Chair for Reliability Engineering

Ali Osman Mert Kilicsoy

Gradient-enhanced neural networks: applications in linear and nonlinear mechanics,

Master thesis

Folio 01

Book 02

Archive 23

Nr. 003.01

2023

Gradient-enhanced neural networks: applications in linear and nonlinear mechanics

To obtain an academic degree
from the Faculty of Mechanical Engineering
at the Technical University of Dortmund

presented by
Ali Osman Mert Kilicsoy, B.Sc.
from
Duisburg, Germany

Dortmund, 2023

Abstract

Neural networks provide alternative models to approximate complex numerical models for regression. Improving approximation capabilities through use of gradient data with respect to the inputs is explored experimentally by applying them to linear and non-linear mechanics of 2D elasticity of a hook object attached to a wall. The training data is generated by a finite element model, an approximation of the hook object itself. Various metrics to compare the increase or decrease in performance are analyzed to show that gradient data with respect to the inputs does improve performance. This increase in performance comes two-fold, for one the quality of approximation improves and secondly less data and training time is required to approximate as well as a basic neural network. By introducing gradient information into the loss function of the neural network the loss of the model is expanded by addends. A method to blend the additional losses with the original model loss for this weighting problem is explored experimenting with ways of adjusting scale difference between training data, even after standardization has occurred. Comparisons are done, mainly with loss and relative l_2 error of the model outputs and processing times are also observed to gauge viability of gradient enhanced neural networks replacing other approximation methods. This work will show an improvement of performance when incorporating gradient data with respect to the inputs. The various methods to solve the weighting problem show worse or equal performance than the basic gradient enhanced model, however certain potential improvable tendencies are observed. Many different dimensions for neuron number, layer number, training data size are considered, but still leave room for greater dimensions to be analyzed. Furthermore, the applied methods to solve the weighting problem in this work are very simplistic and few, which can be expanded on under consideration of the scale difference between output and its gradient with respect to the inputs.

Keywords:
neural networks, gradient-enhanced, sobolev, linear and non-linear elasticity, loss weights, finite element model

Table of contents

Abstract ..... **I**

Table of contents ..... **II**

List of abbreviations ..... **III**

List of figures ..... **IV**

List of Tables ..... **VI**

1 Introduction **1**

2 Literature Review **4**

2.1 Neural Networks 4

2.1.1 General Function 4

2.1.2 Information and Statistical Theory 5

2.2 Types of Neural Networks 8

2.2.1 Artificial Neural Network (ANN) 9

2.2.2 Convolutional Neural Network (CNN) 10

2.2.3 Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) 11

2.3 Neural Network Parts and Subparts 12

2.3.1 Input, Output and Hidden Layer 12

2.3.2 Neurons 14

2.3.3 Activation Function 14

2.3.4 Types of Activation Functions 15

2.3.5 Weights and Biases 18

2.3.6 Training 18

2.3.7 Loss Functions 19

2.3.8 Training Algorithms 21

2.3.9 Data Preprocessing 36

2.4 Neural Network Enhancements 37

2.4.1 Physics Informed Neural Network (PINN) 38

2.4.2 Gradient-Enhanced Physics Informed Neural Network (gPINN) 39

2.5 Finite Element Method, Sensitivity Analysis and Uncertainty Quantification 40

2.5.1 Finite Element Method 40

2.5.2 Sensitivity Analysis 41

2.5.3 Uncertainty Quantification 42

3 Methodology **45**

3.1 Research goal 45

3.2 Gradient Enhanced Neural Network Model 46

3.3 First Variation – Expanding Model Output 47

3.4 Second Variation – Loss addend 50
3.4.1 Neural Network Model Generation 52
3.4.2 Loss Function 55
3.4.3 Backpropagation per Automatic Differentiation 60
3.4.4 Loss Weight Methods 61

4 Analysis 64
4.1 Overview of Models 64
4.1.1 Finite Element Model 64
4.1.2 Model Hyperparameters 65
4.2 Results 66
4.2.1 Training and Prediction times 67
4.2.2 Comparisons – Linear Case 69
4.2.3 Comparison – Weighing of Losses 79
4.3 Discussion 83

5 Conclusion 85

List of References VI

Appendix/Appendices VII

Affidavit VIII

List of abbreviations

Adam	Adaptive moment estimation method
ANN	Artificial neural network
CNN	Convolutional neural network
ELU	Exponential linear unit
FEM	Finite element model
gPINN	Gradient enhanced physics informed neural network
gNN	Gradient enhanced neural network
LSTM	Long short-term memory
MAE	Mean-absolute error
MPE	Mean percentage error
MSE	Mean-squared error
NAG	Nesterov accelerated gradient
NN	Neural network
PDE	Partial differential equation
PINN	Physics informed neural network
ReLU	Rectified linear unit
RMS	Root-mean square error
RNN	Recurrent neural network
SANN	Sobolev neural network

List of figures

Figure 2-1: The first perceptron by Rosenblatt (figure by author)....8
Figure 2-2: 2D linear hyperplane splitting data into two classes. The weight vector runs perpendicular to the hyperplane (figure by author) 9
Figure 2-3: Simplified neural network model, a black box model mapping input to output (figure by author)....9
Figure 2-4: Application of feature mask onto input array (figure by author) 10
Figure 2-5: Recurrent feedback in a model with one neuron $\boldsymbol{a}$ (figure by author)....11
Figure 2-6: A simple LSTM unit. The hidden state stores previous values, the cell state stores long term previous values. The parameters inside this unit are part of the neural network model parameters (figure by author) 12
Figure 2-7: A visual representation of input, hidden and output layer (figure by author) ...13
Figure 2-8: The general calculation occurring inside a neuron, which is very similar to the perceptron (figure by author)....14
Figure 2-9: The rectified linear unit (figure by author)....15
Figure 2-10: The leaky rectified linear unit (figure by author)....16
Figure 2-11: The exponential linear unit (figure by author) 16
Figure 2-12: The hyperbolic tangent (figure by author) 17
Figure 2-13: The sigmoid function (figure by author) 17
Figure 2-14: Generalized difference between the three methods of differentiation. Illustrated by using the function $\boldsymbol{fx = \cos\,(x + \cos x)}$ (figure by author) 22
Figure 2-15: A feed-forward neural network model with 2 hidden layers with 3 neurons each. It has 2 inputs and 1 output. Neuron outputs are defined as $\boldsymbol{a}$ and the linear part of the neuron output as are defined as $\boldsymbol{z}$ (figure by author)....27
Figure 2-16: All 3 Paths for only $\boldsymbol{w110}$ shown with each distinct color (figure by author) 31
Figure 2-17: A Simple structure of a PINN. The PDEs are simply an extension to the basic neural network model, specifically the loss function. Denoted are the model output $\boldsymbol{u}$ and the true output $\boldsymbol{u}$ and the PDE residual $\boldsymbol{R}(\cdot)$ (figure by author).....39
Figure 2-18: The gPINN is an expanded PINN structure. The additional loss of the gradient of the PDE is now added to the total loss (figure by author)....40
Figure 2-19: Propagation of uncertainty through a deterministic method. The input parameter x defines the response y through the probability distributions

$\boldsymbol{f(xi)}$.The deterministic method is creating samples of response to generate probability distributions $\boldsymbol{f(yi)}$ (figure by author) 43
Figure 3-1: Previous example of a neural network expanded to have 2 outputs (figure by author) 47
Figure 3-2: Red marked lines are newly introduced connections in the neural network (figure by author) 49
Figure 3-3: Information Flow Chart of the MATLAB Program Code (figure by author) ... 52
Figure 4-1: Sketch of the hook system approximated by the finite element model. The design parameters $\boldsymbol{x1}, \boldsymbol{x2} = [-\mathbf{1}, \mathbf{1}]$ adjust the blue marked nodes along the green axis (figure by author) 65
Figure 4-2: gNN vs. NN, $\boldsymbol{l2}$ error (figure by author) 69
Figure 4-3: gNN vs. NN, loss (figure by author) 69
Figure 4-4: gNN with different number of hidden layers, $\boldsymbol{l2}$ error (figure by author) 70
Figure 4-5: gNN with different number of hidden layers, loss (figure by author) 70
Figure 4-6: gNN with different number of neurons per hidden layer, $\boldsymbol{l2}$ error (figure by author) 71
Figure 4-7: gNN with different number of neurons per hidden layer, loss (figure by author) 71
Figure 4-8: gNN with different training data sizes, $\boldsymbol{l2}$ error (figure by author) 72
Figure 4-9: gNN with different training data sizes, loss (figure by author) 72
Figure 4-10: Confusion chart, FEM vs. gNN in red and FEM vs. NN in blue. The black diagonal represents a perfect fit (figure by author) 73
Figure 4-11: Exceedance curve, FEM vs. NN. The means of both FEM and NN are drawn in as vertical lines (figure by author) 73
Figure 4-12: Exceedance curve, FEM vs. gNN. The means of both FEM and gNN are drawn in as vertical lines (figure by author) 74
Figure 4-13: : gNN vs. NN, $\boldsymbol{l2}$ error (figure by author) 74
Figure 4-14: gNN vs. NN, loss (figure by author) 75
Figure 4-15: gNN with different number of hidden layers, $\boldsymbol{l2}$ error (figure by author) 75
Figure 4-16: gNN with different number of hidden layers, loss (figure by author) 76
Figure 4-17: : gNN with different number of neurons per hidden layer, $\boldsymbol{l2}$ error (figure by author) 76
Figure 4-18: gNN with different training data sizes, $\boldsymbol{l2}$ error (figure by author) 77
Figure 4-19: gNN with different training data sizes, loss (figure by author) 77
Figure 4-20: Confusion chart, FEM vs. gNN in red and FEM vs. NN in blue. The black diagonal represents a perfect fit (figure by author) 78
Figure 4-21: : Exceedance curve, FEM vs. NN. The means of both FEM and NN are drawn in as vertical lines (figure by author) 78
Figure 4-22: Exceedance curve, FEM vs. gNN. The means of both FEM and gNN are drawn in as vertical lines (figure by author) 79
Figure 4-23: Fixed weight variants of gNN, $\boldsymbol{l2}$ error, linear (figure by author) 80
Figure 4-24: Fixed weight variants of gNN, loss, linear (figure by author) 80
Figure 4-25: Dynamic weight variants of gNN, $\boldsymbol{l2}$ error, linear (figure by author) 81
Figure 4-26: Dynamic weight variants of gNN, loss, linear (figure by author) 81
Figure 4-27: Fixed weight variants of gNN, $\boldsymbol{l2}$ error, nonlinear (figure by author) 82
Figure 4-28: Fixed weight variants of gNN, loss, nonlinear (figure by author) 82
Figure 4-29: Dynamic weight variants of gNN, $\boldsymbol{l2}$ error, nonlinear (figure by author) 83
Figure 4-30: Dynamic weight variants of gNN, loss, nonlinear (figure by author) 83

List of Tables

Table 2-1: Variables and their definition (representation by author) 24
Table 4-1: Hyperparameter initialization values (representation by author) 66
Table 4-2: Hyperparameter adjustment for training data size variation results (representation by author) 66
Table 4-3: Learnrate update table (representation by author) 66
Table 4-4: Prediction and Training times for linear elasticity, Time is in seconds (representation by author) 68
Table 4-5: Prediction and Training times for non-linear elasticity, Time is in seconds (representation by author) 68
Table 4-6: Fixed loss weights table (table by author) 79
Table 4-7: Dynamic Weight Table (table by author) 80

1 Introduction

Machine learning and neural networks cover a wide net [1] of different fields of study, whereby through a combination of the knowledge of the various fields they are capable of tasks, such as classification or regression, as well as more complex functions, such as image recognition or language recognition [1][2][3][4][5]. Although for a time neural networks were still behind alternative preexisting methods in many cases [6][7], neural networks have shown great progress in recent years with breakthroughs in many topics [8][9][10]. Furthermore new research has allowed neural networks to tackle issues of very complex structure, like conversing with a human through language and speech processing [11][12]. Other factors lending to these breakthroughs include technological advancements of available hardware [13][14], but also the coordination between scientific fields, that were previously working parallel to each other [9][15]. Plenty of research has been done and is still underway on further enhancing neural networks [16][17][18]. While there is plenty of research on a neural network's activation functions [19][20], other research tackles the general architecture of layers [21], the loss function of the model [22][23], the optimization algorithm [24][25] and many others [26][27]. While neural networks have shown great error ranges after the training process is completed [28][29][30], the amount of data necessary to train a model, as well as training time are factors to consider for improvement [31][32].

There are a variety of tools available to engineers to solve problems of different nature. Current numerical methods dominate the engineering market for problem solving, like finite element method, finite difference method, Runge-Kutta methods, Monte-Carlo method and many more [33][34][35]. These methods allow for the analysis of systems from simple to very complex nature. While neural networks are more than capable after training to output results with very small error margins, numerical methods are often faster for a few calculations and have a deeper background of research behind them. However there are certain fields and use cases, where neural networks have been shown to be equal to numerical methods or provide an even better model for problem solving [36][37][38]. In the case of uncertainty quantification, where it is necessary to have multiple various results for a scenario/use case to be able to proceed with the calculations and analysis of uncertainty, methods such as Monte-Carlo are the main tools for engineers [39]. Through Monte-Carlo simulation engineers can generate many datapoints necessary for uncertainty quantification. The Monte-Carlo method however can be tedious, slow, and ineffective, as well as inaccurate with its approximations, when dealing with large and complex data [40]. This is one of the possible avenues in which neural networks can deliver an edge to current numerical methods. Since numerical methods are very processing heavy and time consuming, the more complex the system is and depending on the degree of accuracy the model should have, numerical methods prove themselves very taxing [35]. Furthermore, for uncertainty quantification, where it is necessary to produce a great dataset through deterministic methods, replacing those methods with trained neural networks could help reduce the time necessary to produce large datasets. A neural network model, provided it has enough training data and enough prior time to train, can output satisfying results for multiple cases of the same scenario very quickly. If data of the system that is to be approximated by a model is available and the data itself contains the underlying

information and patterns necessary that describe that system, neural networks have been shown to be able to approximate and generalize any such system [41][42]. Therefore, one could make use of a neural network trained on sparse data, to generate and fill out the sparse data of a system or model to then use the generated large data in uncertainty quantification to evaluate the system or model. Some papers are already a step further where they use neural networks do directly estimate uncertainty [43][44]. A neural network in its most basic form can be used for many tasks, but there are a variety of different designs and architectures employed for various tasks and functions. Still, adaptation of neural networks instead of current numerical methods is slow or unnecessary, as often neural networks only improve productivity by a little or not at all depending on the use case and the available data. In addition, neural networks take time to train and need great datasets to train accurately.

Another important part of research in neural networks concerns itself with the enhancement of the existing designs of the models. Various enhancements of neural networks have shown that the general idea of providing more information to the neural network model to constrain it often improves accuracy and speed [45][46][47][48], possibly giving it an additional edge over basic neural networks and possibly over widely used numerical methods. One such enhancement that this work will present is through the addition of derivative information in the dataset used for training a neural network model. By using available gradient data with respect to the inputs during the training process the neural network model is now constrained on its output as well as the derivative of the output resulting in quicker and more accurate training. Additionally, such neural network enhancement can help alleviate the issue of the size of the training dataset necessary for an accurate model. This is because many numerical methods and models use so called adjoint methods which allow the calculation of derivatives without strongly affecting the processing time [49][50].Figure 2-17

This work will concern itself with enhancing a basic regression neural network, that is a feed forward neural network, through constraining the model loss function to be optimized by additional use of gradient information. Gradient information here means the partial derivatives of the output value of our dataset with respect to inputs, not to be confused with the gradient descent training method where the loss functions gradient is used to optimize the model parameters. The general loss function that is formulated in regression neural networks and optimized through algorithms is modified, by adding the loss of the gradient of the model consisting of the mean-squared error of the residual of the gradient with respect to inputs of the model and the true gradient with respect to inputs to the original mean squared error of the model output and the true output. This approach of expanding the loss and adding new constraints has been used in other papers concerning physics informed neural networks, PINNs, where they call it a gradient enhanced physics informed neural network, gPINN [48]. Other examples include a Sobolev neural network, SANN, and a gradient enhanced neural network applied to uncertainty quantification. The former goes into a detailed theoretical framework for using gradient information during training. The latter gives a theoretical framework for a basic 2 layer neural network. In this work the loss function does not contain any partial differential equations or formulae constraints where the output of the model is a parameter of a partial differential equation expressed as residuals. Therefore, only the data and its underlying information will provide the constraints for the neural network. Furthermore, the data used to train the neural network model will be generated by a finite

element model of linear and nonlinear elasticity, which will be expanded on in section 2.5.1 and 4.1.1. This work will also apply different weight schemes to the losses of the neural network and compare results [51].

Concerning the sections of this work, it proceeds as follows: First, the current state of the art concerning neural networks as well as various model enhancements will be examined. This will cover the basics of neural networks pertaining to this work, which includes their general functions and use cases, a list of parts and subparts and their general functions, the process and theory of training a model and the different algorithms and loss functions in use for training neural networks. Brief theoretical explanations of some parts are included as well. The section will also include a detailed example of calculations for a regression neural network model similar to the one used in this work for analysis. This example will focus on automatic differentiation, as in the forward and backwards propagation of the model during training. Afterwards research of current enhancements of neural networks and a brief focus on papers which cover enhanced neural network models through use of gradient information is discussed. Lastly a succinct discussion on finite element models in general and the theory behind the finite element model used for this works analysis is addressed. Additionally, an overview on sensitivity analysis and uncertainty quantification is included to clarify the use case for neural networks.

The next section will describe the gradient enhancement used in this work in detail and the neural network architectures applied to the data of the linear and nonlinear systems generated by the finite element model. The previously made calculations for the base neural network are revisited and expanded on the gradient enhancement part. As the neural network will be programmed in MATLAB, a concise presentation of the calculations and flow of information occurring inside the program will be showcased. Then, the linear and nonlinear examples that are approximated by the finite element model are presented and described. Following this, the generation of the training data through the finite element model will be expanded on. This will include a short descriptive summary of the finite element model's attributes and code. Lastly the methodologies used to analyze the results in the following section are elaborated on.

In the third section, the results will be visualized and analyzed. A concise presentation of all results for each example and the variations will be showcased.

The fourth section will be a conclusion and summary of the work, where its results will be evaluated. Furthermore, possible future developments and further research of interest following this work are discussed.

2 Literature Review

2.1 Neural Networks

A neural network is a general model capable of representing or characterizing different functions used for a multitude of applications in many different fields. Neural network models are capable of approximating mathematical expressions, so if a task can be construed as such, a neural network can achieve that task. While there is a great variety of neural networks when it comes to their function and capabilities, every neural network follows a general design. Their earliest design lends itself to the system of biological neurons found in humans. Compared to other model-based methods neural networks are mainly data driven models that self-adapt. This means a neural network model has less assumptions before it is trained on data. The models use algorithms and optimization of an objective function defined by the engineer to adapt itself to the data given to it for training. Therefore, a neural network simply approximates whatever the data represents in the form of the desired output, nothing else [52]. Through multiple interconnected nodes and different parameters attributed to various mathematical processes, the model can produce an output by a given input. Through a training process defined by an algorithm and a dataset of input/output pairs, the model adjusts its accuracy for a given objective function. In contrast to regular programming models, where "rules" are defined and with given inputs the model produces certain desired outputs, a neural network model consists of a defined "ruleset" which through use of algorithms during the training process is adjusted given the input/output pair dataset. This process is comparable to the process of how humans adjust their ability of a task, given their experience, hence the term "learning" is used as well for neural network training. Neural networks are capable of simple tasks like regression and classification. More complex tasks, such as speech and image recognition, involve more complex neural networks, but are also achievable [53].

2.1.1 General Function

A neural network is comprised of a collection of units, usually called neurons. These neurons are linked together in structured layers, with a concrete flow of information. Depending on the design, a neural network is capable of accomplishing tasks like regression or classification. There are variations of these tasks, like binary classification and multivariable classification. The task of the neural network is important when it comes to designing the model and its hyperparameters. Hyperparameters are any adjustable attribute of a neural network. This includes how many neurons or other parts are used, what algorithms and which parameters are used, etc. Mathematically, a neural network is a series of nested functions which contain a vast array of parameters. How the functions are nested is defined by the structure of the model, another hyperparameter. The functions themselves are simple mathematical equations and its parameters in general are weight parameters and bias parameters. Because of the simplicity of a neural networks and their associated mathematical equations, the models are quick and efficient at learning from data and once trained, outputting correct results. By using a training dataset, the model learns the datasets patterns. This

training process involves algorithms that vary depending on the desired task or the available dataset and its mathematical space.

As an example, in the case of a regression task there are various simple or rigorous numerical methods available to create a model which approximates a dataset. However, with a neural network, the model would only use the available dataset to learn its underlying pattern through training algorithms, which once trained results in faster evaluation. While most of the structure of a neural network is usually defined by the engineer, the set of parameters that make up the mathematical process of the model are adjustable by the model itself during the training process. This allows the model to find an optimal solution for the task "on its own". Rephrased that means, while in classical computer science and engineering approaches one uses predefined rules that are then applied to data to obtain results, neural networks use datasets of input/output pairs to "create its own rules". They generalize and approximate the data and its patterns and information. These models are then able to be applied to new data, given that it follows the same underlying pattern and information of the used training data. The models can then output satisfactory results on data they have never seen [53]. Mathematically, neural networks are general approximators, meaning under certain conditions an adequately sized neural network is capable of approximating any function [42]. Therefore, a dataset needs to contain an underlying pattern that can be "learned" by the model. For example, in the case of binary classification of the very first proven concept of a neural network, the perceptron, the data was dividable through a linear mathematical expression, which the model approximates by adjusting its parameters [54].

2.1.2 Information and Statistical Theory

In general, the trained models are functions capable of mapping a given input to an output. Through the training process, the likelihood of mapping an input to an output occurring correctly is increased, which is why neural networks are very dependent on the used training data. The theory of this machine learning process is the same behind adjusting a probability distribution to correctly mirror a desired distribution of probabilities. By varying the likelihood of an expectation of an event, our model's probability distribution can copy the probability distribution of the dataset which optimally would be equal to the phenomena our dataset represents. The maximum likelihood method allows exactly this. By minimizing the dissimilarities between the distribution of our dataset and the model's distribution of output values, cross-entropy is reduced. Information theory gives us a broader understanding behind machine learning and cross-entropy. Since bits are used to communicate information, one topic information theory concerns itself with is the storage that is necessary for an observed event of a probability distribution. Intuitively, the rarer an observed event is, the more bits should be "reserved" for such an event, as their information is transferred less frequently. In short, the lower the probability of an observed event, the higher its information. The information $I(x)$ of an observed event x is defined through the probability $P(\cdot)$ as follows:

$$I(x) = -log\,(P(x)) \tag{2-1}$$

The entropy $H(x)$ is defined as the average number of bits necessary to transfer the information of an event. It is expressed as:

$$H(x) = -\sum P(x)\log\left(P(x)\right) \quad \textbf{(2-2)}$$

The more even the probabilities in a probability distribution the higher the entropy. Entropy here relates to how surprising the general events are. Following entropy, cross-entropy is called the difference between two probability distributions for a given input. In the case of neural networks, our training algorithm aims at reducing the cross-entropy between the dataset and the model's predictions. Relating to information theory, cross-entropy expresses the average amount of bits necessary to transfer data of the true distribution P gained from the dataset, with the model's distribution Q. For discrete probability distributions cross-entropy $H(P, Q)$ is defined as:

$$H(P, Q) = -\sum P(x)\log\left(Q(x)\right) \quad \textbf{(2-3)}$$

In the case of continuous probability distributions, the expression changes to take the integral across the events instead of the sum:

$$H(P, Q) = -\int P(x)\log\left(Q(x)\right)dx \quad \textbf{(2-4)}$$

In the case of a neural network, the most used loss function is the mean-squared error. Per theory, the mean-squared error is the cross-entropy between the dataset and the model's distribution. The following is going to briefly summarize the theory explaining why the mean-squared error loss function works for machine learning.

If Θ is a parameter defining a probability distribution function and its parameters and X is the joint probability of our dataset, to increase the probability $P(X \mid \Theta)$, means to find Θ, so that the likelihood of X being observed increases. Therefore, it is called likelihood $L(X;\ \Theta)$. X can be split into its various samples leading to $L(x_1, x_2, \dots, x_n;\ \Theta)$. This joint probability distribution is defined as the product of the probability of the sample given Θ for all samples. This product over many probabilities can lead to numerical underflow, therefore the natural logarithm log is taken. The log does not change the argmax of the likelihood but transforms it into a sum, which is more convenient and simpler. Since a loss function is phrased as being minimized, one transforms the

likelihood L and is left with the minimization of the negative log-likelihood. In the case of machine learning, one uses the conditional probability of the output given the input, given the model, so:

$$\underset{\theta}{\operatorname{argmax}}\big(L(y|X;\,\theta)\big) = \underset{\theta}{\operatorname{argmax}}\Big(\sum \log\left(P(y|X;\,\theta)\right)\Big) \tag{2-5}$$

$$OR$$

$$\underset{\theta}{\operatorname{argmin}}\big(-L(y|X;\,\theta)\big) = \underset{\theta}{\operatorname{argmin}}\Big(-\sum \log\left(P(y|X;\,\theta)\right)\Big) \tag{2-6}$$

Whereby $L(y|X;\,\theta)$ expresses the likelihood of the probability distribution of X producing the probability distribution y for given model parameters θ. Through a handful of additional mathematical operations, these equations are transformed into a mean-squared error. A typical regression loss function is the half-mean squared error with the additional denominator N, the number of samples. This shows why maximum likelihood estimation forms a basis for (supervised) machine learning models. The maximum likelihood estimation framework leads to the mean-squared error in the case of a regression task, and the cross-entropy loss function in the case of a classification task [55][56][57].

In this sense the study of machine learning neural networks is tightly connected to statistical theory and information theory. In statistical theory the concern is about datapoints and their patterns. The focus is on understanding how these features inside of datasets can be modeled. Especially when it comes to the optimization and training process of neural networks, statistical theory plays a great role in the design of the optimization function. Methods like the previously discussed maximum likelihood estimation method are used to optimize the neural network function, although they are not referred to as such in the machine learning community [58]. Information theory studies the information stored inside of data and how to quantify it. It also concerns itself with the flow of information and how this can occur. In neural networks, information theory helps understand the effect of input features onto the outputs. The study also defines the uncertainty and randomness in a model's output as entropy and applies methods to reduce this entropy to increase a model's accuracy. Therefore, both theories are an integral part of neural networks and machine learning [59]. If the training data is too biased, it can result in unusable results. In the case of regression, a neural network excels at outputting correct values inside the range of a dataset. However outside of the dataset, depending on the mathematical function the dataset represents, a neural network can have difficulties approximating correctly. It has been shown through the Universal Approximation Theorem, that neural networks are theoretically capable of mapping any

input to an output, as in, neural networks are universal. Therefore, neural networks can approximate the results of any function, no matter its complexity, so long as the network is sufficiently designed to mirror that complexity [42].

2.2 Types of Neural Networks

There has been considerable progress on neural network architecture in the last few decades. The very first neural networks were ultimately performing worse compared to their contemporary methods of optimization and solution finding [60], however newer types of neural networks have proven to be just as good, if not considerably better [61][62]. Early neural networks were very simple in their design, mainly made up of a single neuron/unit. The most known and earliest practical neural network was the perceptron by Frank Rosenblatt in 1958 [54]. The perceptron is a binary classifier, meaning it can classify information into two classes.

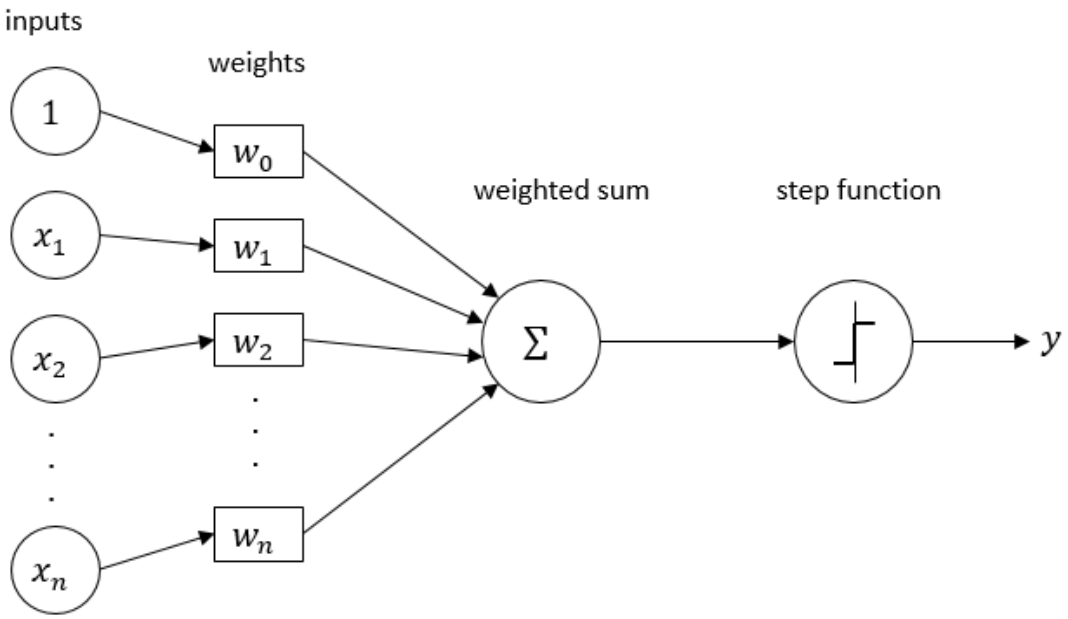

Figure 2-1: The first perceptron by Rosenblatt (figure by author)

Figure 2-1 illustrated Rosenblatt's perceptron. It uses the weighted sum of its inputs and applies a step function. This step function introduces non-linearity and is usually called an activation function. The training process is very similar to today's neural networks training process. Through use of data of input and output pairs, the neural network produces an output given the input and its output is then compared to the dataset output. The difference of these values is then used in an optimization algorithm. To minimize this difference, the perceptron parameters are adjusted by the perceptron learning rule, which is a very simple iterative process.

If the model prediction is false, the model parameters are adjusted by the difference of the outputs multiplied with a learning rate and the inputs. This is akin to a hyperplane rotation to split the data into two classes in the data space, which is why the perceptron is only capable of simple

linear classification tasks and has trouble classifying more complex binary data, where the hyperplane is non-linear. The perceptron represents an important milestone in the study of neural networks, proving that a neural network model can be trained effectively using data [54].

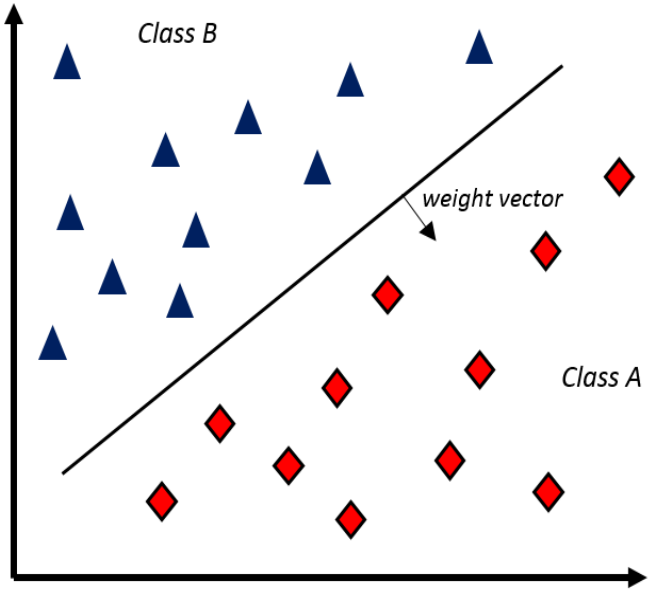

Figure 2-2: 2D linear hyperplane splitting data into two classes. The weight vector runs perpendicular to the hyperplane (figure by author)

Figure 2-2 shows the hyperplane that is rotated to split the data into two classes. The vector of the weights is perpendicular to the hyperplane.

2.2.1 Artificial Neural Network (ANN)

An artificial neural network is a simple feed forward neural network. The term is usually used for neural networks in general, although they are also called multilayer perceptron. The most widely used ANN is the multi-layer feedforward network. In ANNs the flow of information is unidirectional.

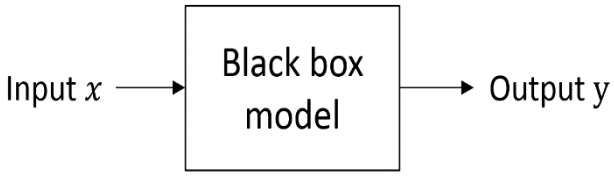

Figure 2-3: Simplified neural network model, a black box model mapping input to output (figure by author)

Figure 2-3 illustrates the simple design of a neural network. It is a black box model, where the training algorithm adjusts parameters of the model to approximate the data. The process of mapping input to output occurs in a "concealed" manner. The model consists of multiple neurons ordered in layers, with an input and output layer at each end of the model. Each neuron contains an activation function. These neural networks are used for a variety of tasks as they represent a

basic form of neural networks and can be found in slightly altered ways in other types of neural networks. The network maps a fixed-size input to a fixed-size output. A neural network is simply a black box model. The information propagates through weighted connections between neurons and layers. Each neuron takes a weighted sum of its inputs and passes it through a non-linear function to generate its output [52].

