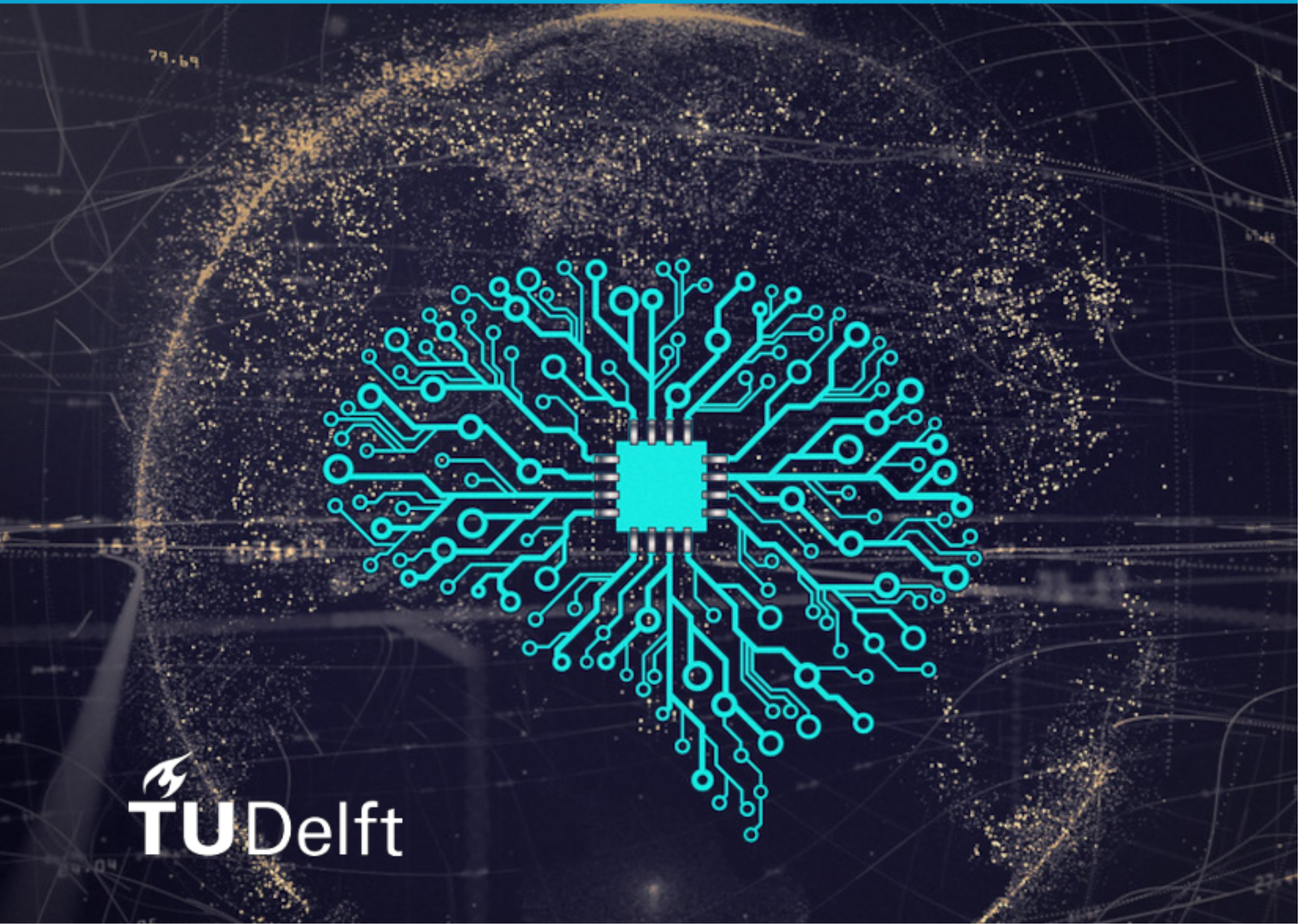
BSc thesis Applied Physics & Applied Mathematics

Solving Partial Differential Equations using Physics-Informed Neural Networks

Vlad Popa

2022



SOLVING PARTIAL DIFFERENTIAL EQUATIONS USING PHYSICS-INFORMED NEURAL NETWORKS

A thesis submitted to the Delft University of Technology in fulfillment of the requirements for the degree of

Bachelor of Applied Physics & Applied Mathematics

by

Vlad Popa

August 2022

Student number:	5113628
Supervisor (AP):	Prof. Dr. J.M. Thijssen
Supervisor (AM):	Dr. M. Möller
Co-reader (AP):	Dr. Q. Tao
Co-reader (AM):	Dr. J.L.A. Dubbeldam

ABSTRACT

In an attempt to find alternatives for solving partial differential equations (PDEs) with traditional numerical methods, a new field has emerged which incorporates the residual of a PDE into the loss function of an Artificial Neural Network. This method is called Physics-Informed Neural Network (PINN). In this thesis, we study dense neural networks (DNNs), including codes developed in the context of this bachelor project. We derive the backpropagation equations necessary for training and use different configurations in a DNN to test its interpolating accuracy. We distinguish between a-PINNs which use automatic differentiation to evaluate a PDE, and n-PINNs which approximate differential operators in a PDE with numerical differentiation. We compare both PINNs on the harmonic oscillator, the 1D heat equation and the 1-soliton and 2-soliton solutions of the Korteweg-De Vries (KdV) equation. Both PINNs could accurately converge to the solution, except to the 2-soliton solution, where the a-PINN outperformed the n-PINN. Furthermore, we tested a highly nonlinear problem of the KdV equation, which can be described by a train of solitons. We observed that PINNs are inaccurate if insufficient training samples are used for training. Adding training samples on the interior from a numerical solution leads to a good qualitative agreement, though more effort is required to find a better network configuration to obtain more accurate predictions.

Additionally, PINNs were used for inverse problems to derive an unknown coefficient in a PDE and proved to be highly accurate for noiseless data. When we generated training samples with 10% noise from a uniform distribution, the PINN results' relative error stayed within a margin of under 2%. However, inverse PINNs are much more inefficient compared to nonlinear least squares methods like the Levenberg–Marquardt algorithm.

As of now, PINNs are still very early in development and stand no match against traditional numerical methods to a known PDE. They may, however, provide a useful alternative in the future as they are constantly being improved.

Python codes used in this thesis are publicly available via https://github.com/vgpopa/BEP-thesis.

ACKNOWLEDGEMENTS

First, I would like to express my utmost gratitude to both supervisors, Prof. dr. J.M. Thijssen and dr. M. Möller. I thank Prof. dr. J.M. Thijssen for proofreading my thesis and providing extremely detailed feedback, and providing very insightful suggestions to improve my work. Furthermore, I thank dr. M. Möller for guiding me and for the insightful discussions about the wide applicability of neural networks. I also thank Mert Karaçelebi for his time, patience, and useful suggestions to improve my work.

Furthermore, I thank all the grading committee members for their time.

Additionally, I thank the author of this beautiful template, Hugo Ledoux, for making this template accessible under license CC0 1.0.

Finally, I have to thank my parents and friends for their unlimited support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYMBOLS

Table 0.1: List of symbols with dimensions used throughout the thesis.

symbol	definition	dimensions
l	layer	1
L	last layer	1
$\mathcal{L}$	loss function	1
N	amount of samples	1
n_l	amount of neurons in layer l	1
w_{jk}^l	weight of the connection from neuron k in layer $l-1$ to neuron j in layer l	1
b_j^l	bias of neuron j in layer l	1
z_j^l	pre-activation value of neuron j in layer l	1
a_j^l	activation value of neuron j in layer l	1
δ_j^l	$\frac{\partial L}{\partial z_j^l}$: the error of neuron j at layer l	1
$f^{(l)}(\cdot)$	activation function of layer l	
$\mathbf{x}_m$	input of the mth sample	n_0 x 1
$\mathbf{X}$	input data with N samples	n_0 x N
$\mathbf{W}^l$	weight matrix at layer l	n_l x n_{l-1}
$\mathbf{b}^l$	bias vector at layer l	n_l x 1
$\mathbf{z}_m^l$	pre-activated vector of the mth sample at layer l	n_l x 1
$\mathbf{a}_m^l$	activated vector of the mth sample at layer l	n_l x 1
$\boldsymbol{\delta}_m^l$	$\frac{\partial L}{\partial \mathbf{z}_m^l}$: the error vector of the mth sample at layer l	n_l x 1
$\mathbf{Z}^l$	pre-activated matrix of N samples at layer l	n_l x N
$\mathbf{A}^l$	activated matrix of N samples at layer l	n_l x N
$\boldsymbol{\delta}^l$	$\frac{\partial L}{\partial \mathbf{Z}^l}$: the error matrix of N samples at layer l	n_l x N

1 INTRODUCTION

Machine learning has become a prevalent and essential method for building mathematical models. In an era with huge quantities of available data, computers are superior to humans in evaluating relations between various data types and drawing conclusions. Computer programs can be built to make decisions solely based on data, without being explicitly programmed to do so. Their complexity and uncertainty to guarantee convergence to a solution for a particular problem has led these models to be seen as ”black boxes”. Machine learning algorithms are used in a very wide variety of fields such as voice, speech and image recognition, personalised advertisements, regression analysis, and genomics (Alipanahi et al. [2015]), to name a few. An example of a machine learning model is a *Dense Neural Network*, which is inspired and modeled after a biological brain. These have proven to be an excellent method for regression and classification problems. Furthermore, these have been proven to be universal function approximators by Hornik et al. [1989], implying that there exists a neural network which can approximate any continuous function to any desirable accuracy. For example, there is a neural network which is able to approximate the hidden relationship between a handwritten number and a digit, to correctly classify unseen data.

