plot and analyze performance, *plot_performance.py*. These two files largely depend on the contents of the *Neural_network* folder.

Within the *Neural_network* folder, the two most frequently used folders are *checkpoints* and *datasets*. These two folders are used to store model checkpoints generated with *NN_function.py* and contain training datasets respectively.

4.3.2 The *NeuralNetwork.generate_performance_data_dict* method

To evaluate *NeuralNetwork* objects, a class method to evaluate neural network performances was added. This method uses the absolute percentage error matrix $\mathbf{P}$ containing all percentage differences between the known outputs and predicted network outputs. The dictionary contents generated by *NeuralNetwork.generate_performance_data_dict* is described in table 4.3

Table 4.3: *Contents of NeuralNetwork.performance_dict*

Data key	Content description
averages	Column averages of $\mathbf{P}$
overall_average	Average of column averages
average_angle	Average of outputs in angle columns
average_voltage	Average of outputs in voltage magnitude columns
Xpercent	Dictionary of threshold specific metrics

Table 4.4: *Contents of Xpercent as described in table 4.3*

Data key	Content description
pred_errors_over_threshold	List of indexes describing the location of single prediction errors exceeding the threshold value X.
sets_with_pred_errror_over_threshold	Number of predictions that contain at least a single prediction error over the threshold value X
sets_worse_than_threshold	Percentage describing the amount of sets containing an error compared to the test dataset.

The *generate_performance_data_dict* method carries out operations in the same order as they were described in table 4.3. Upon a method call, the first operation is the calculation of the column average of $\mathbf{P}$. Further, the average of the column average is calculated to summarize model performance using a single digit. In a similar manner, average_angle and average_voltage are the absolute average errors in voltage angle and magnitude calculations respectively. *Xpercent* is represents a series of dictionaries with threshold specific metrics. The contents of a single *Xpercent* dictionary is described in closer detail by table 4.4.

4.4 Implementing a physics informed neural network

To improve training efficiency and possibly reduce the models dependency on large datasets, a physics informed element was to be introduced to the neural network model developed in earlier stages. Since altering the loss function is a relatively simple and flexible process, this was the physics informed approach utilized in this work.

The first custom loss function utilizing line flows to generate a loss value was called *CustomLoss*, and it inherited attributes and methods from *tf.keras.losses.Loss*. Additional attributes provided to *CustomLoss* where (1) the *y_bus_matrix* generated from a method within the *Load Flow* class, (2) the *output_normalizer*, used to calculate correct line flows and angles because the outputs from the NN were scaled. Finally, the number of buses in the system was stored as an attribute within *CustomLoss* for convenience.

Table 4.5: *Overview of different loss classes existing in [39]*

Class name	**Description**	**Return dimension**
CustomLoss	Calculates the line flow matrix and associates the average flow of all lines connected to the bus with its output variables. The sum of a regular MSE and MSE absolute line flow error is used as loss value.	1*1
SquaredLineFlowLoss	Calculates the sum of the squared errors in average absolute line-flow and regular MSE error.	bs*o
LineFlowLossForAngle	Calculates the sum of squared errors in average absolute line flow and regular MSE error. Average line flow errors are associated only to outputs of angle predictions.	bs*o

As *CustomLoss* contains attributes and methods required to calculate a line flow matrix, all other loss functions inherit from it. An overview and a short description of all loss functions included in [39] is provided in table 4.5. bs*o represents the dimensions *batch_size**outputs*, where batch size is the number of predictions made before weight adjustment during the learning procedure and outputs are the NN models output.

The return dimension of the loss values was included in table 4.5 because it is important for the weight adjustment within the network. With a loss value returning a single value this is used for adjustments starting at all outputs. For error functions with return dimension bs*o, weights are adjusted backwards starting with output specific error values.

4.4.1 Calculating line flows

Because the line flows were to be used within the training process of the neural network requiring network outputs, and a built in training method was used, the line flows needed to be calculated within the loss function class. The line flow calculations were based on equation 2.25, neglecting shunt flows for simplicity. In practice, the line flows were calculated and stored in a matrix $\mathbf{L}$. $\mathbf{L}$ was initialized as a $n * n$ matrix of complex zeroes, where n is the number of buses within the system. A nested loop iterates through this matrix calculating the complex line-flow for all non-diagonal and connected buses. Diagonal and non-connected line flows were set to complex NaNs for later convenience.

4.4.2 The *CustomLoss* class

Upon a call for *CustomLoss*, Tensorflow utilizes *CustomLoss.call* to generate loss values for adjusting weights. This function first calculates $\mathbf{L}$ for both the model predictions and the known actual values. A loss value was obtained by calculating the element-wise absolute value of $\mathbf{L}$. Further, the absolute line-flows were associated to their respective outputs by calculating the *np.nanmean* column average for all buses except the slack. *np.nanmean* calculates the average of all non-NaN values in a *np.array*.

$$L_{CustomLoss} = MSE + MSE_{flow} \tag{4.1}$$

When *CustomLoss* is used as loss function in a NN model, it returns a single loss value as given in equation 4.1. MSE is the regular MSE loss of the output mismatches, whereas MSE_{flow} is the MSE of the flow mismatches. Note that the line flows are multiplied by *CustomLoss.output_normalizer* before any further calculations are carried out.

CustomLoss is used as a parent class for the other implemented cost functions. This means that other loss function classes inherit all its attributes and class methods. Inheriting from a parent class is useful as it makes it easy to pass on core functionality and overwrite for instance the *.call* function to force a return of more than one value. This can be useful to reproduce more efficient training of the neural network model.

The *SquaredLineFlowLoss* class is identical to *CustomLoss*, except for its return statement. *SquaredLineFlowLoss* returns a loss value for each prediction. Thus it returns a matrix of dimensions bs*o. This loss function was implemented to compare the performance of MSE to more detailed losses.

4.4.3 The *LineFlowLossForAngle* class

This loss function inherits from *CustomLoss* and changes only the method *associate_mean_lineflows_to_variables*. Therefore it return statement is of dimensions bs*o, and the values contained in its return statement are described by equation 4.2. In practice this is implemented by setting all MSE_{flow} errors related to voltage magnitudes to zero.

$$\begin{aligned} L_{magnitudes} &= MSE \\ L_{angles} &= MSE + MSE_{flows} \end{aligned} \tag{4.2}$$

LineFlowLossForAngle was implemented in an attempt to guide the voltage angles towards their correct values using greater errors than the voltage magnitudes, inspired by ideas from [8]. Since the voltage magnitudes have a more nonlinear relationship than the magnitudes, this could aid model convergence as the angle related weights and biased would be iterated more strictly.

4.5 Preparing for simulations on Idun

To make the code easier to operate on Idun, the master script was adapted to become a single function given in the file *NN_function.py*. The contents are identical to the previous script, the only difference is that all hyperparameters are submitted as function arguments. Previously, hyperparameters were changed directly in the master script.

Finally, a function to print performance data on the format used in section 5 *Results*. This was done since files pickled on Idun could not be opened off-server. The function is named *print_performance*, and lies within the file *plot_performance.py*

5 Results

All results from the different NN training sessions are given in this section. Section 5.1 describes the performance of the small test network. Section 5.2 describes the medium NN performance, while 5.3 describes the large NN performance. The introduction to each of the sections contains tables describing the baseline hyperparameters for all training sessions in that section. For the medium and large test networks, where multiple training sessions have been conducted for different hyperparameters, only the changed hyperparameters are mentioned as the others are taken from the baseline table.

5.1 Small test network

This section contains the results from the small test network. It consists of two parts, the first presenting the very first results obtained by training a small NN with basic settings and without much experience using tensorflow. The second part gives results where a stricter control over the hyperparameters is applied and an improved set of metrics is used to evaluate NN performance.

5.1.1 Initial pure NN performance

This subsection gives a summary of the results from tests of a NN predicting the system state of a 4 bus network using the code in [39] at commit 63d2697. Data presented in this section should be considered a demonstration of untuned NN performance.

Implementing the parameters as given in table 5.1, a neural network with the ability to estimate the bus states with reasonable accuracy can be established. The network is able to predict the voltage magnitudes and angles with average accuracies and worst-percentage-errors as given in figure 5.1. The data used to produce this figure is provided in tables A.1 and A.2. After training epoch 150, the greatest average deviation is found for $\delta_2 = 1.11\%$, and the greatest worst deviation is $\delta_2 = 1315.8\%$.

Table 5.1: *Baseline hyperparameters used to train a NN to solve a load flow for the small test network with 4 buses*

Parameter	Value
Dataset size	60000 [samples]
Test fraction	0.2
NN architecture	[6, 12, 12, 12, 6]
Batch size	20
Epochs	30
NN weight initializer	tf default
input scaling	/2
output scaling	*10

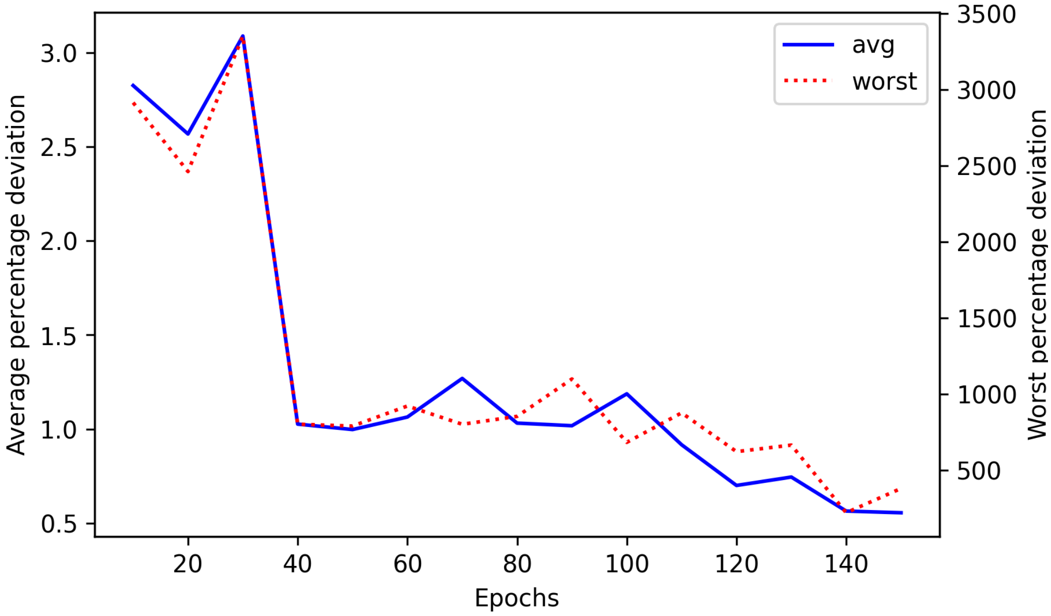

Figure 5.1: *Average and worst errors of the early small NN as a function of training epoch. Plotted using the data in tables A.1 and A.2*

Table 5.2: *Detailed prediction accuracies of the first NN model with the convergence in figure 5.1. Table generated using hyperparameters from table 5.1.*

Threshold	Accuracy	Accuracy angles	Accuracy voltages
3%	95.6%	95.6%	99.9%
20%	99.5%	99.5%	100%

A more detailed analysis of the results revealed that the model more accurately predicts the voltage magnitudes than the angles. Table 5.2 shows how accurately the model predicts network state. All prediction sets that include more than a single variable prediction error above the threshold are counted in the statistics. Thus 95.6% of the sets are predicted with an accuracy greater than the threshold value.