2.2.2 Convolutional Neural Network (CNN)

A convolutional neural network is a special type of artificial neural network that is preceded by at least one convolutional layer. Some convolutional neural networks can be split up resulting in simple multilayer feedforward networks in between layers, however the convolutional layer is its core part. The convolutional layer is a kind of compression layer. Whereas a normal hidden layer in a simple artificial neural network takes the weighted sum of its inputs and together with a bias passes it through a non-linear function to provide an output, a convolutional layer takes an input in parts, where each part is a meld of inputs. A CNN is capable of processing multiple arrays of inputs which is useful for image classification, where an image is split and combined by, for example its pixels. The convolutional layer units are ordered in feature maps, also called kernels.

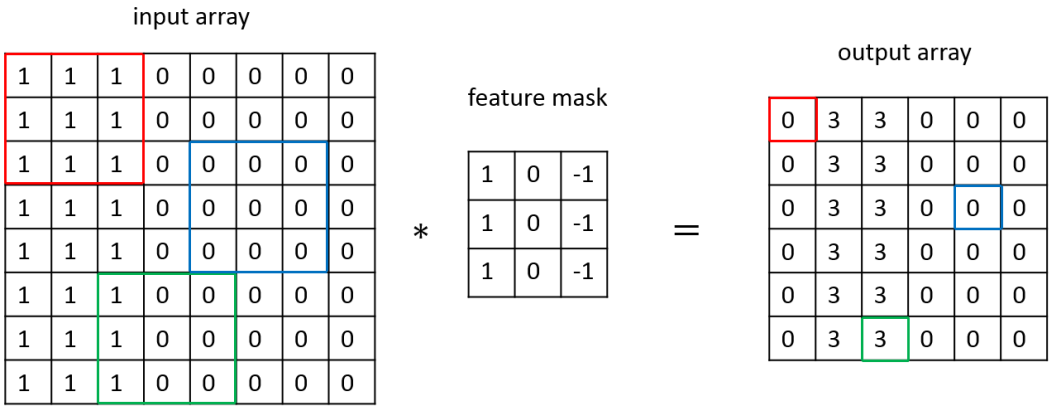

Figure 2-4: Application of feature mask onto input array (figure by author)

Figure 2-4 demonstrates the process of convolution. The depicted feature mask is able to discern vertical edges. As can be seen from the input array and the output array, the feature mask allows the model to extract features the engineer designing the neural network deems relevant. A feature map allows the highlighting of certain patterns in those multiple arrays of input data. Each unit of a feature map is attached to a local part in the feature map of the previous layer through a set of weights. Just like with an ANN, a unit takes the weighted sum of the information from that connection and passes it through a non-linear function. Inside a feature map, all its units share the same weights, because in certain array data inputs like images, local values highly correlate and form clear local patterns that can be observed. Furthermore, these patterns are often invariant to location. Since this process of filtering through feature maps is called discrete convolution, neural networks employing such layers are called convolutional neural networks [63]. To allow the neural network to learn features of hierarchical order, multiple convolutional layers with distinct feature maps are stacked. A convolutional layer is usually followed by a pooling layer which merges

thematically similar features together. They reduce the dimensions of the input data capturing only the most important information and reducing the complexity of the network computation. One way of pooling is max pooling, where only the maximum value of a given space of the input data is used. CNNs are mainly used for image recognition [61] and language processing [64].

2.2.3 Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM)

In the case of sequential data or data that has temporal dependencies simple feed forward networks and convolutional networks are inadequate. Recurrent neural networks are capable of processing input one sequence at a time through storing the history of processed information from previous sequences. This stored history of processed information is relayed through recurrent connections. A recurrent neural network is a deep feed forward neural network, where the weights over the sequences are the same for each individual layer [10]. In a recurrent neural network, each neuron can store and process information from previous runs. This creates a feedback loop where previous sequences can influence later sequences.

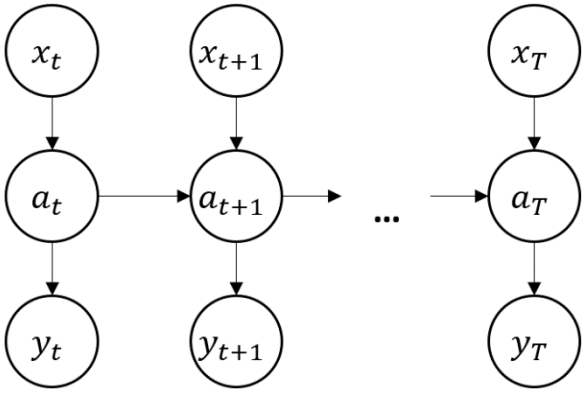

Figure 2-5: Recurrent feedback in a model with one neuron a (figure by author)

A simple recurrent network with only 1 neuron is depicted in Figure 2-5. In the figure, t denotes the current sequence and in such a model, the previous neuron output also influences the following sequence. By combining the previous sequence information with the current sequence input of the neuron and passing it through a non-linear function an output is produced. This output is then used to adjust the memory stored in the neuron. Basic recurrent neural networks had the issue of vanishing gradients leading to difficulties in storing long-term sequence history. The discovery of the long short-term memory unit and gated recurrent unit have solved this issue. A long short-term memory unit consists of multiple smaller units. The first unit relays to itself in the next sequence, with a weight of 1, its own information. This relaying connection is controlled by a second unit which learns to decide whether the information should be relayed or not [65]. For the training process the automatic differentiation backpropagation occurs unfolded through time. Recurrent neural networks are used for language processing [64], speech recognition [66], machine translation [67] and other time dependent tasks [68][69].

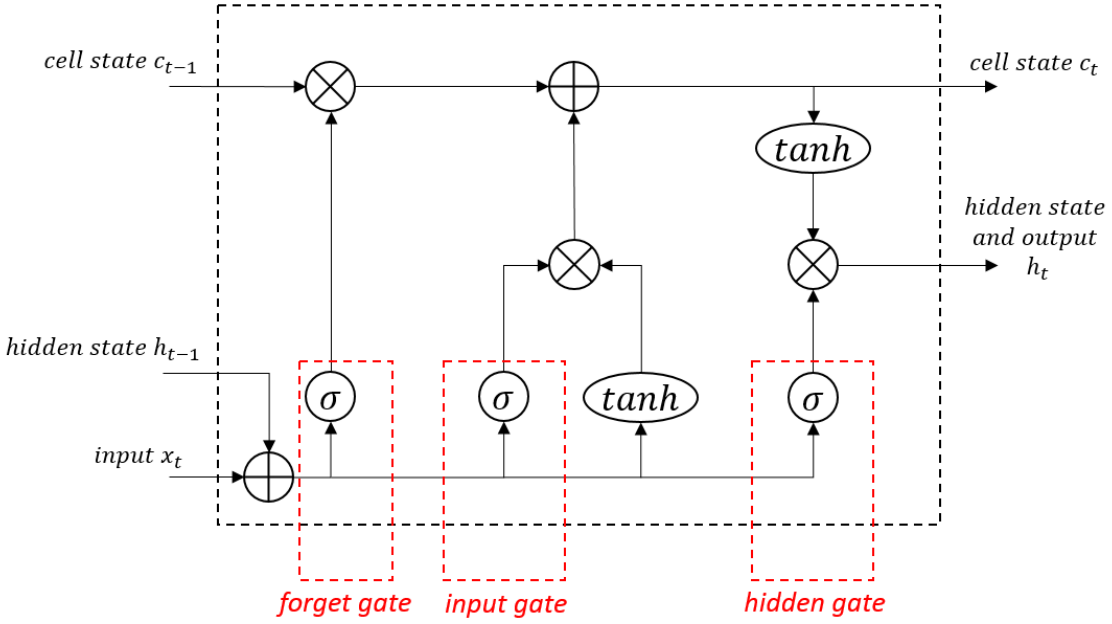

Figure 2-6: A simple LSTM unit. The hidden state stores previous values, the cell state stores long term previous values. The parameters inside this unit are part of the neural network model parameters (figure by author)

The general design of the LSTM can be seen in Figure 2-6. The LSTM has a hidden state and a cell state. While the hidden state carries previous sequence information of short-term nature, the cell state collects long-term information.

2.3 Neural Network Parts and Subparts

A neural network consists of multiple units, called neurons. These neurons are ordered in structures called layers, with each layer having a defined number of neurons. Between every layer there are connections between the neurons. In general, these connections visualize the flow of information, in the case of a unidirectional flow of information it would mean that the outputs of neurons in previous layers are passed to the input of the neurons in the next layers [70].

2.3.1 Input, Output and Hidden Layer

The input layer of a neural network is the very first layer. It is the layer where the input data is inserted into the model. For each feature or parameter there is an input neuron. The output layer is the last layer of a neural network. Its neurons output the final prediction of the model. The input layer and its number of neurons are defined according to the training data and depending on the use case, the output layer varies. A general illustration of the layers can be seen in Figure 2-7.

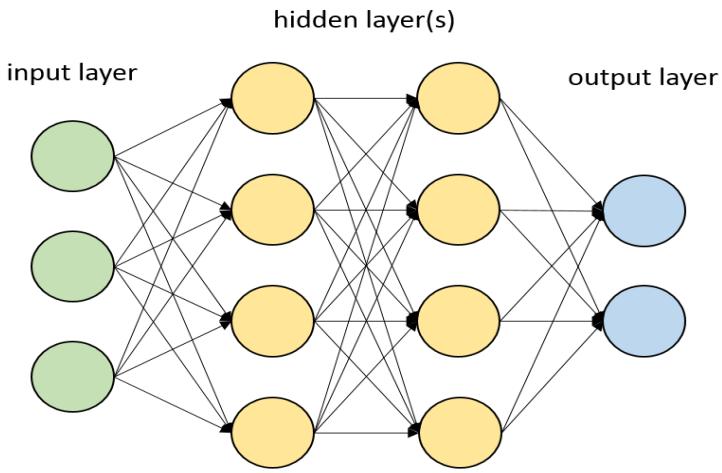

Figure 2-7: A visual representation of input, hidden and output layer (figure by author)

For a simple one output regression network the output would have one neuron, which would be the value to be approximated. For a simple classification network, the output layer would consist of as many outputs as there are classes to identify. In the example of an image recognition network for numbers, the model would have an array of possible classes, $0\ to\ 9$, to identify the numbers in the image. Each of these classes would have an output value representing the probability of the class being the correct output, where the most likely class is picked. Most of the abstraction and work of a neural network occurs in the hidden layer. The hidden layer lies between input and output layers. The hidden layer consists of one or more layers and the number of neurons is not predefined. The purpose of the hidden layers is to extract and learn information from the data. Some hidden layers operate as encoder and decoder layers in certain neural networks like recurrent neural networks. An encoder layer compresses the data, reducing or removing unimportant information, leaving only the important information. The decoder layer then takes the encoder layers compressed information and transforms it into usable data for the task of the neural network. This helps remove noise and other issues from the dataset. While a simple network consisting of a handful of layers and neurons can solve simple tasks, more complex tasks require deeper networks. Deep learning networks have a sizeable hidden layer and neuron count. In the hidden layer one can find the model parameters, which are called weights and biases, of the neural network [70]. The hidden layer contains the main portion of calculations of a neural network. Besides the weights and biases, there are also the non-linear functions like the rectified linear unit or the sigmoid that are used to process and pass the neuron information to the next layer.

2.3.2 Neurons

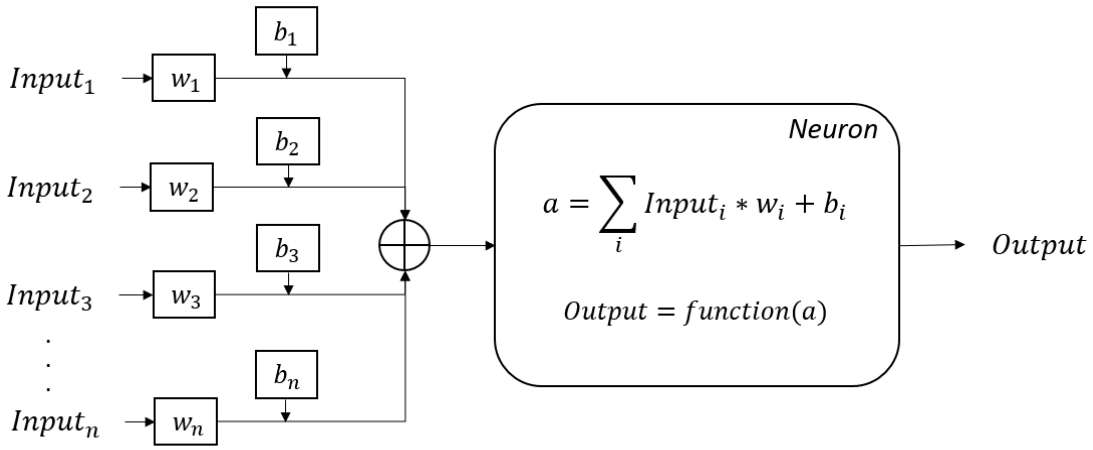

Figure 2-8: The general calculation occurring inside a neuron, which is very similar to the perceptron (figure by author)

Neurons are the core unit of a neural network. Their first designs follow biological neurons found in the nervous system of animals and research since then has provided adjustments and variations improving them. As depicted in Figure 2-8, a general neuron is a unit that consists of an input and an output. Inside the neuron a mathematical calculation is applied onto the input to generate its output. A neuron can get its input from multiple neurons of the previous layer and thereby a neuron can also output to multiple neurons in the next layer. Its input is the weighted sum of all incoming values, each with its own weight and bias. In some neural networks, like Recurrent Neural Networks, this flow of information is not strictly unidirectional. The calculations inside the neuron and the connections do vary between different neural network architectures. However, in general incoming values are multiplied with a weight and a bias is added. The weighted sum of all these inputs is then passed through a non-linear function introducing non-linearity to the neuron, where its value is then passed on to the next layer. The non-linear function allows the neuron to transform its output to a desired form. Depending on the task the neural network is supposed to do, this varies.

2.3.3 Activation Function

Activation functions are different types of mathematical operations to adjust the output of neurons. The desired function of the neural network often dictates an array of suitable activation functions. While basic calculations of a neuron are linear, the activation function usually serves as a non-linearity in the neural network to make the approximation of non-linear functions possible, but also to increase the networks efficiency as approximating with only linear functions is difficult for certain datasets. In general, an activation function is part of every neuron. A neural network model usually uses the same activation function for all its neurons. However, some neural networks use different activation functions for certain layers or certain neurons do not use them at all. For example, for image recognition, a combination of a convolutional neural network and a recurrent neural network has been used effectively. In this case the CNN and the RNN might

have different activation functions as their tasks in the combined neural network are different. In the case of a LSTM network, where memory is stored for its algorithm, the memory units may vary from the general neuron structure [65]. In the case of classification, a bounded range for the output of neurons works better. Since classification neural networks calculate probabilities of multiple classes given an input, it makes sense to limit the predicted probabilities ranges between $0\ and\ 1$ like any probability, or $-1\ and\ 1$ to allow a greater degree or range of non-probability for the output classes.

2.3.4 Types of Activation Functions

Rectified Linear Unit

The rectified linear unit (ReLU) is the most popular activation function in use for machine learning nowadays. The rectified linear unit is a simple half-wave rectifier, where any negative value returns 0 and any positive value is returned.

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \quad \textbf{(2-7)}$$

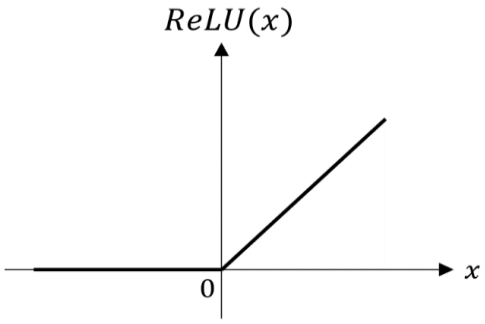

Figure 2-9: The rectified linear unit (figure by author)

The first general idea behind the ReLU was to mimic biology, as human neurons depending on their input decide to "fire" an output or not. For gradient descent purposes the derivative of the ReLU is often separately defined at the discontinuity. The ReLU activation function is very useful for regression [71] and provides better results than other activation functions [72]. Since neural networks have had a sudden boost in progress again, new research into ways of optimizing neural network structure is being conducted, like the activation functions used in a model.

Leaky Rectified Linear Unit

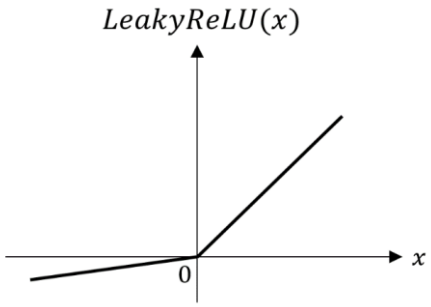

Figure 2-10: The leaky rectified linear unit (figure by author)

The leaky rectified linear unit is an adjustment of the ReLU for values of negative space. Any negative values are returned multiplied with a predefined factor a, usually a very small number.

$$leakyReLU(x) = \begin{cases} x, & x > 0 \\ ax, & x \leq 0 \end{cases} \quad \textbf{(2-8)}$$

This allows neurons to not completely turn off, leading to a less sparse neuron layout of the model.

Exponential Linear Unit (ELU)

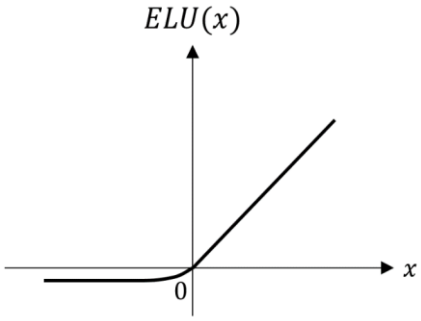

Figure 2-11: The exponential linear unit (figure by author)

$$ELU(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases} \quad \textbf{(2-9)}$$

The exponential linear unit behaves similar to the ReLU, but for negative values it is defined as an exponential, allowing for faster learning. A scaling factor α is introduced [73].

Hyperbolic Tangent

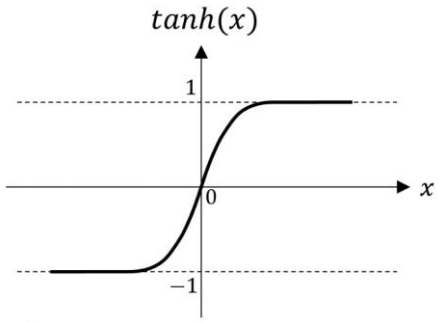

Figure 2-12: The hyperbolic tangent (figure by author)

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad \textbf{(2-10)}$$

The hyperbolic tangent activation function ranges from $-1\ to\ 1$. It is especially useful when a model is classifying between two classes.

Sigmoid

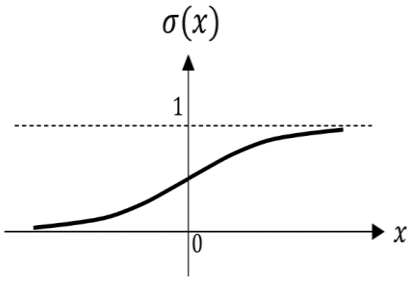

Figure 2-13: The sigmoid function (figure by author)

The sigmoid activation function returns a value between $0\ and\ 1$. Its main use is for classification, where the model must predict the probability of multiple classes/outputs.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad \textbf{(2-11)}$$

The sigmoid function is a very important function historically. It was the most widely used activation function for neural networks in general. Now it is mainly used for classification tasks or in neural network parts like the long short-term memory unit [42].

2.3.5 Weights and Biases

In a neural network, each unit is connected to a previous unit. These connections represent the flow of information inside the model. Depending on the type of neural network, the directions vary. For a simple feed forward neural network like the multilayer perceptron, information flows from layer to layer chronologically, that is from input to output. In the case of recurrent neural networks, the model works with sequential input arrays. In the case of its long short-term memory units, information from previous sequences can flow to previous layers of current sequences.

Each of the neurons in a neural network passes information to its next layer. When information is passed forward, a weight and a bias is attributed to that information. These two parameters are the main model parameters of a neural network and are adjustable by the model itself. The input to a neuron is processed as:

$$Output = f(\sum_{i} Input_i * w_i + b_i) \tag{2-12}$$

With $f(\cdot)$ denoting the activation function and w_i, b_i the weights and biases of each input $Input_i$, with i denoting the current input. Each neurons output, including the input layers neurons, are multiplied with a weight that represents their importance in the mapping of input to output for our neural network model. Same goes for the bias. The weighted sum itself is a simple linear function, allowing neural networks to propagate information quickly with efficient use of processing power. Furthermore, as will be discussed later, this simple expression allows for fast optimization through gradient based methods where calculations of derivatives play an important role.

2.3.6 Training

The basic principle of machine learning and neural networks is the idea that they are trainable with a dataset, that is, learn the patterns inherent to a dataset and through training the model is capable of discerning the important information of the dataset, while removing the unimportant information. The dataset represents the phenomena and its patterns and distinct information the model is supposed to approximate. The training of a neural network can occur supervised and unsupervised, although there is also semi-supervised learning as well. In an unsupervised training method, the dataset provided to the model during its training is not labeled. This means that while input data is provided, usually there is no specific output data labeling. In the case of regression this is useless, which is why unsupervised learning is generally used for classification tasks and its variations. Through unsupervised learning the model learns the dataset and its patterns, allowing for new observations that could not be observed by the engineer themselves, like clustering data points or reducing the input data sets dimensionality by discarding unimportant information and compressing the given information. In the supervised training method, our dataset is labeled correctly, that is each input is paired with an output and therefore guiding the model on the correct answer. The model then learns the desired pattern inherent to the paired dataset one wants to

approximate. The goal of supervised learning is to build a model capable of handling new input data, that was not previously part of the used training data, simply through training the model on patterns and structures inherent to the training data. Most use cases of neural networks are through supervised learning. Semi-supervised learning uses part unlabeled data, part labeled data. The idea is to alleviate the learning process through labeled data, allowing the model to have a small guiding hand during training while using the unlabeled data. This allows the use of huge unlabeled datasets to be used in combination with small or sparse labeled datasets to create a good model compared to just training a model on the sparse labeled dataset[74][70].

The main process of training a neural network with methods such as gradient descent, is done through a computational algorithm called automatic differentiation. Automatic differentiation allows for simple and efficient calculation of derivatives of the neural network model. This is done without relying on numerical approximations or manual derivations. Automatic differentiation is applied mainly to the loss function of the model however the process propagates through the whole model. The training of a neural network occurs over the whole training dataset however the data is sometimes split into parts. There are varying types of training algorithms used in machine learning that access automatic differentiation. More details on this topic are discussed later in this work, see section 2.3.8.

2.3.7 Loss Functions

A loss function represents the accuracy of a neural network to approximate a given dataset. Its value is the error between the model's output and the correct output of the dataset. The loss function and its loss serve as an observable metric of the model's accuracy to predict outputs given inputs. Depending on the model's task certain loss functions operate better than others. When designing the neural network, the loss function is an important aspect to consider. The goal of the training is to minimize this loss function, thereby also minimizing the difference between the model's output and the datasets output. As previously shown, by adjusting the model's parameters, the model converges towards a probability distribution equal to the probability distribution given by dataset. The loss function design follows statistical and information theory.

In the following types of loss functions, $\hat{y}$ denotes the model's prediction and y denotes the true output from the training dataset. Each loss is generally divided by the amount of data used, denoted as N. This is necessary when training occurs over multiple input/output pairs, called batches/minibatches [53][70].

Mean-Absolute Error (MAE)

A regression loss which considers the mean of the absolute residual of prediction and expected result. All predictions have even weight in this loss function. It is expressed as:

$$MAE = \frac{1}{N}\sum_{i=1}^{N} \hat{y}_i - y_i \qquad \textbf{(2-13)}$$

With function value y and model output $\hat{y}$. N denotes the sample size used.

Mean-Squared Error (MSE)

The mean-squared error is the most used loss function in the case of regression, as it has shown the best results, refer to the statistical framework discussed earlier. Compared to the mean absolute error, the mean-squared error considers the size of the residuals for each data sample and the model's prediction. The loss is the squared residual between predictions and expected results. The mean-squared error is applied to a factor of 0.5 to simplify its derivatives for the gradient descent optimization, called half mean-squared error, but simply referred to as the former. Through the squaring operation training data that produces greater residuals for the current model and its parameters have more weight/influence on the loss function and therefore during optimization. It is expressed as:

$$(H)MSE = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \qquad \textbf{(2-14)}$$

With function value y and model output $\hat{y}$. N denotes the sample size used.

Root Mean-Squared Error (RMS)

The root mean-squared error is an extension of the mean-squared error loss function., whereby the root of it is used instead. The root mean-squared error allows for a direct comparison between loss value and target output, as it has the same unit. This is done while retaining the weighting of bigger residuals from the mean-squared error. It is defined as:

$$RMS = \sqrt{\frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{N}} \qquad \textbf{(2-15)}$$

With function value y and model output $\hat{y}$. N denotes the sample size used.

Cross-Entropy

The cross-entropy loss function is mainly used for classification, see statistical framework discussed earlier. It is the difference of probability distributions $P\ and\ Q$, with P being the probability distribution of the training dataset and Q being the probability distribution of the model's predictions. The cross-entropy CE is defined as:

$$CE = -\sum P(x) * \log\left(Q(x)\right) \qquad \textbf{(2-16)}$$

Rewritten as a loss function for a neural network model, one gets:

$$CE = -\frac{1}{N}\sum_{i=1}^{N} y_i * \log(\hat{y}_i) \quad \textbf{(2-17)}$$

A loss function can be extended by a weight regularization term, also called weight decay. When a neural network model is trained with a dataset, just like with any data task, overfitting and underfitting are potential issues. In the case of neural networks, a model's parameters can become very big to fit the training data, resulting in instability of the model, also called overfitting. Just small changes in inputs can lead to great variation in outputs in such networks. To penalize these potential large weights and keep a neural network model simpler a weight regularization term is added to the loss function. Two wide known regularizations are l_1 and l_2. In the case of l_1 regularization weights tend towards 0 leading to sparse weight distribution. In a l_2 regularization large weights are penalized, however there is less sparse weight distribution. The terms include the sum of all model parameter weights. The l_2 regularization usually squares the weights to simplify the gradient calculation. l_1 and l_2 weight regularization terms can be expressed as:

$$L_{l1} = \frac{\lambda}{N} * \sum_{l=1}^{L}\sum_{i=1}^{I}\sum_{j=1}^{J}\left|w_{ij}\right|^{[l]} \quad \textbf{(2-18)}$$

$$L_{l2} = \frac{\lambda}{N} * \sum_{l=1}^{L}\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}^{2\,[l]} \quad \textbf{(2-19)}$$

The parameter λ is a predefined regularization factor. Every weight w_{ij} is defined by its connection to the previous layer and the following layer. Throughout this work, i denotes the neuron of the following layer and j denotes the neuron of the previous layer. Each weight of every layer l is then combined according to one of the above weight regularization equations. As can already be seen, denotations through sub- and superscripts become overwhelming when dealing with neural networks because of the great dimensionality introduced to the model parameters through multiple layers and samples. A set denotation will be discussed in following section 2.3.8. While the l_2 regularization has found general use for neural networks, which of the two regularization terms is used depends on the use case of the model and its dataset [53][55][57][70].

2.3.8 Training Algorithms

The process of training a model follows a predefined algorithm. The algorithm chooses how the model adjusts its model parameters to minimize the objective function, that is its defined loss function. Such algorithms can come with their own hyperparameters in addition to the models

own. By converging towards a minimum of the loss function, the training produces a potential array of the model parameters which approximate the correct output with very small error, meaning an optimal minimum of the loss function. The most used optimization process is the gradient descent method. Although there are other derivative-free methods, the gradient descent method remains quite popular. This lends itself to the automatic differentiation technique, allowing for very simple and quick neural network models even for large and complex data. The gradient descent algorithm faces issues like spurious local minima, where the trained model does not approximate well, and further training is not possible as the model parameters are stuck in the local minima, hence called spurious [76][77]. Still, some papers discuss this issue of local minima and their optimality for a neural network model [75][78][79][80][81][82][83]. Another issue are saddle points, since their gradients are zero as well, a model can also become "stuck" on them during the training process, leading to bad models [84]. However, there is research discussing these issues and potential solutions [85]. Other papers show local minima to be less prevalent in deeper neural networks. Research has also shown that local minima are less of an issue than the saddle points in the case of gradient descent algorithms. These saddle points have been shown to mostly "curve up" in most dimensions, with the rest "curving down" [10][86][87].

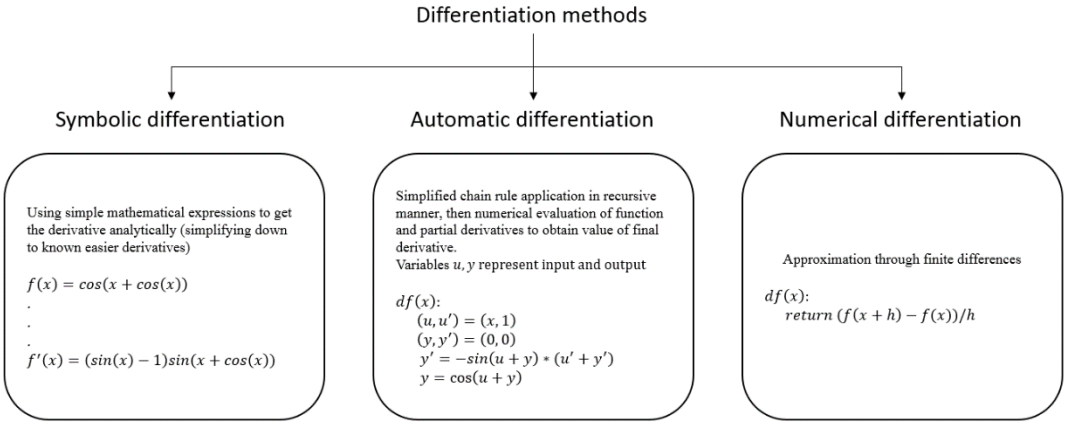

Figure 2-14: Generalized difference between the three methods of differentiation. Illustrated by using the function $f(x) = \cos(x + \cos(x))$ (figure by author)

The main technique used for training neural networks through gradient descent is called automatic differentiation [88][89]. Automatic differentiation is split into two modes which are forward and reverse mode. It is a technique to calculate all partial derivatives of a model with respect to a chosen model parameter. Automatic differentiation differs from the other methods of differentiation, namely symbolic and numerical. Figure 2-14 demonstrates the general difference between symbolic, numerical, and automatic differentiation. Numerical differentiation relies on the definition of the derivative and numerical calculation of the derivative by an approximation through numerical methods. Symbolic differentiation reformulates a derivative expression through simple and well-known derivatives and use of mathematical operations like the product rule to then calculate the derivative. Symbolic differentiation is considered slow and faces more issues when calculating higher and more complex derivatives compared to automatic differentiation. Same issues occur for numerical differentiation, in addition to computation inaccuracies.

Automatic differentiation combines aspects of symbolic and numerical differentiation. Its differentiation occurs recursively through the chain rule, whereby the observed function is simplified into multiple partial derivatives. This recursiveness is an important aspect of any programming task. Instead of manipulating symbolic expressions to simplify and express the observed function differently, to then derive the final derivative of said simplified function through known simple derivatives, automatic differentiation instead recursively derives all partial derivatives of the observed function and then evaluates the observed function and all its partial derivatives at specific points through use of numerical values. Thereby, the final derivative of the observed function is evaluated accurately and efficiently, combining the best aspects of symbolic and numerical differentiation. There are newer papers suggesting that symbolic differentiation and automatic differentiation are equivalent [90]. While the methods of symbolic differentiation and automatic differentiation are similar, the latter operates through numerical values whereas the latter through expressions/symbols, making automatic differentiation more efficient for programming and especially for machine learning gradient based methods where differentiation must be applied repetitively.

In both modes of automatic differentiation, the process occurs according to the chain rule of partial derivatives of a function. In forward mode, the partial derivatives of the model are calculated starting from the input and ending at the output. In the reverse mode it is the opposite way. The reverse automatic differentiation is widely called backpropagation. Backpropagation refers to the second phase of reverse automatic differentiation. In the first phase the model runs a given input forward through itself to populate parameters and trace dependencies. Once this is done, in the second phase the algorithm propagates all derivatives by use of chain rule or propagates the adjoints starting at the output up until the input. When it comes to deep neural networks with large datasets, reverse automatic differentiation is the preferred method over forward mode. With forward automatic differentiation the process of calculating the derivative by propagation with the chain rule needs to occur multiple times. Each independent input variable needs its own independent pass to calculate its partial derivatives. In contrast the reverse automatic differentiation only requires a forward pass and a single backpropagation pass to calculate all its partial derivatives, independent of the amount of input variables.