Physics phenomena are usually described in terms of partial differential equations (PDEs), which can be solved analytically in rare cases only. A whole industry of numerical algorithms has been developed in the last centuries and have proven to be very successful. However, highly nonlinear and high dimensional PDEs are problematic to set up and are very time-consuming to solve. An alternative method was proposed by Raissi et al. [2019], in which neural networks use information not only from data, but also from PDEs that are assumed to describe them. The resulting method is known as Physics-Informed Neural Networks (PINNs). Since then, a new field has emerged and neural networks have become an attractive and popular method for solving PDEs. While neural networks are largely accessible through popular libraries such as TensorFlow or PyTorch and are mostly already pre-programmed, the goal of this thesis is to investigate the mathematics inside ”the black box” of a neural network's learning phase by deriving the main backpropagation equations. These equations are used in our implementation of a dense neural network from scratch, without using already existing popular libraries for neural networks. Furthermore, PINNs' ability to interpolate and, most importantly, extrapolate solutions of mathematical models will be explored and investigated. We shall investigate the damped harmonic oscillator, the heat equation and the Korteweg–De Vries (KdV) equation as examples.

In chapter 2, the working of a dense neural network will be explained and a derivation of the backpropagation equations for the learning phase will be given. Next, the method of modifying neural networks to make them physics-informed will be explored in chapter 3. Afterwards, various architectures of PINNs with different settings will be investigated to research their ability to interpolate and extrapolate solutions of the harmonic oscillator, the heat equation and the KdV equation in chapter 4. Lastly, the performance of PINNs will be discussed in a conclusion in chapter 5. This thesis has been written as part of the double bachelor's degree programme Applied Physics & Applied Mathematics at Delft University of Technology.

All results of this research have been created in Python, and the Python codes are accessible via https://github.com/vgpopa/BEP-thesis.

2 THE DENSE NEURAL NETWORK

In this chapter, neural networks with fully connected layers, which are called dense, will be introduced. Different aspects which make up the configuration of a neural network will be discussed, and we shall discuss issues that may arise during training a dense neural network. The backpropagation equations used in the training phase are derived, which are used for the simplest training algorithm: steepest gradient descent. Neural networks can either be used in classification problems to correctly assign an observation to a set of categories (called a *class*), or to approximate an unknown function in regression analysis. While these networks are largely available in popular libraries, we aim to get a detailed insight into the workings of neural networks by encoding them ourselves. After we have constructed and validated our own neural network, we shall compare its outcomes and performance with PINNs.

2.1 MODELLING A NEURON

The very first idea of modelling a biological neuron was proposed by McCulloch and Pitts [1943], which would initiate a long mission towards simulating the human brain with artificial intelligence. A biological neuron accepts one or multiple electrical signals from other neurons, processes the information and depending on the information received, can fire up, and transmit the information further to other neurons. This process can be modelled as follows. The electric signals received are perceived as inputs, which mathematically can be represented as real numbers. For example, a picture for image recognition can be used as an input, by converting each pixel to a float. In machine learning, it is unnecessary to know the details of how biological neurons process their information, as long as our model keeps some essentials. One of these is the *activation function*: a function which tells the neuron whether to fire or not. Common activation functions will be discussed in Section 2.5. The type of input, together with the activation function defines the artificial neuron, which is the building block of any neural network. This neuron can be used for training in the simplest neural network possible: a *perceptron*.

2.2 A PERCEPTRON

A life cycle of a neural network has two modes of operation that are active in successive stages. The first stage consists of training the network on training data by also providing the network the correct outputs for some series of inputs. In the second stage, the network tries to predict data from new inputs. The objective is to create a network, in which the predicted outputs of the second stage are as close as possible to the true outputs. The error between the predicted outputs and the true outputs is evaluated with a *loss function* (often also called *cost function*), which is discussed in Section 2.6. It will be clear that this loss function is to be minimized.

The working of a neural network is best understood when considering a neural network of only one neuron: the perceptron. Each of the input values x_i is multiplied by a weight (w_i), the sum is taken over all inputs, and a real number is added, the so-called bias term (b). The weights and bias are determined during the training phase. The resulting value is called the *pre-activation* (z), and lastly, applying the activation function on z yields an *activation* (a). It turns out that the bias term is an important parameter for the model, as it can allow a shift of the activation function, which is independent of the inputs, and can significantly improve the training of the model. For a perceptron, a is the predicted output ($\hat{y}$). The workings of a perceptron is summarised in Figure 2.1.

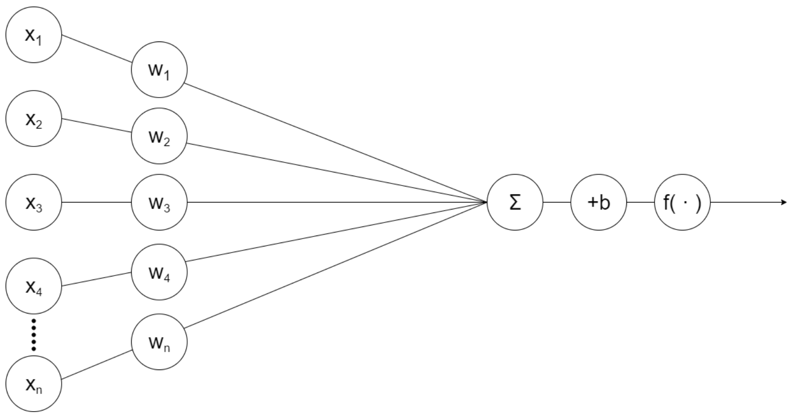


Figure 2.1: Illustration of a working perceptron.

The parameter space (Θ) of the model contains the weights and biases. Initially all parameters of the model are randomized. Then, the operations are performed as illustrated in Figure 2.1, which is also known as a *forward propagation*. The predicted output is compared to the "true" output, often called *target* or *label* (y), with the loss function. The goal is to minimize the error from the loss function, which is done by adapting the model parameters Θ. This is done with a learning algorithm, which will be discussed in detail in Section 2.7. It requires calculating the gradient of the loss function with respect to every model parameter. In Section 2.10 it will be illustrated how this is done. The amount of iterations performed in the training phase is often called *epochs*. Whenever the model has been trained enough according to some given criterion, the model enters the second stage and performs one forward propagation to yield a predicted output. In practice, perceptrons are too simple to be used for more complex problems. In fact, for classification problems, perceptrons can only be used as linear binary classifiers, as shown by Freund and Schapire [1999]. By using the linear activation function $f(x) = x$, a perceptron can be used for linear regression. However, perceptrons are never used for linear regression as the ordinary least squares estimators have proven to be the best linear unbiased estimators by the Gauss-Markov Theorem (Theil [1971]). Instead, more neurons are required to form multilayer perceptrons, as is the case in a dense neural network.

2.3 DENSE NEURAL NETWORK

In the literature, there are countless types of neural networks being developed, each having a unique architecture and custom rules to solve specific types of problems. Multiple neurons are placed in a layer, which can be used with multiple layers to form a *multilayer perceptron*. In a *dense* neural network, each neuron in a layer is connected with each neuron from the previous layer and the next layer. Each connection contains a weight: a model parameter which can be seen as the influence of a neuron from the previous layer. The bias is added at each neuron. This complex system is illustrated in a directed graph in Figure 2.2, in which each neuron is represented by a node.

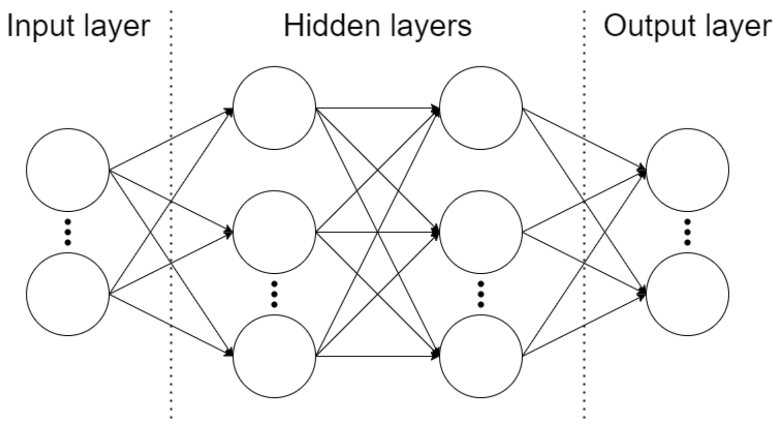


Figure 2.2: Example of a 4-layer dense neural network architecture with 2 hidden layers.

There are three types of layers. The first type is the *input layer*, where the training data used for learning are separated by their features, or physical quantities, by neurons in the first layer. The amount of neurons used in the input layer depends on the purpose of a neural network. In image recognition, for example, the network is trained one picture at a time, where each pixel is represented by a different neuron in the input layer. As a result, the input layer contains thousands of neurons. Since the emphasis is put on predicting physical quantities, *the amount of neurons in the input layer will always match the number of physical quantities in one training sample, i.e. the dimension of one training sample.* For example, when a neural network is built to predict the 3D time-dependent heat equation, four neurons would be used in the first layer. The amount of training samples passed onto each input layer varies and therefore one neuron can either contain a single real number or a vector. One could let the network train with all samples from the training dataset simultaneously, or one could split the training dataset into groups of training samples, so-called *batches*, which are independently used for training. The different methods of training the network in batches are discussed in Section 2.9.

By convention, *the same activation function is applied to each neuron within the same layer.* It is very common to choose the same activation function for each layer, and highly uncommon to use different functions within the same layer. Since the input layer is only used to structure the training samples, no activation function is applied and the input layer is often called the zeroth layer ($l = 0$).

The second type of layer is a *hidden layer*, which always lies between the input and output layer. The hidden layers add extra depth to the complexity of the system and allow the network to predict more advanced features. The process of adding extra layers to make a network deeper to learn more complex problems, is called *deep learning*. The layers are 'hidden' because they are not directly coupled to the outside world, as is the case with the in- and output layer.