5.1.2 Physics informed NN performance

This subsection presents results obtained by training a NN to solve a loadflow for the small test network, using the hyperparameters in table 5.1. The only hyperparameter that was changed was the loss function used during training.

Figure 5.2 shows the convergence performance of NNs trained with different loss functions. Note the inclusion of a reference network trained using a regular MSE function. Average performance data after training epoch 30 is given in table 5.3.

Table 5.4 describes how many single state variable predictions contain at least one prediction with an absolute percentage deviation greater than the threshold value. A clear tendency is that the number of predictions containing at least one error increase as the threshold is reduced.

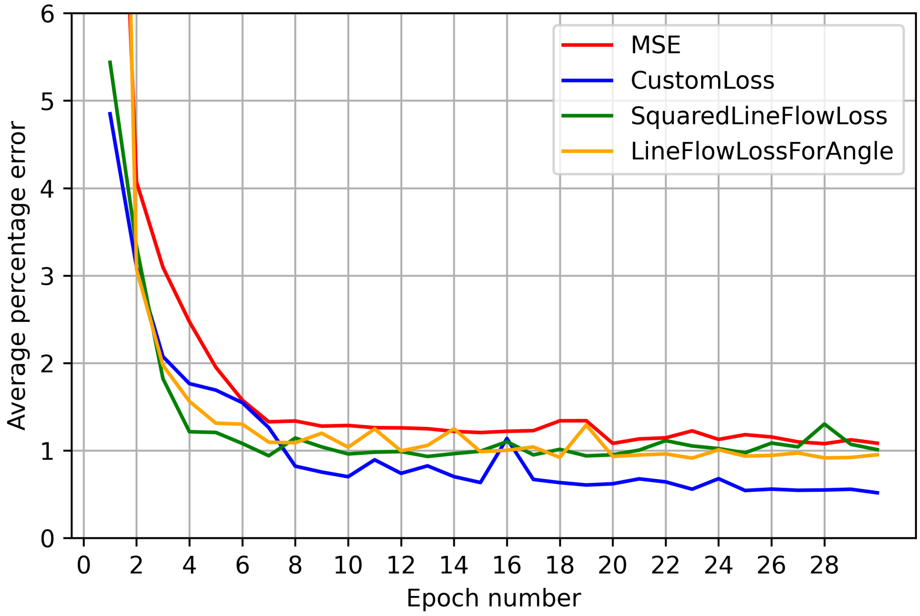

Figure 5.2: *Average error of the small NN as a function of training epoch for different loss functions.*

Table 5.3: *Detailed general performance data for the small NNs trained using different loss functions. Data produced after training epoch 30 and given as percentages.*

	Overall errors	**Magnitude errors**	**Angle errors**
MSE	1.08	0.30	1.86
CustomLoss	0.52	0.09	0.94
SquaredLineFlowLoss	1.01	0.20	1.82
LineFlowLossForAngle	0.95	0.28	1.63

Table 5.4: *Prediction samples containing at least a single state variable prediction error greater than the threshold value. The table is valid for the small NN after training epoch 30, and all values are given as percentages.*

Threshold	**MSE**	**CustomLoss**	**SquaredLineFlowLoss**	**LineFlowLossForAngle**
20	0.97	0.43	1.07	0.78
10	2.73	0.98	2.38	2.03
5	6.86	2.02	5.92	5.55
3	13.50	3.84	10.64	16.04

5.2 Medium test network

This subsection displays the results from test where NNs are used to predict the system state of the 33 bus IEEE radial test system. For each different set of hyperparameters used in a training session, a convergence plot, average performance data and worst case performance data is provided.

Table 5.5: *Baseline hyperparameters used to train a NN to solve a load flow for the medium test network with 33 buses.*

Parameter	**Value**
Dataset	slim, 60000 [samples]
Test fraction	0.2
NN architecture	[64, 128, 128, 128, 64]
Batch size	20
Epochs	50
Learning rate	1e-3
NN weight initializer	glorot uniform
input scaling	/2
output scaling	*10

The results presented were generated using both a private laptop and Idun. Unless stated otherwise, all training sessions utilize the hyperparameters from table 5.5.

5.2.1 Baseline performance

This subsection presents the baseline results for medium NNs trained using the hyperparameters presented in table 5.5. Figure 5.3 illustrates the convergence performance during training while tables 5.6 and 5.7 show the average and worst performance.

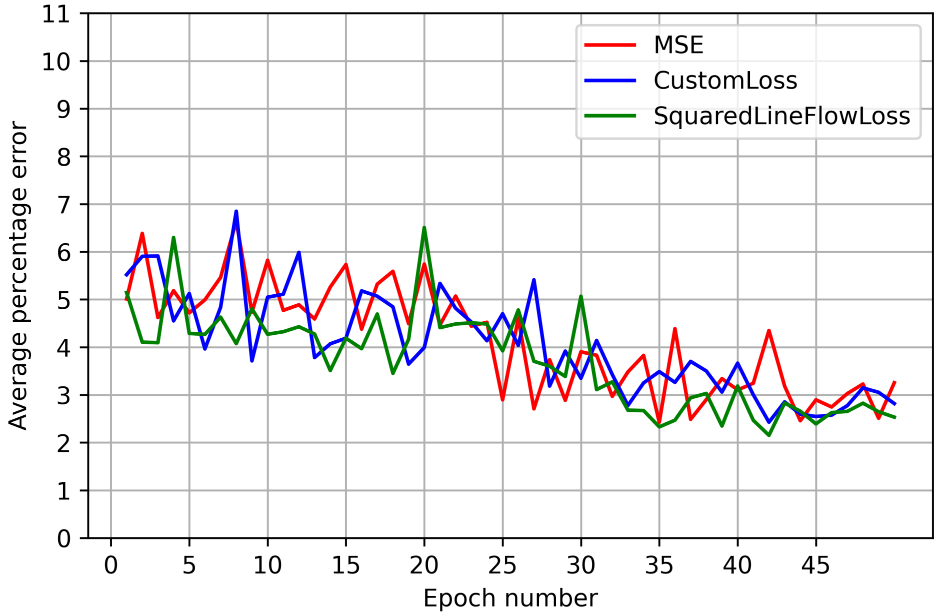

Figure 5.3: *Average error of the medium NN as a function of training epoch for different loss functions. Generated with baseline hyperparameters.*

Table 5.6: *Average performance data for the medium NN trained with different loss functions. Data generated after 50 epochs and with baseline hyperparameters. All values given as percentages.*

	Overall errors	**Magnitude errors**	**Angle errors**
MSE	3.26	0.14	6.37
CustomLoss	2.82	0.078	5.56
SquaredLineFlowLoss	2.53	0.10	4.97

Table 5.7: *Medium prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using baseline hyperparameters after epoch 50. All values given as percentages.*

Threshold	**MSE**	**CustomLoss**	**SquaredLineFlowLoss**
20	100.0	99.9	87.80
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.2.2 Wide load interval performance

Figure 5.4 shows the convergence performance of medium NNs trained using the wide dataset, where the load is varied within the interval [0.8,1.2]. All other hyperparameters are taken from 5.5. Table 5.8 shows the average performance data for the same NNs, and table 5.9 shows the worst results for these models.

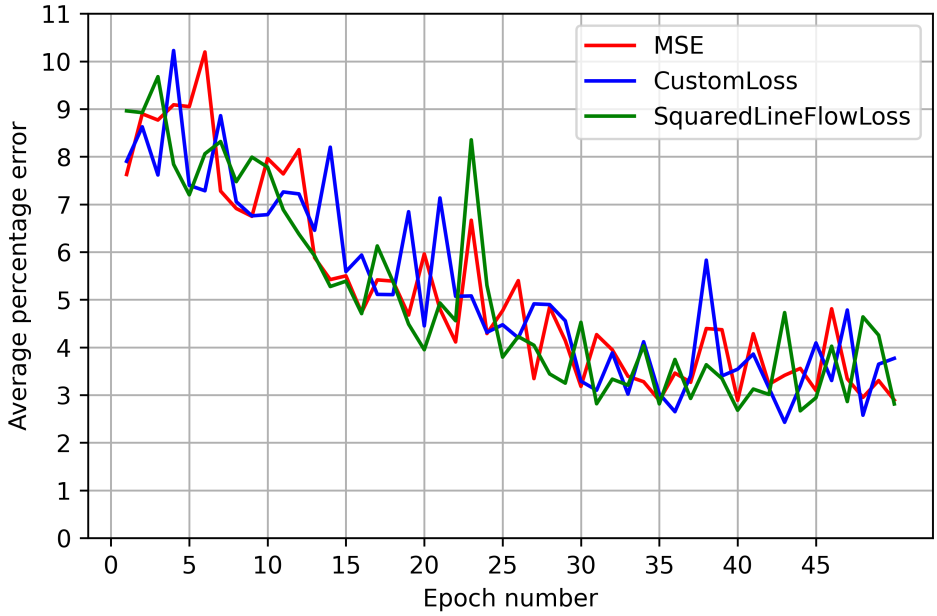

Figure 5.4: *Average error of the medium NN as a function of training epoch for different loss functions. Generated with the wide dataset.*

Table 5.8: *Average performance data for the medium NN trained with different loss functions. Data generated after 50 epochs and with the wide dataset. All values given as percentages.*

	Overall errors	**Magnitude errors**	**Angle errors**
MSE	2.89	0.10	5.69
CustomLoss	3.77	0.04	7.50
SquaredLineFlowLoss	2.81	0.12	5.51

Table 5.9: *Medium prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using the wide dataset after epoch 50. All values given as percentages.*

Threshold	**MSE**	**CustomLoss**	**SquaredLineFlowLoss**
20	98.60	100.0	94.70
10	100.0	100.0	99.70
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.2.3 Reduced learning rate performance

Figure 5.5 shows the convergence performance of a NN trained with a reduced learning rate over 90 epochs, all other hyperparameters taken from table 5.5. Average and worst performance data is provided in tables 5.10 and 5.11.