The following will briefly cover the equations for reverse mode automatic differentiation, which includes the forwards and backwards propagation during training. Keep in mind, these expressions may vary according to the defined model design. The following expressions will parallel the formulation used in the program of this work.

First, let us denote the following in Table 2-1:

Table 2-1: Variables and their definition (representation by author)

Variable	Denotes the following term
x	input
w	weight
b	bias
z	linear neuron output
$g(\cdot)$	activation function
a	non-linear neuron output
y	true output
$\hat{y}$	model output

Furthermore, the current layer of any of these will be denoted as a superscript inside []-brackets, not to be confused with the power function, with superscript $[l]$ denoting the current layer. Since each layer has its own predefined number of neurons, to discern variables shared between neurons of different layers they will be denoted with subscripts $i\ and\ j$, with the former referencing the current layer $[l]$ and the latter referencing the previous layer $[l-1]$:

$$w_{ij}^{[l]}, b_{ij}^{[l]}, z_i^{[l]}, a_i^{[l]}$$

To begin automatic differentiation, the algorithm forward propagates with the following definitions:

The activation function a of any layer l:

$$a_i^{[l]} = g(z_i^{[l]}) \tag{2-20}$$

With the linear neuron output z of any layer l:

$$z_i^{[l]} = \sum_j a_j^{[l-1]} * w_{ij}^{[l-1]} + b_{ij}^{[l-1]} \tag{2-21}$$

With $i = \{1, \dots, I\}$ and $j = \{1, \dots, J\}$. I is the number of neurons in layer l and J is the number of neurons in layer $l-1$. Additionally the output of the model $\hat{y}$ is defined as:

$$\hat{y} = a^{[L]} = z^{[L]} = \sum_j a_j^{[L-1]} * w_j^{[L-1]} + b^{[L-1]} \tag{2-22}$$

With $[L]$ denoting the final layer of the neural network, the output layer, where no activation function is applied to its linear term, although this can vary depending on the used programming library/program.

Once the forward propagation is done, the model has filled all necessary variables and traced them for the backpropagation. For a simple regression loss function defined as the half mean-squared error it follows:

$$L = \frac{1}{2N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2 \tag{2-23}$$

With $n = 1,2,\dots,N$ denoting the training sample. The model parameters are the weights w and biases b, which are expressed as Θ. The loss function L is minimized according to its argument Θ in the following expression:

$$\underset{\theta}{\operatorname{argmin}}\, L = \underset{\theta}{\operatorname{argmin}}\frac{1}{2N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2 \tag{2-24}$$

The partial derivatives $\frac{\delta L}{\delta \theta^{[p]}}$ which are calculated during backpropagation, are defined by the chain rule, with p being the layer of referred parameter Θ. The general expression is:

$$\frac{\delta L}{\delta \theta^{[p]}} = \frac{\delta L}{\delta \hat{y}}\frac{\delta \hat{y}}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}}\frac{\delta a^{[L-1]}}{\delta z^{[L-1]}} * \dots * \frac{\delta a^{[p+1]}}{\delta z^{[p+1]}}\frac{\delta z^{[p+1]}}{\delta \theta^{[p]}} \tag{2-25}$$

With layer $l = \{p, \dots, L\}$. All partial derivatives are then expressed according to the following general expressions. For the partial derivative of the loss L with respect to the model output $\hat{y}$, $\frac{\delta L}{\delta \hat{y}}$, we simply differentiate (2-14) and get:

$$\frac{\delta L}{\delta \hat{y}} = \frac{1}{N}(\hat{y} - y) \tag{2-26}$$

For the partial derivative of the model output $\hat{y}$ with respect to the activation function $a^{[L]}$, $\frac{\delta \hat{y}}{\delta a^{[L]}}$, the expression follows from the detail that no activation function is applied in this work on the last layer:

$$\frac{\delta \hat{y}}{\delta a^{[L]}} = 1 \tag{2-27}$$

For the partial derivative of the activation function $a^{[L]}$, with respect to the linear neuron output $z^{[L]}$, $\frac{\delta a^{[L]}}{\delta z^{[L]}}$, differentiating (2-20) gives:

$$\frac{\delta a^{[L]}}{\delta z^{[L]}} = g'\left(z^{[L]}\right) \tag{2-28}$$

with $g'(\cdot)$ being the derivative of the predefined activation function. For the partial derivative of the linear neuron output $z^{[L]}$ with respect to the previous activation function $a^{[L-1]}$, $\frac{\delta \hat{y}}{\delta a^{[L]}}$ it follows from (2-21):

$$\frac{\delta z^{[L]}}{\delta a^{[L-1]}} = \sum w^{[L-1]} \tag{2-29}$$

Subsequently, this can be applied to any point inside the neural network during backpropagation, that is any layer and any connection between neurons. These computations are done for each sample $n = \{1,2, \dots, N\}$ and standard chain rule procedure applies when partial derivatives split according to the neural network design. To clarify this point a simple neural network will be designed in the following paragraphs.

Lastly, the final partial derivative of the chain rule term in (2-25) is computed differently according to which model parameter the backpropagation currently occurs for, through differentiating (2-21) again.

In the case of the model parameter $\theta^{[p]} = w^{[p]}$ the final partial derivative is:

$$\frac{\delta z^{[p+1]}}{\delta \theta^{[p]}} = a^{[p-1]} \tag{2-30}$$

In the case of the model parameter $\theta^{[p]} = b^{[p]}$ the final partial derivative is instead:

$$\frac{\delta z^{[p+1]}}{\delta \theta^{[p]}} = 1 \tag{2-31}$$

The following equations will demonstrate forward and backward propagation for a simple regression neural network with 2 hidden layers, each with 3 neurons. The neural network will have 2 input neurons and 1 output neuron and is illustrated in Figure 2-15.

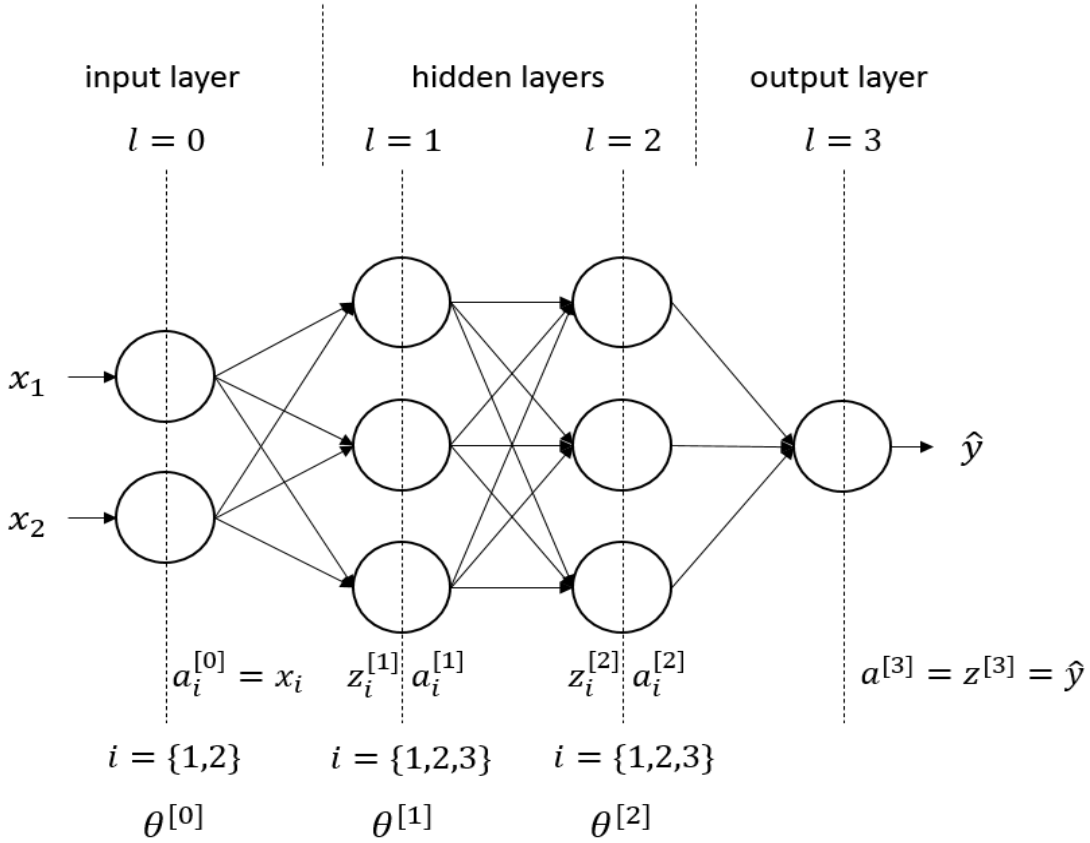

Figure 2-15: A feed-forward neural network model with 2 hidden layers with 3 neurons each. It has 2 inputs and 1 output. Neuron outputs are defined as a and the linear part of the neuron output as are defined as z (figure by author)

The parameters of the model between each layer, denoted by superscript, are $\theta^{[0]}, \theta^{[1]}, \theta^{[2]}$. They are defined as following:

$$\theta^{[0]} = \begin{Bmatrix} w_{11}^{[0]} & w_{21}^{[0]} & b_{11}^{[0]} & b_{21}^{[0]} \\ w_{12}^{[0]} & w_{22}^{[0]}, & b_{12}^{[0]} & b_{22}^{[0]} \\ w_{13}^{[0]} & w_{23}^{[0]} & b_{13}^{[0]} & b_{23}^{[0]} \end{Bmatrix} \quad \textbf{(2-32)}$$

$$\theta^{[1]} = \begin{Bmatrix} w_{11}^{[1]} & w_{21}^{[1]} & w_{31}^{[1]} & b_{11}^{[1]} & b_{21}^{[1]} & b_{31}^{[1]} \\ w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]}, & b_{12}^{[1]} & b_{22}^{[1]} & b_{32}^{[1]} \\ w_{13}^{[1]} & w_{23}^{[1]} & w_{33}^{[1]} & b_{13}^{[1]} & b_{23}^{[1]} & b_{33}^{[1]} \end{Bmatrix} \quad \textbf{(2-33)}$$

$$\theta^{[2]} = \begin{Bmatrix} w_1^{[2]} \\ w_2^{[2]}, b^{[2]} \\ w_3^{[2]} \end{Bmatrix} \quad \textbf{(2-34)}$$

Again, the subscripts attribute the weight to firstly the current layer neuron and secondly the previous layer neuron. This means that, for example, $w_{12}^{[0]}$ is the weight of the area between input layer and first hidden layer, which connects the first neuron of the first hidden layer with the second neuron of the input layer. Each neuron of the current layer is connected to each neuron of the previously layer. Each connection is accompanied with its own weight and bias. In this example the neuron count can be written down as 2x3x3x1, that is 2 input neurons, 3 hidden layer one neurons, 3 hidden layer two neurons, 1 output neuron. Therefore there is 6 weights and 6 biases for the first model parameters, $\theta^{[0]}$, 9 weights and 9 biases for the second model parameters, $\theta^{[1]}$. For the last model parameters, $\theta^{[2]}$, there are 3 weights, and the number of biases is reduced to 1 bias instead of 3 biases. This is because there is no activation function applied to the linear neuron terms, therefore the potential 3 biases can be summarized to 1 bias. The subscripts for the last model parameter also do not include the denotation for the output layer since there is only 1 output in this example.

First, according to the general equations above, beginning (2-20) for the forward propagation, the model equations are expressed as the following:

$$a_i^{[0]} = x_i, i = 1,2 \quad \textbf{(2-35)}$$

$$z_i^{[1]} = \sum_{j=1}^{2} a_j^{[0]} * w_{ij}^{[0]} + b_{ij}^{[0]}, i = 1,2,3;\ j = 1,2 \quad \textbf{(2-36)}$$

$$a_i^{[1]} = g\left(z_i^{[1]}\right), i = 1,2,3 \quad \textbf{(2-37)}$$

$$z_i^{[2]} = \sum_{j=1}^{3} a_j^{[1]} * w_{ij}^{[1]} + b_{ij}^{[1]}, i = 1,2,3;\ j = 1,2,3 \quad \textbf{(2-38)}$$

$$a_i^{[2]} = g\left(z_i^{[2]}\right), i = 1,2,3 \quad \textbf{(2-39)}$$

$$z^{[3]} = \sum_{j=1}^{3} a_j^{[2]} * w_j^{[2]} + b^{[2]}, j = 1,2,3 \quad \textbf{(2-40)}$$

$$a^{[3]} = z^{[3]} = \hat{y} \quad \textbf{(2-41)}$$

$$L = \frac{1}{2N} \sum_{n} (\hat{y} - y)^2, n = 1,2, \dots, N \quad \textbf{(2-42)}$$

Then it follows for backpropagation, that the partial derivatives are defined according to equation beginning (2-23):

$$\frac{\delta L}{\delta \hat{y}} = \frac{1}{N} (\hat{y} - y) \quad \textbf{(2-43)}$$

$$\frac{\delta \hat{y}}{\delta a_j^{[2]}} = \frac{\delta a^{[3]}}{\delta a_j^{[2]}} = \frac{\delta z^{[3]}}{\delta a_j^{[2]}} = w_j^{[2]}, j = 1,2,3 \quad \textbf{(2-44)}$$

$$\frac{\delta a_i^{[2]}}{\delta z_i^{[2]}} = g'\left(z_i^{[2]}\right), i = 1,2,3 \quad \textbf{(2-45)}$$

$$\frac{\delta z_i^{[2]}}{\delta a_j^{[1]}} = \sum_{j=1}^{3} w_{ij}^{[1]}, i = 1,2,3; j = 1,2,3 \quad \textbf{(2-46)}$$

$$\frac{\delta a_i^{[1]}}{\delta z_i^{[1]}} = g'\left(z_i^{[1]}\right), i = 1,2,3 \quad \textbf{(2-47)}$$

These expressions can then be used to formulate the partial derivative of any model parameter Θ. The final derivative term again varies depending on which model parameter, weight or bias, is referred to with the partial derivative. The following is an example for the first 6 weights connecting the input layer with the first hidden layer. For $w_{ij}^{[0]}$ the general equation is:

$$\frac{\delta L}{\delta w_{ij}^{[0]}} = \frac{\delta L}{\delta \hat{y}} \frac{\delta \hat{y}}{\delta a^{[2]}} \frac{\delta a^{[2]}}{\delta z^{[2]}} \frac{\delta z^{[2]}}{\delta a^{[1]}} \frac{\delta a^{[1]}}{\delta z^{[1]}} \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} = Path_1 + Path_2 + Path_3 \quad \textbf{(2-48)}$$

With $Path_1, Path_2, Path_3$ defined as:

$$\begin{aligned} Path_1 = \frac{1}{N}(\hat{y} - y) \\ * w_1^{[2]} * g'\left(z_1^{[2]}\right) * (w_{11}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} \\ + w_{12}^{[1]} * g'\left(z_2^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} \\ + w_{13}^{[1]} * g'\left(z_3^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}}) \end{aligned} \quad \textbf{(2-49)}$$

$$\begin{aligned} Path_2 = \frac{1}{N}(\hat{y} - y) \\ * \ w_2^{[2]} * g'\left(z_2^{[2]}\right) * (w_{21}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} \\ + w_{22}^{[1]} * g'\left(z_2^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} \\ + w_{23}^{[1]} * g'\left(z_3^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}}) \end{aligned} \quad \textbf{(2-50)}$$

$$\begin{aligned} Path_3 = \frac{1}{N}(\hat{y} - y) \\ * w_3^{[2]} * g'\left(z_3^{[2]}\right) * (w_{31}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} \\ + w_{32}^{[1]} * g'\left(z_2^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} \\ + w_{33}^{[1]} * g'\left(z_3^{[1]}\right) * \frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}}) \end{aligned} \quad \textbf{(2-51)}$$

With the final term $\frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}}$, the partial derivative of the linear neuron output of layer 1 with respect to the weights of layer 0, defined as one of the two inputs, depending on the connection:

$$\frac{\delta z^{[1]}}{\delta w_{ij}^{[0]}} = x_j \tag{2-52}$$

Visually, for $w_{11}^{[0]}$, the equation according to $Path_1 + Path_2 + Path_3$ can be seen in Figure 2-16, in green, red, blue respectively. For each partial derivative with respect to the chosen model parameter, the chain rule follows the structure of the neural network along the designed architecture. As per the rules of derivation, a summation of each derivative path occurs during computation.

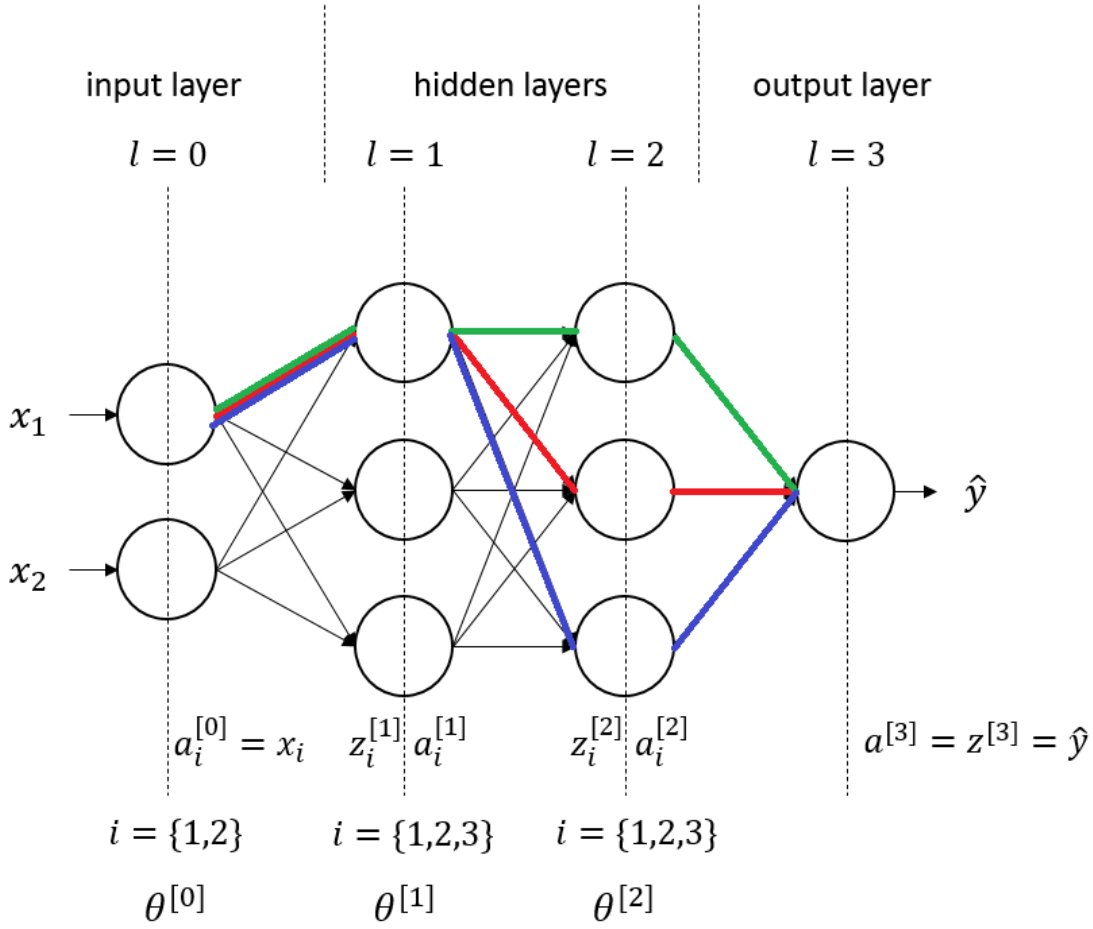

Figure 2-16: All 3 Paths for only $w_{11}^{[0]}$ shown with each distinct color (figure by author)

Through these equations, the neural network backpropagates to calculate all partial derivatives of the model with respect to the model parameters. These values are then used in the next step of the training algorithm, where a parameter update rule is defined to adjust the model parameters.

In the following, different types of training algorithms will be discussed. The algorithms are all based on the gradient descent method and are ordered chronologically to their discovery/creation. Other algorithms will not be discussed as derivative-free algorithms are very task specific

and do not make use of automatic differentiation, therefore being generally outperformed. Since this work will make use of minibatch stochastic gradient descent with momentum, other types of training algorithms will only be briefly shown, while the focus will be on the former and its predecessors.

Gradient Descent

The gradient descent algorithm calculates the direction for steepest decline of the model's loss function with respect to the model's parameters. Depending on the size of the neural network this can result in a gradient of great dimensions. The gradient is then used together with its hyperparameters like the learning rate assigned to the algorithm and the model's weights to update the current model parameters. The gradient descent algorithm is very suitable for convex optimization problems of a model loss function, where it is guaranteed to converge to the global minimum. In non-convex cases, the optimization is guaranteed to fall onto a local minimum. Other potential issues for converging towards a solution are saddle points, vanishing gradients and exploding gradients. As previously discussed, there is active research focused on tackling these issues, although there are already certain solutions available, like regularization or use of specific activation functions such as the rectified linear unit.

Batch Gradient Descent

Batch gradient descent is the most basic version of gradient descent, that introduces the definition of the training samples used to train a model as a batch. In this method the batch consists of our whole training dataset. During each training loop, called epoch, the batch is used for training. This means that the optimization calculations average over the batch, in this case all training samples. This leads to a redundancy, as for large datasets the optimization process is recalculating gradients for similar data samples before it updates the parameters. Depending on the size of the dataset this can be very time consuming. Furthermore, memory storage issues can arise depending on the datatype as well. Since many deep learning neural networks use great datasets, this method has fallen out of favor. The batch gradient descent is simply defined as:

$$\theta_{t+1} = \theta_t - \lambda * Grad_\theta(L) \tag{2-53}$$

With $Grad_\theta(L)$ being the gradients $\frac{\delta L}{\delta \theta}$ of the model's loss function L and λ the learn rate of the neural network. θ_{t+1} is the updated model parameter of sequence/iteration $t+1$.

Stochastic Gradient Descent

The stochastic gradient descent algorithm operates just like the batch gradient descent algorithm. The improvement lies in the difference of the parameter update. For this method, the parameter adjustment is calculated and done for each individual training data pair of input/output. Since this method does one update at a time, it performs the adjustment more frequently than batch gradient descent, however with greater variance. It doesn't have the redundancy problem of batch gradient descent. Furthermore its greater variance, allows the optimization of the loss

function to fluctuate out of potential local minima, while also increasing the difficulty of convergence to a precise minimum. To remedy this issue, the learn rate hyperparameter is usually slowly reduced to decrease the overshooting of the minimum. With (x_i, y_i) denoting the current training data sample of the training iteration, the stochastic gradient descent is expressed as:

$$\theta_{t+1} = \theta_t - \lambda * Grad_\theta(L(x_i, y_i)) \quad \textbf{(2-54)}$$

Minibatch Stochastic Gradient Descent

Minibatch stochastic gradient descent is a combination of the basic batch gradient descent and the stochastic gradient descent algorithms. For the calculation of parameter updates, the training data is shuffled and split into multiple equally sized pieces, called minibatches. The calculations and parameter adjustments are done for each minibatch instead. An iteration covers one minibatch and does one parameter update. Once all minibatches have been iterated through, one epoch has passed. Usually, the minibatches are shuffled after an epoch. This leads to a lower variance than in stochastic gradient descent, but it is still a faster and less redundant procedure than gradient descent. This method has been extended with an additional hyperparameter called momentum. The momentum is an added variable to help support the minibatch stochastic gradient descent algorithm to converge to a minimum. The hyperparameter momentum is used to adjust the model parameter update value. Mathematically, it adds a fraction of the parameter update vector of the previous minibatch update to the current one. This accelerates the optimization gradient in relevant directions in the loss function space and reduces fluctuation of the gradient. Minibatch stochastic gradient descent is defined as:

$$\theta_{t+1} = \theta_t - \lambda * Grad_\theta(\sum L(x_i, y_i)) \quad \textbf{(2-55)}$$

Considering the momentum η, the expression changes to:

$$\theta_{t+1} = \theta_t - \nu_t$$
$$\theta_{t+1} = \theta_t - (\eta * \nu_{t-1} + \lambda * Grad_\theta\left(\sum L(x_i, y_i)\right)) \quad \textbf{(2-56)}$$

With the subscript t denoting the current iteration of the training. The variable ν_t refers to the update vector adjusting the model parameter.

Nesterov Accelerated Gradient (NAG)

The Nesterov accelerated gradient method is an adjustment to the minibatch stochastic gradient descent method with momentum. By using an approximation of the location where the gradient would move the parameters to by using the momentum term, the model can adjust "ahead". The method then calculates the actual gradient not with respect to the parameters but to the approximated future parameters, $Grad_{\theta-\eta*v_{t-1}}$. This prevents the gradient from overshooting and increases the performance of neural networks through higher responsiveness to the data [91]. The equation then are defined as:

$$\theta_{t+1} = \theta_t - v_t$$
$$\theta_{t+1} = \theta_t - (\eta * v_{t-1} + \lambda * Grad_{\theta-\eta*v_{t-1}}\left(\sum L(x_i, y_i)\right)) \quad \textbf{(2-57)}$$

Adagrad

The Adagrad method can adjust the learning rate used for the optimization of the model parameters depending on the frequency of features inside the dataset. If the current minibatch contains features that occur less frequently the model updates parameters with a greater learning rate compared to a minibatch containing features that occur more frequently. This makes the Adagrad method well suited for sparse data but generally improves the training. However, its learning rate vanishes with long training time as it collects all previously calculated gradients to adjust each individual learning rate in its denominator. The final equation is defined as:

$$\theta_{t+1} = \theta_t - v_t$$
$$\theta_{t+1} = \theta_t - \left(\frac{\lambda}{\sqrt{diag(G_t) + \varepsilon}}\right) \odot g_t \quad \textbf{(2-58)}$$

The variable ε is a smoothing term to prevent dividing by zero. G_t is a matrix where the diagonal contains the sum of the squared past gradients with respect to θ up to iteration t, g_t is a vector containing the past gradients with respect to θ up to iteration t. The equation performs an element-wise matrix-vector product $' \odot '$. While the general equation uses the whole matrix G_t, a simplified version only applies the diagonal elements of G_t, where each diagonal element of G_t refers to the sum of the squared gradients with respect to θ of the respective model parameter θ. In this simplified equation we simply have an element wise vector product, also called Hadamard product, between the elements of $\frac{\lambda}{\sqrt{diag(G_t)+\varepsilon}}$ and the elements of g_t.

Adadelta

The Adadelta method is an adjustment to the Adagrad method to eliminate the vanishing learn rate. This is done by restricting the number of gradients accumulated in the denominator to a defined amount. The sum of gradients is instead defined as a decaying average of all previous squared gradients. Its final equation is defined as:

$$\theta_{t+1} = \theta_t + v_t$$
$$v_t = -\left(\frac{RMS(v_{t-1})}{RMS(g_t)}\right) * g_t \quad \textbf{(2-59)}$$

With RMS being the root mean-square and g_t a vector containing all past gradients with respect to θ up to iteration t.

Adam

The adaptive moment estimation method also adjusts learning rates for each individual parameter. It does so by storing the exponentially decaying average of past squared gradients and past gradients. The Adam method has been shown to work well and compared to other methods [92]. The Adam update equation is defined as:

$$\theta_{t+1} = \theta_t - \left(\frac{\lambda}{\sqrt{\hat{v}_t} + \varepsilon}\right) * \hat{m}_t \quad \textbf{(2-60)}$$

With $\hat{m}_t$ being the bias-corrected first moment estimate:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} \quad \textbf{(2-61)}$$

And the decaying average of past gradients m_t:

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t \quad \textbf{(2-62)}$$

Where β_1 is the first decay rate and g_t is the past gradients with respect to θ. Analog for the bias corrected second moment estimate $\hat{v}_t$:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \quad \textbf{(2-63)}$$

And the decaying average of past gradient v_t:

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2 \quad \textbf{(2-64)}$$

Where β_2 is the first decay rate and g_t^2 is the squared past gradient with respect to θ. There is of course many more types of training algorithms used for neural networks such as AdaMax [92], Nadam [93]. There is also new methods being researched like AdamW [24], QUAdam [94], AggMo [95].

2.3.9 Data Preprocessing

When training a neural network an important part of the process is the data used. The dataset contains information that the model is supposed to learn. Therefore, the more data there is the better. However, if our data does not include many examples of certain features, the model will not be able to learn those features well enough. Furthermore, the data must include the underlying pattern the model is supposed to approximate. The data itself is split into three parts of training, validation, and testing data. However, usually data is only split into training and testing. The training data is used to train the model using the algorithm chosen for the neural network. The training data should include a wide array of examples expressing the information the model should learn. The validation data is used to adjust hyperparameters in between training, as they are not adjustable by the training algorithms. These hyperparameters are hand engineered to improve the training. The testing data is used in the final step of the training. While the model has been trained on the training data it is not supposed to have seen the testing data once. The testing data allows a blind test on the trained model. By evaluating how our trained model performs on data expressing the same information as the training data, but having never integrated this data into its optimization, it is possible to grade the trained neural network. The ratio of the data split into training and testing is predefined like other hyperparameters. Usually, the training data is the biggest data set to cover a wide range of information [53].

Even more important is the preparation of the data for training. While some papers suggest that there is no need to pre-process the dataset used for training [96][97], generally datasets are prepared by cleaning up strong outliers and non-processable data like values of 0. Furthermore, normalization and standardization are important to for example hinder exploding gradients which lead to a model not converging [53][70]. With training data of input x_i and output y_i for N samples, for standardization, the mean μ, standard deviation σ, and the final processed data x_s, y_s, are as follows to basic standardization formula:

The mean μ_x of training data x and the mean μ_y of training data y are expressed as the sum of all samples x_i, y_i divided by the sample size N:

$$\mu_x = \frac{\sum x_i}{N} \quad \textbf{(2-65)}$$

$$\mu_y = \frac{\sum y_i}{N} \tag{2-66}$$

The standard deviation of training data x and y, σ_x and σ_y are expressed as the square root of the sum of squared difference of each sample x_i, y_i to its mean μ_x, μ_y divided by $N - 1$:

$$\sigma_x = \sqrt{\frac{\sum (x_i - \mu_x)^2}{N - 1}} \tag{2-67}$$

$$\sigma_y = \sqrt{\frac{\sum (y_i - \mu_y)^2}{N - 1}} \tag{2-68}$$

And lastly it then follows for the standardized training data x_{is} and y_{is}, where the mean μ_x, μ_y is subtracted from each sample x_i, y_i and then divided by the standard deviation σ_x, σ_y:

$$x_{is} = \frac{x_i - \mu_x}{\sigma_x} \tag{2-69}$$

$$y_{is} = \frac{y_i - \mu_y}{\sigma_y} \tag{2-70}$$

2.4 Neural Network Enhancements

The enhancement of neural networks seeks to improve current structures of models, by extending or adding new parts to the model or replacing them completely. There are various types of enhancement for various types of neural networks. This can often include the combination of two neural networks, like for the first image recognition neural networks where a convolutional neural network is coupled with a recurrent neural network allowing therefore splitting the main task of image recognition into smaller tasks. In this case the convolutional neural network extracts the features from the data array, that is, the image, and the recurrent neural network transforms this information into something useable like a sentence describing the image. In this process the convolutional neural network would be the encoder and the recurrent neural network the decoder. Other enhancements may adjust or replace subparts of a neural network like its layers, neurons, activation function or loss function. Certain enhancements are specific to certain use cases, where

the enhanced neural network does perform better for its inherent task but is not applicable to other tasks [98][99].

In general, neural network enhancements allow for increasing speed and accuracy during training, reduced processing power necessary, increased information extraction or noise reduction of the given dataset, etc. The multiple training algorithms that have been developed can be considered such enhancements. The introduction of minibatches, decay of learn rate and others are enhancement methods that are applicable to any neural network based on gradient descent optimization. Similarly residual networks which apply skips to the backpropagation during the optimization to mend issues like vanishing gradients, can be applied to a multitude of neural networks, even though it is mainly used for image classification. There are many more enhancements for general use like Gated Recurrent Units [100], Long Short-Term Memory [65], Dropout [101], Batch Normalization[16], Ensemble Learning [102] and many more that improve certain capabilities of neural networks. The following sections will briefly cover physics informed neural networks and its gradient enhanced variation. There are also Sobolev trained neural networks and finite-element-informed neural networks, which all operate in a similar manner of loss function expansion[47][103].

2.4.1 Physics Informed Neural Network (PINN)

A physics informed neural network, called PINN, is an enhanced regression neural network model that incorporates partial differential equations (PDE) governing the dataset that the model is supposed to approximate. A simplistic sketch of a PINN can be seen in Figure 2-17, which shows the PINN being an expansion of the basic neural network design. In addition to the basic loss function of a neural network, like the mean squared error, the loss is extended in a PINN by the partial differential equations:

$$L_{PINN} = L_{Total} = L_M + L_F + L_C \tag{2-71}$$

Where the total loss L_{Total} is the sum of the basic neural network model loss L_M, the loss of the residual of the partial differential equation L_F and the loss of the boundary and initial constraints L_C. By formulating a residual of this PDE, where the output of the neural network is a parameter inside of the equation, the model is being constrained by the PDE. In addition to the PDE, initial and boundary constraints can be added as additional loss expressions of residuals. Doing so allows a PINN to outperform its basic neural network counterpart in multiple ways.