The third type is the *output layer*, which contains values predicted by the network. Like with the input layer, choosing the amount of neurons is dependent on the form of the output. For regression, the amount of output neurons should match the dimension of the target. In classification, when the model has to predict between k classes, k output neurons are used, each representing a class.

The choice of the amount of layers to use is a trade-off between computation time and the learning power. Although adding extra layers introduces extra parameters which need to be optimized, the minimum amount of network parameters per layer required to approximate a target function with equal accuracy, decays exponentially and less training data is needed overall (Aggarwal [2018]). Except for an increase in computation time, arbitrarily increasing the depth of the network may lead to other problems such as overfitting, in which more training data is needed for learning, see Section 2.8. Though increasing depth is not always a necessity, as there exist other types of neural networks with more complex architectures which yield very similar results, but go beyond the scope of this thesis. Ultimately, there is no universal answer as to which architecture is the best for a given problem, and this question is still being widely researched.

2.4 FORWARD PROPAGATION EQUATIONS

Once the architecture of a dense neural network and the activation functions used in each layer are known, the forward propagation equations fully describe the network's prediction. Suppose we have a sample x_i, with i features and thus i neurons in the input layer. Let w^l_{jk} be the weight of the edge from neuron k in layer $l-1$ to neuron j in layer l. Denote b^l_j as the bias of neuron j in layer l, with preactivation z^l_j and activation a^l_j. Suppose $f^{(l)}$ is the activation function applied in layer l which contains n_l neurons. Figure 2.3 provides an illustration of the notation used in a neural network.

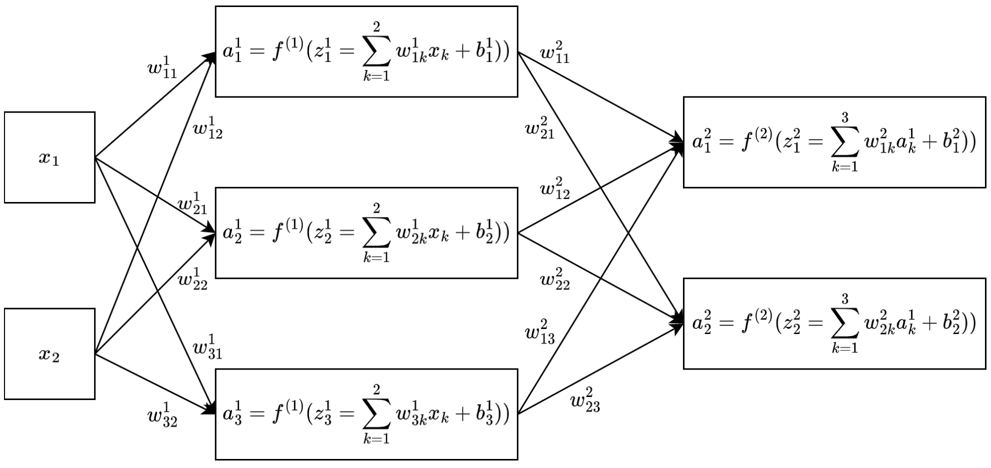

x_1
x_2
$a^1_1 = f^{(1)}(z^1_1 = \sum_{k=1}^{2} w^1_{1k}x_k + b^1_1))$
$a^1_2 = f^{(1)}(z^1_2 = \sum_{k=1}^{2} w^1_{2k}x_k + b^1_2))$
$a^1_3 = f^{(1)}(z^1_3 = \sum_{k=1}^{2} w^1_{3k}x_k + b^1_3))$
$a^2_1 = f^{(2)}(z^2_1 = \sum_{k=1}^{3} w^2_{1k}a^1_k + b^2_1))$
$a^2_2 = f^{(2)}(z^2_2 = \sum_{k=1}^{3} w^2_{2k}a^1_k + b^2_2))$
w^1_{11}
w^1_{12}
w^1_{21}
w^1_{22}
w^1_{31}
w^1_{32}
w^2_{11}
w^2_{21}
w^2_{12}
w^2_{22}
w^2_{13}
w^2_{23}

Figure 2.3: Illustration of the notation in a dense neural network.

Because the network is fully connected, the activation in layer l can be related to the activations in layer $l-1$

for $l = 1$:
$$a^1_j = f^{(1)}\left(\sum_{k=1}^{n_0} w^1_{jk}x_k + b^1_j\right), \tag{2.1}$$

and

for $l > 1$:
$$a^l_j = f^{(l)}\left(\sum_{k=1}^{n_{l-1}} w^l_{jk}a^{l-1}_k + b^l_j\right). \tag{2.2}$$

Note that in the last layer $l = L$, one immediately obtains the predictions as $\hat{y}_j = a^L_j$.

While Equations 2.1 and 2.2 fully describe the network's forward propagation, it is more useful to rewrite them into matrix form where all training samples are used. Suppose there are N samples such that the input data are represented by a n_0 x N matrix $\mathbf{X}$, where n_0 is the amount of input neurons. Each row in $\mathbf{X}$ represents samples of the same physical quantity. It is important to keep two important features into mind when rewriting the equations into matrix form. First, each element of a row of $\mathbf{X}$ is multiplied by *the same* weight. Second, the *same* bias is added to each sample in the row (∗). The preactivation matrix $\mathbf{Z}^l$ and activation matrix $\mathbf{A}^l$ are both n_l x N matrices, containing the respective preactivations and activations for each neuron in layer l. The weight matrix $\mathbf{W}^l$ and bias vector $\mathbf{b}^l$ only depend on the architecture of the network. They have dimensions n_l x n_{l-1} and n_l x 1 respectively. We use a notation in which applying $f^{(l)}(\cdot)$ to a matrix is the same as applying the function to each element of the matrix. The matrix forms of Equations 2.1 and 2.2 can then be formulated

for l = 1:
$$\mathbf{A}^1 = f^{(1)}(\mathbf{W}^1\mathbf{X} + \mathbf{b}^1\mathbf{J}_{1,N}), \tag{2.3}$$

and

for $l > 1$:
$$\mathbf{A}^l = f^{(l)}(\mathbf{W}^l\mathbf{A}^{l-1} + \mathbf{b}^l\mathbf{J}_{1,N}), \tag{2.4}$$

where $J_{1,N}$ is the 1 x N matrix of ones, needed to obey (∗), see above. Equations 2.3 and 2.4 fully describe the forward propagation process, omitting an abundance of indices. If only one training sample is considered at a time, then $\mathbf{X}, \mathbf{A}, \mathbf{Z}$ are column vectors $\mathbf{x}, \mathbf{a}, \mathbf{z}$ and Equations 2.3 and 2.4 are still valid. Furthermore, most common libraries have a very fast framework for matrix manipulations and vectorizations, which would make the implementation of the forward propagation very efficient.

2.5 ACTIVATION FUNCTIONS

The choice of which activation function to choose for a neural network design ultimately dictates the learning efficiency and most importantly, the prediction outcome. The choice also depends on the range of the targets. For unbounded regression problems, an unbounded activation function is needed in the last layer. For binary classification problems, the outcome of the activation function should always be in [0,1], where the outcome represents the probability of a binary class. Which activation function to choose to yield the best results is debatable and dependent on the neural architecture and on the metric space, and will often come down to trial and error. There are many types of functions which can serve as a universal function approximator (UFA). Hornik et al. [1989] have shown that any continuous nonconstant function over a bounded set as an activation function is an arbitrarily accurate approximation to any target function. Furthermore, he shows any non-decreasing squashing function can serve as a UFA, where a squashing function $f(x)$ has the property that $\lim_{x\to-\infty} f(x) = 0$ and $\lim_{x\to\infty} f(x) = 1$. However, the big assumption was that there are "sufficiently enough" hidden units, and Hornik et al. provide no guarantee for applications to work whenever there is "inadequate learning" and a "lack of deterministic relationship between input and target". Needless to say, the requirements of activation functions can be more relaxed. Hornik [1991] has shown that a bounded nonconstant function can serve as a UFA. Furthermore, Hornik showed that there exists unbounded functions which can be used as a UFA, such as the exponential function. For gradient based learning, an activation function needs to be differentiable almost everywhere. There are three activation functions which are most commonly used in machine learning. The sigmoid function is defined as $(1 + e^{-x})^{-1}$, and was very common for classification problems in the 1990s as it has a range of [0,1]. However, one big drawback of the sigmoid function is how it saturates for very high and low values of the domain, making it only sensitive to values around zero. This complicates gradient-based learning, where it becomes too difficult due to saturation to distinguish between different values. Another common activation function is the hyperbolic tangent function, which is closely related to the sigmoid function. Though, the hyperbolic tangent function generally performs better than the sigmoid function, and is very popular for regression (Bengio [2016]). One of the most popular activation function in classification is the Rectified Linear Unit (ReLU), defined as max(0,x). It is the simplest non-linear function, which allows for much faster training compared to the previous functions and yields better results in classification (Nair and Hinton [2010]), (Glorot et al. [2011]). Notice, however, that the derivative does not exist in 0, which is why the derivative is often set to 1 there. There are more difficulties than meets the eye that may arise during learning, which is related to the choice of the activation function. The differences between these three functions become more apparent in Section 2.8.