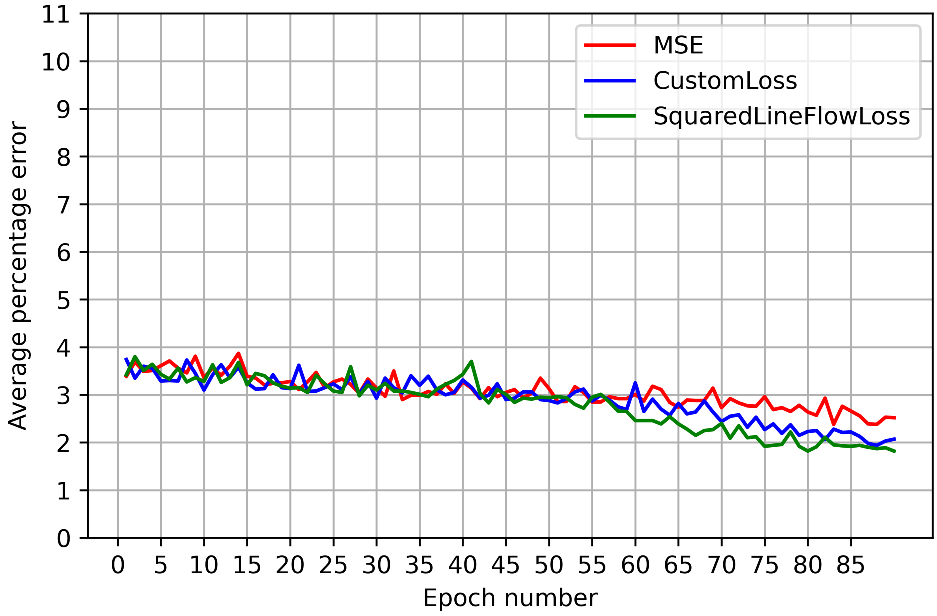

Figure 5.5: *Average error of the medium NN as a function of training epoch for different loss functions. Generated with a learning rate of 1e-4 over 90 epochs.*

Table 5.10: *Average performance data for the medium NN trained with different loss functions. Data generated after 90 epochs and with a learning rate of 1e-4. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	2.52	0.04	4.99
CustomLoss	2.07	0.07	4.06
SquaredLineFlowLoss	1.82	0.04	3.61

Table 5.11: *Medium prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using a learning rate of 1e-4 after epoch 90. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	71.0	65.4	44.8
10	97.3	95.9	90.7
5	100.0	99.9	99.7
3	100.0	100.0	100.0

5.2.4 Greater batch size performance

Figure 5.6 shows the performance of a NN trained with a batch size of 30, all other hyper-parameters the same as in table 5.5. Average and worst case performance is provided in tables 5.12 and 5.13.

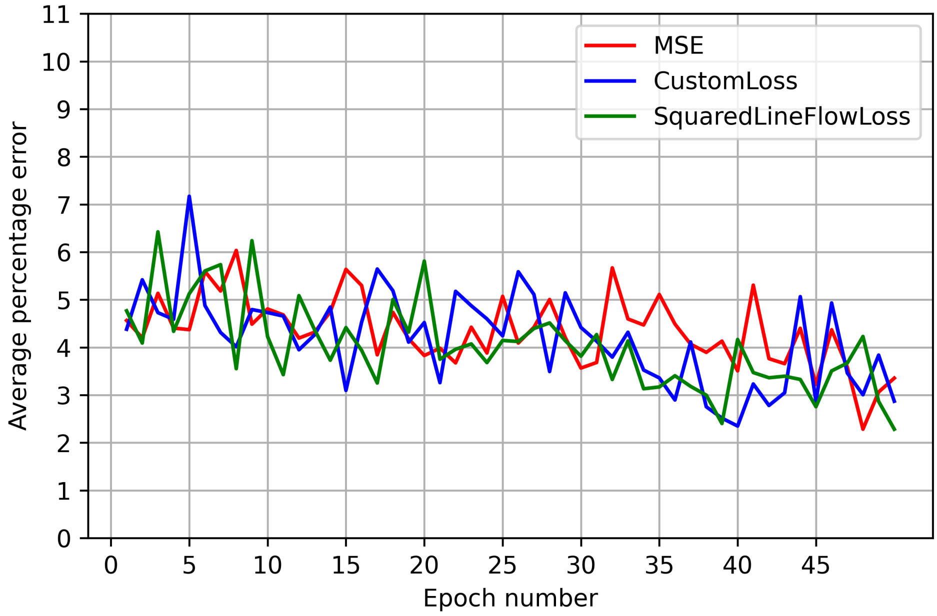

Figure 5.6: *Average error of the medium NN as a function of training epoch for different loss functions. Generated with a batch size of 30.*

Table 5.12: *Average performance data for the medium NN trained with different loss functions. Data generated after 50 epochs and with a batch size of 30. All values given as percentages.*

	Overall errors	**Magnitude errors**	**Angle errors**
MSE	3.36	0.21	6.51
CustomLoss	2.87	0.06	5.68
SquaredLineFlowLoss	2.28	0.05	4.52

Table 5.13: *Medium prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using a batch size of 30 after epoch 50. All values given as percentages.*

Threshold	**MSE**	**CustomLoss**	**SquaredLineFlowLoss**
20	97.4	97.4	77.1
10	100.0	100.0	99.9
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.2.5 Deeper NN performance

Figure 5.7 shows the performance of a NN trained with an additional layer of 128 neurons, all other hyperparameters the same as in table 5.5. Average and worst case performance is provided in tables 5.14 and 5.15.

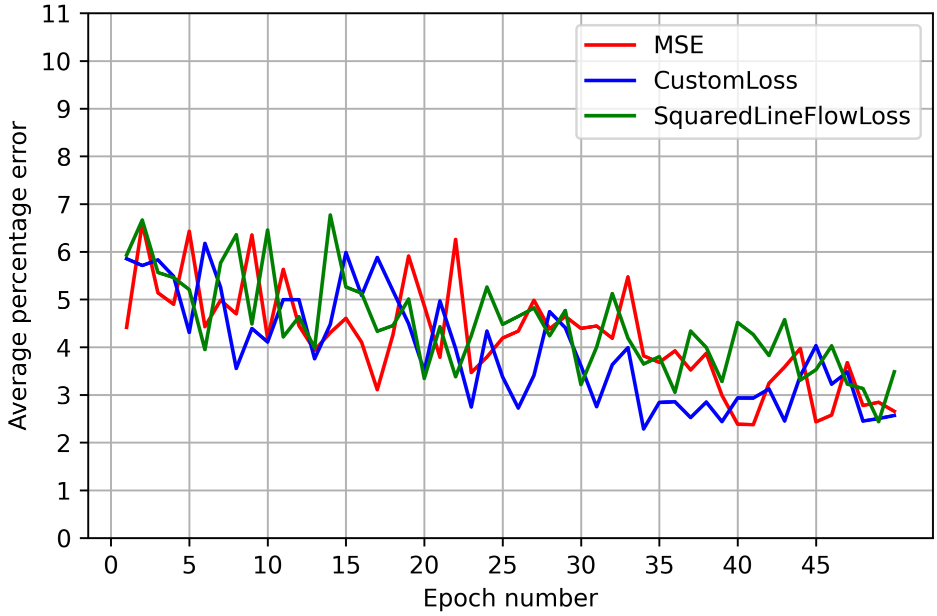

Figure 5.7: *Average error of the medium NN as a function of training epoch for different loss functions. Generated with an additional layer of 128 neurons.*

Table 5.14: *Average performance data for the medium NN trained with different loss functions. Data generated after 50 epochs and with an additional layer of 128 neurons. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	2.66	0.17	5.14
CustomLoss	2.57	0.16	4.98
SquaredLineFlowLoss	3.48	0.15	6.82

Table 5.15: *Medium prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using an additional layer of 128 neurons after epoch 50. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	95.8	78.6	100.0
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.2.6 Combining a low Learning rate with a deeper system architecture

Figure 5.8 shows the convergence performance of a medium NN trained with two additional layers of 128 neurons, and a learning rate of 1e-4. Average and worst performances are given in tables 5.16 and 5.17.

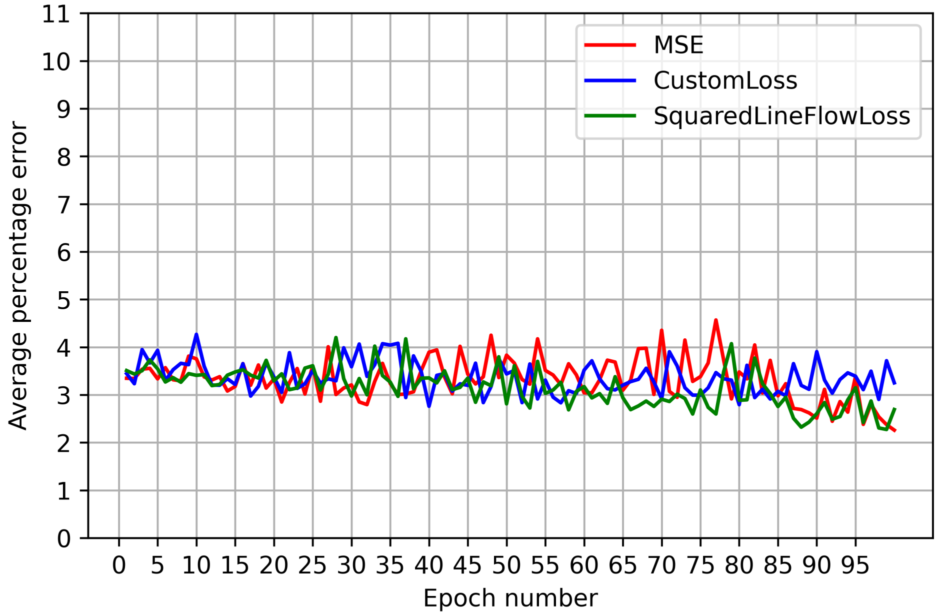

Figure 5.8: *Average error of the medium NN as a function of training epoch for different loss functions. Generated with a learning rate of 1e-4 and two additional layers of 128 neurons.*

Table 5.16: *Average performance data for the medium NN trained with different loss functions. Data generated after 100 epochs with a learning rate of 1e-4 and two additional layers of 128 neurons. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	2.26	0.05	4.47
CustomLoss	3.25	0.07	6.43
SquaredLineFlowLoss	2.69	0.05	5.34

Table 5.17: *Medium prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using a learning rate of 1e-4 and two additional layers of 128 neurons after epoch 100. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	41.5	89.0	90.0
10	94.3	99.5	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.3 Large test network

This subsection gives a summary of the results from training sessions where a NN is used to predict the system state of the 69 bus IEEE radial test system from figure 4.3. The results presented were generated on Idun exclusively, using [39] at commits *8080bea* and outward. Unless stated otherwise, all training sessions utilize the hyperparameters from table 5.18.