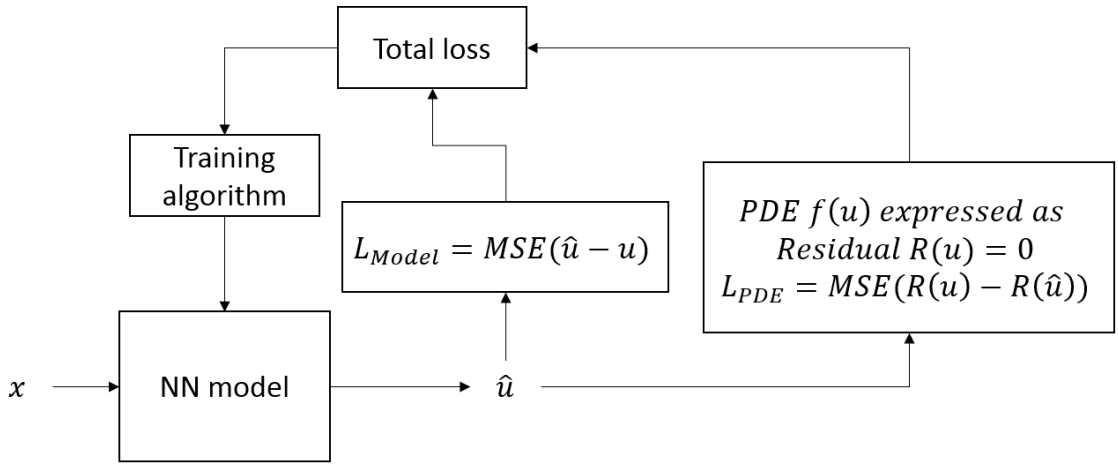

Figure 2-17: A Simple structure of a PINN. The PDEs are simply an extension to the basic neural network model, specifically the loss function. Denoted are the model output $\hat{u}$ and the true output u and the PDE residual $R(\cdot)$ (figure by author)

For one, a PINN is more accurate with the same dataset. Additionally, the PINN is able to approximate faster, potentially being able to achieve the same results with less data as the basic neural network with more data. They offer a great improvement when trying to analyze engineering problems of thermodynamics, structural integrity, etc. As these problems are always governed by PDEs, introducing them to the neural network that approximates solutions for those problems improves them. PINNs face the issue of these governing PDEs, and the initial and boundary conditions being formulated and integrated into the neural network. Depending on the complexity of the problem and its solution, including the equations governing them into the loss function can be quite a difficult task[46].

2.4.2 Gradient-Enhanced Physics Informed Neural Network (gPINN)

The gradient enhanced physics informed neural network, gPINN, is an enhancement of the PINN, see Figure 2-18. Similar to the PINN, the gPINN incorporates partial differential equations in the neural network optimization of its loss. In addition to the basic partial differential equations governing the physics of system, the gPINN also makes use of the derivatives of those partial differential equations and incorporates them in the loss in the form of residuals as well:

$$L_{Total} = L_{PINN} + L_G \tag{2-72}$$

Where L_G is the loss of the residual of the gradient of the original partial differential equation that governs the physics of the observed system. By including a constraint on the model's output

through incorporation of the derivative of the partial differential equations which govern the physical system of the data the model is supposed to approximate, the neural network performs generally better.

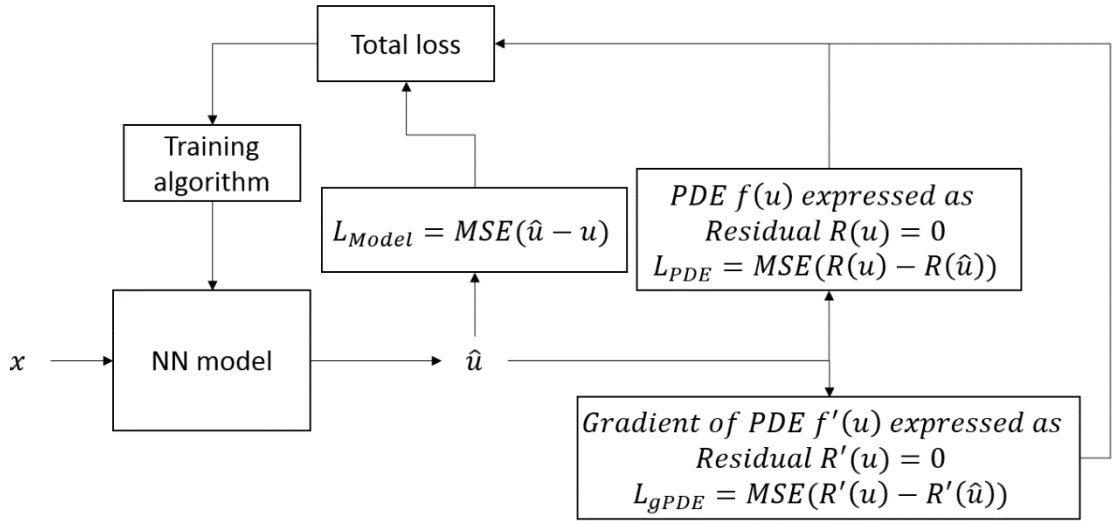

Figure 2-18: The gPINN is an expanded PINN structure. The additional loss of the gradient of the PDE is now added to the total loss (figure by author)

The gPINN is faster and more efficient than the base PINN, which is already an improvement to base neural networks. However just as it is the case with PINNs, gPINNs face issues when it comes to formulating the partial differential equations. On top of this, the derivatives of the partial differential equations need to be formulated as well. In the case of complex systems with great dimensionality, this can be a very difficult task [48].

2.5 Finite Element Method, Sensitivity Analysis and Uncertainty Quantification

2.5.1 Finite Element Method

The finite element method is a numerical method employed to approximate models of different fields of engineering. In this method, the system to be analyzed is divided into multiple smaller and simpler parts. These parts are called finite elements. These are usually triangles or quadrangles for 2D systems and tetrahedra or hexahedra in the case of 3D systems. Because of their simple design it is possible to numerically solve the weak form of equilibrium equation governing the physics of a problem through use of shape functions. The weak form of equilibrium in structural mechanics is generally defined as:

$$R(u;v) = \int P^K(u) : Gradv\ dV - \int b_0 * v\ dV - \int t_0 * v\ dA = 0 \qquad \textbf{(2-73)}$$

Where P^K is the Piola-Kirchoff stress tensor. Depending on the use case, other stress tensors can be used. The vectors b_0 and t_0 are the body and traction forces. The displacement field is denoted as u and the test function as v. In this equation the forces of internal and external nature should equalize, turning the residual to zero. Through use of various test functions v, the finite element model can find a solution for the displacement field u. The purpose of the weak form, compared to the strong form, is to simplify the equilibrium equation for easier approximation through the finite element model. For a more detailed explanation see the referenced paper [51].

The process of dividing a system into finite elements happens through use of a defined discretization in space of the object to be analyzed. This yields a mesh of the object made up of all the finite elements with multiple equations and boundary conditions for each finite element. These equations are then combined to create a general set of equations, the finite element model, which approximates a solution of the original object through numerical methods. The finite element method is a widely used tool. The method allows for the approximation of many systems by simplification into finite elements. Since numerical methods of equation solving are fast, the finite element method allows for quick and precise solution finding. They can solve linear and nonlinear problems, as well as time dependent problems. The difficulties of a finite element model are various. For one the generation of a mesh through discretization can prove difficult. Depending on the complexity of the original object, finding a mesh that accurately represents it while minimizing the number of finite elements is a balancing task. The more finite elements there are, the more nodal points exist and need to be considered in the calculations, leading to longer processing time. Another issue is numerical instability of numerical methods [104][49].

2.5.2 Sensitivity Analysis

Sensitivity analysis concerns itself with methods to discern the rate of change of mathematical models given variations of their governing parameters. In other words, the goal of such analysis is to understand how changes in certain values like input values affect the output of a model. In a wide range of engineering problems understanding the root of the difference in a model's output compared to the desired output can help fine tune a model. In this regard, sensitivity analysis is most useful to quantify the effect of aleatoric or epistemic uncertainty. Sensitivity analysis is also used as part of uncertainty quantification [105]. This can be applied onto a finite element model. To analyze the response sensitivity of such a model, it is possible to determine the change of the residual under certain parameter changes. The following equation defines multiple properties:

$$\begin{aligned} \delta R &= \delta_u R + \delta_s R + \delta_{h_n} R = 0 \\ &= k(v, \delta u) + p(v, \delta s) + h(v, \delta h_n) \end{aligned} \tag{2-74}$$

Where the slight variations of the weak form of the equilibrium are summarized as variations through changes in displacement $k(v, \delta u)$, design parameters $p(v, \delta s)$ and deformation history $h(v, \delta h\_n)$ respectively. For a more detailed explanation on the response sensitivity calculation, see the referenced source [51].

2.5.3 Uncertainty Quantification

Uncertainty quantification is the analysis of uncertainty in data, systems, and models. In any real-world event or model prediction there is a degree of uncertainty because of unknown parameters, bias, etc. These are divided into aleatoric and epistemic uncertainty, although a combination of both is usually the case. Aleatoric uncertainty describes the inherent randomness of any given system that is being approximated. Random variations such as process fluctuations or errors in measurements are some examples of aleatoric uncertainty. Epistemic uncertainty covers uncertainty of unknown nature. Any information not available to the model which approximates a system is epistemic. In contrast to aleatoric uncertainty, epistemic uncertainty can be reduced through better datasets, better models, etc. Uncertainty quantification tries to understand the likelihood of certain events to occur under these uncertainties and therefore quantify them. To quantify uncertainty there needs to be a considerable amount of data to extract the necessary information. This information is synthesized into probability distributions, that characterize the unknown parameters of a certain model. Then, it is necessary to propagate this uncertainty through the model, to determine the uncertainty associated with the response of a given model. This step is usually known as "forward uncertainty propagation". One of the mainly used methods for this forward uncertainty propagation is the Monte Carlo simulation. The Monte Carlo method is a technique to create many random numerical results of a system through repeated simulation for different realizations of the uncertain parameters of a model. The general pattern of the Monte Carlo simulation is to define a domain of potential inputs via its probability distribution and generate outputs. This propagation of the uncertainty is done through a deterministic method like a finite element model, see . The Monte Carlo method is considered very time consuming, especially with large, complex systems and technically its results are only approximations. When trying to quantify such systems, acquiring the necessary data for this through Monte Carlo can turn into a bottleneck [106]. Other techniques are the Latin Hypercube Sampling, which is a form of Monte Carlo Simulation, and Polynomial Chaos Expansion [107]. The produced output samples allow us to analyze them statistically with various methods. For example, to calculate the mean μ of the response of a model, the equation follows as:

$$\mu = E\{r\} \approx \frac{\sum_{i=1}^{N} r_i}{N} \tag{2-75}$$

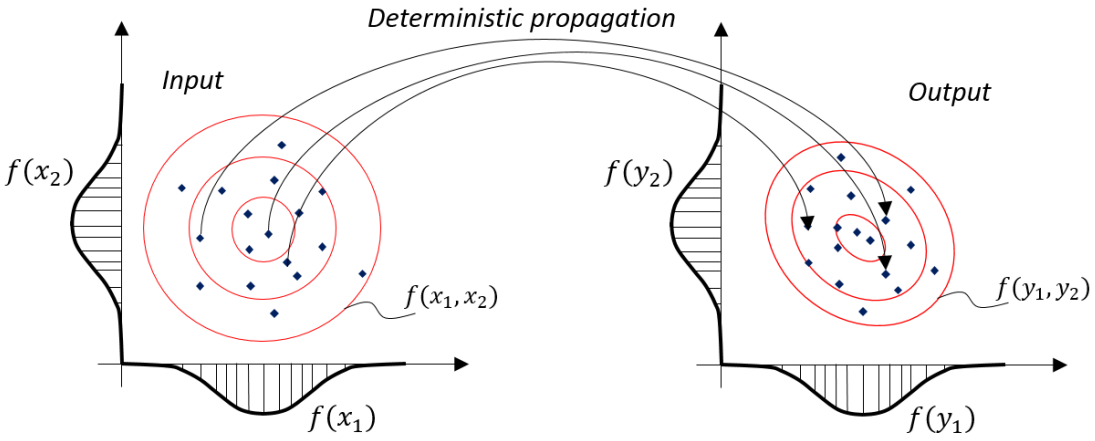

Figure 2-19: Propagation of uncertainty through a deterministic method. The input parameter x defines the response y through the probability distributions $f(x_i)$.The deterministic method is creating samples of response to generate probability distributions $f(y_i)$ (figure by author)

With $E\{r\}$ being the expectation of the response r and N being the number of samples. The variance $Var\{r\}$ of the response r, or squared standard deviation σ^2, is expressed as:

$$Var\{r\} = \sigma^2 = E\{(r-\mu)^2\} \approx \frac{\sum_{i=1}^{N}(r_i-\mu)^2}{N-1} \tag{2-76}$$

Another metric to analyze data is the probability of exceedance, also called probability of failure. The probability of exceedance compares various metrics of analyzed data to glean if a defined value of failure will be exceeded and with which probability. The probability of exceedance can be observed visually by plotting all generated samples sorted over a probability axis from $0\ to\ 1$ and marking the value of exceedance, for example the mean of the generated data together with its variance, and then noting the intersection. The probability of exceedance can be calculated through integration as well. This can be done analytically, numerically, or through other methods like a cumulative distribution function [108][109]. Formally, the probability of a model's response r exceeding a predefined threshold b is defined as:

$$P\{r \geq b\} = \int_{\theta\in\Theta} I(r(\theta) \geq b) f_{\Theta}(\theta) d\theta \approx \frac{1}{N}\sum_{j=1}^{N} I\left(r\left(\theta^{(j)}\right) \geq b\right) \tag{2-77}$$

Where θ denotes the uncertain input parameters of a model, which belong to a set Θ and whose uncertainty is characterized by the joint probability distribution $f_{\Theta}(\theta)$ and $I()$ is an indicator function whose value is equal to 1 in case that the expression contained in parentheses is true and zero, otherwise. Additionally, $\theta^{(j)}, j = 1, \dots, N$ denoted N independent, identically distributed samples of θ distributed according to $f_{\Theta}(\theta)$.

3 Methodology

During this chapter the general question of how and why concerning gradient data and its incorporation in neural networks is addressed. In the following sections, the theory and methodology behind the gradient enhanced neural network are discussed. A brief overview of how to take advantage of gradient data during the training of a neural network is discussed considering two different variations, including advantages and disadvantages of the two mentioned variations. The final used variation in this work will be elaborated and current state of the art is explored. In addition expansion on this current state of the art is discussed as further motivation and reason of this work.

3.1 Research goal

The goal of this work is to take advantage of additional gradient information of a given dataset during the training of a neural network and compare its performance with its base neural network counterpart. Many neural networks still do not employ the use of gradient information for training. For one, computing additional gradient information of a system to be observed can be quite costly depending on the method used. Secondly, the difference of scale and magnitude between function value and gradient value can lead to numerical issues when training a neural network. There have been a handful of papers [111][112] in which the use of gradient information has been exploited for better performance of the neural network. In this paper, the research on this topic will be expanded, by using a variety of design hyperparameters, that is, varying designs of a neural network referring to its layers, neuron count and others. The observations are done using data generated by a finite element model using sensitivity analysis, instead of simple academic function evaluations. In cases where gradient information is readily available, applying it to the training of a neural network should lead to performance improvement. Through specific numerical methods, gradient information can be obtained for certain cases. Sensitivity analysis, previously discussed in section 2.5.2, computes this gradient information at a relatively efficient numerical cost, thus opening a potential avenue to train neural networks by taking advantage of this available gradient information. The gradient information will be applied during training of the neural network similar to Sobolev-Training [111], whereby the neural networks output and its derivative with respect to input are optimized via an expanded loss function. In this work specifically, the data used for the training of the neural network is generated by a finite element model for linear and nonlinear elasticity through sensitivity analysis, thereby applying the neural networks observed in this work to linear and nonlinear mechanics. Any PDEs governing the system will not be used for training, making any performance strictly data based. The neural networks are also compared to the finite element model for accuracy and processing time, to gauge if replacing the finite element model with the gradient enhanced neural network is viable. This goal is particularly relevant for uncertainty quantification, where uncertainty needs to be propagated and replacing the finite element model with a gradient enhanced neural network could potentially solve issues such as long processing times. These comparisons are applied to different designs of neural networks. Through a

variation of predefined hyperparameters, like training data size, layer number, neuron number, etc., results will be compared. Lastly, the issue of weighting the different losses will be explored. As the loss function will be expanded with additional addends and these addends contain different information, function and gradients respectively, each of these loss addends have varying importance for the optimization of the neural network during training. Therefore, each loss should be weighed differently. In a previous paper [112], weights had been applied to the gradient information addends in the loss function of a Sobolev-Training model. In short, the weight of the loss addends containing the gradient information was linearly increased over the training time or to slowly introduce the gradient information to the training. The target in this work will be instead, to analyze various methods of solving this weighting problem instead of a simple linear increase of weighting, whereby the weighing of the gradient loss will slowly increase from 0 to 1. This will be done by adjusting the redefined loss function of the gradient enhanced neural network, whereby a weight factor will be added to the respective loss terms in the model loss function and varied through various methods. First, fixed weights will be observed to understand correlations of weighing to performance of the models. Secondly, dynamic weighing will be experimented with, whereby the gradient loss weight will be linearly decreased from 1 to 0 and another variation whereby the weight of the output loss will be linearly increased from 1 to 2. Another last variation of dynamic loss weights will consider weights for the model output loss and the gradient loss, which are expressed as cumulative distribution functions to bound the weight values. The weights will then be trained by the model itself for optimization.

3.2 Gradient Enhanced Neural Network Model

When given a dataset containing function values mapped to specific inputs, a neural network can approximate the probability relationship between these inputs and outputs. Through expressing the approximation error of our model through a loss function and using and optimization algorithm to minimize it in relation to the model parameters, the models can obtain accurate output capabilities mirroring the presented data. When given additional information of any system, optimization tasks can become easier. However this can also have the opposite effect of turning the optimization problem very complex and thus reducing optimization performance. When trying to take advantage of gradient data in neural networks, this needs to be kept in mind. Whatever method is applied, needs to reduce complexity, and increase performance.

There is a variety of ways to include gradient data with respect to the inputs in a neural network. For one, the model could be expanded to include an additional output which would hold the gradient information, thereby forcing the model to also approximate the gradient. This would be a simple neural network with bigger dimensionality. The second variation is to expand the loss function of the model, by including an additional loss addend which holds the gradient information, thereby taking advantage of the gradient information during the training process. This would differ from the general design of loss functions, where usually only the output residual is computed including any regularization addend. In the following, both variations will be analyzed to determine which variation is better.

3.3 First Variation – Expanding Model Output

The first variation of expanding a model's output size is just a simple neural network, whereby none of the previously equations change drastically. The only difference is that now there are additional loss addends for each additional output. In the case of expanding the outputs of the model, thereby increasing the output layer neurons from for example 1 to 2, the model would approximate the output values y_1, y_2. In this specific case of function values and gradient data, y_1 would be the function value and y_2 the gradient value, as can be seen in Figure 3-1.

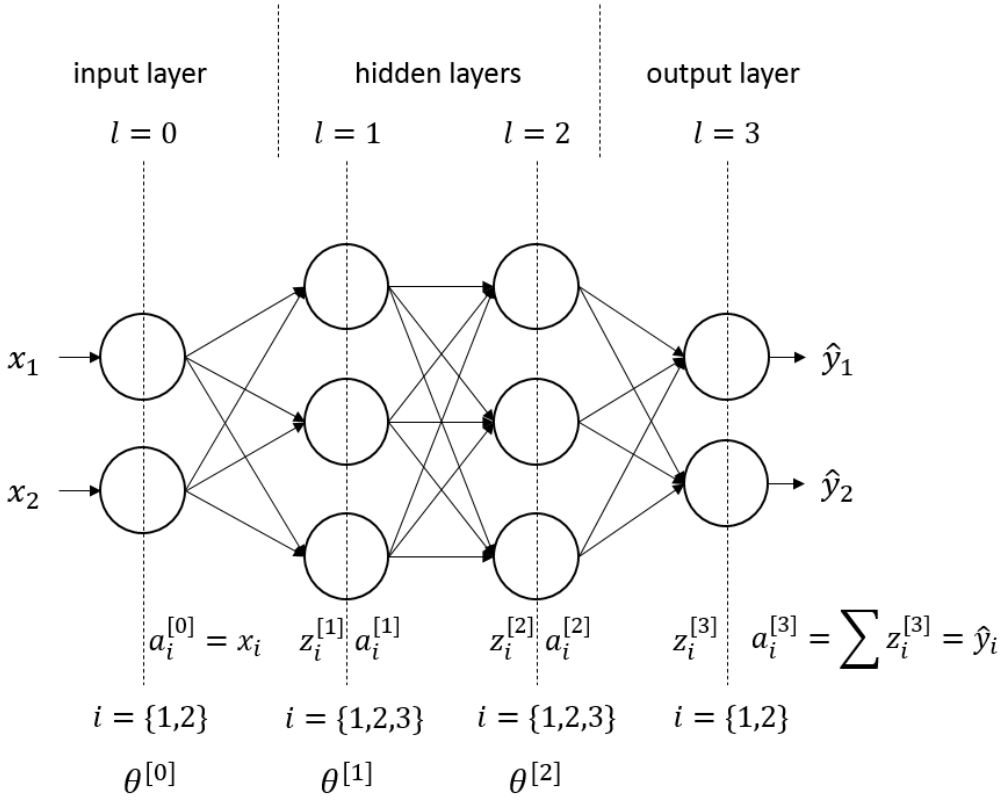

Figure 3-1: Previous example of a neural network expanded to have 2 outputs (figure by author)

This is usually called a multi-output neural network, where the vector of model parameters $\theta = \{w, b\}$ would be adjusted to approximate the training data of y and $\frac{\delta y}{\delta x}$. The loss L would then be a simple addend expansion of the basic neural network loss defined in (2-23) , now expressed as:

$$L = \frac{1}{2N}\sum_{n}(\hat{y}_1 - y_1)^2 + \frac{1}{2N}\sum_{n}(\hat{y}_2 - y_2)^2 \quad \textbf{(3-1)}$$

As will be seen later, when the partial derivatives of the used gradient enhanced model are explored in this section, this gradient computed through automatic differentiation for this loss function is different. With applying the chain rule to this loss, its gradient would be expressed as:

$$\begin{aligned}\frac{\delta L}{\delta \theta^{[p]}} = \frac{\delta L}{\delta \hat{y}_1}\frac{\delta \hat{y}_1}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}}\frac{\delta a^{[L-1]}}{\delta z^{[L-1]}} * \dots * \frac{\delta a^{[p+1]}}{\delta z^{[p+1]}}\frac{\delta z^{[p+1]}}{\delta \theta^{[p]}} \\ + \frac{\delta L}{\delta \hat{y}_2}\frac{\delta \hat{y}_2}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}}\frac{\delta a^{[L-1]}}{\delta z^{[L-1]}} * \dots * \frac{\delta a^{[p+1]}}{\delta z^{[p+1]}}\frac{\delta z^{[p+1]}}{\delta \theta^{[p]}}\end{aligned} \quad \textbf{(3-2)}$$

$$, with\ l = \{p, \dots, L\}$$

With p denoting the layer of the parameter θ with respect to which the derivative is computed. Here the exact same dependencies can be observed for both partial derivative terms.

In MATLAB, neural networks are designed by the following general expressions:

$$\frac{\delta \hat{y}_i}{\delta a^{[L]}} = 1 \quad \textbf{(3-3)}$$

$$\frac{\delta a^{[L]}}{\delta z^{[L]}} = 1 \quad \textbf{(3-4)}$$

The only difference therefore between the two loss derivatives $\frac{\delta L}{\delta \hat{y}_1}\frac{\delta \hat{y}_1}{\delta \theta^{[p]}}$ and $\frac{\delta L}{\delta \hat{y}_2}\frac{\delta \hat{y}_2}{\delta \theta^{[p]}}$ would be the terms $\frac{\delta L}{\delta \hat{y}_1}\frac{\delta \hat{y}_1}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}}$ and $\frac{\delta L}{\delta \hat{y}_2}\frac{\delta \hat{y}_2}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}}$, expressed as:

$$\frac{\delta L}{\delta \hat{y}_1}\frac{\delta \hat{y}_1}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}} = \left(\frac{1}{N}\sum_{n}\hat{y}_1 - y_1\right) * w_{1j}^{[L]} \quad \textbf{(3-5)}$$

$$\frac{\delta L}{\delta \hat{y}_2}\frac{\delta \hat{y}_2}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}} = \left(\frac{1}{N}\sum_n \hat{y}_2 - y_2\right) * w_{2j}^{[L]} \quad \textbf{(3-6)}$$

In the case of $p = (L-1)$, the expressions change to:

$$\frac{\delta L}{\delta \hat{y}_1}\frac{\delta \hat{y}_1}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta \theta^{[p]}} = \left(\frac{1}{N}\sum_n \hat{y}_1 - y_1\right) * a_j^{[p]} \quad \textbf{(3-7)}$$

$$\frac{\delta L}{\delta \hat{y}_2}\frac{\delta \hat{y}_2}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta \theta^{[p]}} = \left(\frac{1}{N}\sum_n \hat{y}_2 - y_2\right) * a_j^{[p]} \quad \textbf{(3-8)}$$

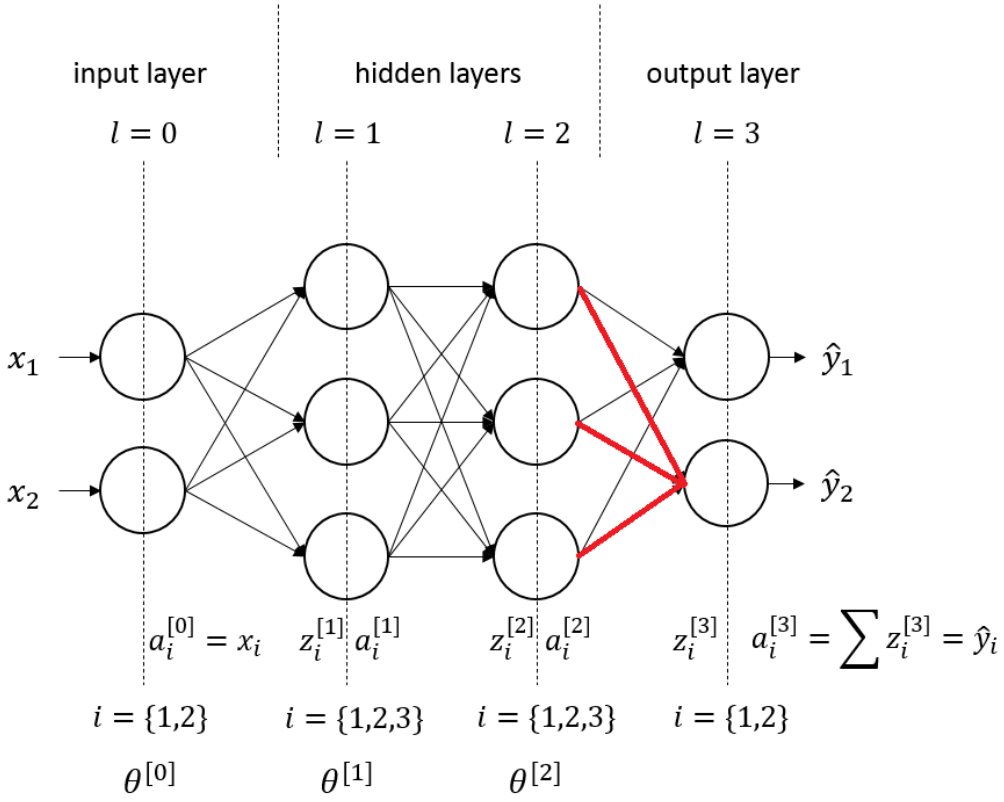

Figure 3-2: Red marked lines are newly introduced connections in the neural network (figure by author)

As can be seen visually in Figure 3-2, when the previous neural network is expanded to now have 2 outputs, there is new connections added only between the last hidden layer and the output layer. These connections do not affect the first output y_1. Therefore the only difference of the two loss terms of this model's loss function expressed in (3-5), (3-6), (3-7), (3-8) only derive from this area. With $w_{1j}^{[L]}, w_{2j}^{[L]}$ being the weights used for the weighted sum of the linear output of the last hidden layer and $a_j^{[p]}$ being the non-linear output of the neuron j of layer p. Therefore, the only difference of influence on the gradient descent method and therefore the update rule for the model parameters besides the error term, like the one in (2-55) or (2-56), would derive from the variation of the last layer weights $w_{1j}^{[L]}, w_{2j}^{[L]}$. This means the addition of a second output which would be the gradient data $\frac{\delta y}{\delta x}$ in this case, would add the additional weights $w_{2j}^{[L]}$ to the optimization as well as the additional bias $b_2^{[L]}$, separate from the model parameters used to determine the first output value which is the value y_1. These weights and bias would not affect our first output at all, therefore being an additional computation with no gain in performance concerning the approximation of y_1. An additional output therefore means more computation for approximating the wrong output y_2.

Since neural network neurons in a simple feed-forward regression model are interconnected layer to layer unidirectionally and chronologically, this model architecture would have the rest of the model parameters be shared between the two outputs, see Figure 3-1. While a function and its derivative are connected mathematically, approximating both as two separate outputs means approximating two different functions. While a model is most certainly capable of approximating both values, this is unlikely to increase the performance of the model and rather decrease the processing performance concerning the desired approximation of data y, as was said previously. The shared model parameters will generalize between the functions of y and its derivative $\frac{\delta y}{\delta x}$, instead of optimizing the model parameters solely for data y. Furthermore, since the model parameters are now used to approximate 2 outputs instead of just 1 output, according to the discussed universal approximation theorem in section 2.1.2, the model would have to be expanded in dimensions, underlining the decrease in performance.

3.4 Second Variation – Loss addend

Instead of adding a second output to approximate the derivative data $\frac{\delta y}{\delta x}$, the model could simply compute its own derivative with respect to its own input and add an additional loss term to the loss function. This would also not expand the output layer of the model and therefore the model does not compute anything new on top of previous computation of the basic neural network without gradient enhancement. This has various advantages. For one, because of automatic differentiation, see section 2.3.8, most partial derivatives used to calculate the gradient of the defined loss function are also used when calculating the partial derivatives of the model output with respect to its inputs. Even if certain partial derivatives are not retained originally, when they are retained, then the processing time is not affected too much by additional calculations, as they are still fundamentally part of the automatic differentiation process and thereby still calculated. The additional memory needed is therefore negligible. Furthermore, the model does not split its model

parameters to approximate 2 different function values like in the multi-output model. Instead it adjusts its own mathematical derivative with respect to the inputs, all the while approximating the desired output y. The model's partial derivatives $\frac{\delta \hat{y}}{\delta x}$ of its output $\hat{y}$ with respect to its inputs do not introduce new model parameters like the multi output neural network variation does in the last layer of the neural network. Rather and only does it give the model additional information to adjust its preexisting model parameters to better approximate the data y. This is similar to regularization addends of the loss function, like the l_2 regularization, where only preexisting model weights are used to regulate weights so as to avoid vanishing and exploding gradients, see section 2.3.7. Another loss addend method to enhance the optimization process that is quite similar in nature is the Lipschitz continuity regularization [110]. The Lipschitz continuity defines the degree or range of change of a function's value with respect to its input parameters. When applying Lipschitz continuity as a regularization addend to the loss of a neural network, the optimization/training is constrained or rather bounded to a predefined range or value in its gradient with respect to the model's inputs. A general expression of Lipschitz continuity is:

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2| \tag{3-9}$$

Whereby K is the predefined Lipschitz constant that defines the rate of change of the function. This Lipschitz regularization occurs without gradient data. It simply puts a bound on rate of change of the loss function of a neural network. Similarly, but with use of gradient information, the Sobolev-trained neural network, or SANN for Sobolev artificial neural network, applies an additional loss addend to the loss of a neural network model. Since the data y and its derivative $\frac{\delta y}{\delta x}$ are connected mathematically, so should also the model output $\hat{y}$ and its derivative $\frac{\delta \hat{y}}{\delta x}$ have that same mathematical connection. This enables the gradient enhanced model to not only approximate on the given data y, but also on unseen data, as the inherit patterns and structures of the data y are constrained by its derivative $\frac{\delta y}{\delta x}$, limiting the possibilities of approximation solutions of data y to a great degree. In general, this use of gradient data can be proven through use Sobolev spaces, that is, through the adjusted or expanded loss function by exploiting gradient information, it is possible to reduce the error of a neural network's approximation in Sobolev spaces [111].