2.6 LOSS FUNCTION

The choice of which loss function to use is critical when designing a neural network. It evaluates the relative error between prediction $\hat{y}$ and label y, and maps deviations for all components of the output into one number. The loss function is directly used for training algorithms to update all model parameters and thus directly affects the efficiency of the learning phase. The loss must clearly represent the purpose of the model and choosing the 'wrong' loss function may even result into low correct prediction levels Reed and Marks [1999]. For regression problems, the most common loss function is the *mean squared error*. Many variations exist, such as the sum of squares or the L_2 norm. For the remainder of this thesis, unless stated otherwise, we use the loss function

$$\mathcal{L}_i(\Theta) = \frac{1}{2}||\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}(\Theta)||_2^2 \tag{2.5}$$

to train a dense neural network, where Θ is the model parameter space. The subscript i and superscript (i) emphasizes that the loss function is calculated *per sample*. For classification problems, using cross entropy loss function has been shown by Simard et al. [2003] to yield higher prediction success rates with faster training on average compared to the traditional mean squared error or the sum of squares. The cross entropy loss function for scalar y and $\hat{y}$ is defined as

$$\mathcal{L}(\Theta) = -\sum_{i=1}^{N} y^{(i)} \log \hat{y}^{(i)}(\Theta) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}(\Theta)).$$

As it turns out, the cross entropy loss function is derived from Bayesian statistics, in which the loss function is simply the maximum log likelihood of N i.i.d. samples from the Bernoulli distribution, which is equivalent to minimizing the negative log. When considering multiclass classification problems with K classes, the loss function is

$$\mathcal{L}(\Theta) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}(\Theta) + (1 - y_k^{(i)}) \log(1 - \hat{y}_k^{(i)}(\Theta)).$$

It becomes clear why the loss function from above is so popular in machine learning when considering both scenarios, when the true label (y) equals either 1 or 0. Assume the true label equals 1, but the network ends with a low $\hat{y}$. Then, the second term disappears and the first term blows up, because $\lim_{\hat{y}\to 0} \log(\hat{y}) = -\infty$. Vice versa, the same holds. This property of the loss function is why it is so attractive to use in machine learning: large mistakes made by the model are heavily punished.

2.7 LEARNING ALGORITHMS

To minimize the loss function, an optimization algorithm is needed. Optimizing deep models differs in essence but is mathematically equivalent to traditional optimization problems, where the goal is to directly minimize or maximize some function. In machine learning, optimization algorithms are used to minimize the loss function with respect to a training dataset, in the hopes of improving the overall performance of the network on unseen data. In other words, achieving a low training loss is no guarantee for a network to also obtain a small prediction loss. Due to the high dimensionality of the parameter space, it is almost impossible to find the global minimum. Fortunately, Choromanska et al. [2014] shows that most local minima are relatively close to the global minimum, and searching for this global minimum may even lead to overfitting (see Section 2.8).

The most popular optimization algorithms used in machine learning are extensions of the steepest gradient descent algorithm, which was first used by Cauchy et al. [1847]. The idea of gradient descent is simple in 1D and can be extended the same way in higher dimensions. Consider a differentiable 1D function $f(x)$, where the slope is known. It is easily verifiable analytically or illustratively that for any differentiable function, taking a step of size ϵ in the opposite direction of the slope for the next iteration step will guarantee a lower output for small enough ϵ and smooth f: $f(x - \epsilon \text{ sign}(f'(x)) \leq f(x)$. The multivariate case works similarly. Therefore all model parameters can be updated as:

$$\mathbf{W}^l_{n+1} = \mathbf{W}^l_n - \epsilon \nabla_{\mathbf{W}} \mathcal{L}(\Theta), \tag{2.6}$$

where $\nabla_{\mathbf{W}} \mathcal{L}(\Theta)$ is the Jacobian matrix with respect to all model parameters in layer l. Note that to calculate the Jacobian, Hessian or use a Taylor expansion, $\mathbf{W}$ is always first converted from a matrix to a vector. The stepsize ϵ is often called the *learning rate*, and is held fixed during steepest gradient descent. Although performing gradient descent as described in Equation 2.6 will always yield lower loss over time, the algorithm is still too simple. In fact, whenever the gradient is the zero matrix, one could also end up at a saddle point instead of a local minimum. Furthermore, a fixed learning rate over the whole learning process is very undesirable. Choosing ϵ too big could lead to divergence, while a very small ϵ leads to very slow convergence. Therefore it is very common in more advanced algorithms to make the learning rate adaptive, in which ϵ changes during each iteration step. For example, the learning rate could be set to decay exponentially over time, taking smaller steps when near a local minimum. Another problem is that the gradient descent method is very sensitive to the initialisation of the model parameters, which may lead to a convergence of an undesirable high local minimum.

In order to make the gradient descent method more accurate, one could also take into account the curvature of the loss function, or also known as the Hessian. The information from the Hessian allows us to calculate the optimal step size to use in Equation 2.6. To see how to update ϵ, we perform a second-order Taylor expansion around $\mathbf{W}^l_n$:

$$\begin{aligned}\mathcal{L}(\mathbf{W}^l_{n+1}) = \mathcal{L}(\mathbf{W}^l_n) + (\mathbf{W}^l_{n+1} - \mathbf{W}^l_n)^\top \nabla_{\mathbf{W}} \mathcal{L} + \frac{1}{2}(\mathbf{W}^l_{n+1} - \mathbf{W}^l_n)^\top H(\mathcal{L})(\mathbf{W}^l_{n+1} - \mathbf{W}^l_n) \\ + R_2(\mathbf{W}^l_{n+1} - \mathbf{W}^l_n),\end{aligned} \tag{2.7}$$

where R_2 is the remainder term. Substituting Equation 2.6 into 2.7 yields:

$$\mathcal{L}(\mathbf{W}^l_{n+1}) = \mathcal{L}(\mathbf{W}^l_n) - \epsilon(\nabla_{\mathbf{W}} \mathcal{L})^\top \nabla_{\mathbf{W}} \mathcal{L} + \frac{1}{2}\epsilon^2 (\nabla_{\mathbf{W}} \mathcal{L})^\top H(\mathcal{L}) \nabla_{\mathbf{W}} \mathcal{L} + R_2(\mathbf{W}^l_{n+1} - \mathbf{W}^l_n). \tag{2.8}$$

When neglecting the remainder term, $\mathcal{L}(\mathbf{W}^l_{n+1})$ is minimized by taking

$$\epsilon^* = (\nabla_{\mathbf{W}} \mathcal{L})^\top \nabla_{\mathbf{W}} \mathcal{L}((\nabla_{\mathbf{W}} \mathcal{L})^\top H(\mathcal{L}) \nabla_{\mathbf{W}} \mathcal{L})^{-1}. \tag{2.9}$$

Another method to make the algorithm more precise is to use a second-order approximation according to Newton's method, unlike in Equation 2.6. This can be done by minimizing $\mathcal{L}(\mathbf{W}^l_{n+1})$ in Equation 2.7, taking the gradient with respect to $\mathbf{W}^l_{n+1}$, setting it to zero and solving for $\mathbf{W}^l_{n+1}$, to obtain:

$$\mathbf{W}^l_{n+1} = \mathbf{W}^l_n - H^{-1}(\mathcal{L}(\mathbf{W}^l_n))(\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W}^l_n)). \tag{2.10}$$

Optimization algorithms using only the gradient like gradient descent, are called *first-order optimization algorithms*. Additionally, optimization algorithms that also incorporate the Hessian like Newton's method, are called *second-order optimization algorithms*. Such a second-order method like in Equation 2.10 requires an exact Hessian matrix, which may be computationally too expensive to calculate exactly, let alone its inverse. Hence in modern optimization algorithms, such as the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) that uses a Quasi-Newton approximation, an approximation of the Hessian matrix is calculated at each iteration step with a secant method (Dennis and Schnabel [1983]), and an update according to 2.10 is made for some learning rate. Because an approximation of the Hessian is used, an exact learning rate can no longer be obtained. In BFGS, the learning rate is found via a line search and with the Wolfe conditions, to ensure the gradient converges to zero (Wolfe [1969]). Another very popular optimization algorithm used in machine learning is called Adam (adaptive moment estimation), which is a first-order algorithm that computes an adaptive learning rate for each model parameter. Unlike BFGS, no Hessian matrix is used in Adam and the algorithm is explained in detail by Kingma and Ba [2014]. Both have proven to be successful, but there is no evidence to suggest one is preferred over the other as the performance strongly varies from problem to problem. In fact, Wolpert and Macready [1997] have shown that "for both static and time dependent optimization problems the average performance of any pair of algorithms across all possible problems is exactly identical", which is known as the "no free lunch" theorem.

2.8 COMMON ISSUES DURING TRAINING

During training of a dense neural network, unprecedented issues may arise, depending on the problem and on the configuration used. In this section, the most common issues are discussed that can arise during training using steepest gradient descent. To illustrate how effortlessly these issues are replicable on the easiest regression problems, all architectures have identically initialised model parameters and are set to test the same simple regression problem: predict values on a parabola.

2.8.1 Exploding gradients

Increasing the depth and thus the complexity of a dense network may lead to various practical problems. A consequence of adding depths is adding nonlinearity, resulting in regions with exploding gradients in the parameter space of the loss function, which are also known as cliffs. As a result, when updating the model parameters in an iteration step, the exploding gradient results in a very large iteration step, and drastically changes the model parameters. Hence the loss function explodes, making the model unstable and making previous progress futile. Exploding gradients during training may even occur when using only two hidden layers, as shown in Figure 2.4.