Table 5.18: *Baseline hyperparameters used to train a NN to solve a load flow for the large test network with 69 buses.*

Parameter	Value
Dataset	slim, 60000 [samples]
Test fraction	0.2
NN architecture	[136, 272, 272, 272, 136]
Batch size	20
Epochs	150
Learning rate	1e-4
NN weight initializer	glorot uniform
input scaling	/2
output scaling	*10

5.3.1 Baseline performance

Figure 5.9 shows the convergence performance of a large NN trained with baseline hyperparameters. Average and worst performances after the final training epoch are given in tables 5.19 and 5.20.

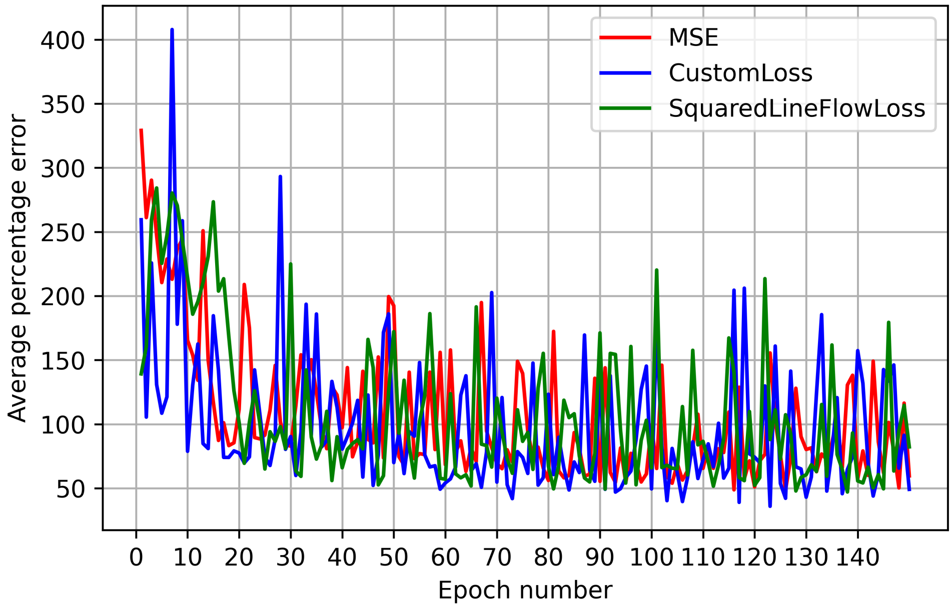

Figure 5.9: *Average error of the large NN as a function of training epoch for different loss functions. Generated with baseline hyperparameters.*

Table 5.19: *Average performance data for the large NN trained with different loss functions. Generated with baseline hyperparameters after 150 epochs. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	59.66	0.03	119.29
CustomLoss	49.13	0.02	98.25
SquaredLineFlowLoss	82.27	0.06	164.48

Table 5.20: *Large prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using baseline hyperparameters after epoch 150. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	100.0	100.0	100.0
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.3.2 Wide load interval performance

Figure 5.10 shows the convergence performance of a large NN trained using the wide dataset. Average and worst performances after the final training epoch are given in tables 5.21 and 5.22.

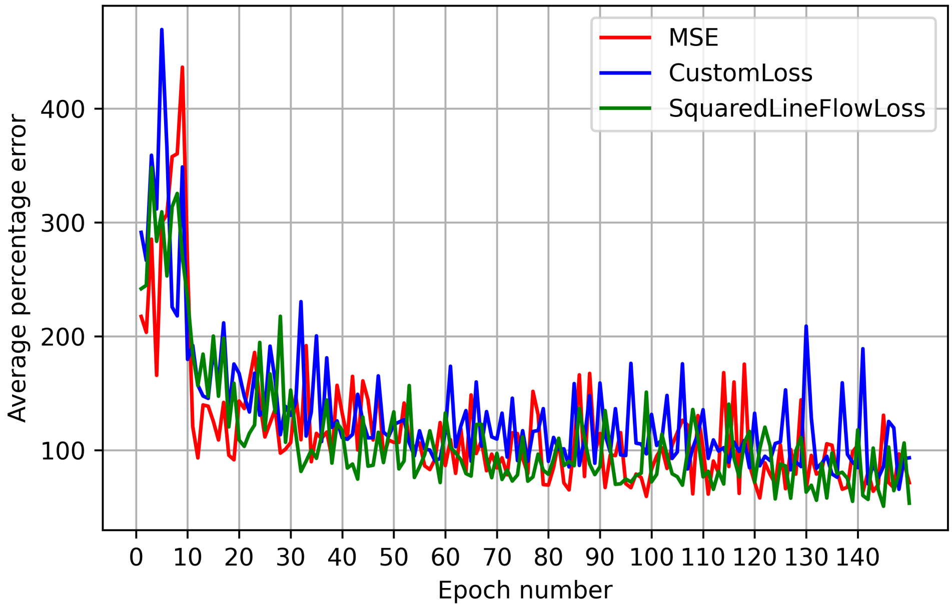

Figure 5.10: *Average error of the large NN as a function of training epoch for different loss functions. Generated with the wide dataset.*

Table 5.21: *Average performance data for the large NN trained with different loss functions. Data generated after 150 epochs and with the wide dataset. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	71.28	0.03	143.53
CustomLoss	93.20	0.04	186.36
SquaredLineFlowLoss	53.35	0.05	106.65

Table 5.22: *Large prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using the wide dataset after epoch 150. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	100.0	100.0	100.0
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.3.3 Reduced learning rate performance

Figure 5.11 shows the convergence performance of a large NN trained with a reduced learning rate of 1e-5 over 200 epochs. Average and worst performances after the final training epoch are given in tables 5.23 and 5.24.

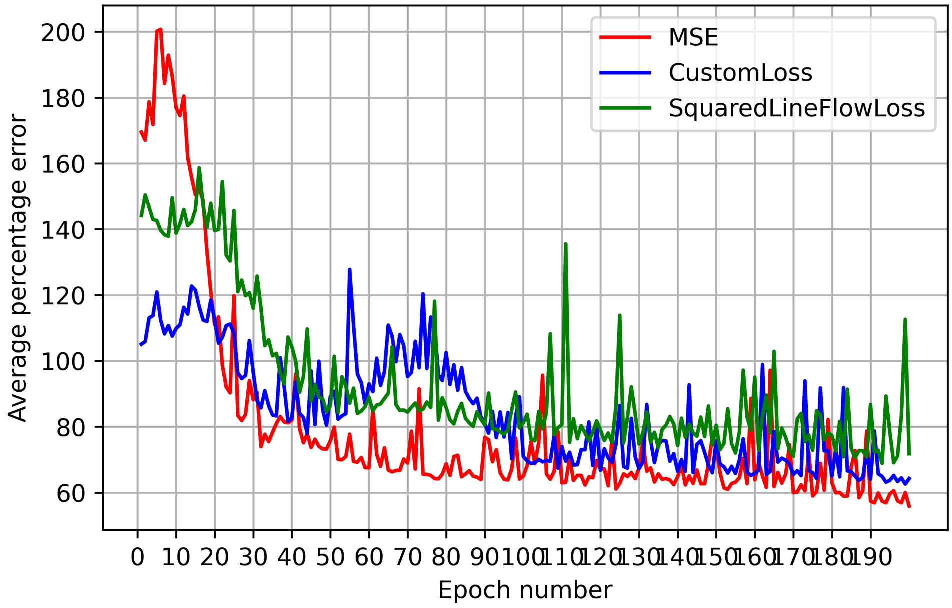

Figure 5.11: *Average error of the large NN as a function of training epoch for different loss functions. Generated with a learning rate of 1e-5.*

Table 5.23: *Average performance data for the large NN trained with different loss functions. Data generated after 200 epochs and with a learning rate of 1e-5. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	55.90	0.04	111.77
CustomLoss	64.26	0.03	128.50
SquaredLineFlowLoss	71.80	0.03	143.56

Table 5.24: *Large prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using a learning rate of 1e-5 after epoch 200. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	100.0	100.0	100.0
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.3.4 Greater batch size performance

Figure 5.12 shows the convergence performance of a large NN trained with a batch size of 30. Average and worst performances after the final training epoch are given in tables 5.25 and 5.26.

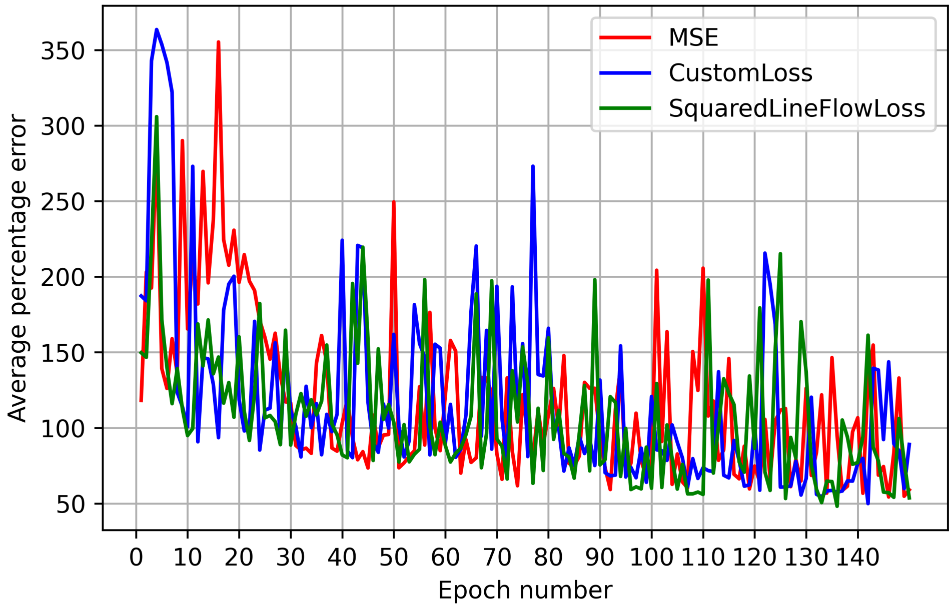

Figure 5.12: *Average error of the large NN as a function of training epoch for different loss functions. Generated with a batch size of 30.*

Table 5.25: *Average performance data for the large NN trained with different loss functions. Data generated after 150 epochs and with a batch size of 30. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	58.99	0.04	117.94
CustomLoss	89.05	0.01	178.09
SquaredLineFlowLoss	53.67	0.11	107.24

Table 5.26: *Large prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using a batch size of 30 after epoch 150. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	100.0	100.0	100.0
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

5.3.5 Deeper NN performance

Figure 5.13 shows the convergence performance of a large NN trained with an additional layer of 272 neurons. Average and worst performances after the final training epoch are given in tables 5.27 and 5.28.