Therefore, gradient enhanced neural network models have the same general architecture as basic neural network models, when it comes to layers, neurons and other hyperparameters. However the difference in the gradient enhanced model will be the expansion of its loss function through the simple addition of a gradient loss L_{dY} and with it, the few additional computations inside the automatic differentiation to acquire the partial derivatives with respect to the inputs and their retention for further computations. The equations governing the gradient enhanced model are elaborated in the next section.

3.4.1 Neural Network Model Generation

The MATLAB program code used for the neural network models is illustrated in Figure 3-3, which details the general flow and processing of information through a flow chart. Going step by step through the flow chart in Figure 3-3, first, the generated data is split into training and validation data after being standardized according to the equations in section 2.3.9. An important issue with using a gradient enhanced neural network relates to the standardization of the data. The magnitude or scale difference between the datasets of output y and its derivative $\frac{\delta y}{\delta x}$ have direct effects on the performance of the training of the neural network. When training a neural network model it is common knowledge in machine learning to normalize/standardize the training data for various reasons. Generally, it improves convergence as it reduces scale differences between samples, thereby also increasing gradient stability.

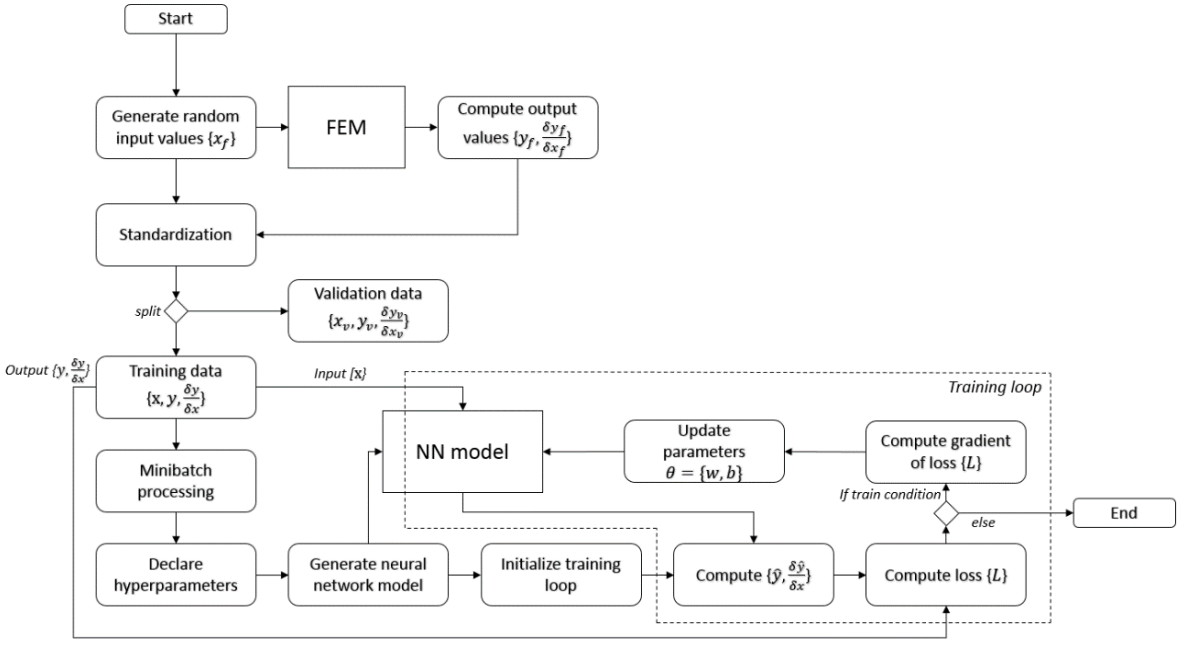

Figure 3-3: Information Flow Chart of the MATLAB Program Code (figure by author)

This can also improve the weight regularization implemented in the loss function, as adjusting the scale of data samples allows the weight penalty to be applied uniformly. Especially when using derivative data to enhance a neural network, it needs to be preprocessed, because of the potential large scale difference between function value and derivatives. Additionally, to standardize the derivative data, simply using the same equation procedure as for the input data x and the output data y is not possible. Doing so destroys and changes the gradient information contained in the data. When using the standard deviation $\sigma_{\frac{\delta y}{\delta x}}$ and the mean $\mu_{\frac{\delta y}{\delta x}}$ of the derivative data to calculate the standardized derivative data $(\frac{\delta y}{\delta x})_s$, the following equation, similar to (2-69) (2-70), would follow:

$$(\frac{\delta y}{\delta x})_s = \frac{\frac{\delta y}{\delta x} - \mu_{\frac{\delta y}{\delta x}}}{\sigma_{\frac{\delta y}{\delta x}}} \tag{3-10}$$

Here, the calculated value of this equation has lost the information that relates the derivative $\frac{\delta y}{\delta x}$ to the output y. To illustrate this, using an example like a simple quadratic function of $f(x) = y = x^2$, one has the following values for the given x:

$$x = \{-1,0,1\},\ y = \{1,0,1\} \text{ and } \frac{\delta y}{\delta x} = \{-2,0,2\}$$

By applying basic normalization to each data set, the values change to:

$$x_n = \{0,0.5,1\},\ y_n = \{1,0,1\} \text{ and } (\frac{\delta y}{\delta x})_n = \{0,0.5,1\}$$

As can be observed, the normalized derivative data $(\frac{\delta y}{\delta x})_n$ has lost certain gradient information, like the correct position of the global minima. It no longer has the correct gradients, and this means it does not contain the derivative of the normalized output y_n. When training a gradient enhanced neural network with gradient data normalized according to (3-10) the training is not able to correctly converge to a solution and leads to numerous numerical instabilities. The normalized derivative data must contain the gradient information of the derivative of y_n with respect to x_n instead. Thus, the correctly standardized derivative data $\frac{dy_{is}}{dx_{is}}$ is calculated according to the following written equations. Previously, in equations (2-69) and (2-70), the following equations for the standardized input x_{is} and output y_{is} were given:

$$x_{is} = \frac{x_i - \mu_x}{\sigma_x} \tag{3-11}$$

$$y_{is} = \frac{y_i - \mu_y}{\sigma_y} \tag{3-12}$$

With $\frac{dy_{is}}{dx_{is}}$ being the derivative of the standardized output dy_{is} with respect to the standardized input dx_{is}, the value of $\frac{dy_{is}}{dx_{is}}$ will still contain the gradient information while simultaneously adjusting the derivative data to the standardized input x_{is} and output y_{is} data, thereby standardizing it as well.

First, an expression for dx_{is} is formulated by differentiating (3-11), which is the general derivative of the standardized input x_{is}:

$$dx_{is} = \frac{dx_i}{\sigma_x} \tag{3-13}$$

Similarly, differentiating (3-12), it follows:

$$dy_{is} = \frac{dy_i}{\sigma_y} \tag{3-14}$$

By combing these two and adjusting for $\frac{dy_{is}}{dx_{is}}$, the final expression changes to:

$$\frac{dy_{is}}{dx_{is}} = \frac{\frac{dy_i}{dx_i}}{\frac{\sigma_x}{\sigma_y}} \tag{3-15}$$

And finally the correct standardized derivative data $\frac{dy_{is}}{dx_{is}}$ is:

$$\frac{dy_{is}}{dx_{is}} = \frac{\sigma_x}{\sigma_y} * \frac{dy_i}{dx_i} \tag{3-16}$$

Without this correct preprocessing, the neural network will not accurately converge or not converge at all. Not using this equation leads to the case where the optimization algorithm tries to minimize the loss of the output and its derivative with respect to its input, the loss of the derivative will not apply to the same data as the loss of the output, meaning the two loss terms will define two completely different functions which the neural network model then tries to approximate, which is not possible. To expand on the previous example, going back to the simple quadratic function, the correctly normalized data now is:

$$x_n = \{0,0.5,1\},\ y_n = \{1,0,1\} \text{ and } \frac{dy_n}{dx_n} = \{-4,0,4\}$$

As can be observed, the correctly normalized derivative data contains the global minimum at (0.5,0). The derivative has also increased in scale, which is correct since the normalized output

y_n has been compressed into the smaller range of the normalized input x_n. Still, after standardizing the output and the derivative data, a difference in scale between them remains. To determine if the remaining scale difference is an issue, a model variation with the loss weights is introduced and analyzed with a variation of fixed values, as mentioned previously in this section.

Returning to the flow chart in Figure 3-3, once the training data has been preprocessed, it is split into minibatches of a predefined size. Then the neural network model is initialized with predefined options for all its hyperparameters. After this, the training loop is initiated, whereby according to the predefined options, through automatic differentiation, a unique loss function is used after forward and backward propagation, to calculate all gradients. The gradients are then parsed into the update rule of the stochastic gradient descent momentum algorithm and the model parameters are updated. This is repeated for each iteration until an epoch has passed. After each iteration t, the program updates the new learning rate λ_t inside the update rule with the decay rate c, a predefined hyperparameter, according to the time-based decay:

$$\lambda_{t+1} = \frac{\lambda_t}{(1 + c * t)} \tag{3-17}$$

Afterwards, the order of the minibatches is shuffled and the process repeats until the training is stopped or has reached the maximum epochs. During each iteration, metrics, such as loss and relative l_2 error, are computed and stored. All results are stored to then be used in various calculations to provide metrics for the analysis of the results. This time-based decay of the learning rate will be replaced with a more streamlined learning rate update rule. More on this in section 4.

3.4.2 Loss Function

The regression gradient enhanced neural network approximates an output according to the training data through use of an optimization algorithm like gradient descent. A neural network does this through adjusting its model parameters $\theta = \{w, b\}$ iteratively. These parameters are used in its nested non-linear neuron output functions to calculate the output for each layer to then predict the final output $\hat{y}$. Through defining an objective function, that is, the loss function for the optimization algorithm to optimize, the neural network model is capable of converging towards an optimal solution. The solution is only an approximation, and the optimization algorithm only adapts itself according to the residual of its output $\hat{y}$ to the true output y. By adding another additional loss in the form of the gradient of y, the optimization algorithm has more information during the training process. The gradient $\frac{\delta \hat{y}}{\delta x}$ of the model output $\hat{y}$ with respect to the model input x, adds new information to the optimization problem and thereby removes a certain amount of previously possible solutions of model parameter variations and combinations that worked for the basic neural network. With this new information, these variations and combinations of model parameters can no longer reduce the total loss of our model, as they now must consider a constellation of model parameters that reduces the gradient loss. This allows the optimization to converge easier along the space of the loss function. Because the gradient $\frac{\delta y}{\delta x}$ defines rate and direction of

how the function value y moves through its function space, it limits the model function obtained through training to a certain shape more in line with the to be approximated function . The total loss L of the gradient enhanced neural network is defined as:

$$L = L_Y + L_{dY} + L_R \tag{3-18}$$

With the basic neural network loss L_Y being the error of our model output $\hat{y}$ to the true output y:

$$L_Y = \frac{1}{2N}\sum_n (\hat{y} - y)^2 \tag{3-19}$$

And the gradient loss L_{dY}, which is the error of the gradient $\frac{\delta \hat{y}}{\delta x}$ of our model output $\hat{y}$ with respect to the input x, to the true gradient $\frac{\delta y}{\delta x}$ of the true output y with respect to the input x:

$$L_{dY} = \frac{1}{2N}\sum_n (\frac{\delta \hat{y}}{\delta x} - \frac{\delta y}{\delta x})^2 \tag{3-20}$$

With the sum over $n = [1, \dots, N]$ training data samples of the predefined minibatch. Whereby depending on the number of inputs, $\frac{\delta \hat{y}}{\delta x}$ and $\frac{\delta y}{\delta x}$ are vectors of size NxI. In this case each residual for each individual partial derivative is computed and squared before being summed up together. In the case of multiple model outputs and model inputs these two equations can be expressed as:

$$L_Y = \frac{1}{2N}\left[\sum_n (\hat{y}_1 - y_1)^2 + \sum_n (\hat{y}_2 - y_2)^2 + \cdots + \sum_n (\hat{y}_o - y_o)^2\right] \tag{3-21}$$

$$L_{dY} = \frac{1}{2N}\left[\left(\sum_n (\frac{\delta \hat{y}_1}{\delta x_1} - \frac{\delta y_1}{\delta x_1})^2 + \sum_n (\frac{\delta \hat{y}_2}{\delta x_1} - \frac{\delta y_2}{\delta x_1})^2 + \cdots + \sum_n (\frac{\delta \hat{y}_o}{\delta x_1} - \frac{\delta y_o}{\delta x_1})^2\right)\right.$$
$$+ \left(\sum_n (\frac{\delta \hat{y}_1}{\delta x_2} - \frac{\delta y_1}{\delta x_2})^2 + \sum_n (\frac{\delta \hat{y}_2}{\delta x_2} - \frac{\delta y_2}{\delta x_2})^2 + \cdots\right.$$
$$\left. + \sum_n (\frac{\delta \hat{y}_o}{\delta x_2} - \frac{\delta y_o}{\delta x_2})^2\right) + \cdots \quad \textbf{(3-22)}$$
$$+ \left(\sum_n (\frac{\delta \hat{y}_1}{\delta x_i} - \frac{\delta y_1}{\delta x_i})^2 + \sum_n (\frac{\delta \hat{y}_2}{\delta x_i} - \frac{\delta y_2}{\delta x_i})^2 + \cdots\right.$$
$$\left.\left. + \sum_n (\frac{\delta \hat{y}_o}{\delta x_i} - \frac{\delta y_o}{\delta x_i})^2\right)\right]$$

With i denoting the number of inputs and o denoting the number of outputs. These equations are used with standardized data. Reasons for preprocessing of training data are discussed in section 2.3.9. The loss of weight regularization L_R, which is a l_2 regularization of all weights $w_i^{[l]}$ of the model parameters θ:

$$L_R = \frac{1}{2N}\sum_l \sum_i {w_i^{[l]}}^2 \quad \textbf{(3-23)}$$

Whereby the weights $w_i^{[l]}$ denote the model parameter weights used for each linear part for each layer l. They are the weights of each of the model's neurons i, used in the weighted sum calculation as defined in section 2.3.2. All these weights are squared and summed together for the l_2 regularization to apply a weight penalty.

The partial derivative of the output with respect to the input parameters $\frac{\delta \hat{y}}{\delta x}$ is calculated by automatic differentiation. Per the chain rule, it is expressed as:

$$\frac{\delta \hat{y}}{\delta x} = \frac{\delta \hat{y}}{\delta a^{[L]}}\frac{\delta a^{[L]}}{\delta z^{[L]}}\frac{\delta z^{[L]}}{\delta a^{[L-1]}}\frac{\delta a^{[L-1]}}{\delta z^{[L-1]}} * \ldots * \frac{\delta a^{[1]}}{\delta z^{[1]}}\frac{\delta z^{[1]}}{\delta a^{[0]}}, with\ l = \{0, \ldots, L\} \quad \textbf{(3-24)}$$

All partial derivatives, except the last, follow the same expressions as shown in section 2.3.8 starting from equation (2-26) onwards. For the last partial derivative $\frac{\delta z^{[1]}}{\delta a^{[0]}}$ it follows from this previous section:

$$\frac{\delta z^{[1]}}{\delta a^{[0]}} = w^{[0]} \tag{3-25}$$

To extend with this on the previous simple regression neural network example with 2 hidden layers with 3 neurons each, detailed in section 2.3.8, the following is obtained:

$$\begin{aligned}
\frac{\delta \hat{y}}{\delta x} &= \frac{\delta \hat{y}}{\delta a^{[2]}}\frac{\delta a^{[2]}}{\delta z^{[2]}}\frac{\delta z^{[2]}}{\delta a^{[1]}}\frac{\delta a^{[1]}}{\delta z^{[1]}}\frac{\delta z^{[1]}}{\delta x} \\
&= w_1^{[2]} * g'\left(z_1^{[2]}\right) * (w_{11}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\
&\quad + w_{12}^{[1]} * g'\left(z_2^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\
&\quad + w_{13}^{[1]} * g'\left(z_3^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i}) \\
&+ w_2^{[2]} * g'\left(z_2^{[2]}\right) * (w_{21}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\
&\quad + w_{22}^{[1]} * g'\left(z_2^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\
&\quad + w_{23}^{[1]} * g'\left(z_3^{[1]}\right)) * \frac{\delta z^{[1]}}{\delta x_i}) \\
&+ w_3^{[2]} * g'\left(z_3^{[2]}\right) * (w_{31}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\
&\quad + w_{32}^{[1]} * g'\left(z_2^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\
&\quad + w_{33}^{[1]} * g'\left(z_3^{[1]}\right)) * \frac{\delta z^{[1]}}{\delta x_i})
\end{aligned} \tag{3-26}$$

With the final part of each term defined as one of the two inputs, depending on the connection:

$$\frac{\delta z^{[1]}}{\delta x_i} = \frac{\delta z^{[1]}}{\delta a^{[0]}} = w_{ij}^{[0]} \tag{3-27}$$

For $w_{11}^{[0]}$, one of the first 6 weights connecting the first neuron of the input layer with the first neuron of the first hidden layer, (3-26) shows the chain rule path for computation. The path is the exact same in this case. The difference now is that the model outputs gradient with respect to its inputs is differentiated with respect to the model parameter as well. Then, making use of the

previously defined expression of $\frac{\delta \hat{y}}{\delta x}$, one concludes the following simplified expression for $\frac{\frac{\delta \hat{y}}{\delta x}}{\delta w_{11}^{[0]}}$, the partial derivative with respect to $w_{11}^{[0]}$:

$$\begin{aligned}\frac{\frac{\delta \hat{y}}{\delta x}}{\delta w_{11}^{[0]}} &= w_1^{[2]} * g'\left(z_1^{[2]}\right) * w_{11}^{[1]} * g'\left(z_1^{[1]}\right) \\ &+ w_2^{[2]} * g'\left(z_2^{[2]}\right) * w_{21}^{[1]} * g'\left(z_1^{[1]}\right) \\ &+ w_3^{[2]} * g'\left(z_3^{[2]}\right) * w_{31}^{[1]} * g'\left(z_1^{[1]}\right)\end{aligned} \quad \textbf{(3-28)}$$

A simpler derivation explanation is by observing (3-26). Any term in which the final part is defined as $\frac{\delta z^{[1]}}{\delta x_i} = w_{11}^{[0]}$, is simply dropped, thereby concluding to (3-28). For the connection between the first and second hidden layer, the partial derivative with respect to the weight $w_{11}^{[1]}$ is computed and it is expressed as:

$$\begin{aligned}\frac{\frac{\delta \hat{y}}{\delta x}}{\delta w_{11}^{[1]}} &= w_1^{[2]} * g'\left(z_1^{[2]}\right) * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\ with\ \frac{\delta z^{[1]}}{\delta x_1} &= w_{11}^{[0]}\ \ OR\ \ \frac{\delta z^{[1]}}{\delta x_2} = w_{21}^{[0]}\end{aligned} \quad \textbf{(3-29)}$$

Simply put, observing (3-26), any term which does not include $\delta w_{11}^{[1]}$ is simply dropped. And the partial derivative with respect to the weight $w_1^{[2]}$, which is between the final hidden layer and the output layer, is expressed as:

$$\begin{aligned}\frac{\frac{\delta \hat{y}}{\delta x}}{\delta w_1^{[2]}} &= g'\left(z_1^{[2]}\right) * w_{11}^{[1]} * g'\left(z_1^{[1]}\right) * \frac{\delta z^{[1]}}{\delta x_i} \\ with\ \frac{\delta z^{[1]}}{\delta x_1} &= w_{11}^{[0]}\ \ OR\ \ \frac{\delta z^{[1]}}{\delta x_2} = w_{21}^{[0]}\end{aligned} \quad \textbf{(3-30)}$$

Again, simply put, observing (3-26), any term which does not include $\delta w_1^{[2]}$ is simply dropped. These exemplary partial derivatives with respect to model parameters θ show the interconnected nature of using the partial derivative $\frac{\delta \hat{y}}{\delta x}$ of the model output $\hat{y}$ with respect to the inputs, just like with the partial derivatives of the model output $\hat{y}$ with respect to model parameters θ. It shows precisely why there is no additional computation occurring when previously computed partial derivatives are traced and retained. Each of the terms in the equations above are already part of the computations for a non-gradient enhanced neural network. For the partial derivative

$\frac{\delta \hat{y}}{\delta x}$, the expressions also do not include any non-linear neuron outputs $a_i^{[l]}$, making them completely linear and simple in design. This lends itself to the basic idea behind neural networks, where its calculations are simple mathematical functions. In other terms, this means less computational load when using automatic differentiation. Of course, depending on the chosen activation function, this may not be the case, however in this work the ReLU is used.

3.4.3 Backpropagation per Automatic Differentiation

When backpropagation occurs, for the gradient of the loss function $\frac{\delta L}{\delta \theta}$, the first equation is:

$$\frac{\delta L}{\delta \theta} = \frac{\delta L_Y}{\delta \theta} + \frac{\delta L_{dY}}{\delta \theta} + \frac{\delta L_R}{\delta \theta} \tag{3-31}$$

The gradient $\frac{\delta L_Y}{\delta \theta}$ is the same as the gradient for the basic neural network discussed in section 2.3.8, equation (2-25). Next, the other terms need to be defined. When calculating the gradient of the loss function L_{dY} with respect to a model parameter θ, one obtains:

$$\frac{\delta L_{dY}}{\delta \theta} = \frac{\delta L_{dY}}{\frac{\delta \hat{y}}{\delta x}} \frac{\frac{\delta \hat{y}}{\delta x}}{\delta \theta} \tag{3-32}$$

Depending on the number of inputs, the loss L_{dY} and therefore its loss gradients $\frac{\delta L_{dY}}{\delta \theta}$ are vectors. Computationally, this only means that each element of this vector is treated as an additional addend to the loss term, meaning the vector is summed up elementwise. And for the gradient of the loss L_{dY} with respect to the gradient of the output with respect to the input parameters $\frac{\delta \hat{y}}{\delta x}$, the expression is:

$$\frac{\delta L_{dY}}{\frac{\delta \hat{y}}{\delta x}} = \frac{1}{N} \sum_n \frac{\delta \hat{y}}{\delta x} - \frac{\delta y}{\delta x} \tag{3-33}$$

This follows the same derivative structure as (2-23). For the regularization term L_R, the following equation follows from differentiating (3-36), for its gradient $\frac{\delta L_R}{\delta \theta}$:

$$\frac{\delta L_R}{\delta \theta} = \frac{\alpha}{N} * \theta \tag{3-34}$$

3.4.4 Loss Weight Methods

As mentioned earlier, because of the scale and magnitude difference between function value y and its gradient $\frac{\delta y}{\delta x}$ even after standardization/normalization, weight factors of the loss addends are considered. The goal of these loss weights is to improve the weights of each loss to increase training convergence performance.

Since the function value y and the gradient data $\frac{\delta y}{\delta x}$ contain different information for the same function space, both have differing importance for the optimization of the model through training. Furthermore standardization/normalization is not always optimally reducing the scale and magnitude difference between function value and its gradient. This can again, greatly affect the training process and the accuracy of the neural network model obtained after training. In (3-18) the losses L_Y and L_{dY} can vary greatly in magnitude, leading to numerical issues or to a certain loss to dominate the optimization process. An adjusted ratio of function value loss to gradient loss in the total loss function could improve the training process. In one published application, the gradient data is gradually introduced to the training process by adding a weight factor to the gradient loss which is then linearly increased over the training period [112]. The aspect of weight factors added to each loss addend can still be expanded. For one, instead of only 1 weight factor for the gradient loss, 2 weight factors defining the ratio of basic loss and gradient loss could improve performance, since depending on the data one or the other could dominate the optimization process. Secondly, instead of simple predefined weight factors, including these weight factors of the loss addends in the optimization algorithm, treating them like any other model parameter, could also improve performance. Third and lastly, the behavior of the optimization performance and the accuracy of the final trained model could be heavily dependent on the data and function the model is approximating. In the previously mentioned publication, [112], only a linear increase of the gradient information weight factor occurs for simple academical functions. This behavior should be replicable when applied to linear and non-linear mechanics such as elasticity or behave completely differently. Further research could reveal previously unknown correlation between performance and increase/decreased/adjustment of these loss weights. These are several reasons to improve the optimization process through new and different methods of loss weights.

First, let us describe the loss weight implementation of this work. When considering the weights given to each component of the total loss function of our model, they are also included in the regularization term, which regulates all model weights in the model layers. The loss weight of the output error is denoted as w_Y, and the loss weight of the gradient error is denoted as w_{dY}. The total loss L is then expressed as:

$$L = w_Y L_Y + w_{dY} L_{dY} + L_R \tag{3-35}$$

And the regularization term L_R, previously discussed in 2.3.7, is now expanded with these loss weights and defined as:

$$L_R = \frac{\alpha}{2N}((\sum_l \sum_i w_i^{[l]^2}) + w_Y^2 + w_{dY}^2) \tag{3-36}$$

With α being a predefined regularization factor.

First, fixed loss weights are applied to the model and the various metrics, like loss and relative l2 error, are observed. The first variant will apply a weight w_Y of 0.1 gradient loss L_{dY}, while the output loss L_Y will have a weight w_{dY} of 1. The second variant will swap the weight values. The goal of this is to observe the correlation of loss weight values to the overall performance of the gradient enhanced neural network. By applying a linear set of different loss weights, the results should provide a general overview of any correlation.

Then, two variations with training algorithms updating the loss weights just like all other model parameters are updated through stochastic gradient descent, are observed. The first variation will have no constraints on the loss weights, initializing both w_Y and w_{dY} as 1. Only its loss will be analyzed. The second variation will use an equation of a normal cumulative distribution for the loss weights. The reason for this is to constrain the loss weights into a desired range, in this case 0 and 1, thus attributing importance along this range to each loss addend. Since the optimization process's goal is to minimize the loss function, these factors will naturally tend to lower values. By limiting them to this range, the goal is to avoid negative values and numerical instability. The loss weights are then defined as:

$$\begin{aligned} w_Y &= \frac{1}{2} * \left(1 - \operatorname{erf}\left(-\frac{r_1}{\sqrt{2}}\right)\right) \\ with\ \operatorname{erf}\left(-\frac{r_1}{\sqrt{2}}\right) &= \frac{2}{\sqrt{\pi}} \int_0^{-\frac{r_1}{\sqrt{2}}} e^{-t^2} dt \end{aligned} \tag{3-37}$$

And

$$w_{dY} = \frac{1}{2} * (1 - \operatorname{erf}\left(-\frac{r_2}{\sqrt{2}}\right)) \tag{3-38}$$

$$with\ \text{erf}\left(-\frac{r_2}{\sqrt{2}}\right) = \frac{2}{\sqrt{\pi}} \int_0^{-\frac{r_2}{\sqrt{2}}} e^{-t^2} dt$$

Where both loss weight parameters r_1 and r_2 are initialized as 0, which results in a value of 1 through the expressions (3-37) and (3-38) for the final weights attached to each loss term. The gradient of the loss with respect to these two parameters is used to update them. The learning rate of these loss weights is separate from the model parameters learning rate. This second variation will be compared to the fixed weight model and the gradient enhanced model by loss and l_2 error.

4 Analysis

4.1 Overview of Models

The neural network will be designed in MATLAB through a combination of predefined functions of libraries and self-defined functions. The gradient enhanced neural network models can have any number of inputs and outputs. In general, the application of the discussed methods is independent of the dimensions of a model and can be applied or extended to any model dimensionality. When using high dimensional models, it should be noted that the design of the model would need to be adjusted accordingly [42]. For simplicity, the neural network models used in this work are feed-forward regression models with 2 inputs and 1 output. The activation function for all neurons will be the ReLU activation function. The gradient enhanced neural network will be compared with the basic neural network with 1000 training points, 2 hidden layers and 10 neurons in each hidden layer. Afterwards, the gradient enhanced neural network will be analyzed for 2, 4 and 6 hidden layers, with 10 neurons each and 1000 training points. Then, the fixed loss weight models will be compared to the gradient enhanced model with 1000 training points, 2 hidden layers and 10 neurons in each hidden layer. The training data will be generated by the finite element model, the input parameters will be generated randomly in a range of -1 and 1. The validation data will cover 1000 data points. Finally, the gradient enhanced neural network will be analyzed across increasing training data sizes. The training data will vary to 1000, 500, 100. These last models will use 2 hidden layers with 10 neurons each.

The analysis metrics will consist of the defined loss of the models over the iterations, the relative l_2 error over the iterations, execution times for the finite element model, the training of the neural network models, the prediction time of the neural network models, a confusion chart comparing all three and lastly the probability of exceedance curve of the neural network predictions and the finite element model outputs. The loss and relative l_2 error will allow the comparison of approximation performance between models. Processing times will allow the comparison of time performance between models. The probability of exceedance curve will be used as an exemplary metric for output data assessment, in which the neural network models are compared to the finite element model they are based on. The confusion chart will serve as a visualization of areas of difference between the finite element model and the neural networks.

4.1.1 Finite Element Model

The finite element model computes the response and the response sensitivity information of a linear elastic 2D mechanical system and a non-linear elastic 2D mechanical system. The core parameters of the finite element model are:

- x: design parameters
- opt: options
- U: structural response (displacement)
- dU: response sensitivity

- svM: von Mises stresses (integration points)
- $dsvM$: von Mises stresses sensitivity
- C: compliance
- dC: compliance sensitvity
- V: volume
- dV: volume sensitivity

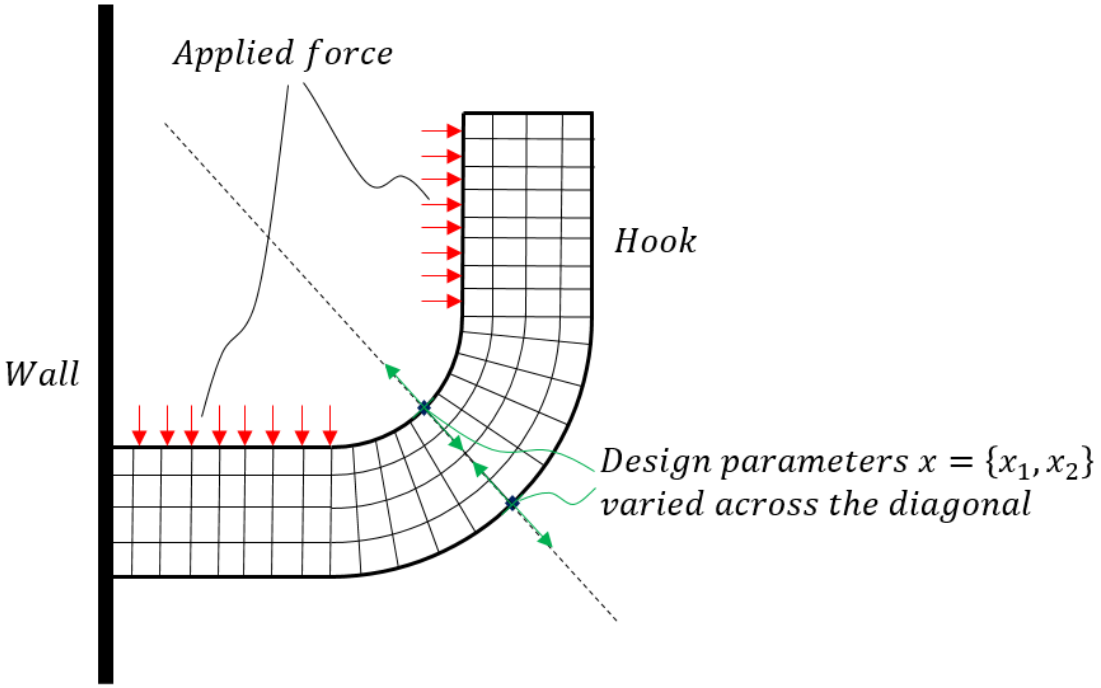

Figure 4-1: Sketch of the hook system approximated by the finite element model. The design parameters $x_1, x_2 = [-1, 1]$ adjust the blue marked nodes along the green axis (figure by author)

The hook system is illustrated in Figure 4-1. The non-linear variation of the finite element model can specify the material used for the system. The von Mises stresses are derived from the second Piola-Kirchoff stress tensor, which are predefined in the reference configuration. The finite element model will be used to generate training data for one, the linear case and second, the non-linear case for the St. Venant-Kirchoff material in the plane stress state, whereby this plane stress state is an approximation in contrast to an exact solution of a plane strain state.

4.1.2 Model Hyperparameters

Afterwards, the model applies the minibatch stochastic gradient descent method with momentum to update the model parameters each minibatch iteration. The following options are applied to all neural network models:

Table 4-1: Hyperparameter initialization values (representation by author)

Hyperparameter	Initialization value
Minibatch size	250
α	0.1
Initial learn rate	0.1
Momentum	0.9
Total number of epochs	100

The neural network models are coded in MATLAB. Models which are compared use the same training data and validation data to avoid undesired performance difference through randomly generated data. A multitude of models for each setup are performed and the average of these models is used to remove randomness of results introduced by model parameter initialization and the stochastic gradient descent training. With 100 epochs, a training data size of 1000 samples and a minibatch size of 250, there are 4 iterations per epoch. In the case of the comparison of the gradient enhanced neural network for different training data sizes the minibatch size and validation data size are adjusted accordingly, see Table 4-2.