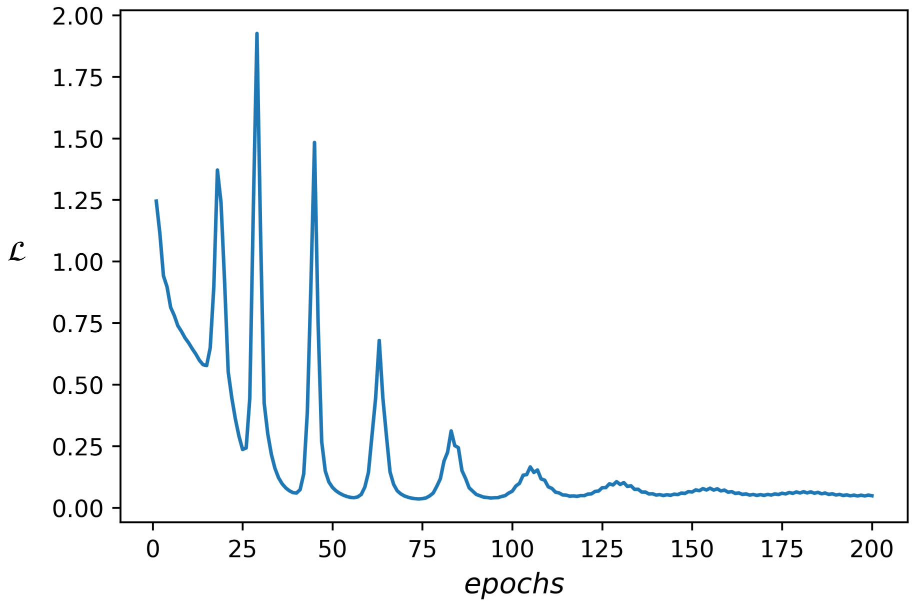


Figure 2.4: Illustration of exploding gradients during training for regressing a parabola with 4 neurons in the first hidden layer and 4 neurons in the second hidden layer. Learning rate is set to 0.001 and the hyperbolic tangent is used as activation function. $\mathcal{L}$ represents the mean squared error and 500 equidistant training samples were used.

Instead of removing excessive hidden layers, to prevent exploding gradients one could also use a simple technique called *gradient clipping*. Gradient descent only tells the direction corresponding to the steepest descent, without taking into account the curvature. To prevent making drastic changes, one could clip the norm of the gradient, which would prevent exploding update step sizes (Pascanu et al. [2013]).

2.8.2 Vanishing gradients

As opposed to the exploding gradient problem, another issue may be that gradients vanish during training. As a result, the parameter update is negligible and the model is set to reach saturation, where no learning takes place. To see how this is possible, notice that the model parameters are updated via backpropagation, where the gradient in layer l can be related to the gradient in layer $l+1$ via a backpropagation error. The cause of vanishing gradients is in fact the derivative of the activation function, see Section 2.10. The derivative of the sigmoid function attains a maximum of 0.25 and goes to 0 as $x \to \infty$ or $x \to -\infty$. Hence in the last layer, if the gradient update is small, the gradient update in previous layers drops exponentially. As a consequence, the first layers will receive very little update. Because the derivative of the hyperbolic tangent is higher for values near the origin compared to the sigmoid function and attains a maximum of 1, the hyperbolic tangent is less sensitive to vanishing gradients than the sigmoid function. This can be graphically illustrated by comparing the derivative of the sigmoid function (f_1') and hyperbolic tangent (f_2').

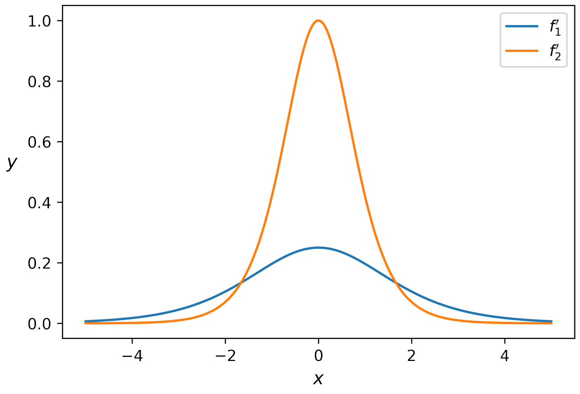


Figure 2.5: Plot of the derivative of the sigmoid function (f_1') and hyperbolic tangent (f_2').

An illustration of the vanishing gradient problem and comparison between the sigmoid function and hyperbolic tangent is shown in Figure 2.6.

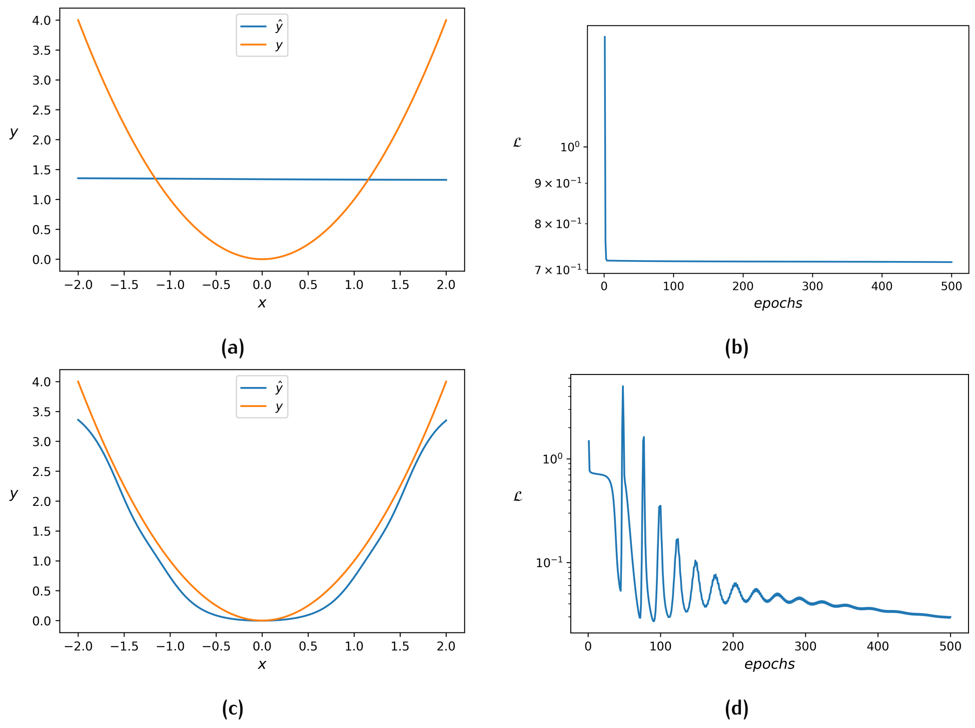


Figure 2.6: Illustration of the vanishing gradient problem during training for regressing a parabola with 3 neurons in the first hidden layer and 3 neurons in the second hidden layer. The learning rate is set to 0.001 and the sigmoid function is used as activation function (a) with training MSE (b) and the hyperbolic tangent in (c) with training MSE (d). 500 equidistant training samples were used.

An alternative option is to use the ReLU function, the derivative of which is the step function and is therefore more resistant against the vanishing gradient problem. Though the ReLU function is prone to a more serious problem, known as dead neurons, where an analogy can be drawn as having brain damage. If for some reason the preactivation is negative, the derivative of the ReLU is zero and no update will be performed. Hence the neuron is seen as dead, because it cannot learn. There exist other methods to resolve vanishing gradients, such as using an adaptive learning rate, different weight initialization schemes or batch normalization (Aggarwal [2018]), but these will not be elaborated further in this thesis.

2.8.3 Overfitting

Overfitting is a phenomenon where fitting a deep model on a training dataset does not guarantee good performance on unknown data, even if the loss function converges to zero during the learning phase. Therefore, there is a clear distinction between the performance on the training and prediction dataset. This problem always occurs when the number of training points is less than the amount of model parameters. In this case, it will always be possible to find infinitely many solutions for the model parameters that yield zero loss for the training data. No matter how complex or deep a model is, the model is said to generalize poorly on unseen data and contain a high prediction variance. An example is illustrated in Figure 2.7, where only 5 training points have been used for learning.

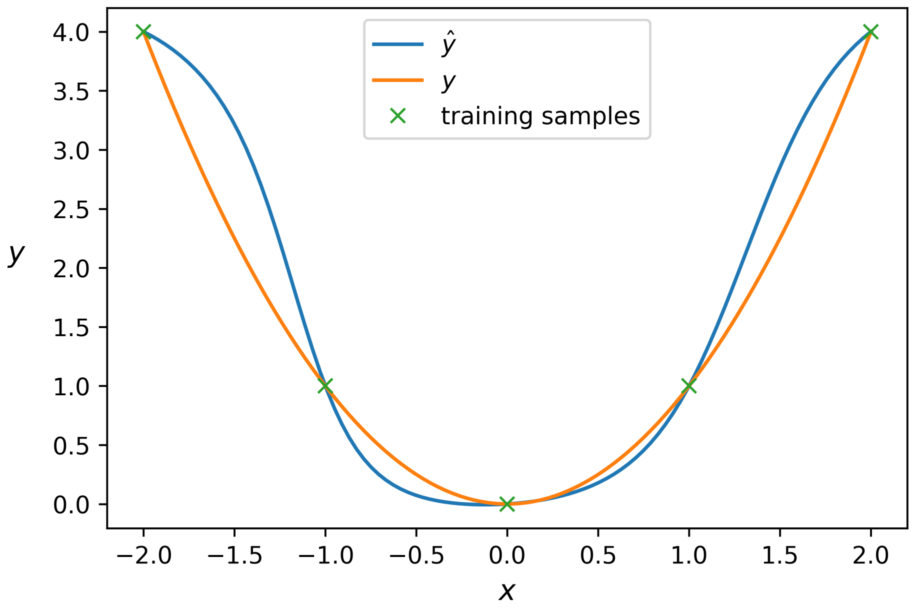


Figure 2.7: Illustration of overfitting a parabola with only 5 training samples. The neural network used a constant learning rate of 0.001, with 50000 epochs, had a [1,8,4,1] neuron structure and used the hyperbolic tangent activation function in the two hidden layers.

Even though after 50000 iterations a mean squared training error of $9.0 \cdot 10^{-28}$ was obtained, the prediction of the model $\hat{y}$ clearly does not resemble a parabola. Aggarwal [2018] recommends a general rule of thumb to use at least twice as many training samples as there are model parameters.