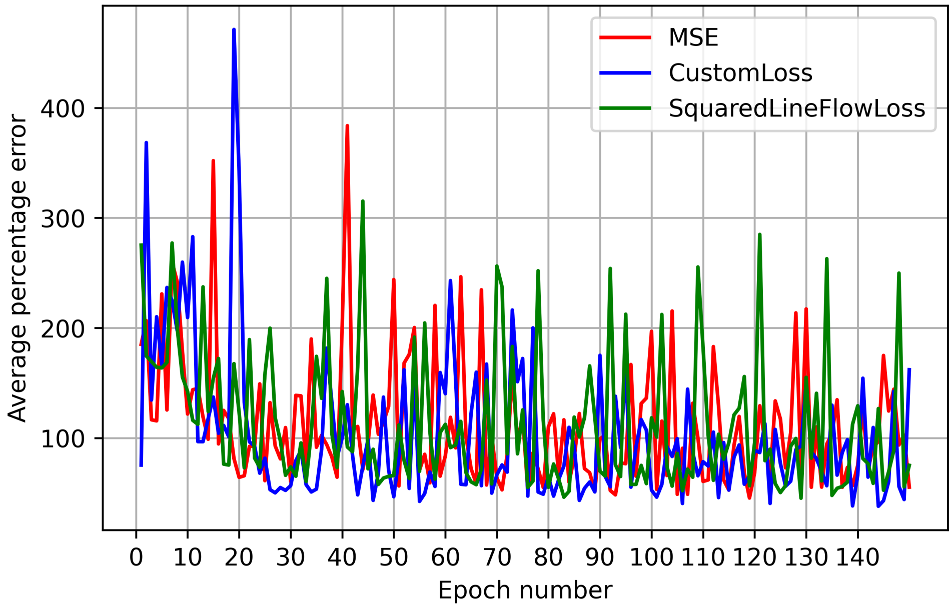

Figure 5.13: *Average error of the large NN as a function of training epoch for different loss functions. Generated with an additional layer of 272 neurons.*

Table 5.27: *Average performance data for the large NN trained with different loss functions. Data generated after 150 epochs and with an additional layer of 272 neurons. All values given as percentages.*

	Overall errors	Magnitude errors	Angle errors
MSE	55.44	0.03	110.85
CustomLoss	162.13	0.06	324.21
SquaredLineFlowLoss	75.28	0.03	150.53

Table 5.28: *Large prediction samples containing at least a single state variable prediction greater than threshold. The table was generated using an additional layer of 272 neurons after epoch 150. All values given as percentages.*

Threshold	MSE	CustomLoss	SquaredLineFlowLoss
20	100.0	100.0	100.0
10	100.0	100.0	100.0
5	100.0	100.0	100.0
3	100.0	100.0	100.0

6 Discussion

This part discusses the results presented in section 5. It starts by commenting important contributions from the literature review in section 3. Then the most important results from the different NN training sessions are discussed starting from the small through the large NNs. For each network size, the discussion starts by evaluating the pure NN performance, ie. results obtained from a training session based upon the MSE error function.

Further, there is a section containing more general comments to the observed NN performance. Section 6.3 describes factors that are particularly to the NN performance presented in this thesis.

6.1 Literature and this work

Section 3 describes a selection of recent efforts to apply NNs to power system analysis. Due to the NNs generalization capability it can be applied in a broad range of different scenarios. The authors of reference [1] propose using a NN based power flow model instead of DC power flow models. Their results are promising because they are more precise than similar numbers from the linear model. Thus, a reasonable area of application would be scenarios where a DC power flow model is currently used but where a greater accuracy is desirable. While the authors do not mention the use of physics informed concepts, this could be a reasonable continuation of their work.

In reference [8], several interesting proposals to improvement of neural network based load flow are provided. Their approach is slightly different than usual as they blend power system theory with NNs in an elegant way. Perhaps inspired by FDLF, where selected sensitivities are set to zero for improved calculation speed, the authors propose altering the training process to achieve faster training.

The authors of [36] note that a simple neural network structure does not give satisfying results for load flow analyses. In their work they propose splitting the NN conducting the load flow into separate networks for angle and magnitude predictions. While this has not been verified in this thesis, the difference between angle and voltage magnitude predictions in the results from this thesis suggest that this could improve NN accuracy.

References [15] and [35] both propose the addition of worst case guarantees to OPF models. An important drawback of using NNs close to critical infrastructure is a lack of control over the data-flow and thus the results. Providing worst case guarantees could therefore reduce the threshold for using these tools in practice.

The two works also demonstrate how the use of collocation points may reduce the dependency on data. As data availability is important for NN training, this is also an important step towards seeing practical implementations of NNs. This work did not utilize collocation point training as the principles used in [15] and [35] were not directly applicable. Consequently the models trained in this work are heavily reliant on synthetic data.

6.2 Neural network performances

This section will be used to discuss general performance observations of NNs. Results are discussed in three main parts, where the two first comment on the average accuracies and worst case performance of networks trained with an MSE loss function. The final part highlights any observed performance differences obtained by training using physics informed loss functions.

6.2.1 Small NN, Accuracy

Figure 5.1 shows both the average and average-worst performance of the NN described in table 5.1. As expected, both the average and worst case prediction accuracy increases as the NN is trained through the epochs.

Both small NNs trained with MSE error function are capable of predicting the system state accurately on average. There is a notable difference in prediction accuracy for voltage magnitudes and angles. This difference can be seen in both tables 5.2 and 5.3. Most of the erroneous predictions occur due to errors in angle predictions.

This difference in prediction accuracy is likely induced by the more complex relationship between power injections and voltage angles. It has also been observed in literature, where several ways to mitigate this problem have been suggested. A simple way to improve angle prediction accuracy could be to predict angles and magnitudes with two separate networks. An other possibility could be to reduce the amount of guidance provided to the voltage magnitudes relative to the voltage angles.

Since the active and reactive power demands were generated independent of each other, highly unrealistic operating scenarios may have been used for NN model training. Consequently, this work should be considered a demonstration of what can be achieved. Other models would have to be generated for use in other cases.

Furthermore, it is worth noting that branch flow accuracy has not been evaluated by this approach. Branch flows are dependent on complex nodal voltages, and will therefore be affected by inaccuracies in nodal voltages. As branch flows are of great importance for the power system this should be considered a major disadvantage of this approach.

6.2.2 Small NN, Worst case performance

After training the initial small NN over 150 epochs, there was a significant difference between average accuracy and worst case accuracy. The greatest average and worst deviation is found for δ_2, the two values were 1.11% and 1315.8% respectively. This shows that though average performance gives promising results, the models worst case performance is dissatisfactory.

Enforcing more strict control over hyperparameters and using another set of metrics, evaluating how many state variable predictions in a single set of predictions are worse than a given threshold provides another perspective. Table 5.4 shows that it is possible to train a small NN capable of predicting the system state of a small 4 bus network with little worst case errors at a threshold of 10%. Lowering the threshold worsens the perceived worst case performance. At a threshold of 3%, 13.5% of all prediction samples contain an error.

Comparing table 5.3, with table 5.4 gives an idea of what causes the reduced worst case performance. It is known that the relationship between bus injections and bus angles is more complex than the same relationship to voltage magnitude. This is also illustrated by the fact that voltage angles on average contain significantly more prediction errors. Therefore it reasonable to assume that the poor worst case accuracy to a large extent is caused by the complex relationship between power injection and angle.

Wether the worst case performance is acceptable depends entirely on the usecase of the NN. If it were to be used for power flow calculations, where a high accuracy of angles is required, this model is likely not applicable without a significant improvement in angle prediction accuracy.

6.2.3 Small PINN performance

Subsection 5.1.2 describes the performance of the small neural network in predicting the state of the small test network. From table 5.4 it is is clear that the amount of affected sets increases significantly as the threshold value decreases. In all cases, except for the network trained with the CustomLoss function, more than 10% of the predicted sets include a prediction error. The network trained using the CustomLoss error function shows the best worst performance, where only 3.84% of the predicted sets contain an error at threshold 3%.

Studying table 5.3 it is possible to see that a small NN trained with CustomLoss shows overall better performance. This is also shown in figure 5.2 where CustomLoss shows a slightly improved convergence behavior. Thus it is reasonable to state that CustomLoss shows the best performance for the small NN.

6.2.4 Medium NN, accuracy

Studying figures 5.3 to 5.8 it is obvious that the networks do not converge properly, regardless of the error function used during training. Tables 5.6 to 5.17 show the average and worst performance of difference NNs. Studying these tables, illustrate that medium NN performance is worse than the small NN.

Average performances of the medium NNs trained with the MSE loss function are typically around 3% overall error, lower than 0.2% for voltage magnitude, and between 3-6% error for voltage angle predictions. Neglecting worst case performance, these results look promising.

In a similar way as for the small NN the voltage angle predictions are less accurate than the voltage magnitude predictions. Moreover, this difference between is more pronounced for the medium NN. A possible explanation for this is that there exists a much broader variety of operating cases for the medium network compared to the small. Thus, since the network architecture for is simply scaled up from the small NN, network performance is worsened.

The best average results are produced by a MSE trained NN, as shown in figure 5.8 and its corresponding tables 5.16 and 5.17. Though the results are better than the remaining results from the medium NN, it is possible that it is a coincidence. Since the NNs are initialized randomly, the convergence rate may behave differently due to different starting points. Furthermore, by studying the convergence plot more closely, it seems that the MSE trained NN seems to suddenly converge faster towards the end. This can likely be characterized a (fortunate) stochastic event.

The closest candidate is the MSE trained NN with a reduced learning rate as given in figure 5.5, together with tables 5.10 and 5.11. Compared the the previous noisy convergence, this model converges without too great spikes. This means that its results are less likely to be a mere coincidence, and should be possible to replicate more easily. Finally it is worth noting that this models worst case performance is better than most other medium MSE trained NNs.

6.2.5 Medium NN, Worst case performance

Worst case performances of the medium test networks show significantly lower accuracy than the small NN. Virtually none of the medium NNs are capable of predicting all network outputs with a smaller error than 10%. This means that none of the MSE trained medium NN models are capable of predicting the system states without any one state prediction being less than 10% off.

Keeping in mind that there is a pronounced difference in magnitude and angle prediction accuracy, it is likely that many of the erroneous outputs are angles. Still, since most of the trained models show a poorer worst case performance than this, it is not reasonable to recommend this approach state estimation in power systems where angles are important.

6.2.6 Medium PINN performance

Comparing results from the medium NNs trained using the MSE loss with NNs trained with CustomLoss and its derivatives, there does not seem to be any significant convergence performance gains. Still, in 4 out of 6 hyperparameter variations the custom loss function outperformed pure NNs trained with an MSE loss function.