Table 4-2: Hyperparameter adjustment for training data size variation results (representation by author)

Training data size	Minibatch size	Validation data size
1000	250	1000
500	125	500
100	25	100

The learnrate is updated according to Table 4-3. The time-based decay learnrate update rule was replaced with this fixed case update rule. The reason for this is that the results showed the learnrate decreasing too fast. To make sure the training would create workable results, this fixed update rule was introduced.

Table 4-3: Learnrate update table (representation by author)

Epoch	Learnrate
0	0.1
50	0.05
150	0.01
300	0.005

4.2 Results

The analysis of the linear and nonlinear models will happen in this section. In the following, all the various results are presented. As mentioned previously, a confusion chart between finite

element and neural network, as well as between finite element and gradient enhanced neural network have been done. Furthermore there is various tables concerning the processing and training times. Since the gradient-enhanced network has a larger loss because of its additional gradient loss addends, comparing models by loss is inconclusive, see equation (2-23) and equations starting (3-18). Therefore, the loss metric can be discarded in the analysis, however the graphs will be included for completeness and discussed. The most significant graphs follow the relative l_2 error. The relative l_2 error is a good metric to compare the neural network models to the finite element model. This is done by validation of the prediction of trained models to unseen predictions of the finite element model.

$$l_2 = \frac{\|\hat{y} - y\|_2}{\|y\|_2} \tag{4-1}$$

It is an expression using the l_2-norm. The analysis will also cover an evaluation of the distribution and extremes of the model's predictions via probability exceedance curves.

4.2.1 Training and Prediction times

The Table 4-4 shows the time results of all models for the linear elasticity case. For the FEM, the validation data size is the generated data of the FEM. The training data size is what the models use to train. As can be seen, the processing times for the neural network models after they have been trained is immensely smaller than the time it takes the finite element model to compute its outputs. It can also be observed that the gradient enhanced neural network is not slower at reaching the final number of training epochs, even though it technically computes more expressions than the basic neural network.

For the linear case it is necessary to mention, that the trained neural network models, basic or gradient enhanced, only compute a single output, whereas the finite element model computes multiple various outputs as mentioned in section 4.1.1. Therefore the comparison between the times of FEM, NN and gNN with each other should be considered biased and unequal. When considering training data size, the training time does not seem affected by it. The data size does not slow down or speed up the gradient neural networks training. Same can be said about the time it takes to predict 1000 validation values.

Increasing the number of layers positively correlated with training time. This is explained because each layer adds additional model parameters. Since a neural network is a big collection of nested functions including weighted sums, more layers can quickly lead to exponential increase of computation effort, as each individual neuron in the new additional layer provides 2xN new model parameters, with N being the number of neurons in the previous layer, and 2xP new model parameters, with P being the number of neurons in the next layer. While the training time increased, the time it took to predict 1000 validation data was not affected at all. The same observation can be made for the change of the number of neurons in each layer. However increasing the number of neurons affects the training time stronger than increasing layer size. The times of

the table clearly show that once a neural network is trained, it is extremely quick compared to the finite element model. The training time of the neural network models however is considerably longer. This does not take into consideration however the time it takes to setup and derive a finite element model. In the case of neural networks, having the data available, a neural network of simple design can immediately be trained.

Table 4-4: Prediction and Training times for linear elasticity, Time is in seconds (representation by author)

Model	Hidden Layer Size	Neurons per Hidden Layer	Training Data Size	Validation Data Size	Prediction Time	Training Time
FEM	-	-	-	100	12s	-
	-	-	-	500	58s	-
	-	-	-	1000	117s	-
NN	2	10	1000	1000	0.0053s	142s
gNN	2	10	1000	1000	0.0054s	147s
	4	10	1000	1000	0.0051s	313s
	6	10	1000	1000	0.0086s	494s
	2	20	1000	1000	0.0058s	403s
	2	30	1000	1000	0.0060s	843s
	2	10	500	500	0.0054s	140s
	2	10	100	100	0.0052s	138s

Table 4-5: Prediction and Training times for non-linear elasticity, Time is in seconds (representation by author)

Model	Hidden Layer Size	Neurons per Hidden Layer	Training Data Size	Validation Data Size	Prediction Time	Training Time
FEM	-	-	-	100	88s	-
	-	-	-	500	439s	-
	-	-	-	1000	881s	-
NN	2	10	1000	1000	0.0069s	141s
gNN	2	10	1000	1000	0.0066s	145s
	4	10	1000	1000	0.0072s	304s
	6	10	1000	1000	0.0071s	490s
	2	20	1000	1000	0.0062s	404s
	2	30	1000	1000	0.0063s	841s
	2	10	500	500	0.0069s	142s
	2	10	100	100	0.0064s	138s

When looking at the nonlinear elasticity case in Table 4-5, we can observe the same behaviors. The finite element model takes considerably longer to compute the nonlinear case. Just increasing the training data size tenfold leads to a tenfold increase of computing time for the finite element model. Meanwhile, the training time of the neural networks does not seem as affected by the nonlinear nature of the data. What this shows is that in the case of complex non-linear systems, a neural network can be trained and output results faster, than the finite element model can output results. Just like in the linear case, increasing layer size or neuron number results in longer training times. This is, again, for the same reasons as mentioned for the linear case. The prediction times of the neural network models seems to have increased as well compared to the linear timetable.

4.2.2 Comparisons – Linear Case

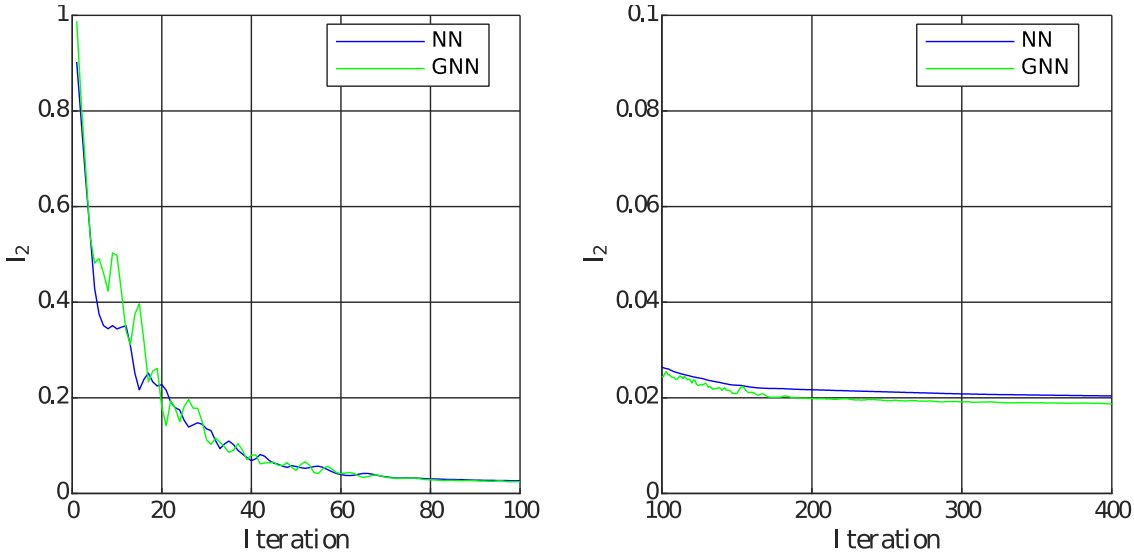

Figure 4-2: gNN vs. NN, l_2 error (figure by author)

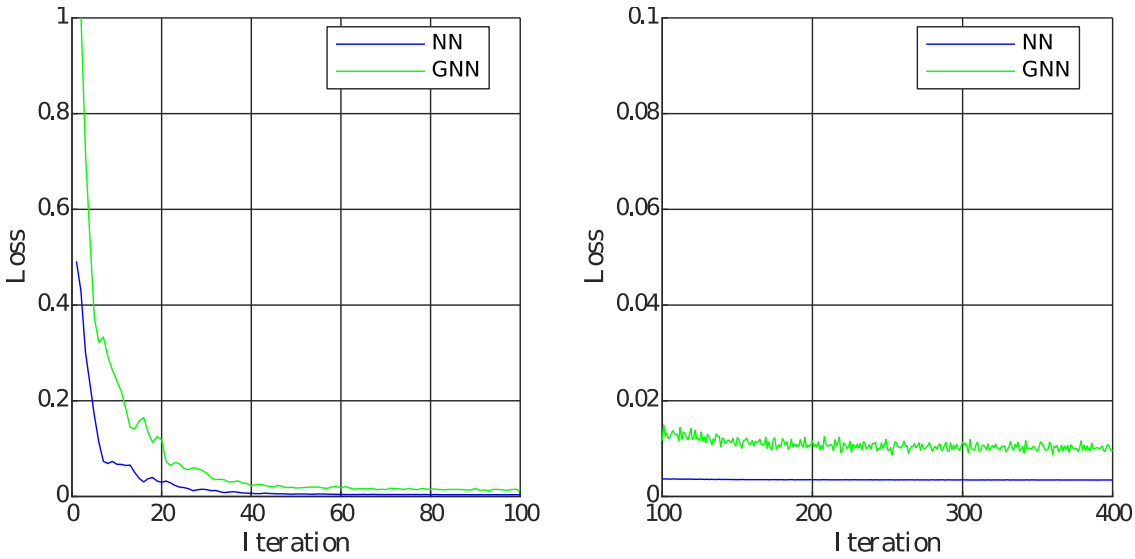

Figure 4-3: gNN vs. NN, loss (figure by author)

In Figure 4-2 and Figure 4-3 the l_2 error and the loss are plotted over the training iterations. It can be observed that the loss of the gradient enhanced neural network is larger than the loss of the basic neural network as expected. Concerning the l_2 error, the gNN is outperforming the NN by a slight margin. This shows that the gradient enhanced neural network performs better than the base neural network.

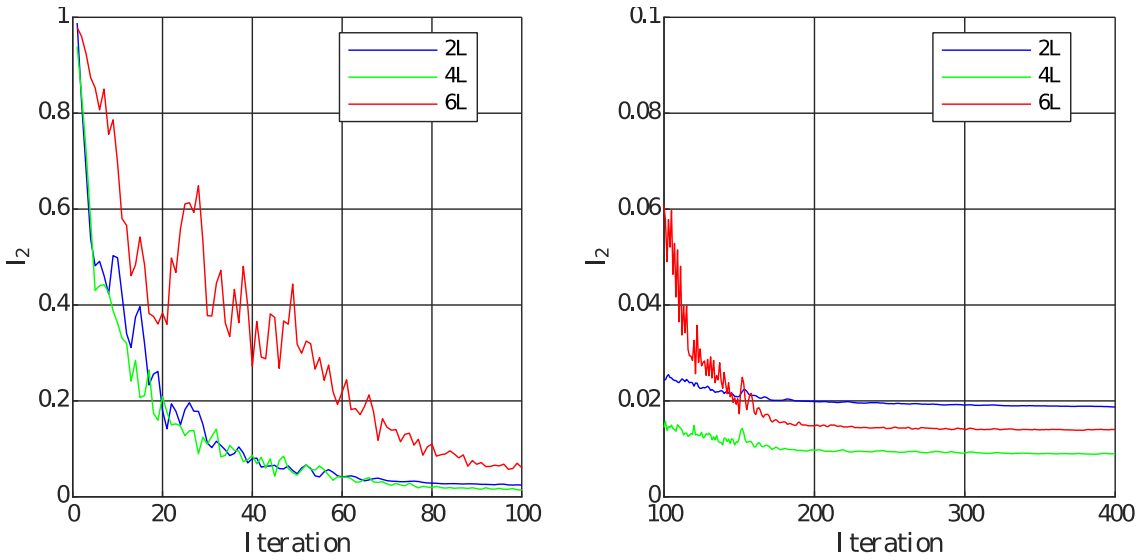

Figure 4-4: gNN with different number of hidden layers, l_2 error (figure by author)

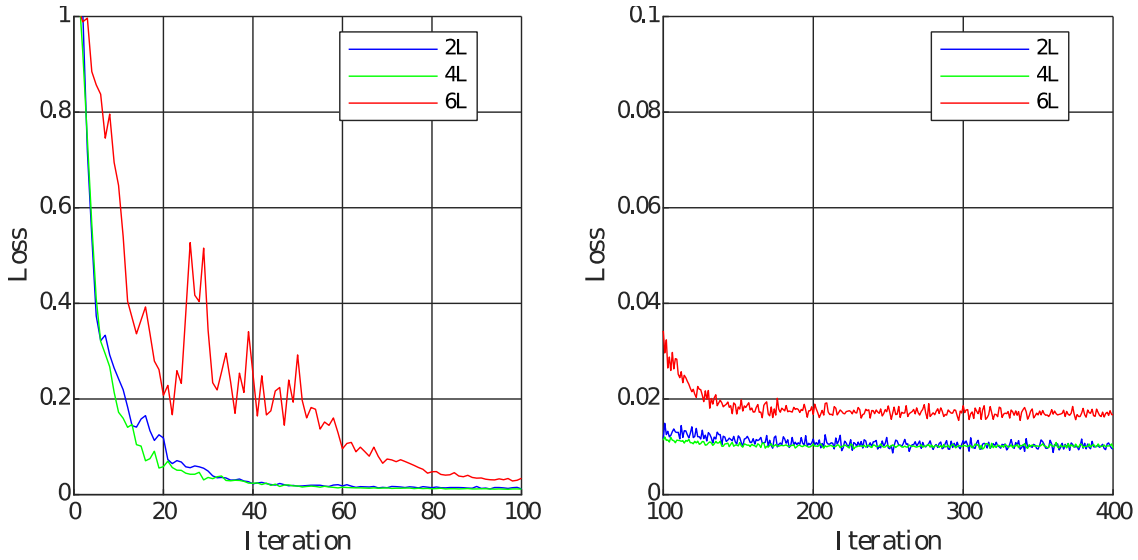

Figure 4-5: gNN with different number of hidden layers, loss (figure by author)

In Figure 4-4 and Figure 4-5 the gNN is compared with different number of hidden layers. Most notable is the 6 hidden layer model taking more iterations to converge. The early iterations it is very unstable. It also shows the highest loss of all three variations. However when observing the l_2 error the 6 hidden layer model performs better than the 2 hidden layer model. The best performance is achieved by the 4 hidden layer model. More than likely for the linear case and the provided training data, 4 hidden layers is closer to the optimal layer design than 2 or 6 hidden

layers. This shows that an oversized neural network model reduces performance just like an undersized model, refer to the universal approximation theorem [42].

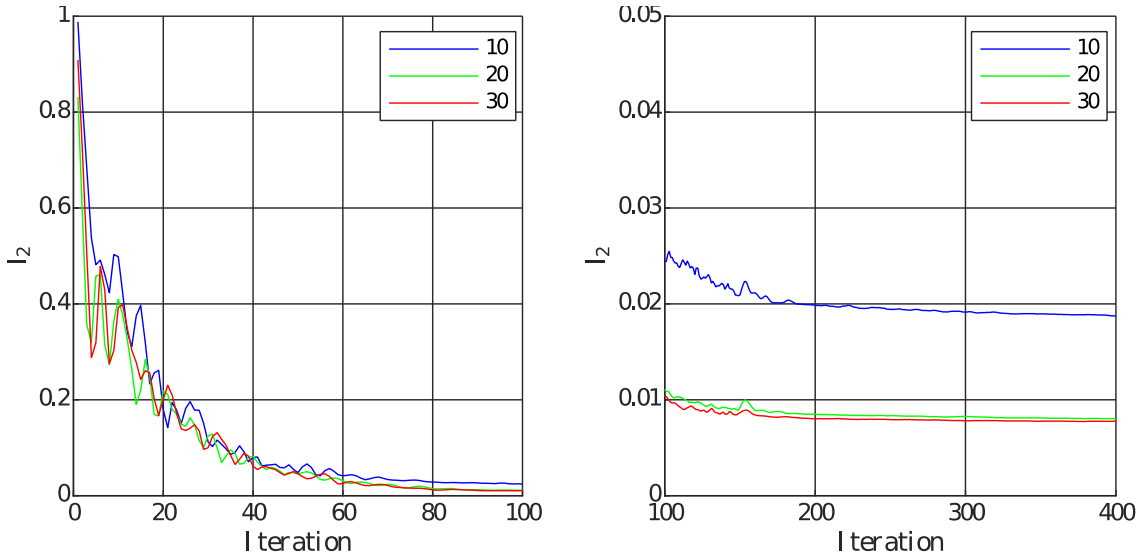

Figure 4-6: gNN with different number of neurons per hidden layer, l_2 error (figure by author)

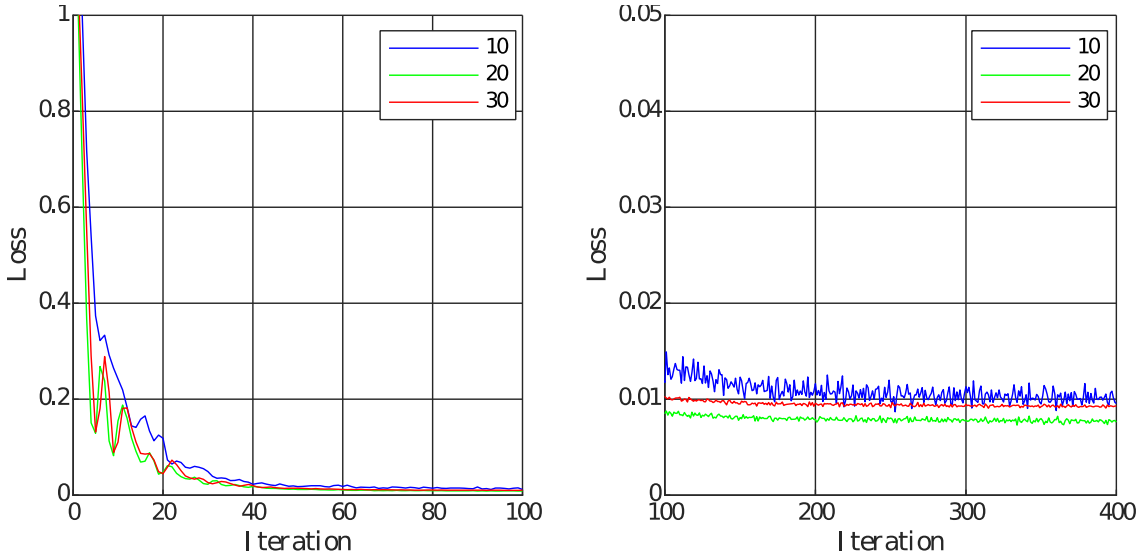

Figure 4-7: gNN with different number of neurons per hidden layer, loss (figure by author)

When changing the number of neurons per layer, one can observe in Figure 4-6 and Figure 4-7 that increasing the number of neurons improves the performance of $\boldsymbol{l_2}$ error. The losses do not differ too much.

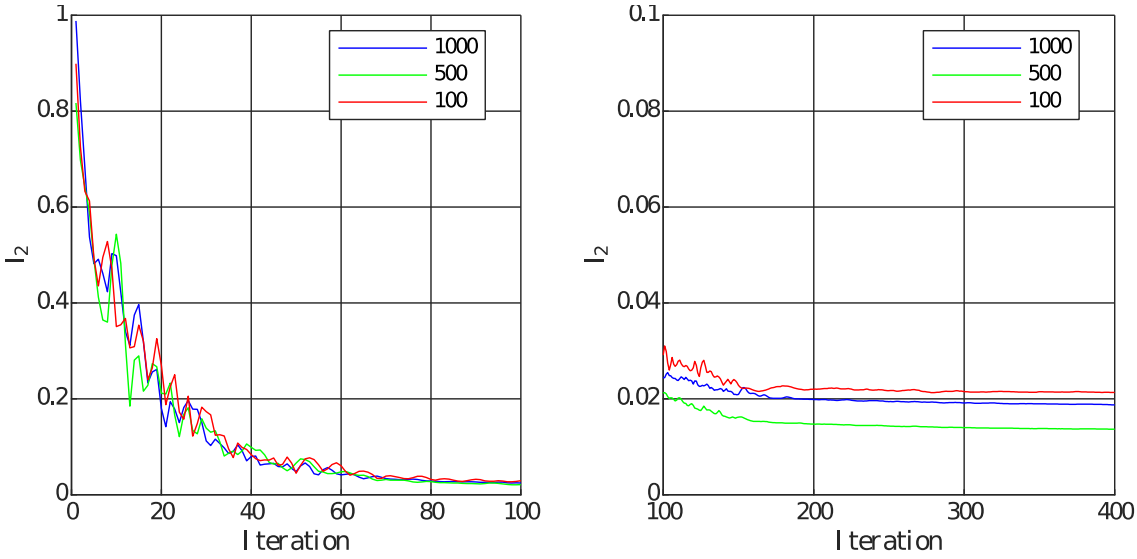

Figure 4-8: gNN with different training data sizes, l_2 error (figure by author)

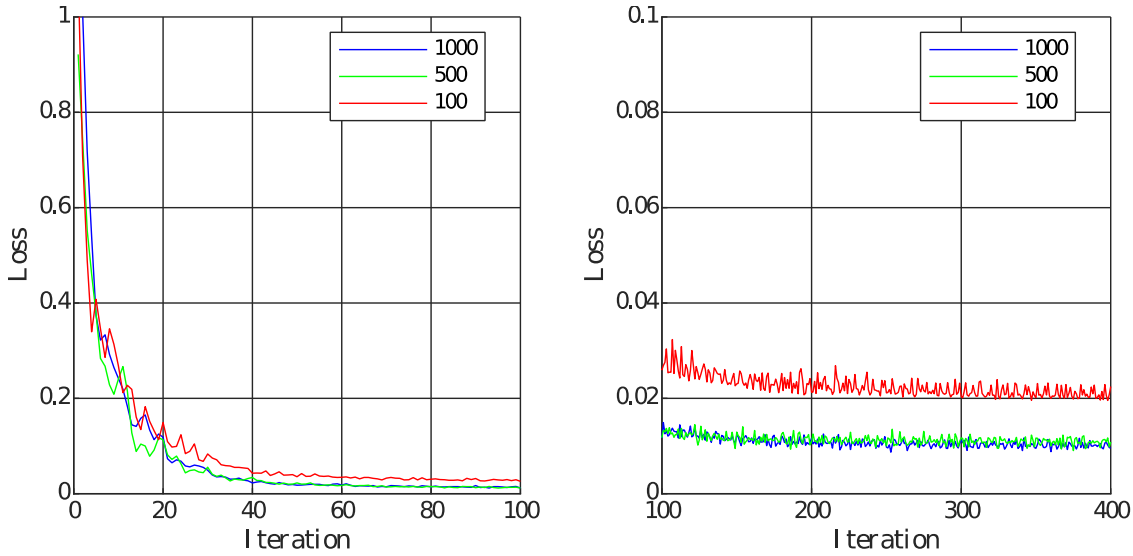

Figure 4-9: gNN with different training data sizes, loss (figure by author)

In Figure 4-8 and Figure 4-9 one can observe that the variant with 500 training samples performs the best when it comes to the l_2 error. However since the variants are trained on different sized training data, the potential issue arising is that certain training and validation samples are not included in one or the other variation. In this case, observing the loss holds more insight in the performance of the training. It can be seen that the more training data is provided, the lower and better the loss converges, with the 1000 training data size variant having a very small edge compared to the 500 training data size variant.

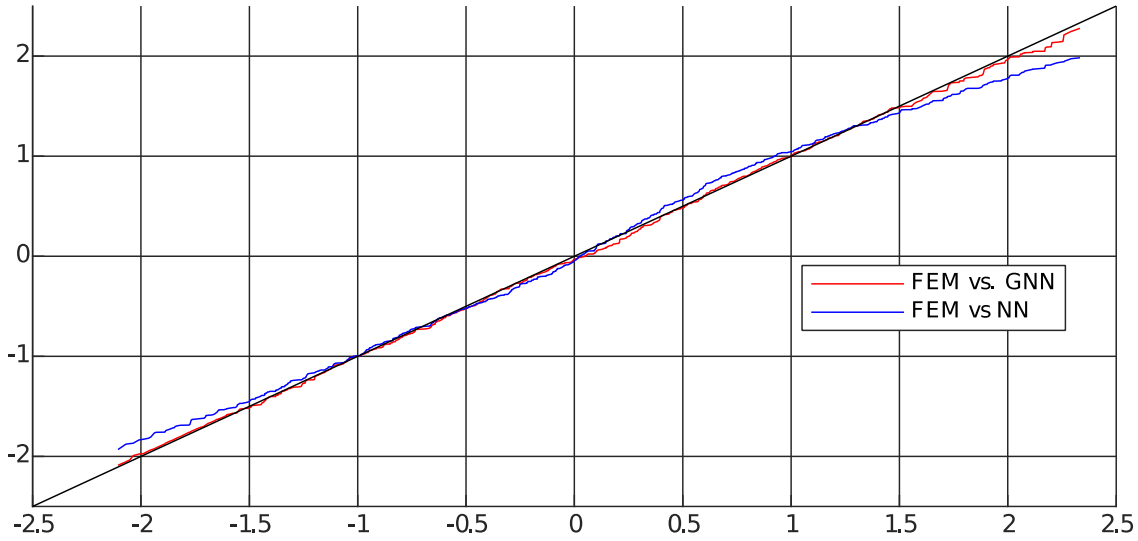

Figure 4-10: Confusion chart, FEM vs. gNN in red and FEM vs. NN in blue. The black diagonal represents a perfect fit (figure by author)

As can be observed in Figure 4-10, the gNN is closer to the optimal fit, represented by the black diagonal line, than the basic neural network. The basic neural network seems to have accuracy trouble especially at the edges of the data range. The gradient information provided in the gNN allows a certain degree of extrapolation from the range of given training values, which is why the gNN is not showing this same behavior at the edges of the data range.

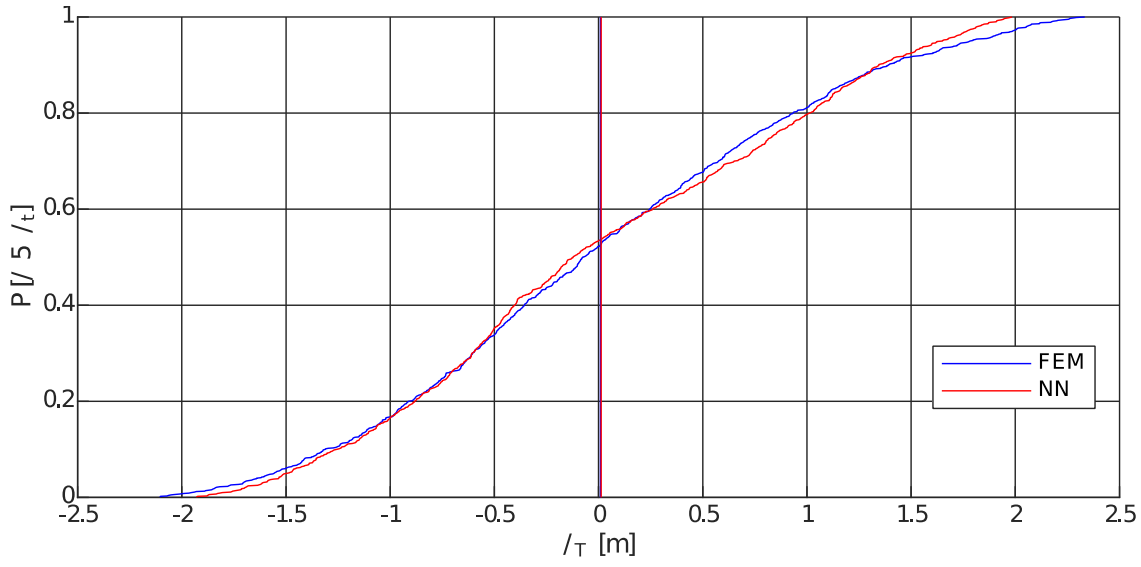

Figure 4-11: Exceedance curve, FEM vs. NN. The means of both FEM and NN are drawn in as vertical lines (figure by author)

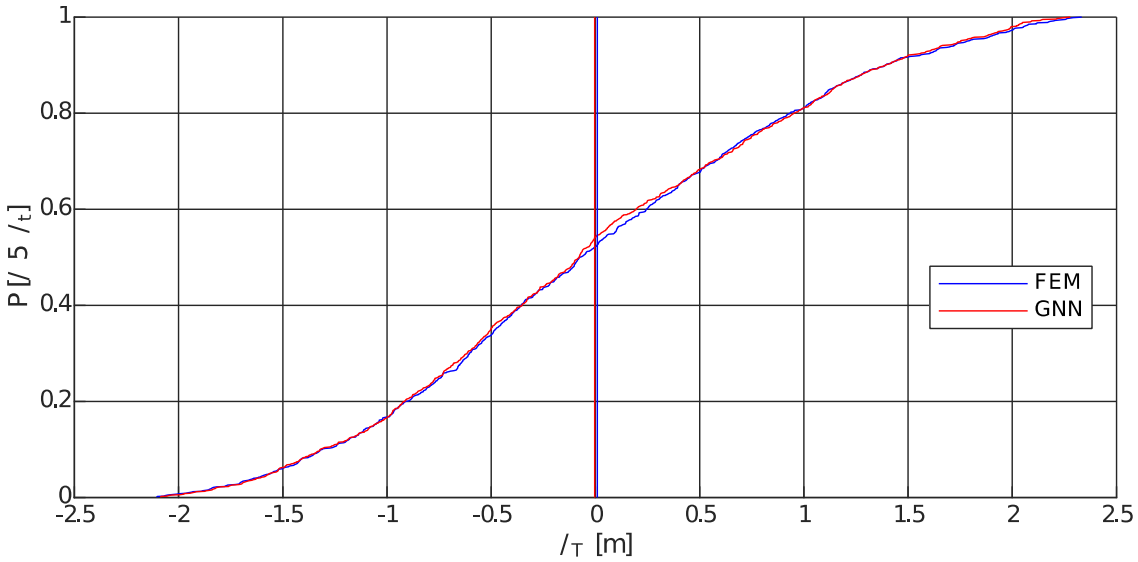

Figure 4-12: Exceedance curve, FEM vs. gNN. The means of both FEM and gNN are drawn in as vertical lines (figure by author)

Both Figure 4-11 and Figure 4-12 also confirm this observation. The exceedance curve of the gNN is closer to the exceedance curve of the FEM, compared to the exceedance curve of the NN. In both cases the mean is nearly equal.

Comparisons – Nonlinear Case

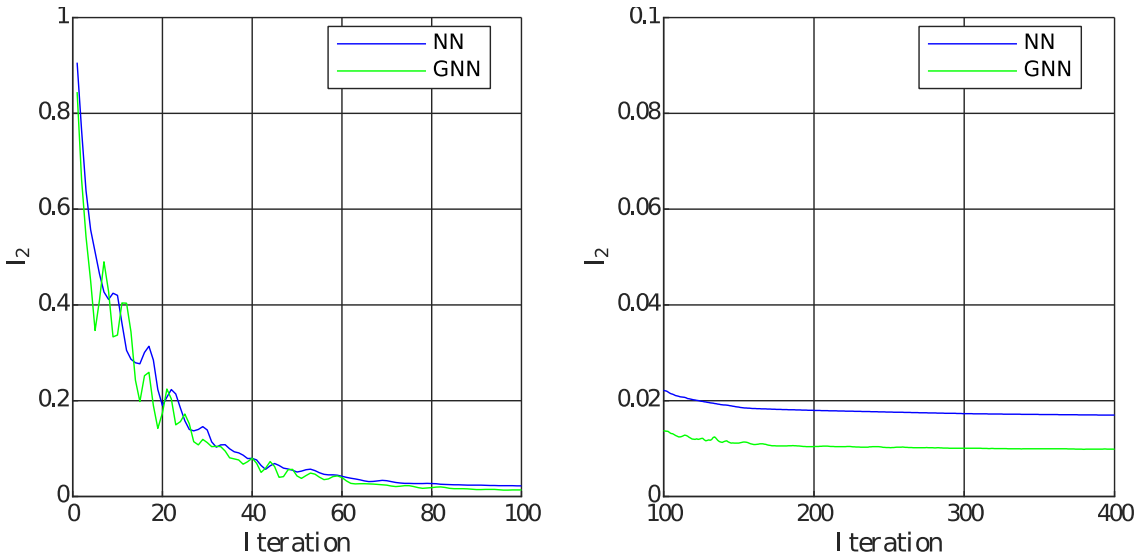

Figure 4-13: : gNN vs. NN, l_2 error (figure by author)

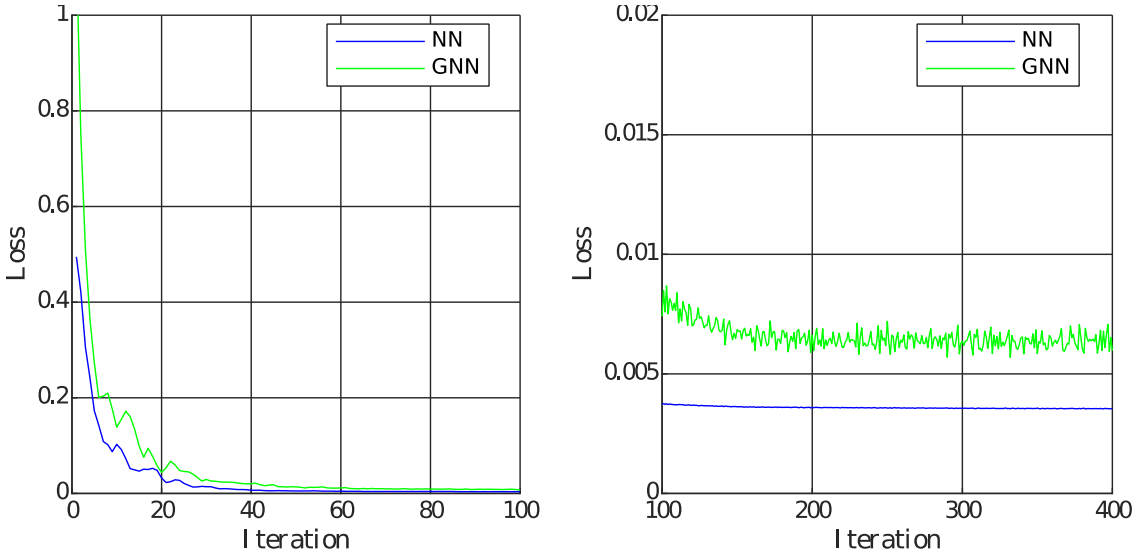

Figure 4-14: gNN vs. NN, loss (figure by author)

In the nonlinear case, when comparing the gNN with the NN in Figure 4-13 and Figure 4-14, the results show the gNN again outperforming the NN. While the loss of the gNN is again greater and more unstable than that of the NN, its l_2 error is considerably lower. Taking advantage of gradient information seems more essential in more complex systems.