2.8.4 Stuck in high local minimum

In steepest gradient descent, the algorithm is very sensitive to the initialisation of the model parameters. The initial parameters will ultimately determine to which local minimum the algorithm will converge towards. In the worst case scenario, one could immediately get trapped inside a very high local minimum or saddle point, with no way out. An example is shown in Figure 2.8, where the only difference between the two dense networks is the weight initialisation.

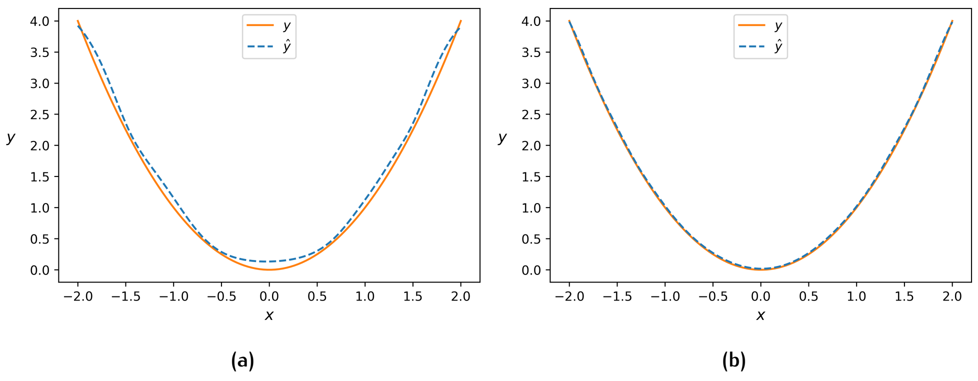


Figure 2.8: Illustration of two identical dense networks with different weight initialisations. Both networks use a [1,4,4,1] neuron structure, a learning rate of 0.001, the hyperbolic tangent activation function and 500 training samples. After 10000 epochs, network (a) obtained a $7.6 \cdot 10^{-3}$ prediction MSE and network (b) obtained a $3.1 \cdot 10^{-4}$ prediction MSE.

In fact, in a high multidimensional space like the parameter space of a dense network, Dauphin et al. [2014] shows via statistical physics and random matrix theory that saddle points are much more common than local minimum. The intuition behind the reasoning lies in the Hessian, where a local minimum only occurs when the Hessian is positive-definite, which becomes highly unlikely as the parameter space grows. To bypass this problem, Dauphin et al. proposed as an optimization algorithm a saddle-free Newton method that outperforms Quasi-Newton methods.

Furthermore, the initialisation of the model parameters has extensively been studied and Glorot and Bengio [2010] has proposed a very successful initialisation scheme, which would later become an industry standard and be known as Xavier initialisation. In particular, they proposed a method for drawing model parameters from a probabilistic distribution in such a way as to keep the model parameters' and activations' variance in each layer equal. While Glorot and Bengio have used a uniform distribution, He et al. [2015] proposed a normal distribution which theoretically yields the same goal as Glorot and Bengio. As a result, deep models were less prone to exploding or vanishing gradients, quicker convergence was reached and overall more accurate results were obtained compared to no or other initialisation schemes.

2.8.5 Depth of the network

The more complex a problem is, the more neurons are needed in a dense network to yield good results. One has the choice between making a network more wide, or adding more neurons in extra layers to make the network deeper. As discussed earlier, deeper networks require fewer neurons per layer as using more activation functions makes the model more nonlinear and thus more powerful. Though deeper networks come with certain limitations, such as very long computation times. In practice, very deep networks are more prone to the vanishing gradient problem. Also, even when using an equal amount of neurons, using more layers makes the parameter space more nonlinear, which increases the likelihood of exploding gradients. To show an example, we have trained two identical dense neural networks on the same training set with 5000 epochs. The first network used one hidden layer with 10 neurons, and the second network used a [3,3,2,2] hidden layer structure. We compare their performance by comparing their predictions with the true labels via the mean squared error. Even though the same amount of neurons were used, network 1 ended with a $1.5 \cdot 10^{-3}$ MSE, and the second network with a $1.2 \cdot 10^{-2}$ MSE.

2.9 BATCHES

The *batch size* is the number of training samples used in the training phase to perform one iteration on all model parameters. In particular, there are three options to train a model: stochastic gradient descent, minibatch or full batch. The choice boils down to a tradeoff between accuracy of the gradient versus computation time.

One popular method to train a network is with stochastic gradient descent (SGD). In that approach, only one sample at a time is used to calculate the gradients to update all model parameters. Consequently, the loss function uses only one sample, such as the one defined for regression in Equation 2.5. Then, another sample is randomly chosen to update all model parameters, until all samples have been used. In that case, the algorithm starts over until a satisfactory minimum is reached. In each iteration step, the "true" gradient, which needs the whole training dataset to compute, is approximated by the gradient of only one sample. This method is prone to a very large variance during learning. Hence the update process is most of the time "noisy". Because neural networks are very sensitive to the initialisation of the weights, while using a correct gradient descent step may lead to ending up in a local minimum far higher than the global minimum, the high volatility of SGD may be seen as a biased random walk that jumps between local minima and may prevent from getting trapped in a undesirable high local minimum. Since only one gradient is calculated at a time, it is computationally much faster to calculate the gradient and update the model parameters. In fact, if the sample size is too large, often it is even impossible due to memory limitations to calculate the true gradient. Furthermore, as shown by Saad [1999], under lenient assumptions, SGD will converge almost surely to a local minimum. However, whenever possible, it will always be computationally better to use the full batch, because modern CPUs can use vectorization libraries, which drastically reduces the computation time compared to iterating over each sample individually. On the other side of the spectrum lies full batch, in which all training samples are used to calculate the true gradient and perform one update to the model parameters. If N is the amount of samples in the training dataset, then the total loss is simply the sum of squares error:

$$\mathcal{L} = \sum_{i=1}^{N} \mathcal{L}_i = \frac{1}{2} \sum_{i=1}^{N} ||\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}||_2^2. \tag{2.11}$$

Unless stated otherwise, full batch descent will always be used, with the loss function defined in Equation 2.11. Whenever it is infeasible to calculate the gradient over the entire batch, a subset of the batch, or minibatch can be used for training as a middle of the road option. This option uses K samples for training, with $1 < K < N$. As this method balances out efficiency and accuracy, it is the most common method used in machine learning with overall better performance on a variety of applications, as pointed out by Masters and Luschi [2018]. However, at the end of the day, each option comes with its own up- and downsides, which very much depend on other configurations of the network such as the loss function, Woodworth et al. [2020].

2.10 BACKPROPAGATION EQUATIONS

In presenting the backpropagation equations used for deep learning, two proofs will be presented. In the first proof, the equations are proven in scalar form and then shown to be correct in matrix form with linear algebra. Because the essence of the proof is to use the chain rule of partial differentiation, it is easiest to calculate the gradients of the loss function in the last layer, and then work the gradients out backwards through the network. Due to the chain rule, the gradients in the early layers will always contain common terms already used in the calculation of the gradient in later layers. Storing the common terms as local variables and passing them backwards is computationally more efficient, and considerably simplifies the expression for the gradients in early notations. Hence, in literature, it is very common to define an intermediate variable

$$\delta_j^l \equiv \frac{\partial \mathcal{L}}{\partial z_j^l}, \tag{2.12}$$

or in matrix-form

$$\boldsymbol{\delta}^l \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^l}. \tag{2.13}$$

Because δ_j^l is computed at each neuron, and then passed on backwards in the backpropagation equations, the intermediate variable is often called the *error* of neuron j in layer l. In what follows, let $\odot$ denote the Hadamard product, which is the element-wise multiplication of matrices. Furthermore, we define the indicator function as

$$\mathbb{1}_{\{n=m\}} = \begin{cases} 1 & \text{if } n = m, \\ 0 & \text{if } n \neq m \end{cases}$$

for some $n, m \in \mathbb{N}$.

The four central backpropagation equations are presented on the next page, and will be proved chronologically.

The Four Backpropagation Equations

$$\boldsymbol{\delta}^L = \frac{\partial \mathcal{L}}{\partial \boldsymbol{A}^L} \odot f'^{(L)}\left(\mathbf{Z}^L\right) \tag{2.14}$$

$$\boldsymbol{\delta}^l = \left(\mathbf{W}^{l+1^\top} \boldsymbol{\delta}^{l+1}\right) \odot f'^{(l)}\left(\mathbf{Z}^l\right) \tag{2.15}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} = \boldsymbol{\delta}^l \mathbf{A}^{l-1^\top} \tag{2.16}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l \mathbf{J}_{N,1} \tag{2.17}$$

Proof
Fix one sample $\mathbf{x}_i$ ($N = 1$), with $i \in \{1, \ldots, n_0\}$. First, the equations are obtained for one training sample, and will then be rewritten to apply gradient descent for a batch of N samples. For simplicity, since only one training sample is evaluated at a time, matrices with dimensions dependent on N will be denoted with small letters as they will become column vectors: $\boldsymbol{a}$, $\boldsymbol{z}$, likewise the input and output. Furthermore, a subscript m is added to column vectors to underline the fact that the result follows for the mth training sample. For example, $\boldsymbol{\delta}^l_m$ denotes the column error vector in layer l of the mth training sample. Since the notation will become cumbersome, table 0.1 provides an overview of the definitions of all symbols. By definition, recall

$$\delta^L_j = \frac{\partial \mathcal{L}}{\partial z^L_j}.$$

However, because $\mathcal{L}$ depends on a^L_j, the activations of each neuron in the last layer, by the chain rule one obtains

$$\delta^L_j = \sum_{k=1}^{n_L} \frac{\partial \mathcal{L}}{\partial a^L_k} \frac{\partial a^L_k}{\partial z^L_j}.$$