Studying the differences between the worst case performance results of the MSE trained NNs to the custom loss trained NNs, their performance is very similar. The best worst-case performance occurred for all loss functions simultaneously for the model trained using a low learning rate. This is shown in table 5.11 describing the worst case performance. In this particular case, the NNs are capable of predicting up to 45% of all state variables within an error of 20% or better.

It is difficult to argue that the PINNs trained for the medium network present any significant performance benefits. On the one hand, they often show better performance than their MSE trained counterparts. On the other hand, the performance is only marginally better, and the convergence patterns appear to be similar. Due to the inferior worst case performance of all medium NNs, none should be used for live power system analysis.

6.2.7 Large NN, accuracy

All results from training large NNs with various hyperparameter changes are show in figures 5.9 through 5.13 in addition to tables 5.19 to 5.28. Studying the convergence of the different NNs, none of them converge properly. The most stable convergence is found for the NN trained with the MSE error function and a reduced learning rate (fig 5.11). However, even during the training of this NN, significant spikes are present during training. These spikes occur when a training epoch significantly worsens or improves the average performance of the model.

The average overall prediction errors of the large NN range between 50% and 162%. Consequently it is obvious that none of these models are capable of making an accurate state prediction of the distribution grid. In a similar fashion as for the medium NN, the average prediction accuracies are worse for angle predictions compared to magnitude predictions.

However, it is worth noting that the difference between magnitude and angle prediction accuracy is much more pronounced for the larger network. While voltage prediction accuracies are satisfactory, deviating up to 0.11% on average, voltage angles frequently deviate more than 100%. This suggests that most prediction errors are caused by the angle predictions.

6.2.8 Large NN, Worst case performance

None of the NNs trained in this thesis are capable of predicting the large distribution grid state without including at least one state variable prediction error greater than 20%. Consequently, none of the developed NNs should be used for practical power system analysis as they are. However, as discussed later, there is significant improvement potential.

Worst case performances are gradually deteriorating with NN size. This can be found by comparing the worst case performances of the small and medium NNs to the worst case performances of the large NNs. While worst case performances of the small NN were within in an acceptable range, the worst case performances of the medium test network were significantly worse, and finally the worst case performances of the large NNs where even worse than the medium test network.

A possible explanation for the deteriorating performance could be that there exists a greater variety of operating scenarios for larger power grids. As a consequence, the NN needs to generalize this broader range of operating scenarios, which in turn can lead to worse performance. Though the approach to NN design used in this thesis did not yield models directly applicable for live power system analysis, its contents identify important areas that can be exploited for significant performance gains.

6.2.9 Large PINN performance

While some of the large NNs show a performance improvement when training with a CustomLoss loss function, this is not the case for all combinations of hyperparameters.

There does not seem to be any clear trend that the custom loss functions yield more accurate results.

Despite this, there are some interesting observations to be made. For the baseline NN trained using the wide dataset, the SquaredLineFlowLoss function yields the most accurate results. This is interesting because the wide dataset (fig. 5.10) should contain a greater variety of operating points, and therefore more complex patterns that need to be learned by the NN.

Furthermore, the MSE trained NN with a lower learning rate (fig. 5.11) also shows a promising convergence. A possible explanation for the good convergence could be that the loss value returned by the loss function has an better magnitude than for the other two loss functions. Training a NN with a relatively large loss value would cause larger changes in system weights in each batch. Still, as the figure only shows the average performance per epoch and not per processed batch, it is difficult to justify an absolute conclusion based on the plot.

Finally, it seems that only one of of the large NNs is capable of converging (fig 5.11). All others (figs. 5.9- 5.13) show fluctuations of around 100% or more. It is hard to provide an exact reason for the poor convergence performance, but it is likely that it is related to the increasing complexity and diversity brought by the inclusion of more buses to the system.

6.2.10 Comparing medium and large NN performance

Table 6.1 was compiled to compare how the PINNs performed relative to the NNs for the medium and large test networks. It shows the relative ranks of the NNs trained with *MSE*, *CustomLoss* or *SquaredLineFlowLoss* function for the medium and large test networks (section 5.2 and 5.3). Points are awarded by comparing the relative performances of the different loss functions. Consequently, a lower number indicates superior performance. Note that one more set of hyperparameters has been tested for the medium test network, and that the final results are therefore slightly biased towards the medium NN performances.

Table 6.1: *Table showing the relative performance ranks of the MSE and CustomLoss loss function. Greater numbers indicate worse performance.*

	MSE	CustomLoss	SquaredLineFlowLoss
Medium	15	12	9
Large	8	12	10
Overall	23	24	19

There are performance differences between the loss functions. For the medium test network, the *SquaredLineFlowLoss* loss function performs significantly better than its competitors. For the large NNs, the MSE loss function shows the best performance. Overall, *SquaredLineFlowLoss* shows the best performance, mostly due to its superiority for the medium NN. It is important to acknowledge that this table does not take into account how much better each loss function is. Nevertheless, it shows that implementing a physics informed loss function can give performance improvements.

The CustomLoss loss function and its derivatives require a much longer training time because they are incompatible with Tensorflow graph execution. Consequently, there are few reasons to prioritize the use of the developed custom loss functions. The saved training time could instead be used for parameter tuning to improve model performance.

A common feature seems to be that the models trained with a reduced learning rate display better convergence performances, shown in figures 5.5 and 5.11. Assuming that this is the case, it indicates that most models are trained with parameters making the NNs unstable. This can be seen from virtually all convergence plots presented in section 5. The observed instability in the different NN convergences can have many different explanations. It is possible that the difference between input and output scaling contributes to the observable differences. This proposition is described in closer detail by section 6.4.2.

6.3 General comments to NN performance

This section contains discussions on the limiting similarity between NN and grid topologies, how a limited background knowledge about NN affects this work and how important hyperparameter selection is.

6.3.1 Network topology and neural network architecture

A possible explanation for the relatively weak NN performance on the larger radial test networks could be the limited coherence between the NN architecture and the power network topology. In the NN all nodes are interconnected, passing a signal to all neurons in the downstream layer. However, since the medium test network is radial most power system buses are not connected to more than two other buses. Moreover, the power flows outward through the radial branches, meaning that the voltage magnitudes are decreasing and the power angles are increasing when moving out in the network.

Since the NN passes signals from one node to all other nodes, it is possible to argue that it passes information between buses that have no physical connection. As a consequence the signal processing cannot be assigned any physical meaning. To obtain a signal processing more coherent with the underlying physical principles a substantial amount of network weights would have to be adjusted to blocks signals between nodes that are not connected.

There are probably multiple ways to force the signal processing to become more similar to the physical power flow. One way would be initializing and fixing system weights in such a way that information can only flow between nodes with connections. Doing this, no signals would be passed between nodes without connections, but the network would have to be made as deep as the amount of network buses. Obvious disadvantages of this approach are that it would give a large NN fixed to a single topology.

Differences in magnitude and angle prediction accuracies have also been discussed in literature. In [36], separating the NN into two parts one predicting voltage angles and one predicting voltage magnitudes is suggested. An advantage to this approach include the possibility to utilize different neural network depths for the two networks. Moreover, it iw worth noting that this approach does not eliminate the possibility of using a physics informed architecture.

Assuming this approach yields accurate results with a satisfactory worst case performance, it is not certain that it can be applied to the evolving power grid. In radial grid operation scenarios, there exist backup feeders which are used in the case of a grid fault. Because neural networks are trained for specific operating scenarios, NN architecture would likely have to be modular and pre-trained for all possible operating scenarios.

However, this is without considering bilateral flows within the power grid. Reversing power flows would introduce more variables to the power system and possibly disturb the normally decreasing voltage magnitude trend in the branches of a radial power system.

6.3.2 Dataset width and NN performance

For the medium NN, results show that the wide load interval (section 5.2.2 has slightly better average performance than the baseline parameters (section 5.2.2). Moreover, the convergence of the model trained on the wide dataset appears slightly more pronounced than the baseline model.

For the large NN, average results are unsatisfactory for both the baseline (section 5.3.1) and for the wide model (section 5.3.2). Though final performance is similar, the baseline model displays marginally better average performance. Still, the performance convergence appears slightly better for the wide model.

These results are counterintuitive, because it is expected that a wider load variance would yield more complex patterns to be learned by the NN. As a consequence, worse performance is expected. On the other hand, heavier system loading may provide more pronounced patterns that impact learning positively. Either way, since the performance of the two models is comparable there is not enough evidence to state the impact dataset width has on NN performance.

6.3.3 Starting from scratch

When designing NNs to perform a calculation, there are many parameters that need to be selected. During this work, an important limitation has been experience with neural networks. Consequently, much of the different training sessions were conducted as experiments where hyperparameters were selected more or less randomly in an attempt to improve NN performance.

Since an important part of this work was to compare NN and PINN performance, a physics informed loss function was implemented early. This slowed down training time significantly, which in turn left less time for hyperparameter tuning. Starting over, it would likely have been beneficial to keep training time low to allow for more efficient crude hyperparameter tuning. In practice this implies avoiding Tensorflow eager execution and to keep batch sizes larger. In the repository containing the python code for this thesis, eager execution is disabled for any simulation using the standard MSE loss function

6.3.4 The importance of hyperparameter selection

According to literature, hyperparameter selection is of great importance to neural network performance. This includes changing for instance the input and output scaling of the NN, utilizing different loss functions or changing NN architecture. The different NNs trained in this thesis respond differently to changing hyperparameters.

For the medium NNs, best performance was obtained for a lower learning rate combined with a deeper system architecture trained over 100 epochs using MSE loss (fig. 5.8. This model was trained over 50 epochs more than the remaining models to compensate for the reduced learning rate. Keep in mind that this particular model showed a sudden improvement in accuracy over the last 20 epochs, down from an accuracy similar to that of the other models at epoch 50. Thus, it is possible that the good performance was a coincidence caused by the additional training epochs.

For the large network, the best model performance was observed for the baseline model trained using CustomLoss (fig. 5.9). None of the changes made to the hyperparameters after training this model gave any performance improvements.

Finally, it is wort noting that the two (medium and large) models demonstrating the best performance did not utilize the same same hyperparameters. This shows that there does not necessarily exist one set of hyperparameters which will guarantee good performance for several neural networks. Consequently, for all networks trained in this work, more time should have been spent tuning the hyperparameters to improve final model performance.

6.4 Limitations and suggested improvements

NN performance is dependent on a wide variety of factors. During the work on this thesis, efforts have been made to evaluate their individual impact on NN performance. Still, it is not possible to strictly control every aspect governing NN performance. This section describes several factors that can deteriorate NN performance.