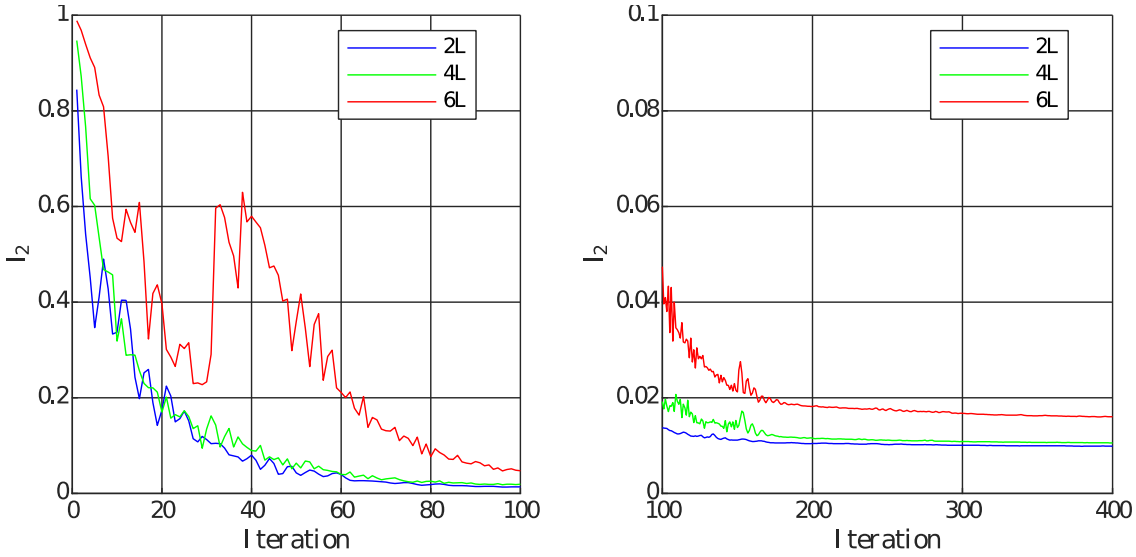

Figure 4-15: gNN with different number of hidden layers, l_2 error (figure by author)

Next, variation of hidden layer amount can be seen in Figure 4-15 and Figure 4-16. Again the 6 hidden layer variant shows great instability and difficulty to converge at the beginning of the training. However it does not improve and performs the worst. The 2 hidden layer variant has a slight edge to the 4 hidden layer variant when observing the l_2 error. Its loss is lower as well.

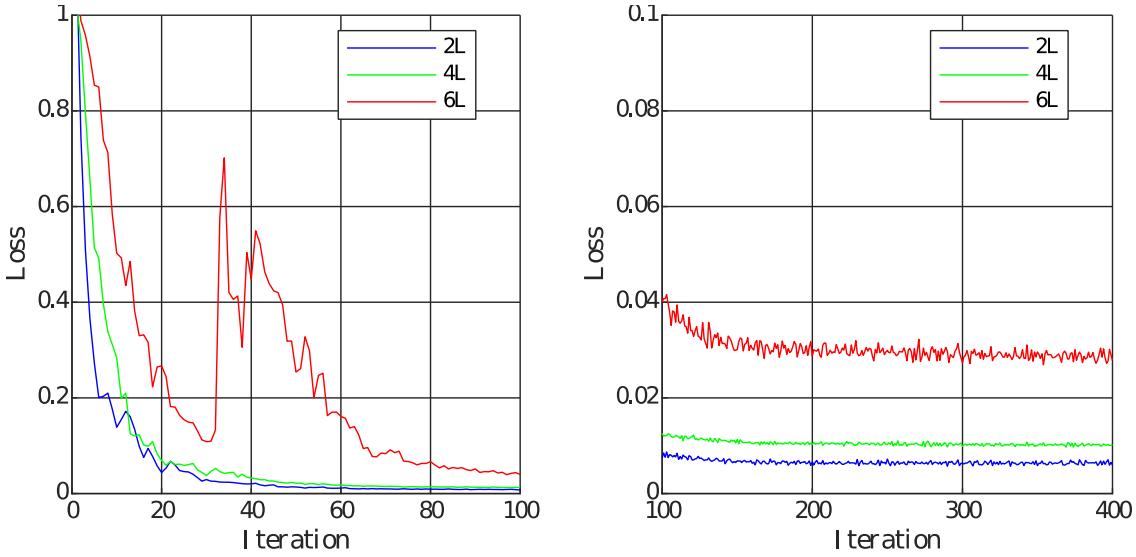

Figure 4-16: gNN with different number of hidden layers, loss (figure by author)

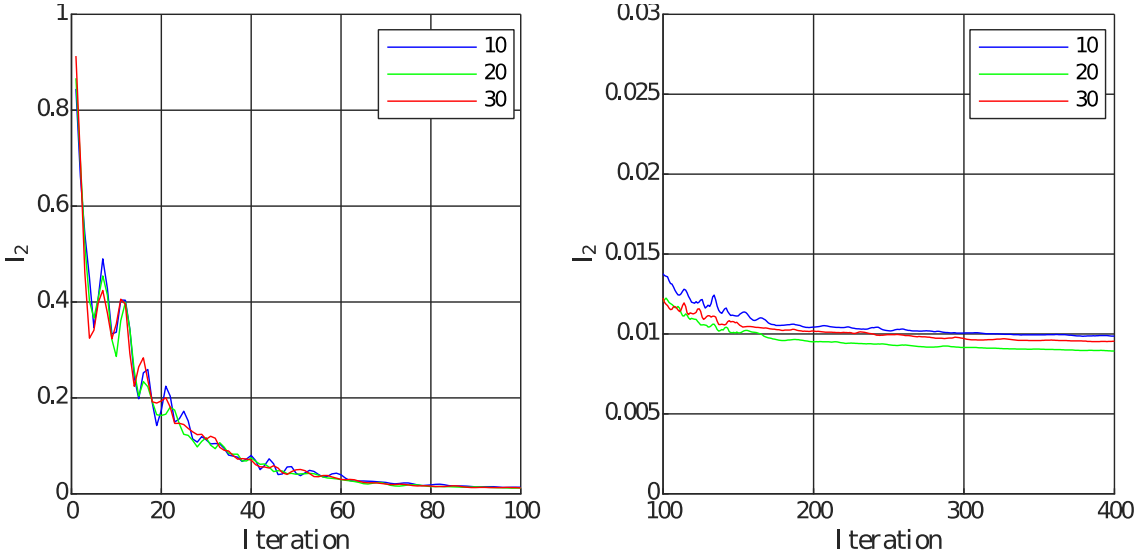

Figure 4-17: : gNN with different number of neurons per hidden layer, l_2 error (figure by author)

When varying the number of neurons per hidden layer, see Figure 4-16 and Figure 4-17, 20 neurons per hidden layer perform the best in the case of l_2 error. However all three variants are close in their performance. It can be seen that the optimal amount of neurons per hidden layer seems to lie around 20 neurons per layer. The loss of variant with 30 neurons per layer seems very unstable at the beginning of training.

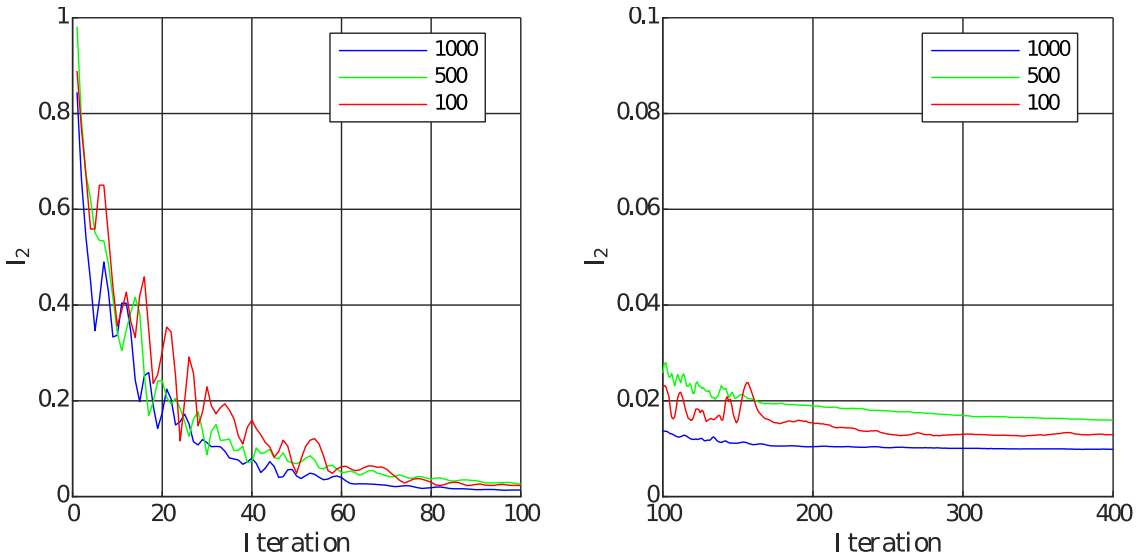

Figure 4-18: gNN with different training data sizes, l_2 error (figure by author)

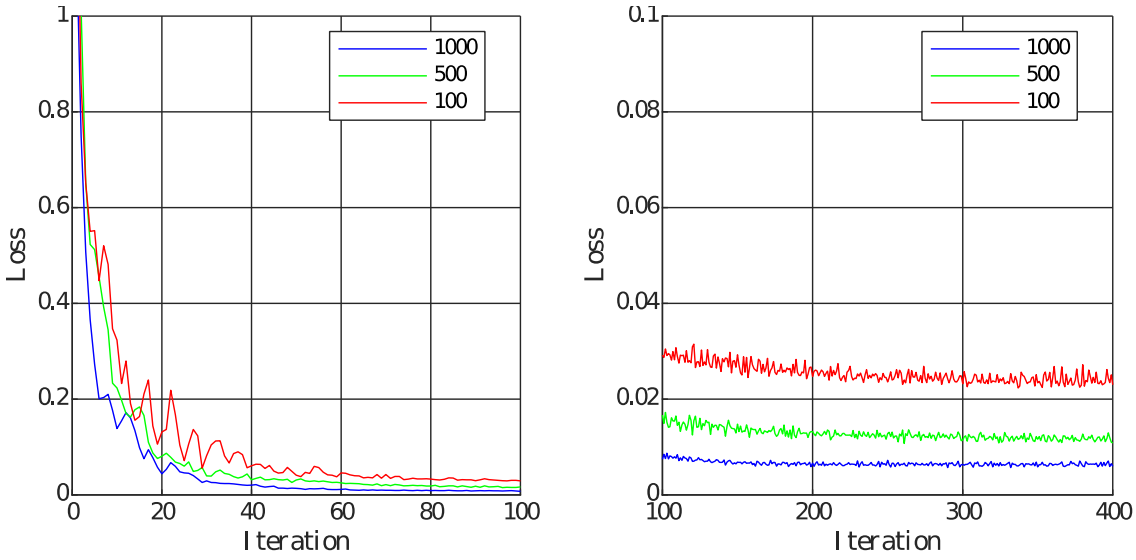

Figure 4-19: gNN with different training data sizes, loss (figure by author)

Lastly, results of varying the training data size for then nonlinear case can be seen in Figure 4-18 and Figure 4-19. Here one can clearly see a positive correlation between performance and training data size when observing the loss. Again, since the training data size varies, the validation data used to determine the l_2 error may differ, making this metric not as decisive as the other model variation results. The loss is therefore a more unbiased indicator of performance.

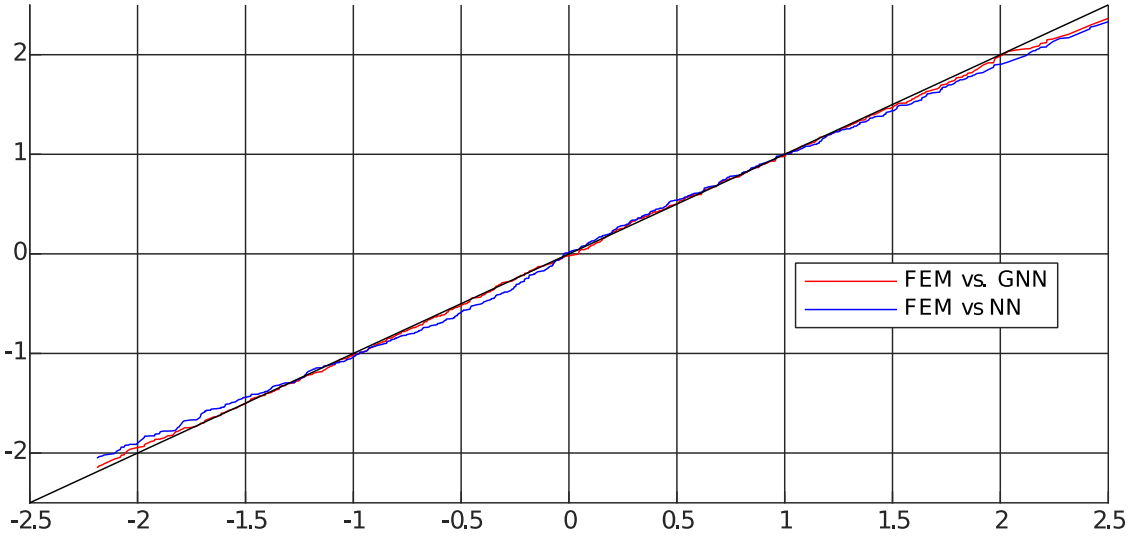

Figure 4-20: Confusion chart, FEM vs. gNN in red and FEM vs. NN in blue. The black diagonal represents a perfect fit (figure by author)

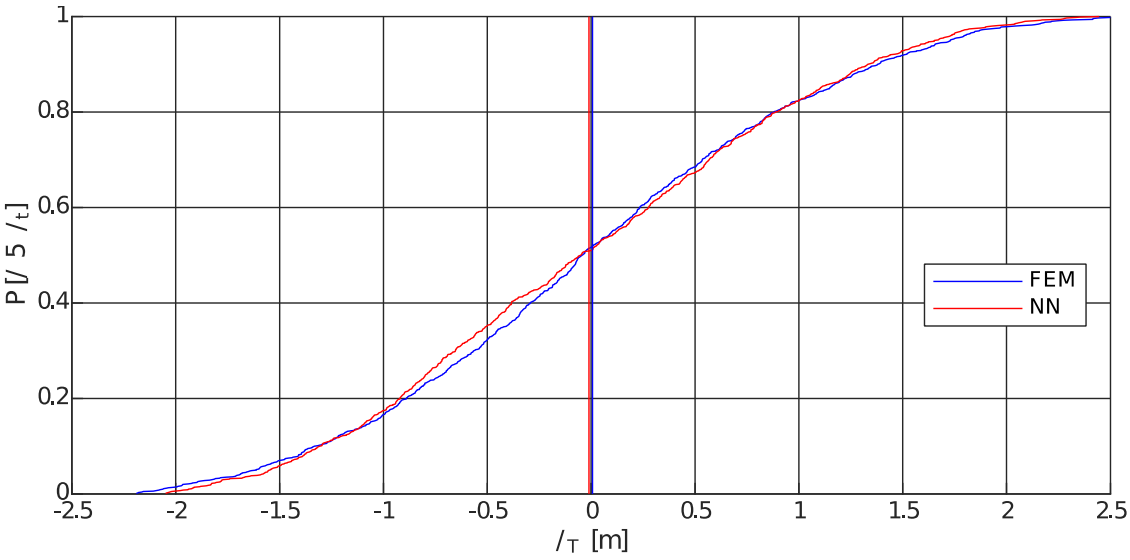

Figure 4-21: : Exceedance curve, FEM vs. NN. The means of both FEM and NN are drawn in as vertical lines (figure by author)

The confusion chart in Figure 4-20 shows the gNN having a better fit with the FEM predictions than the NN. The NN again shows greater accuracy errors at the edges, but also close to the middle. When observing the exceedance curves in Figure 4-21 and Figure 4-22 the same can be seen. The gNN conforms better to the FEM, than the NN does. The mean of the gNN is also closer to the mean of the FEM, than the NN mean is.

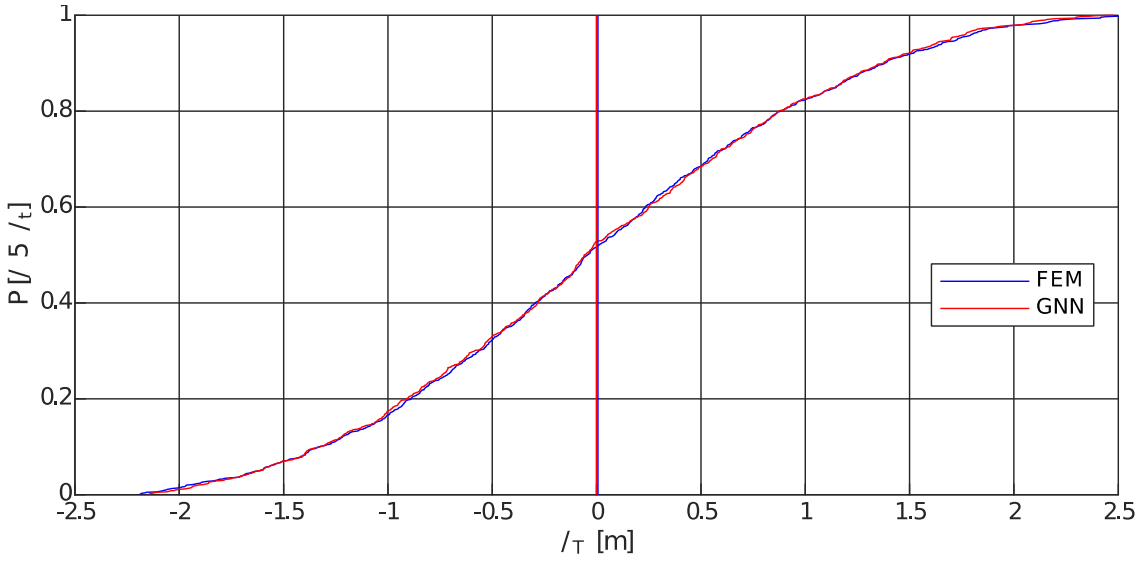

Figure 4-22: Exceedance curve, FEM vs. gNN. The means of both FEM and gNN are drawn in as vertical lines (figure by author)

4.2.3 Comparison – Weighing of Losses

In this section, the different variations of weights attached to the losses in the loss function will be presented. For the fixed weight variants, the gNN will be presented according to Table 4-6.

Table 4-6: Fixed loss weights table (table by author)

Variant	w_Y	w_{dY}
A	1	1
B	1	0.55
C	1	0.1
D	0.55	1
E	0.1	1

The results of these 5 variants for the linear case can be seen in Figure 4-23 and Figure 4-24. Concerning the loss, all variants are rather close to each other. Variant B performs best on loss, while variant A, where the weights are not changed at all, performs worst. However when considering the l_2 error, Variant A again performs worst, and Variant C performs best. Concerning the dynamic weight variations, there is the linear increase of the weight attributed to the base loss, w_Y. and the other variation, the linear decrease of the weight attributed to the gradient loss, w_{dY}. Their values can be seen in Table 4-7. The last variation is the cumulative distribution variant, from here on refer to with NORM, detailed previously in 3.4.4.

Table 4-7: Dynamic Weight Table (table by author)

Variant	w_Y	w_{dY}
LI	1, 1.2, 1.4, 1.6, 1.8, 2	1
LD	1	1, 0.8, 0.6, 0.4, 0.2, 0

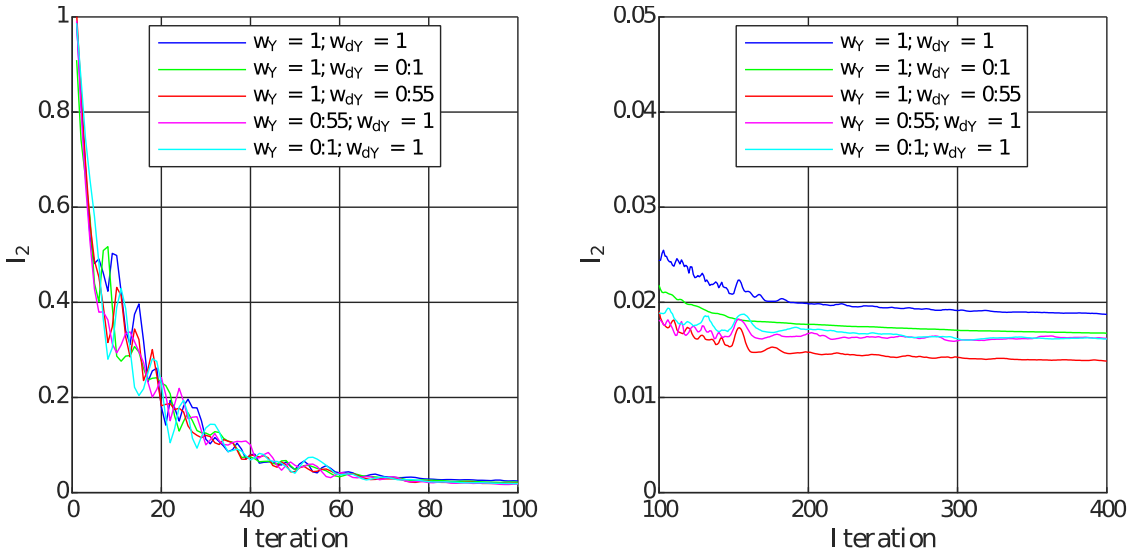

Figure 4-23: Fixed weight variants of gNN, l_2 error, linear (figure by author)

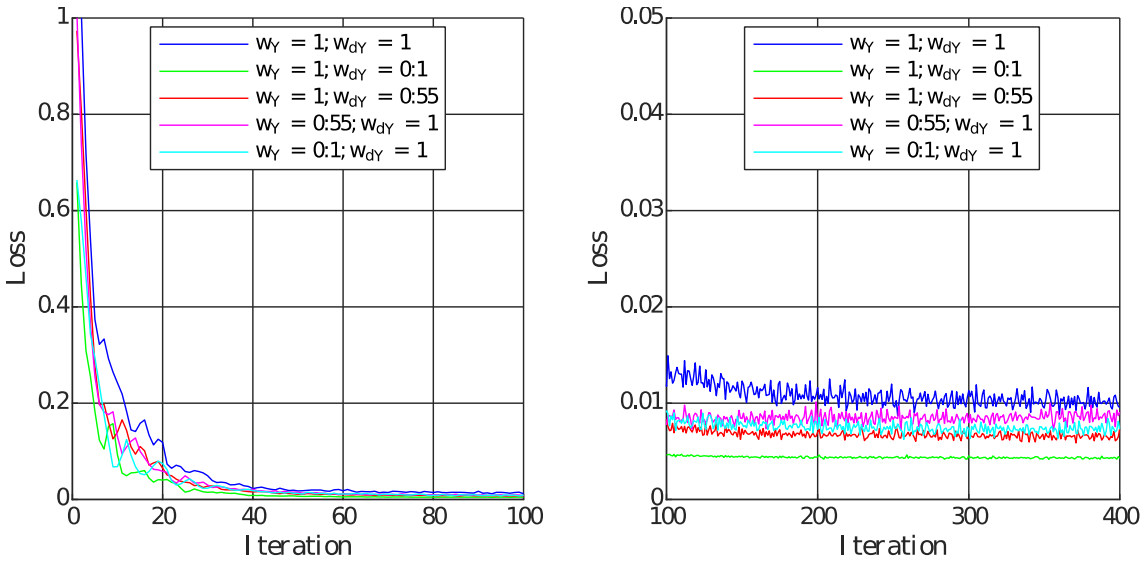

Figure 4-24: Fixed weight variants of gNN, loss, linear (figure by author)

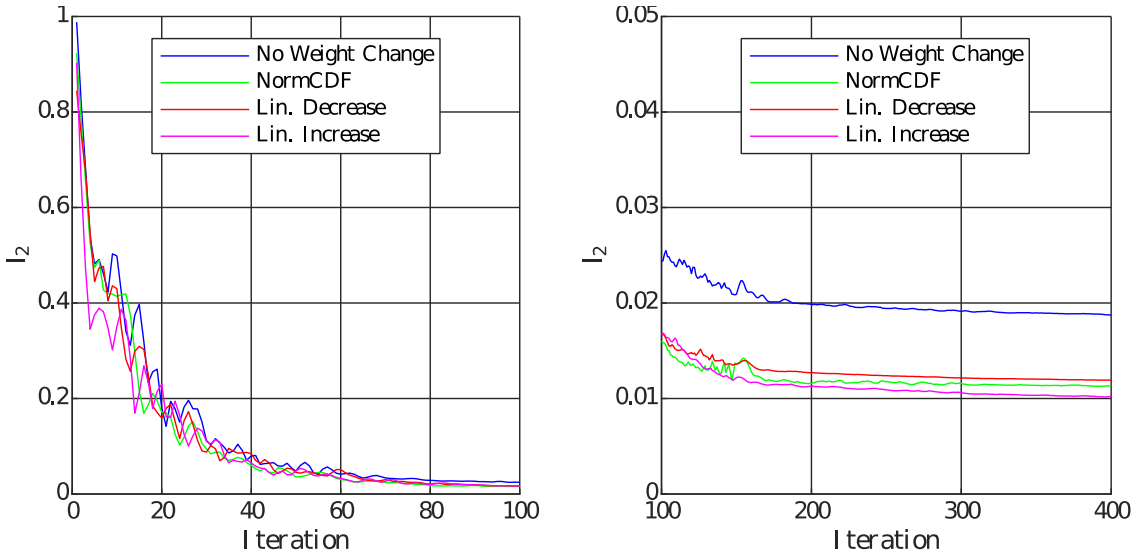

Figure 4-25: Dynamic weight variants of gNN, l_2 error, linear (figure by author)

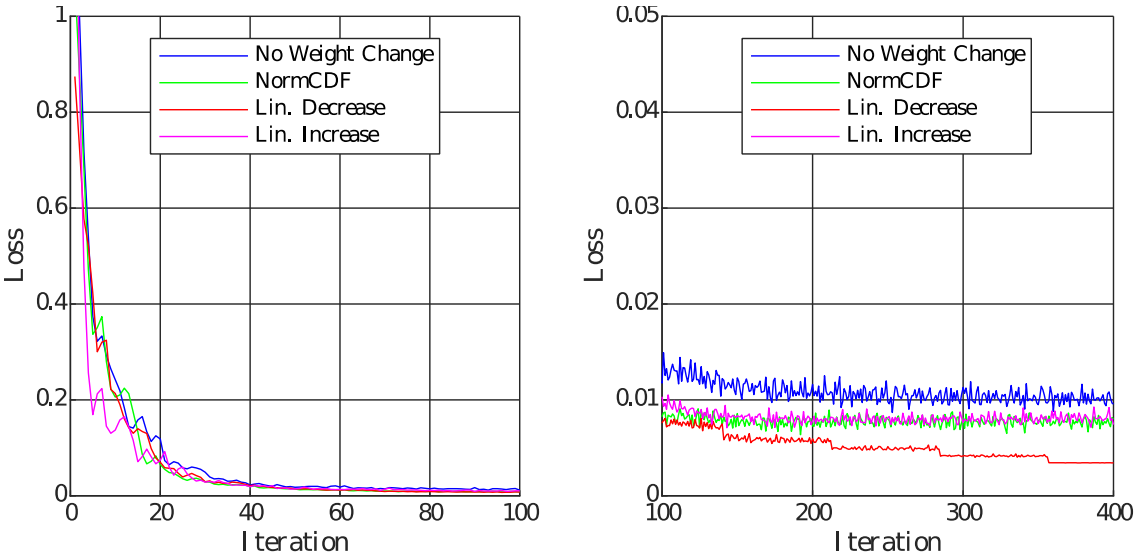

Figure 4-26: Dynamic weight variants of gNN, loss, linear (figure by author)

From Figure 4-25 and Figure 4-26 it can be observed that the no weight change gNN performs worst. The LI variant performs best with the NORM and LD variant close behind. The loss also shows the better loss converges for LI and NORM variants. However important to note is the very low loss of the LD variant. This is because the decrease of the one loss weight automatically reduces the total loss, which does necessarily not coincide with optimization performance increase. From these results it can be seen that the weights attached to the losses of the loss function can improve performance.

The same tables Table 4-6 and Table 4-7 are used for the nonlinear case. The results in Figure 4-27 and Figure 4-28show variant B performing worst and variant A performing best. Variant B shows the lowest loss, which has more to do with the smaller weights reducing the total loss value and not necessarily coinciding with an increase in performance. But variant E shows highest loss. When considering the dynamic weights in Figure 4-29 and Figure 4-30, the LD variants performs

worst in relation to the l_2 error. Meanwhile the basic gNN performs best, close or equal to the LI variant. For the loss, the LD variant is lowest, which is mainly because of the decrease of the weight and not because of better performance. The other variants are very close to each other concerning the loss. These results show a very different behavior than the linear case. Potentially there might have been simulations errors for the basic gNN for the linear case, or some simulation error during the nonlinear case. The previously mentioned paper, [112], shows improvement of results by linear increase of the weight of the gradient loss. This does not coincide with the results of the linear case, since the linear decrease of w_{dY} and the linear increase of w_Y should have the opposite effect of the linear increase of w_{dY}. Still, the nonlinear results of dynamic weights show that the LI and LD variant are not equal. Concerning the NORM variant, in both linear and non-linear cases it performs relatively close to the best variant.