Because

$$a^l_j = f^{(l)}\left(z^l_j\right),$$

all terms will vanish for $k \neq j$, which simplifies to

$$\delta^L_j = \frac{\partial \mathcal{L}}{\partial a^L_j} \frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial \mathcal{L}}{\partial a^L_j} f'^{(L)}\left(z^L_j\right).$$

Since the left and right hand side have matching components, one can immediately rewrite the expression above in vector form:

$$\boldsymbol{\delta}^L_m = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^L_m} \odot f'^{(L)}\left(\mathbf{z}^L_m\right). \tag{2.18}$$

In Equation 2.15, δ^l is related towards δ^{l+1}. This can be done with the help of the chain rule:

$$\delta^l_j = \frac{\partial \mathcal{L}}{\partial z^l_j} = \sum_{k=1}^{n_{l+1}} \frac{\partial \mathcal{L}}{\partial z^{l+1}_k} \frac{\partial z^{l+1}_k}{\partial z^l_j} = \sum_{k=1}^{n_{l+1}} \delta^{l+1}_k \frac{\partial z^{l+1}_k}{\partial z^l_j}.$$

To calculate the last term, recall from Equation 2.2

$$z^{l+1}_k = \sum_{j=1}^{n_l} w^{l+1}_{kj} a^l_j + b^{l+1}_k = \sum_{j=1}^{n_l} w^{l+1}_{kj} f^{(l)}\left(z^l_j\right) + b^{l+1}_k.$$

The last term is simply

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} f'^{(l)}\left(z_j^l\right).$$

Plugging this term back from the earlier expression, one obtains

$$\delta_j^l = \sum_{k=1}^{n_{l+1}} \delta_k^{l+1} w_{kj}^{l+1} f'^{(l)}\left(z_j^l\right).$$

To rewrite the expression from above in matrix form, observe

$$\begin{bmatrix} \delta_1^l \\ \delta_2^l \\ \vdots \\ \delta_{n_l}^l \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{n_{l+1}} w_{k,1}^{l+1} \delta_k^{l+1} f'^{(l)}\left(z_1^l\right) \\ \sum_{k=1}^{n_{l+1}} w_{k,2}^{l+1} \delta_k^{l+1} f'^{(l)}\left(z_2^l\right) \\ \vdots \\ \sum_{k=1}^{n_{l+1}} w_{k,n_l}^{l+1} \delta_k^{l+1} f'^{(l)}\left(z_{n_l}^l\right) \end{bmatrix} =$$

$$\begin{bmatrix} w_{1,1}^{l+1} & w_{1,2}^{l+1} & \dots & w_{1,n_l}^{l+1} \\ w_{2,1}^{l+1} & & & w_{2,n_l}^{l+1} \\ \vdots & & & \vdots \\ w_{n_{l+1},1}^{l+1} & w_{n_{l+1},2}^{l+1} & \dots & w_{n_{l+1},n_l}^{l+1} \end{bmatrix}^\top \begin{bmatrix} \delta_1^{l+1} \\ \delta_2^{l+1} \\ \vdots \\ \delta_{n_{l+1}}^{l+1} \end{bmatrix} \odot \begin{bmatrix} f'^{(l)}\left(z_1^l\right) \\ f'^{(l)}\left(z_2^l\right) \\ \vdots \\ f'^{(l)}\left(z_{n_l}^l\right) \end{bmatrix}$$

$$\iff$$

$$\boldsymbol{\delta}_m^l = (\mathbf{W}^{l+1\top} \boldsymbol{\delta}_m^{l+1}) \odot f'^{(l)}\left(\mathbf{z}_m^l\right). \tag{2.19}$$

To show Equation 2.16, again the chain rule is used:

$$\frac{\partial \mathcal{L}}{w_{kj}^l} = \sum_{m=1}^{n_l} \frac{\partial \mathcal{L}}{\partial z_m^l} \frac{\partial z_m^l}{\partial w_{kj}^l} = \sum_{m=1}^{n_l} \delta_m^l \frac{\partial z_m^l}{\partial w_{kj}^l}.$$

Because

$$z_k^l = \sum_{j=1}^{n_{l-1}} w_{kj}^l a_j^{l-1} + b_k^l$$

and

$$\frac{\partial z_m^l}{\partial w_{kj}^l} = a_j^{l-1} \mathbb{1}_{\{m=k\}},$$

one immediately obtains

$$\frac{\partial \mathcal{L}}{w_{kj}^l} = \sum_{m=1}^{n_l} \delta_m^l \frac{\partial z_m^l}{\partial w_{kj}^l} = \sum_{m=1}^{n_l} \delta_m^l a_j^{l-1} \mathbb{1}_{\{m=k\}} = \delta_k^l a_j^{l-1}.$$

In matrix form, this reads

$$\begin{aligned}
\frac{\partial \mathcal{L}_m}{\partial \mathbf{W}^l} &= \begin{bmatrix} \frac{\partial \mathcal{L}_m}{\partial w^l_{1,1}} & \frac{\partial \mathcal{L}_m}{\partial w^l_{1,2}} & \dots & \frac{\partial \mathcal{L}_m}{\partial w^l_{1,n_{l-1}}} \\ \frac{\partial \mathcal{L}_m}{\partial w^l_{2,1}} & & & \frac{\partial \mathcal{L}_m}{\partial w^l_{2,n_{l-1}}} \\ \vdots & & & \vdots \\ \frac{\partial \mathcal{L}_m}{\partial w^l_{n_l,1}} & \frac{\partial \mathcal{L}_m}{\partial w^l_{n_l,2}} & \dots & \frac{\partial \mathcal{L}_m}{\partial w^l_{n_l,n_{l-1}}} \end{bmatrix} \\
&= \begin{bmatrix} \delta^l_1 a^{l-1}_1 & \delta^l_1 a^{l-1}_2 & \dots & \delta^l_1 a^{l-1}_{n_{l-1}} \\ \delta^l_2 a^{l-1}_1 & & & \delta^l_2 a^{l-1}_{n_{l-1}} \\ \vdots & & & \vdots \\ \delta^l_{n_l} a^{l-1}_1 & \delta^l_{n_l} a^{l-1}_2 & \dots & \delta^l_{n_l} a^{l-1}_{n_{l-1}} \end{bmatrix} \\
&= \begin{bmatrix} \delta^l_1 \\ \delta^l_2 \\ \vdots \\ \delta^l_{n_l} \end{bmatrix} \begin{bmatrix} a^{l-1}_1 & a^{l-1}_2 & \dots & a^{l-1}_{n_{l-1}} \end{bmatrix} = \boldsymbol{\delta}^l_m \mathbf{a}^{l-1\top}_m . \qquad (2.20)
\end{aligned}$$

The last equation to show is the gradient of the loss with respect to the bias term, i.e. Equation 2.17. Fortunately, this is yet again the chain rule:

$$\frac{\partial \mathcal{L}}{\partial b^l_j} = \sum_{k=1}^{n_l} \frac{\partial \mathcal{L}}{\partial z^l_k} \frac{\partial z^l_k}{\partial b^l_j},$$

where

$$\frac{\partial z^l_k}{\partial b^l_j} = \mathbb{1}_{\{k=j\}}.$$

Plugging the term above back into the expression immediately gives the result:

$$\frac{\partial \mathcal{L}}{\partial b^l_j} = \sum_{k=1}^{n_l} \frac{\partial \mathcal{L}}{\partial z^l_k} \frac{\partial z^l_k}{\partial b^l_j} = \sum_{k=1}^{n_l} \frac{\partial \mathcal{L}}{\partial z^l_k} \mathbb{1}_{\{k=j\}} = \frac{\partial \mathcal{L}}{\partial z^l_j} = \delta^l_j.$$

Because all components already match, the multivariable representation immediately follows

$$\frac{\partial \mathcal{L}_m}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l_m. \qquad (2.21)$$

In fact, Equations 2.18, 2.19, 2.20 and 2.21 are the backpropagation equations of *Stochastic Gradient Descent*. While the derivation for the case $N = 1$ comes down to using the chain rule, for arbitrary N for minibatch training or full batch training, the equations can be generalized with elementary linear algebra. Observe that Equations 2.14 and 2.15 simply describe the method for calculating all error vectors for each sample, which comes down to applying Equations 2.18 and 2.19 componentwise. To see how, focus how each column in every matrix corresponds to a specific sample.

$$\begin{aligned}
\boldsymbol{\delta}^L &= \begin{bmatrix} \boldsymbol{\delta}^L_1 & \boldsymbol{\delta}^L_2 & \dots & \boldsymbol{\delta}^L_N \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \mathbf{a}^L_1} \odot f'^{(L)}\left(\mathbf{z}^L_1\right) & \frac{\partial \mathcal{L}}{\partial \mathbf{a}^L_2} \odot f'^{(L)}\left(\mathbf{z}^L_2\right) & \dots & \frac{\partial \mathcal{L}}{\partial \mathbf{a}^L_N} \odot f'^{(L)}\left(\mathbf{z}^L_N\right) \end{bmatrix} \\
&= \frac{\partial \mathcal{L}}{\partial \boldsymbol{A}^L} \odot f'^{(L)}\left(\mathbf{Z}^L\right)
\end{aligned}$$

$$\begin{aligned}
\boldsymbol{\delta}^l &= \left[\begin{array}{cccc} \boldsymbol{\delta}_1^l & \boldsymbol{\delta}_2^l & \dots & \boldsymbol{\delta}_N^l \end{array}\right] \\
&= \left[\begin{array}{ccc} \mathbf{W}^{l+1^\top}\boldsymbol{\delta}_1^{l+1} \odot f'^{(l)}\left(\mathbf{z}_1^l\right) & \dots & \mathbf{W}^{l+1^\top}\boldsymbol{\delta}_N^{l+1} \odot f'^{(l)}\left(\mathbf{z}_N^l\right) \end{array}\right] \\
&= \mathbf{W}^{l+1^\top}\left[\begin{array}{cccc} \boldsymbol{\delta}_1^{l+1} & \boldsymbol{\delta}_2^{l+1} & \dots & \boldsymbol{\delta}_N^{l+1} \end{array}\right] \odot f'^{(l)}\left(\mathbf{Z}^l\right) = \left(\mathbf{W}^{l+1^\top}\boldsymbol{\delta}^{l+1}\right) \odot f'^{(l)}\left(\mathbf{Z}^l\right)
\end{aligned}$$