All metrics used in this thesis evaluate the performance of different neural networks trained on a particular dataset. Consequently no guarantees are made that results can be replicated using other test networks and NN models trained with different datasets. For commercial use, and for applications close to critical infrastructure it is obviously important that NN outputs are accurate. Therefore, providing guarantees that NN outputs are within a specific interval is an important factor for applications that require reliable high accuracy. It was much for this reason that papers [15] and [35] were included in this thesis.

6.4.1 Choice of performance metrics

Performance metrics are essential to NN performance evaluation. The analysis of results in this thesis rely heavily on performance metrics developed early in the work on this thesis. In hindsight there is some improvement potential, especially concerning the threshold specific metrics.

Considering the increasing difference in prediction accuracies between voltage magnitudes and voltage angles, it would be interesting to know which type of prediction causes most worst case violations. Based on the observed performance difference, it is reasonable to assume that most prediction errors were caused by angle predictions.

6.4.2 Data processing

Dataset generation and hyperparameter selection for NN training were both conducted early while working on this thesis. Lacking background on this topic, input and output scaling factors were set based upon intuition. Section 2.1.5 explains that data scaling is of great importance for NN performance. Consequently it is possible that the selected input and output scaling factors have reduced NN performance. Since the same input and output scaling has been used for all NN training sessions, this will have affected all results in this thesis.

As the per unit system was used to calculate the system state the typical input values are smaller than two, consequently scaling inputs by a factor of two brings inputs to the NN down to values closer to the interval [0,1]. Voltage magnitude outputs of distribution grids are typically very close to 1 in magnitude, while the voltage angles are of a different magnitude. Thus, using an output scaling factor of 10 makes NN inputs smaller by a significant amount, compared to NN outputs.

To accommodate the difference in magnitude between inputs and outputs, the neural network weights must deviate further from unity. It is possible that this effect is magnified by the difference in magnitude between voltage magnitudes and angles. Either way, this can have a negative impact on performance. It may give a situation where network weights are of different orders of magnitude, inducing instabilities during training. For this reason, it would be interesting to study neural network performance using a more sophisticated data scaling scheme, where both all inputs and outputs are kept within a similar magnitude range.

6.4.3 Computational efficiency

Much of the code implemented to train the NNs in this work is inefficient. There is likely a large improvement potential both when considering computational resources and memory usage. NN training using the implemented custom loss functions takes a long time because the loss functions are incompatible with graph execution mode. Adapting the loss functions to a format that allows graph execution would drastically improve training speed. Consequently, more time can be spent tuning parameters to improve NN performance.

Especially the current implementation of evaluation metrics require a substantial amount of memory. The current approach generates one *NeuralNetwork* object per epoch to evaluate. As each *NeuralNetwork* is a substantial class with memory intensive attributes, optimizing evaluation metric computation could make the model compatible with cheaper hardware as well as save time.

6.4.4 A changing power grid

All training sets used in this thesis are based on a simple radial neural network structure where all load nodes have a net negative power injection. As the share of renewables and deployment of DERs is increasing, the power injections at the load nodes will become subject to greater load variation. In extreme scenarios, bilateral power flows may also occur. These changes can affect NN performance in a negative way if the networks are not re-trained using a different dataset.

As future power grids more frequently operate closer to their limits, it is possible that topological reconfiguration will be used more frequently to handle system overloading. Because NNs are trained on a specific dataset, in this case where the topology is fixed, they will not be capable of predicting results accurately for varying topologies.

6.4.5 Collocation point training

Collocation point training can reduce a NN models dependence on data while maintaining NN accuracy. It has not been used in this thesis, because examples in literature utilize the KKT conditions for implementations within OPF studies. The KKT conditions cannot be transferred directly to regular power flow studies.

A possible way to implement this for load-only radial distribution grids could be to penalize all predictions where a parent node with a smaller voltage magnitude or greater angle than its child. By doing this, it should be possible to pre-train a NN to predict a declining voltage magnitude / increasing voltage angle throughout a radial power grid.

6.4.6 Multiple ways to implement a PINN

As described in section 2.2, there are multiple ways to implement a PINN. In this thesis, only a single type of physics informed loss function has been evaluated. Based on this work, it is not possible to exclude that other implementations of physics informed loss functions can improve neural network accuracy.

Furthermore, a physics informed loss function is only one way to implement a PINN. It is possible that other PINN approaches can improve these results. Following the discussion in 6.3.1, it is likely that careful design of NN topology may improve NN performance.

All NNs in this thesis have used the same weight initializer. Utilizing a different initialization strategy could (1) provide a better starting point for NN training, and (2) impact how the neurons communicate within the NN. Suggestion 2 is somewhat similar to changing NN topology, but would also allow for more complex NN structures. For instance, setting and fixing some weights could block all communication between magnitude and angle predicting neurons.

Finally, it would also be possible to force information to flow unchanged through some layers. This would allow the design of a NN with different depths for the magnitude and angle prediction parts, while keeping trainable variables at a minimum.

6.4.7 Reference loss function

All physics informed loss functions have been compared to the MSE loss function which returns a single loss value. This implies that all weights will be nudged one step in the right direction, based on the magnitude of the MSE error. *SquaredLineFlowLoss* returns more than a single loss value, which allows for more precise adjustment of weights. It is possible that its performance advantage for the medium NN is caused partly by the more detailed loss return.

7 Conclusions and further work

This thesis investigates the performance of and possible applications of neural networks, while using radial load flow calculations as a case. To assess this, a brief literature review has been conducted, and a python library has been developed. The python library was used to train and evaluate different neural networks, while varying hyperparameters governing the training session. Both should be considered important contributions by this thesis.

This work concludes that that neural networks (NNs) are capable of predicting the state of a small test network accurately with good worst case performance. Similar NNs predicting the state of larger radial distribution grids show significantly worse performance. While average voltage magnitude prediction errors remain below 0.2%, voltage angle prediction errors are frequently around 6% for the medium NN and up to 320% on average for the large NN. Further, average large NN errors remained ranging between 50-70%, with one outlier at 160%. Consequently, the developed models should not be considered candidates for live power system applications as they are.

Voltage magnitude predictions were found to be significantly more accurate than angle predictions, independently of network size. Because of this, it is reasonable to conclude that it is the angle prediction inaccuracies that deteriorate performance as network size increases.

The trained NNs were responsive to changes in hyperparameters, yet the best performance was not found for the same set of parameters. In the process of changing different hyperparameters to optimize NN performance, the learning rate of the NN turned out to be an important parameter. Both for the medium and the large NNs, the training sessions utilizing lower learning rates showed better convergence performances.

In this work, additional physics informed loss functions allocating average adjacent line flows to NN outputs have been developed. Applying these loss functions during neural network training yields marginally better performance compared to the non-physics informed Mean Squared Error baseline.

A compelling explanation for the relatively weak NN performance with increasing network size is the lacking similarity between network power flows and data-flow in the neural network. As a radial power grid contains many child nodes, where information from nodes without direct connections could be considered a disturbance. All neurons pass information to all downstream neurons, as is the case for all NNs in this thesis.

Improved data processing is expected to enhance NN performances. All networks in this work were trained for a load only scenario. This is insufficient for the future power grid. Utilizing a broader dataset and combining multiple physics informed approaches could increase model robustness.

7.1 Further work

Though there are multiple ways PINNs can be implemented, only one approach has been tested in this work. Other potential implementations should be tested before discarding

NN based load flow as a viable option for practical power system analysis. Relevant implementations include physics informed architecture, separating voltage angle from magnitude predictions or to force NN data flow closer to the actual radial power flow. Furthermore, it is likely that other physics informed loss functions also can improve performance.

The possibilities for collocation point training for regular load flow should be evaluated. Due to the limited amount of data available in practical implementations, this should improve NN performance in real environments. For radial power flows, it could be possible to use collocation training to force NNs to recognize that voltages are reduced further out in the power grid.

The angle prediction accuracy in the medium NN and convergence performance for the larger NNs was not satisfactory. This is likely caused by the angle prediction accuracy. Therefore, improving angle prediction accuracies in NN based load flow should be subject to future research.

It is possible that the approach to pre- and post processing of data had a negative impact on final thesis results. Further work should examine the impact data processing has on neural network performance.

Finally, to ensure that neural network based power flow can be used in the future power grid, it is important to develop a neural network based approach that allows for changing network topologies and bilateral power flows.

References

[1] Thuan Pham and Xingpeng Li. “Neural Network-based Power Flow Model”. In: *2022 IEEE Green Technologies Conference (GreenTech)*. 2022, pp. 105–109. DOI: 10.1109/GreenTech52845.2022.9772026.

[2] Lizhi Liao et al. “An Empirical Study of the Impact of Hyperparameter Tuning and Model Optimization on the Performance Properties of Deep Neural Networks”. eng. In: *ACM transactions on software engineering and methodology* 31.3 (2022), pp. 1–40. ISSN: 1049-331X. DOI: 10.1145/3506695.

[3] Bin Huang and Jianhui Wang. “Applications of Physics-Informed Neural Networks in Power Systems - A Review”. In: *IEEE Transactions on Power Systems* 38.1 (2023), pp. 572–588. DOI: 10.1109/TPWRS.2022.3162473.

[4] Clemens Martin Müller. *Optimal network reconfiguration*. Project report in TET4510. Department of electric energy, NTNU – Norwegian University of Science and Technology, 2022.

[5] Stephen Marsland. *Machine learning : an algorithmic perspective*. eng. Chapman & Hall/CRC machine learning & pattern recognition series. Boca Raton, Fla: Chapman & Hall/CRC, 2009. ISBN: 9781420067187.

[6] Christopher M Bishop. *Neural networks for pattern recognition*. eng. Oxford: Oxford University Press, 1995. ISBN: 0198538499.

[7] Xiaoheng Jiang et al. “Deep neural networks with Elastic Rectified Linear Units for object recognition”. eng. In: *Neurocomputing (Amsterdam)* 275 (2018), pp. 1132–1139. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2017.09.056.

[8] Yan Yang et al. “Fast Calculation of Probabilistic Power Flow: A Model-Based Deep Learning Approach”. In: *IEEE Transactions on Smart Grid* 11.3 (2020), pp. 2235–2244. DOI: 10.1109/TSG.2019.2950115.

[9] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feed-forward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[10] R. Aggarwal and Yonghua Song. “Artificial neural networks in power systems. III. Examples of applications in power systems”. In: *Power Engineering Journal* 12.6 (1998), pp. 279–287. DOI: 10.1049/pe:19980609.

[11] Rezaul Karim. *TensorFlow: Predict Valuable Insights of Your Data with TensorFlow*. eng. Birmingham: Packt Publishing, Limited, 2018. ISBN: 1789136911.