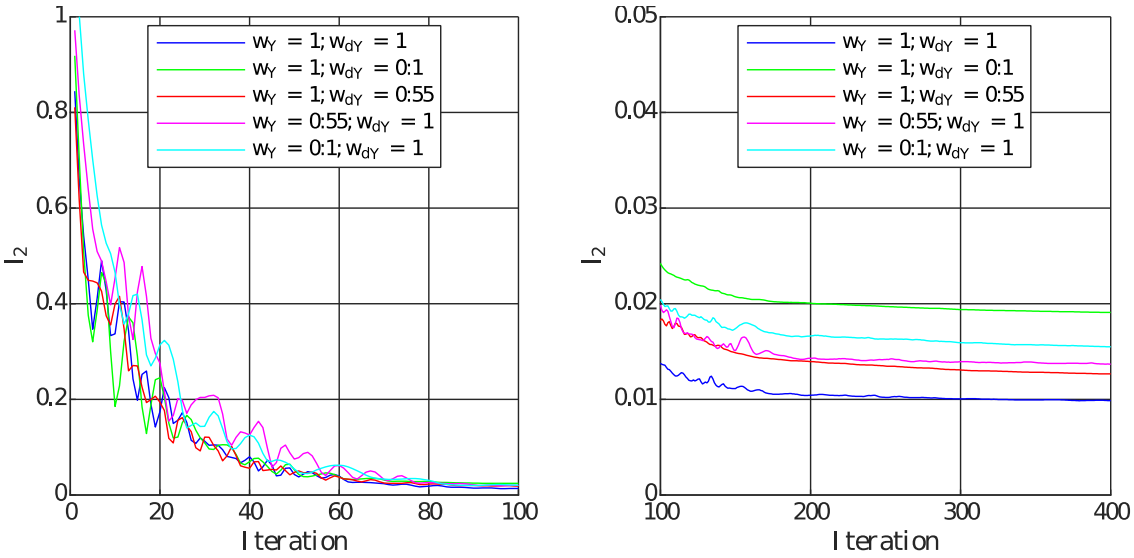

Figure 4-27: Fixed weight variants of gNN, l_2 error, nonlinear (figure by author)

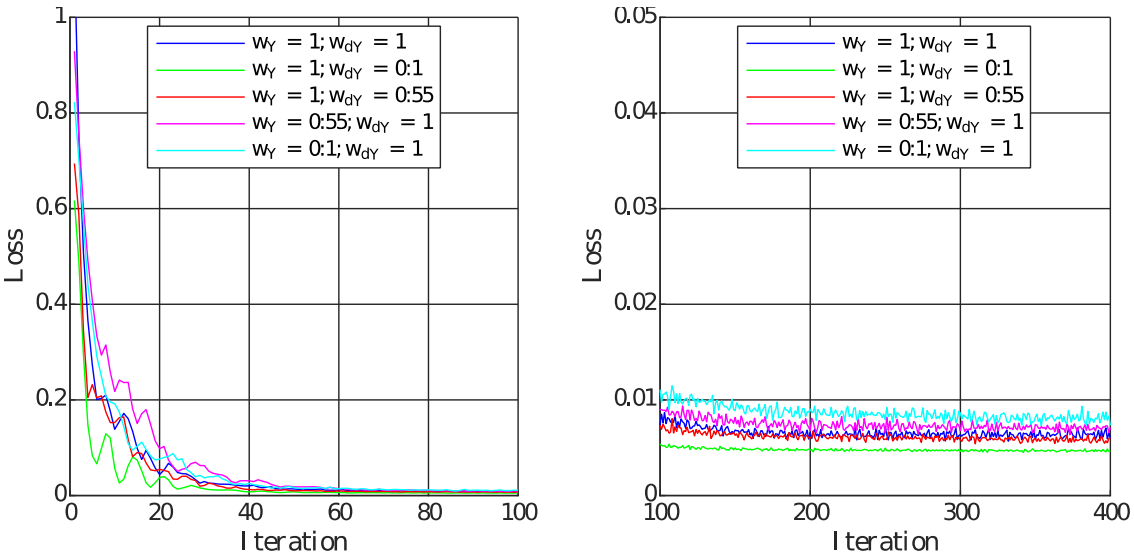

Figure 4-28: Fixed weight variants of gNN, loss, nonlinear (figure by author)

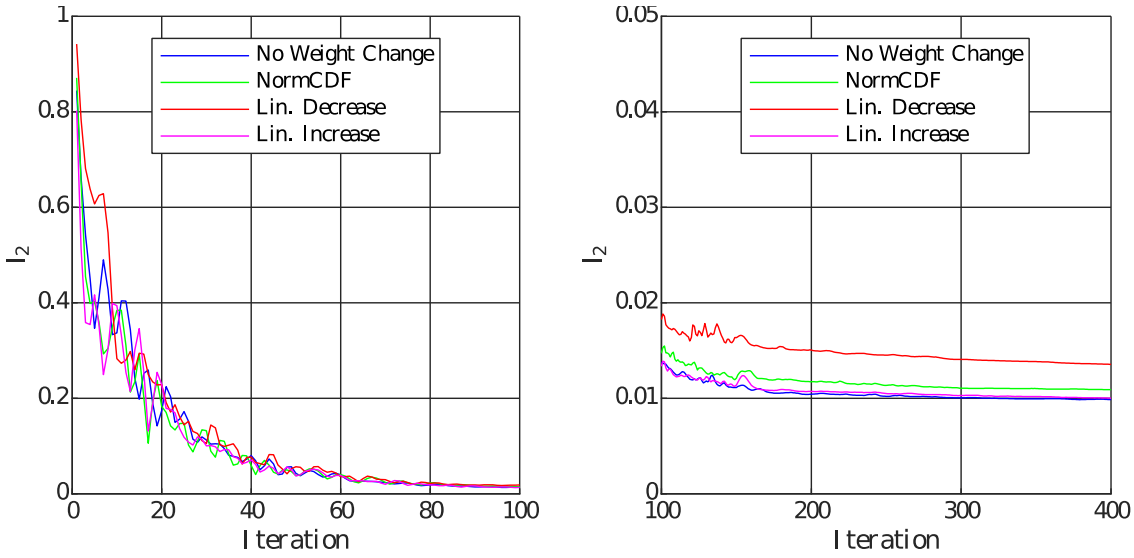

Figure 4-29: Dynamic weight variants of gNN, l_2 error, nonlinear (figure by author)

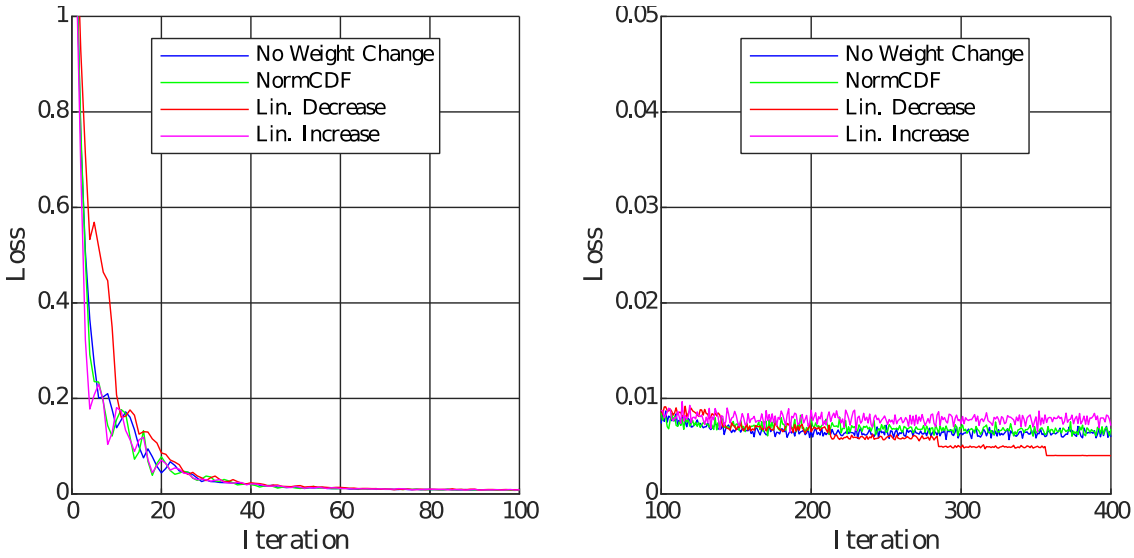

Figure 4-30: Dynamic weight variants of gNN, loss, nonlinear (figure by author)

4.3 Discussion

The results show certain clear tendencies, but in other aspects are rather unclear. Starting from the top of the results, it has been shown that taking advantage of gradient information during training improves the training of a neural network. In both linear and nonlinear cases, the accuracy of the gradient enhanced neural network is better and thereby closer to the FEM it approximates, reaching error values of <2%, with the best models that were trained going to <0.6% error. This is also shown strongly in the exceedance curves and the confusion charts. If the possibility exists to obtain gradient information, training a neural network with it, together with the usual function values, should always be considered. The training times or prediction times do not seem to be

affected at all by the additional gradient information. This is mainly because through automatic differentiation no new computations are done. Any additional computation of the gNN was part of the automatic differentiation of the NN to begin with. When adjusting layer size, neuron numbers or training data size there were clear correlations. There seems to be an optimal range of hyperparameters or design variables of the neural network model, that improve its performance substantially. However adjusting these hyperparameters seems to greatly influence training time. The only exception was the training data size. Since the minibatch size was adjusted according to the training data used, the time it took to train the models was no affected. In general however the universal approximation theorem holds true. Since the models in this work are only 2 input 1 output neural networks, the largest variations observed showed worse performance because of overparameterization. When considering the time it took each model to predict the 1000 validation data compared to the FEM, in both linear and nonlinear cases the neural networks are incredible quick compared to the FEM. This difference of prediction/generation time is especially great for the nonlinear case. Extrapolation the time it took the FEM to generate 1000 training samples for the nonlinear case, it would take ~9000 seconds, or 2 and a half hours, to generate 10000 training samples. Meanwhile a neural network could produce these values in under a second. Of course the training would not take considerably longer, as adjusting the minibatch size shows that it has no effect on the training time.

Concerning the weighing of the losses, the fixed weight variations showed that loss weights can have an impact on the performance of the trained model. However linear and nonlinear case showed different results making it rather inconclusive. It is not clear from the obtained results, which weight combination is best. Perhaps linear and nonlinear case behave completely differently concerning the weights of the losses. Similarly, the dynamic weight variants showed the same potential impact on the performance of a model. But again, linear and nonlinear case results behaved differently. Especially the comparison to the other published work, [112], could not be seen in the results exactly. While there was no results for a similar slow linear increase of the weight of the gradient loss in this work, the linear decrease was expected to not show any improvement as it did in the linear case.

Related to this is the issue of scale and magnitude difference between function value and gradient value. The results show mixed interpretations, when considering the fixed weight results. In the linear case, reducing the function value or the gradient value in the loss function improved results. In the case of the nonlinear elasticity results, none of the reduced weights performed best. This leaves the question whether the scale and magnitude difference is an actual issue for optimization besides the preprocessing for the case of vanishing or exploding gradients. More notably in all results, the decreased or increase value of the loss function through the adjusted weights rather influences the size of the gradient of the model, which ultimately adjusts the model parameters.

5 Conclusion

The main goal of this work was to analyze if gradient enhanced neural networks perform better than the basic neural networks. This was to be done by applying the models to linear and non-linear systems of elasticity. Additionally, the performance of the gradient enhanced model under different loss weighting methods was to be observed. Just as in the case of gPINN and SANN, the incorporation of gradient data improves performance. Furthermore, the gradient enhanced neural network in this thesis does not include tedious partial derivative expressions but relies simply on the provided data and its inherent structure and patterns. Combined with the simplistic design of neural networks, adding the gradients with respect to the inputs as a loss addend is an obvious choice. The comparison of basic neural network and gradient enhanced neural network was done through various metrics applied to the data of the finite element model and the data of the neural networks, like the loss and the relative l_2 error. An additional goal was to assess the feasibility of gradient enhanced neural networks as surrogate models replacing tedious numerical models like the finite element model used in this thesis to compute the training data. For this training and prediction times were compared. For fields of study like uncertainty quantification, a large number of samples need to be computed for its various uncertainty assessment methods. By visualizing exceedance probability curves of neural networks and the finite element model together, a comparison could be made to evaluate this feasibility. Lastly, the scale difference between training data $y \; and \; \frac{dy}{dx}$ was considered as an issue for gradient enhanced neural networks, which is why weights were applied to each loss term in the loss function. To remedy this scale difference standardization was applied. This however still left a considerable scale difference between both training data $y \; and \; \frac{dy}{dx}$. Therefore, weight factors according to described weight methods were applied to the losses and the results compared, to gauge the effect of these weights on the performance. Since choosing the weight factors could be potentially challenging, an additional late goal of this work was weight optimization algorithms akin to the training algorithms of a neural network. The goal was to allow the models to optimize these weight factors to see if a simple optimization design would allow a neural network to optimize them on its own.

The results show that incorporating the gradient data during loss optimization greatly improves performance of a neural network. While the total loss is bigger because of the additional gradient losses, the relative l_2 error shows better approximation for the gradient enhanced neural network to the finite element model's output. Because of this, the loss is considerably less valuable as a metric when comparing between basic neural network and gradient enhanced neural network. When looking at the exceedance probability curves, the gradient enhanced neural network shows great potential for substituting the finite element model. The gradient enhanced model adheres extensively well to the FEM computations, compared to the basic model. Since this is done with previously unseen data for the neural network models, this shows how well the models are able to approximate the inherent structure of the computed FEM data. The dimensions used in this work for the gradient enhanced neural network are rather small (layer size, neuron number) leading to relative l_2 errors well above 10^{-3}. In general, a neural network approximation with a

relative l_2 error of smaller than 10^{-3} can be considered usual. Besides the dimensions, other issues arise from hyperparameters. The learn rate schedule used in this work, a time based decay, coupled with the decay hyperparameter led to quickly decreasing learning rates the bigger the training data was, thereby constraining the potential of the chosen dimensions of models. Which is why it was swapped with a more basic case based learn rate update rule. The results also showed that for most models, the most optimal state during training was always the last or close to the last epoch, meaning if the models had been trained longer, they could have achieved greater accuracy. The results show the gradient enhanced neural network is already approximating the finite element model very well, even with the hyperparameters used in this work. Especially the processing times are promising in this regard. If a model is trained once, it is then capable of propagating large uncertainties necessary for uncertainty quantification. In the non-linear case, the finite element model takes considerable time to compute values. The processing time for all models after training are very small in contrast. Of course it needs to be considered that the models in this work are 1 output regression models, whereas the finite element model computes a multitude of values, making direct comparisons difficult, however the models of this work can be expanded for more dimensionality. Still, it has been shown that a gradient enhanced neural network performs better than a basic neural network. The gradient enhanced model does so by using less time and training points. In addition, considering the graphs of probability of exceedance, even with the given relative l_2 error values, the gradient enhanced neural network compares relatively well to the finite element model.

Further research should be done on greater dimensionality of gradient enhanced neural networks. If current results are possible with low dimensioned neural networks, an adequately sized model, with fine-tuned hyperparameters to allow for longer training of the model could converge to a more optimal state, even if there is greater input and output vectors. Models with more layers and more neurons per layer could potentially show the limits of gradient enhancement, which were not explored to such a degree in this work. Here, it would be important to consider the effects of the scale of the model on the necessary training time for optimal results, as oversizing the model leads to loss of performance. In addition, an adjusted learning rate schedule together with increased training data could allow for better performance as well. Another aspect to consider are multiple output gradient enhanced neural networks against a combination of multiple single output gradient enhanced neural networks. It could be interesting to consider optimizing the training time of these single potential models. In the case of multiple single output gradient enhanced neural networks, the possibility of training a model on a single output and then using this trained model to retrain for the other outputs could reduce any increase of training time introduced by greater dimensionality. Since the outputs are part of the same function structure, perhaps the retraining of a trained model is faster than the complete training of a new model. Furthermore varying activation functions could potentially help reduce the error of the models. Since the data in this work was standardized around a mean of 0, the ReLU might potentially not be an optimal function to consider. Instead, leaky ReLU or other activation functions that include negative values should be considered as well for increasing the model accuracy.

Addressing the loss weights that were used, coming to a solution for the difference of scale between model output and its gradient with respect to input needs to be explored. Since the func-

tion value and gradient value could potentially out scale each other in the loss function, this potentially limits the improvement of performance through incorporation of gradient information. Various methods could be explored to avoid this, such as using a relative based loss function instead of the MSE, like the mean percentage error (MPE), or even a combination of the MPE and the MSE. There is of course many more ways of varying a neural network, of which many were not considered in this work because of scope.

Lastly and in addition to this, the fixed and dynamic weight methods have shown potential in adjusting the loss terms in the loss function through use of predefined fixed weights and dynamic, trained weights. However this was rather inconclusive in many points. More data and results are necessary to make clear conclusions on the effects of loss weights on the performance of neural networks. Especially exploring the use of multiple different use cases, experimental data, or simulation data for training, or simply academical function evaluations could provide insight to this. Concerning the dynamic weight methods, the use of different optimization algorithms could provide interesting results. As it is, the weights tend towards 0 or -Inf, as they try to minimize the loss function and they are simply factors attached to the loss terms. This was fixed by defining the weights as the output of normal cumulative distribution functions. Perhaps a different function expression for the weights could lead to performance improvement. Lastly, while in this work, the weight of the losses were optimized according to the gradient of the loss function, perhaps optimizing the weights according to the gradient of the gradient of the loss function with respect to the model parameters is more logical for improving performance.

With all of this said and to summarize, taking advantage of gradient information for regression tasks should always be done as it clearly improves performance of trained neural networks. Furthermore weights can potentially increase performance and a method of finding optimal predefined hyperparameters, such as layer size, neuron count, activation function, etc., can also improve the performance. And approximating a finite element model with a neural network, like a gradient enhanced neural network, to use as replacement for arduous and time consuming computations is possible with an acceptable error of $<10^{-2}$.

List of References

[1] L. Zhang, L. Zhang and B. Du, "Deep Learning for Remote Sensing Data: A Technical Tutorial on the State of the Art," in *IEEE Geoscience and Remote Sensing Magazine*, vol. 4, no. 2, pp. 22-40, June 2016, doi: 10.1109/MGRS.2016.2540798

[2] Riccardo Miotto and others, Deep learning for healthcare: review, opportunities and challenges, *Briefings in Bioinformatics*, Volume 19, Issue 6, November 2018, Pages 1236–1246, https://doi.org/10.1093/bib/bbx044

[3] Kaleem, Eemaan & Vallecillo, Cinthia & Al-Nasser, Heba & Haweyou, Melad & Sharma, Mitali & Abdelnour, Mena. (2020). Structural Health Monitoring Using A Numerical Model And An Artificial Neural Network For Damage Detection

[4] Runge, J.; Zmeureanu, R. Forecasting Energy Use in Buildings Using Artificial Neural Networks: A Review. *Energies* **2019**, *12*, 3254. https://doi.org/10.3390/en12173254

[5] Chan HP, Samala RK, Hadjiiski LM, Zhou C. Deep Learning in Medical Image Analysis. Adv Exp Med Biol. 2020;1213:3-21. doi: 10.1007/978-3-030-33128-3_1. PMID: 32030660; PMCID: PMC7442218

[6] *Thrun, Sebastian. "Learning to Play the Game of Chess." NIPS (1994).*

[7] Schraudolph, Nicol & Dayan, Peter & Sejnowski, Terrence. (1994). Temporal Difference Learning of Position Evaluation in the Game of Go. Advances in Neural Information Processing. 6.

[8] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14). MIT Press, Cambridge, MA, USA, 2672–2680.

[9] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90

[10] LeCun, Yann & Bengio, Y. & Hinton, Geoffrey. (2015). Deep Learning. Nature. 521. 436-44. 10.1038/nature14539.

[11] Vinyals, Oriol & Le, Quoc. (2015). A Neural Conversational Model. ICML Deep Learning Workshop, 2015.

[12] *Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners.*

[13] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... & Yoon, D. H. (2017, June). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture* (pp. 1-12).

[14] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09). Association for Computing Machinery, New York, NY, USA, 873–880. https://doi.org/10.1145/1553374.1553486

[15] He, Kaiming & Zhang, Xiangyu & Ren, Shaoqing & Sun, Jian. (2016). Identity Mappings in Deep Residual Networks. 9908. 630-645. 10.1007/978-3-319-46493-0_38.

[16] Ioffe, Sergey & Szegedy, Christian. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

[17] G. Huang, Z. Liu, L. Van Der Maaten and K. Weinberger, "Densely Connected Convolutional Networks," in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017 pp. 2261-2269. doi: 10.1109/CVPR.2017.243

[18] Zhang, Hongyi & Cisse, Moustapha & Dauphin, Yann & Lopez-Paz, David. (2017). mixup: Beyond Empirical Risk Minimization

[19] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10). Omnipress, Madison, WI, USA, 807–814.

[20] Ramachandran, Prajit & Zoph, Barret & Le, Quoc. (2017). Swish: a Self-Gated Activation Function.

[21] Tan, Mingxing & Le, Quoc. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.

[22] T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, "Focal Loss for Dense Object Detection," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 318-327, 1 Feb. 2020, doi: 10.1109/TPAMI.2018.2858826.

[23] Li, Xiaoya & Sun, Xiaofei & Meng, Yuxian & Liang, Junjun & Wu, Fei & Li, Jiwei. (2020). Dice Loss for Data-imbalanced NLP Tasks. 465-476. 10.18653/v1/2020.acl-main.45.

[24] Loshchilov, I., & Hutter, F. (2017). Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*

[25] Michael R. Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. 2019. Lookahead optimizer: k steps forward, 1 step back. Proceedings of the 33rd International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA, Article 861, 9597–9608.

[26] Sohn, K., Berthelot, D., Carlini, N., Zhang, Z., Zhang, H., Raffel, C. A., ... & Li, C. L. (2020). Fixmatch: Simplifying semi-supervised learning with consistency and confidence. Advances in neural information processing systems, 33, 596-608.

[27] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He and P. Dollár, "Designing Network Design Spaces," 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 2020, pp. 10425-10433, doi: 10.1109/CVPR42600.2020.01044.

[28] Glowinski, R. & Pan, T. (2022). Numerical Simulation of Incompressible Viscous Flow: Methods and Applications. Berlin, Boston: De Gruyter. https://doi.org/10.1515/9783110785012

[29] Sahin, I., Moya, C., Mollaali, A., Lina, G., & Paniagua, G. (2023). Deep Operator Learning-based Surrogate Models with Uncertainty Quantification for Optimizing Internal Cooling Channel Rib Profiles.

[30] Stefania Fresca, Luca Dede', and Andrea Manzoni. 2021. A Comprehensive Deep Learning-Based Approach to Reduced Order Modeling of Nonlinear Time-Dependent Parametrized PDEs. J. Sci. Comput. 87, 2 (May 2021). https://doi.org/10.1007/s10915-021-01462-7

[31] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. Understanding deep learning (still) requires rethinking generalization. Commun. ACM 64, 3 (March 2021), 107–115. https://doi.org/10.1145/3446776

[32] Papernot, Nicolas & McDaniel, Patrick & Jha, Somesh & Fredrikson, Matt & Celik, Z. Berkay & Swami, Ananthram. (2016). The Limitations of Deep Learning in Adversarial Settings. 372-387. 10.1109/EuroSP.2016.36.

[33] LeVeque, R. J. (2007). *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. Society for Industrial and Applied Mathematics.

[34] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. Numerical Recipes 3rd Edition: The Art of Scientific Computing (3rd. ed.). Cambridge University Press, USA.

[35] Hoffman Joe D. 2001. *Numerical Methods for Engineers and Scientists* 2Nd ed. rev. and expanded ed. New York: Marcel Dekker.

[36] *Raissi, M., Perdikaris, P., & Karniadakis, G.E. (2017). Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. ArXiv, abs/1711.10561.*

[37] Raissi, Maziar & Perdikaris, Paris & Karniadakis, George. (2017). Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations.

[38] Blechschmidt, Jan & Ernst, Oliver. (2021). Three Ways to Solve Partial Differential Equations with Neural Networks -- A Review.

[39] Montomoli, Francesco & Carnevale, Mauro & Massini, Michela & D'Ammaro, Antonio & Salvadori, Simone. (2015). Uncertainty Quantification in Computational Fluid Dynamics and Aircraft Engines. 10.1007/978-3-319-92943-9.

[40] *Ghanem, R.G., Higdon, D.M., & Owhadi, H. (2017). Handbook of Uncertainty Quantification.*

[41] Schoenholz, Samuel & Gilmer, Justin & Ganguli, Surya & Sohl-Dickstein, Jascha. (2016). Deep Information Propagation.

[42] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, 2(5), 359-366

[43] José Miguel Hernández-Lobato and Ryan P. Adams. 2015. Probabilistic backpropagation for scalable learning of Bayesian neural networks. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15). JMLR.org, 1861–1869.

[44] Lee, Minhyeok & Seok, Junhee. (2020). Estimation with Uncertainty via Conditional Generative Adversarial Networks.

[45] Scaman, Kevin & Virmaux, Aladin. (2018). Lipschitz regularity of deep neural networks: analysis and efficient estimation.

[46] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational physics, 378, 686-707.

[47] Czarnecki, W. M., Osindero, S., Jaderberg, M., Swirszcz, G., & Pascanu, R. (2017). Sobolev training for neural networks. Advances in neural information processing systems, 30.

[48] Yu, J., Lu, L., Meng, X., & Karniadakis, G. E. (2022). Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems. Computer Methods in Applied Mechanics and Engineering, 393, 114823.

[49] Zienkiewicz, O. C., Taylor, R. L., & Zhu, J. Z. (2005). *The finite element method: its basis and fundamentals*. Elsevier.

[50] Larson, M. G., & Bengzon, F. (2013). The finite element method: theory, implementation, and applications (Vol. 10). Springer Science & Business Media.

[51] Liedmann, J., & Barthold, F. J. (2020). Variational sensitivity analysis of elastoplastic structures applied to optimal shape of specimens. *Structural and Multidisciplinary Optimization*, *61*(6), 2237-2251.

[52] Zhang, G., Patuwo, B. E., & Hu, M. Y. (1998). Forecasting with artificial neural networks:: The state of the art. International journal of forecasting, 14(1), 35-62.

[53] Chollet, Francois. 2017. Deep Learning with Python. New York, NY: Manning Publications.

[54] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.

[55] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

[56] Murphy, K. P. (2018). Machine learning: A probabilistic perspective (adaptive computation and machine learning series). *The MIT Press: London, UK*.

[57] Reed, R., & MarksII, R. J. (1999). *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press.

[58] Bartlett, P. L., Montanari, A., & Rakhlin, A. (2021). Deep learning: a statistical viewpoint. Acta numerica, 30, 87-201

[59] Tishby, N., Pereira, F. C., & Bialek, W. (2000). The information bottleneck method. arXiv preprint physics/0004057.

[60] Deng, L., Hinton, G., & Kingsbury, B. (2013, May). New types of deep neural network learning for speech recognition and related applications: An overview. In 2013 IEEE international conference on acoustics, speech and signal processing (pp. 8599-8603). IEEE.

[61] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 25.

[62] Silver, David & Huang, Aja & Maddison, Christopher & Guez, Arthur & Sifre, Laurent & Driessche, George & Schrittwieser, Julian & Antonoglou, Ioannis & Panneershelvam, Veda & Lanctot, Marc & Dieleman, Sander & Grewe, Dominik & Nham, John & Kalchbrenner, Nal & Sutskever, Ilya & Lillicrap, Timothy & Leach, Madeleine & Kavukcuoglu, Koray & Graepel, Thore & Hassabis, Demis. (2016). Mastering the game of Go with deep neural networks and tree search. Nature. 529. 484-489. 10.1038/nature16961.

[63] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[64] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.

[65] Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735

[66] Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., ... & Ng, A. Y. (2014). Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*.

[67] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ... & Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144.

[68] Jordao, A., Nazare Jr, A. C., Sena, J., & Schwartz, W. R. (2018). Human activity recognition based on wearable sensor data: A standardization of the state-of-the-art. arXiv preprint arXiv:1806.05226.

[69] Salinas, D., Flunkert, V., Gasthaus, J., & Januschowski, T. (2020). DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, *36*(3), 1181-1191.

[70] Burkov, A. (2019). The hundred-page machine learning book (Vol. 1, p. 32). Quebec City, QC, Canada: Andriy Burkov.

[71] Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013, June). Rectifier nonlinearities improve neural network acoustic models. In Proc. icml (Vol. 30, No. 1, p. 3).

[72] Kamalov, F., Nazir, A., Safaraliev, M., Cherukuri, A. K., & Zgheib, R. (2021, November). Comparative analysis of activation functions in neural networks. In 2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS) (pp. 1-6). IEEE.

[73] Clevert, D. A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289.

[74] Meena, K. & Suriya, Santhi. (2020). A Survey on Supervised and Unsupervised Learning Techniques. 10.1007/978-3-030-24051-6_58.

[75] Freeman, C. D., and Bruna, J. 2016. Topology and geometry of half-rectified network optimization. arXiv preprint arXiv:1611.01540

[76] Safran, I., and Shamir, O. 2017. Spurious local minima are common in two-layer relu neural networks. arXiv preprint arXiv:1712.08968.

[77] Yun, Chulhee, Suvrit Sra, and Ali Jadbabaie. "A Critical View of Global Optimality in Deep Learning." arXiv preprint arXiv:1802.03487 (2018).

[78] Choromanska, A.; Henaff, M.; Mathieu, M.; Arous, G. B.; and LeCun, Y. 2015. The loss surfaces of multilayer networks. In Artificial Intelligence and Statistics, 192–204.

[79] Kawaguchi, K. 2016. Deep learning without poor local minima. In Advances in Neural Information Processing Systems, 586–594.

[80] Nguyen, Q., and Hein, M. 2017. The loss surface of deep and wide neural networks. arXiv preprint arXiv:1704.08045.

[81] Yun, C.; Sra, S.; and Jadbabaie, A. 2017. Global optimality conditions for deep neural networks. arXiv preprint arXiv:1707.02444.

[82] Du, S. S.; Lee, J. D.; Tian, Y.; Poczos, B.; and Singh, A. 2017b. Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima. arXiv preprint arXiv:1712.00779.

[83] Laurent, T., and Brecht, J. 2018. Deep linear networks with arbitrary loss: All local minima are global. In International Conference on Machine Learning, 2908–2913.

[84] Du, S. S.; Jin, C.; Lee, J. D.; Jordan, M. I.; Singh, A.; and Poczos, B. 2017a. Gradient descent can take exponential time to escape saddle points. In Advances in Neural Information Processing Systems, 1067–1077.

[85] Lee, J. D., Simchowitz, M., Jordan, M. I., & Recht, B. (2016). Gradient descent converges to minimizers. arXiv preprint arXiv:1602.04915.

[86] Dauphin, Y. N.; Pascanu, R.; Gulcehre, C.; Cho, K.; Ganguli, S.; and Bengio, Y. 2014. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Advances in neural information processing systems, 2933–2941.

[87] Jin, C.; Ge, R.; Netrapalli, P.; Kakade, S. M.; and Jordan, M. I. 2017. How to escape saddle points efficiently. arXiv preprint arXiv:1703.00887.

[88] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. Journal of Marchine Learning Research, 18, 1-43.

[89] Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. Wiley interdisciplinary reviews: data mining and knowledge discovery, 9(4), e1305.

[90] Laue, S. (2019). On the Equivalence of Automatic and Symbolic Differentiation. arXiv preprint arXiv:1904.02990.

[91] Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2013, May). Advances in optimizing recurrent networks. In 2013 IEEE international conference on acoustics, speech and signal processing (pp. 8624-8628). IEEE.

[92] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[93] Dozat, T. (2016). Incorporating nesterov momentum into adam.

[94] Ma, J., & Yarats, D. (2018). Quasi-hyperbolic momentum and Adam for deep learning. arXiv preprint arXiv:1810.06801.

[95] Lucas, J., Sun, S., Zemel, R., & Grosse, R. (2018). Aggregated momentum: Stability through passive damping. arXiv preprint arXiv:1804.00325.

[96] Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., ... & De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. Advances in neural information processing systems, 29.

[97] Dai, W., Dai, C., Qu, S., Li, J., & Das, S. (2017, March). Very deep convolutional neural networks for raw waveforms. In 2017 IEEE international conference on acoustics, speech and signal processing (ICASSP) (pp. 421-425). IEEE.

[98] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[99] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[100] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

[101] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural

networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

[102] Sagi, O., & Rokach, L. (2018). Ensemble learning: A survey. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 8(4), e1249.

[103] Le-Duc, T., Nguyen-Xuan, H., & Lee, J. (2023). A finite-element-informed neural network for parametric simulation in structural mechanics. Finite Elements in Analysis and Design, 217, 103904.

[104] Klein, B. (2005). FEM: Grundlagen und Anwendungen der Finite-Element-Methode im Maschinen-und Fahrzeugbau. Springer-Verlag.

[105] Saltelli, Andrea & Annoni, Paola. (2010). Sensitivity Analysis. 10.1007/978-3-642-04898-2_509.

[106] Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., ... & Tarantola, S. (2008). Global sensitivity analysis: the primer. John Wiley & Sons.

[107] Yang, S., Xiong, F., & Wang, F. (2017). Polynomial chaos expansion for probabilistic uncertainty propagation. Uncertainty Quantification and Model Calibration, 13.

[108] Schueller, G. I. (2007). On the treatment of uncertainties in structural mechanics and analysis. Computers & structures, 85(5-6), 235-243.

[109] Miller, J. D. (2017). Statistics for data science: Leverage the power of statistics for data analysis, classification, regression, machine learning, and neural networks. Packt Publishing Ltd.

[110] Gouk, H., Frank, E., Pfahringer, B., & Cree, M. J. (2021). Regularisation of neural networks by enforcing lipschitz continuity. *Machine Learning*, *110*, 393-416.

[111] Son, H., Jang, J. W., Han, W. J., & Hwang, H. J. (2020). Sobolev training for the neural network solutions of pdes.

[112] Bouhlel, M. A., He, S., & Martins, J. R. (2020). Scalable gradient–enhanced artificial neural networks for airfoil shape design in the subsonic and transonic regimes. *Structural and Multidisciplinary Optimization*, *61*, 1363-1376.

Appendix/Appendices

Appendix A: MATLAB Program Code

Appendix B: Raw Data

Affidavit

Eidesstattliche Versicherung

(Affidavit)

Kilicsoy, Ali Osman Mert — 229749

Name, Vorname (surname, first name) — Matrikelnummer (student ID number)

☐ Bachelorarbeit (Bachelor's thesis) ■ Masterarbeit (Master's thesis)

Titel (Title)

Gradient-enhanced neural networks: applications in linear and nonlinear mechanics

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Duisburg, 15.08.23 — Ort, Datum (place, date)

Mert Kilicsoy — Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Duisburg, 15.08.23 — Ort, Datum (place, date)

Mert Kilicsoy — Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**