[12] The TensorFlow Authors. *TensorFlow docs - Introduction to graphs and tf.function*. 2020. URL: https://github.com/tensorflow/docs/blob/ba879887790ded183c04540d5386df4962f48eeb/site/en/guide/intro_to_graphs.ipynb (visited on 10/05/2023).

[13] M. Raissi, P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. eng. In: *Journal of computational physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.10.045.

[14] George S. Misyris, Andreas Venzke, and Spyros Chatzivasileiadis. “Physics-Informed Neural Networks for Power Systems”. In: *2020 IEEE Power & Energy Society General Meeting (PESGM)*. 2020, pp. 1–5. DOI: 10.1109/PESGM41954.2020.9282004.

[15] Rahul Nellikkath and Spyros Chatzivasileiadis. “Physics-Informed Neural Networks for Minimising Worst-Case Violations in DC Optimal Power Flow”. In: *2021 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. 2021, pp. 419–424. DOI: 10.1109/SmartGridComm51999.2021.9632308.

[16] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. “Extreme learning machine: Theory and applications”. eng. In: *Neurocomputing (Amsterdam)* 70.1 (2006), pp. 489–501. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2005.12.126.

[17] Xingyu Lei et al. “Data-Driven Optimal Power Flow: A Physics-Informed Machine Learning Approach”. In: *IEEE Transactions on Power Systems* 36.1 (2021), pp. 346–354. ISSN: 1558-0679. DOI: 10.1109/TPWRS.2020.3001919.

[18] Magnus Själander et al. *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. 2019. arXiv: 1912.05848 [cs.DC].

[19] Clemens Martin Müller and Sigrid Eliassen Sand. “A study of microgrids in Norway”. Bachelor’s thesis. NTNU, 2021. URL: https://hdl.handle.net/11250/2779960.

[20] Toshihisa Funabashi. “Chapter 1 - Introduction”. In: *Integration of Distributed Energy Resources in Power Systems*. Ed. by Toshihisa Funabashi. Academic Press, 2016, pp. 1–14. ISBN: 978-0-12-803212-1. DOI: https://doi.org/10.1016/B978-0-12-803212-1.00001-5.

[21] R. James Ranjith Kumar and Amit Jain. “Load flow methods in distribution systems with dispersed generations: A brief review”. In: *2015 1st Conference on Power, Dielectric and Energy Management at NERIST (ICPDEN)*. 2015, pp. 1–4. DOI: 10.1109/ICPDEN.2015.7084498.

[22] J. Duncan Glover. *Power system analysis & design*. eng. Sixth edition. Boston, Mass: Cengage Learning, 2017. ISBN: 9781305636187.

[23] Olav Fosso. “PyDSAL - Python Distribution System Analysis Library”. In: Sept. 2020. DOI: 10.1109/POWERCON48463.2020.9230554.

[24] Deepjyoti Deka, Vassilis Kekatos, and Guido Cavraro. *Learning Distribution Grid Topologies: A Tutorial*. June 2022. DOI: 10.48550/arXiv.2206.10837.

[25] Zhifang Yang et al. “A General Formulation of Linear Power Flow Models: Basic Theory and Error Analysis”. In: *IEEE Transactions on Power Systems* 34.2 (2019), pp. 1315–1324. DOI: 10.1109/TPWRS.2018.2871182.

[26] E Janecek and D Georgiev. “Probabilistic Extension of the Backward/Forward Load Flow Analysis Method”. eng. In: *IEEE transactions on power systems* 27.2 (2012), pp. 695–704. ISSN: 0885-8950. DOI: 10.1109/TPWRS.2011.2170443.

[27] M.H. Haque. “Load flow solution of distribution systems with voltage dependent load models”. In: *Electric Power Systems Research* 36.3 (1996), pp. 151–156. ISSN: 0378-7796. DOI: https://doi.org/10.1016/0378-7796(95)01025-4.

[28] R. James Ranjith Kumar and Amit Jain. “Load flow methods in distribution systems with dispersed generations: A brief review”. In: *2015 1st Conference on Power, Dielectric and Energy Management at NERIST (ICPDEN)*. 2015, pp. 1–4. DOI: 10.1109/ICPDEN.2015.7084498.

[29] D. Shirmohammadi et al. “A compensation-based power flow method for weakly meshed distribution and transmission networks”. In: *IEEE Transactions on Power Systems* 3.2 (1988), pp. 753–762. DOI: 10.1109/59.192932.

[30] Jaianand Jayaraman and Arun Sekar. “Study of reactive power/voltage sensitivities in interconnected power system networks”. In: *2010 42nd Southeastern Symposium on System Theory (SSST)*. 2010, pp. 161–164. DOI: 10.1109/SSST.2010.5442842.

[31] A. Monticelli. *State estimation in electric power systems : a generalized approach*. eng. 1st ed. 1999. Power Electronics and Power Systems. Boston: Kluwer Academic Publishers, 1999. ISBN: 1-4615-4999-X.

[32] Barbara Borkowska. “Probabilistic Load Flow”. In: *IEEE Transactions on Power Apparatus and Systems* PAS-93.3 (1974), pp. 752–759. DOI: 10.1109/TPAS.1974.293973.

[33] Mingxu Xiang et al. “Probabilistic power flow with topology changes based on deep neural network”. In: *International Journal of Electrical Power & Energy Systems* 117 (2020), p. 105650. ISSN: 0142-0615. DOI: https://doi.org/10.1016/j.ijepes.2019.105650.

[34] Henrik Ronellenfitsch, Marc Timme, and Dirk Witthaut. “A Dual Method for Computing Power Transfer Distribution Factors”. In: *IEEE Transactions on Power Systems* 32.2 (2017), pp. 1007–1015. DOI: 10.1109/TPWRS.2016.2589464.

[35] Rahul Nellikkath and Spyros Chatzivasileiadis. “Physics-Informed Neural Networks for AC Optimal Power Flow”. In: *Electric Power Systems Research* 212 (2022), p. 108412. ISSN: 0378-7796. DOI: https://doi.org/10.1016/j.epsr.2022.108412.

[36] Heloisa H. Müller, Marcos J. Rider, and Carlos A. Castro. “Artificial neural networks for load flow and external equivalents studies”. In: *Electric Power Systems Research* 80.9 (2010), pp. 1033–1041. ISSN: 0378-7796. DOI: https://doi.org/10.1016/j.epsr.2010.01.008.

[37] Kyri Baker. “Emulating AC OPF Solvers With Neural Networks”. In: *IEEE Transactions on Power Systems* 37.6 (2022), pp. 4950–4953. DOI: 10.1109/TPWRS.2022.3195097.

[38] Laurent Pagnier and Michael Chertkov. *Physics-Informed Graphical Neural Network for Parameter & State Estimations in Power Systems*. 2021. DOI: 10.48550/ARXIV.2102.06349. URL: https://arxiv.org/abs/2102.06349.

[39] Clemens Martin Müller. *Python code repository for masters thesis - NN_LF_Topology*. URL: https://github.com/clmu/NN_LF_topology.

[40] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. DOI: 10.5555/1953048.2078195.

[41] Sarineh Hacopian Dolatabadi et al. “An Enhanced IEEE 33 Bus Benchmark Test System for Distribution System Studies”. In: *IEEE Transactions on Power Systems* 36.3 (2021), pp. 2565–2572. DOI: 10.1109/TPWRS.2020.3038030.

[42] J. S. Savier and Debapriya Das. “Impact of Network Reconfiguration on Loss Allocation of Radial Distribution Systems”. In: *IEEE Transactions on Power Delivery* 22.4 (2007), pp. 2473–2480. DOI: 10.1109/TPWRD.2007.905370.

A Data: small test network

Data in this appendix is generated using input neural network input data as described in table 5.1, and the code in [39] at commit 63d2697. Table A.1 Shows the average prediction accuracy of all output variables as a function of conducted training epochs. The *Average* column contains all row averages in the table, and is plotted in figure 5.1.

The structure of table A.2 is identical to A.1, but instead it contains the worst predictions of each state variable. The worst values are selected independently of the test set they belong to, meaning that the values are not representative of the entire NN output.

Table A.1: *Average percentage accuracies of the different output variables as a function of training epoch. Averages are plotted in figure 5.1*

Epochs	V_1	V_2	V_3	δ_1	δ_2	δ_3	**Average**
10	0.649	0.668	0.585	4.815	5.798	4.442	2.826
20	0.348	0.365	0.301	4.644	5.401	4.352	2.568
30	0.447	0.482	0.403	5.379	6.224	5.600	3.089
40	0.200	0.204	0.181	1.771	2.079	1.722	1.026
50	0.171	0.180	0.144	1.707	2.093	1.690	0.997
60	0.277	0.291	0.252	1.694	2.238	1.633	1.064
70	0.197	0.209	0.172	2.200	2.455	2.382	1.269
80	0.225	0.236	0.182	1.689	2.230	1.628	1.032
90	0.206	0.209	0.173	1.613	2.069	1.836	1.018
100	0.385	0.377	0.357	1.659	2.141	2.204	1.187
110	0.166	0.175	0.160	1.298	1.697	2.004	0.917
120	0.180	0.190	0.161	1.148	1.425	1.099	0.700
130	0.110	0.123	0.106	1.461	1.389	1.280	0.745
140	0.135	0.133	0.132	0.991	1.060	0.933	0.564
150	0.108	0.113	0.096	0.978	1.113	0.924	0.555

Table A.2: *Worst percentage accuracies of the different output variables as a function of training epoch. Averages are plotted in fig 5.1*

Epochs	V_1	V_3	V_3	δ_1	δ_2	δ_3	**Average**
10	10.12	10.44	7.70	6409.50	8387.15	2660.20	2914.19
20	9.39	9.88	7.13	5223.12	7179.81	2339.38	2461.45
30	8.68	9.11	6.50	7572.81	9249.04	3265.43	3351.93
40	5.66	5.96	4.09	1459.00	2452.55	880.82	801.35
50	5.95	6.20	4.38	1594.42	2295.41	830.09	789.41
60	5.37	5.63	3.85	1730.49	2816.49	964.58	921.07
70	6.21	6.51	4.59	502.13	3923.82	363.43	801.11
80	5.66	5.89	4.15	1429.17	2923.39	749.54	852.97
90	5.58	5.86	4.08	1427.50	3163.00	1985.65	1098.61
100	6.07	6.33	4.58	553.68	1257.36	2266.37	682.40
110	5.21	5.53	3.89	519.36	2965.77	1757.27	876.17
120	4.16	4.56	3.16	816.07	2379.00	520.31	621.21
130	4.08	4.27	3.04	1086.09	1954.18	933.94	664.27
140	3.64	3.77	2.52	456.98	516.54	335.28	219.79
150	3.77	3.93	2.68	668.94	1315.80	282.11	379.54

NTNU

Norwegian University of
Science and Technology