To show the corrections of Equations 2.20 and 2.21, the assumption in 2.11 is used, where the total loss can be expressed as a sum of the loss of each sample. Furthermore, the linearity of the gradient operator is used.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^l} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}_i}{\partial \mathbf{W}^l} = \sum_{i=1}^{N} \boldsymbol{\delta}_i^l \mathbf{a}_i^{l-1^\top} = \left[\begin{array}{cccc} \boldsymbol{\delta}_1^L & \boldsymbol{\delta}_2^L & \dots & \boldsymbol{\delta}_N^L \end{array}\right] \left[\begin{array}{c} \mathbf{a}_1^{l-1} \\ \mathbf{a}_2^{l-1} \\ \vdots \\ \mathbf{a}_N^{l-1} \end{array}\right] = \boldsymbol{\delta}^l \mathbf{A}^{l-1^\top}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^l} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}_i}{\partial \mathbf{b}^l} = \sum_{i=1}^{N} \boldsymbol{\delta}_i^l = \left[\begin{array}{cccc} \boldsymbol{\delta}_1^l & \boldsymbol{\delta}_2^l & \dots & \boldsymbol{\delta}_N^l \end{array}\right] \left[\begin{array}{c} 1 \\ 1 \\ \vdots \\ 1 \end{array}\right]_{N \times 1} = \boldsymbol{\delta}^l \mathbf{J}_{N,1}$$

$\square$

3 PHYSICS-INFORMED NEURAL NETWORKS

In dense neural networks, the model parameters are updated solely based on training samples by minimizing a loss function such as the mean squared error. Without training samples, a dense neural network cannot be trained. These training samples are obtained from numerical simulations or experimental measurements, which are always limited to a bounded range. Data driven models cannot be expected to perform well outside the vicinity of experimental data. Intuitively, if the data can be generated from a PDE, it can be useful to include the PDE during optimization. In an attempt to generalize dense neural networks, Raissi et al. [2019] proposed to add the residual of the PDE to the loss function together with the initial and boundary conditions. As a result, a model is heavily penalized for making physically infeasible predictions. Such a model is called a 'Physics-Informed Neural Network' (PINN). Let an arbitrary PDE be described as

$$\begin{aligned} \mathcal{N}(u(\mathbf{x},t)) &= 0, && \mathbf{x} \in \Omega, \quad t \in [0,T] \\ \mathcal{B}(\mathbf{x},t) &= f(t), && \text{on } \partial\Omega \\ \mathcal{I}(\mathbf{x},t_0) &= g(\mathbf{x}), && t_0 \in [0,T] \end{aligned} \tag{3.1}$$

where $\mathcal{N}$ is a (non)linear combination of (non)linear differential operators acting on u, which is a scalar function of $d + 1$ variables and $\Omega \subseteq \mathbb{R}^d$ is the spatial domain. Furthermore, $\mathcal{B}$ and $\mathcal{I}$ denote some arbitrary boundary conditions (BCs) and initial conditions (ICs), respectively. A PINN can be set up to solve an arbitrary PDE as described by Equation 3.1 as follows. If $(\mathbf{x}, t) \in \mathbb{R}^{d+1}$, then $d + 1$ neurons are chosen in the input layer and for scalar u we take one output neuron. Suppose there are N training samples available. Furthermore, we choose a set of points $(\mathbf{x}, t)$ for the model to evaluate, including values at the initial time, and some points on $\partial\Omega$. These predetermined points are called *collocation points*, and we emphasize the fact that the exact solution in the collocation points is unknown. The loss function can now be defined as:

$$\begin{aligned} \mathcal{L} = &\frac{1}{N}\sum_{i=1}^{N}(u_i - \hat{u}_i)^2 + \frac{1}{N_C}\sum_{i=1}^{N_C}\mathcal{N}(\hat{u}(\mathbf{x}_i, t_i))^2 \\ &+ \frac{1}{N_B}\sum_{i=1}^{N_B}(\hat{\mathcal{B}}(\mathbf{x}_i, t_i) - f(t_i))^2 + \frac{1}{N_I}\sum_{i=1}^{N_I}(\hat{\mathcal{I}}(\mathbf{x}_i, t_i) - g(\mathbf{x}_i))^2, \end{aligned} \tag{3.2}$$

where N_B is the amount of the collocation points on the boundary, N_I the amount of initial points and N_C the remaining points on the interior. The first term is the mean squared error, as is normally used in a dense neural network. The second term contains the residual of the PDE and the final two terms are the loss of the prediction at the boundary ($\hat{\mathcal{B}}$) and initial conditions ($\hat{\mathcal{I}}$), respectively. We will not elaborate further upon if the loss in Equation 3.2 is guaranteed to converge towards zero, but rather perform empirical tests. For an optimization algorithm to update the model parameters, at least the gradient of the loss function in Equation 3.2 is required. The gradient of the first term can be analytically determined via the backpropagation equations, see Section 2.10. For the remaining terms, the differential operator needs to be applied on each predicted $\hat{u}$ and then the gradient with respect to the model parameters needs to be taken. This becomes infeasible to derive by hand and is very cumbersome to do for every different PDE. There are two different methods in which the nonlinear differential operators from PDEs can be evaluated, and we therefore distinguish between two different kinds of PINNs. In particular, a-PINNs rely on the fact that the output of a neural network is a function of its input, which can analytically be calculated via automatic differentiation (AD). Baydin et al. [2018] provides an excellent overview on how automatic differentiation works, and we will not comment on the workings thereof but rather directly use AD from PyTorch. Rather than analytically calculating the differential operators from the PDE of the outputs, one can also approximate these terms with numerical differentiation using, for example, central differences. PINNs using numeric differentiation for differential operators from the PDE and analytically calculated gradients with respect to model parameters are called n-PINNs. Although these two types of PINNs are very similar to each other, they may provide very different outcomes, as has been shown by Chiu et al. [2022]. Both PINNs will be implemented and their performance will be compared when tested upon a harmonic oscillator, the 1D heat equation and the KdV equation. In particular, the solution of the PDEs will be approximated using models on the opposite spectrum: data-driven dense neural networks relying only on limited measurements and PDE-driven neural networks, like a-PINNs and n-PINNs, which only contain training samples on the boundary or on some given initial condition. To show how powerful PINNs are in solving PDEs, as little training samples as possible will be used on the interior and hence the first term in Equation 3.2 will not be used for training. To draw fair comparisons between the two different PINNs, both models will be identically initialised.

It is possible that after training, some terms in 3.2 are much larger than the rest, for example a network may be physically correct on the interior but not obey the boundary conditions. To force the model to still yield satisfactory results, it is very common to vary the weights of the different loss terms. If the model cannot optimize the boundary conditions, then the weight of the corresponding loss term is increased, to emphasize its importance and force the model to adjust its parameters differently to obey the boundary conditions. This technique has been studied by van der Meer et al. [2020], where optimal weights have been derived. In addition, this paper proves that the loss function as defined in Equation 3.2 is mathematically justified.

4 EXPERIMENTAL RESULTS

4.1 HARMONIC OSCILLATOR

A well known problem in classical physics is the damped harmonic oscillator. It is a simple mass-spring system, which can be fully described by the following ordinary differential equation (ODE):

$$\begin{aligned} m\frac{d^2u}{dt^2} + \mu\frac{du}{dt} + ku &= 0 \\ u(0) &= x_0 \\ \frac{du}{dt}(0) &= v_0, \end{aligned} \tag{4.1}$$

where m is the oscillator's mass, μ is the coefficient of friction and k is the spring constant. Based on the coefficients m, μ and k, the oscillator can be in three regimes: overdamped, underdamped and critically damped. We will only consider the underdamped case, which only holds if $\delta < \omega_0$, with the parameters defined in Equation 4.2. The analytical solution is given by

$$\begin{aligned} u(t) &= A\cos(\omega t + \phi)e^{-\delta t}, \\ \text{where:} \qquad w &= \sqrt{w_0^2 - \delta^2} \\ w_0 &= \sqrt{\frac{k}{m}} \\ \text{and} \qquad \delta &= \frac{\mu}{2m}. \end{aligned} \tag{4.2}$$

We take $m = 2$, $\mu = 0.5$ and $k = 2$ for all simulations. The amplitude A and phase angle ϕ are determined by the initial conditions. For simplicity, we take $x_0 = 1$ and $v_0 = 0$ such that the motion is given by

$$u(t) = \cos(\omega t)e^{-\delta t}.$$

The performance of the dense neural network, a-PINN and n-PINN will be compared with the same given initial conditions. In particular, their ability to interpolate and extrapolate will be assessed using different tests.

4.1.1 Performance of Dense Network

To test a dense network's ability to interpolate and extrapolate, we assume there are 500 equidistant training samples in the domain [0,15] and we let the model predict 3000 equidistant points lying on [0,30]. For training, the network uses the loss as defined in Equation 2.11, so no physics information is provided. We evaluate the training and prediction error with the mean squared error (MSE) and use 2 hidden layers with 8 neurons in each layer. The tangent hyperbolic function is used as activation function using BFGS with 1000 epochs, with results shown in Figure 4.1.

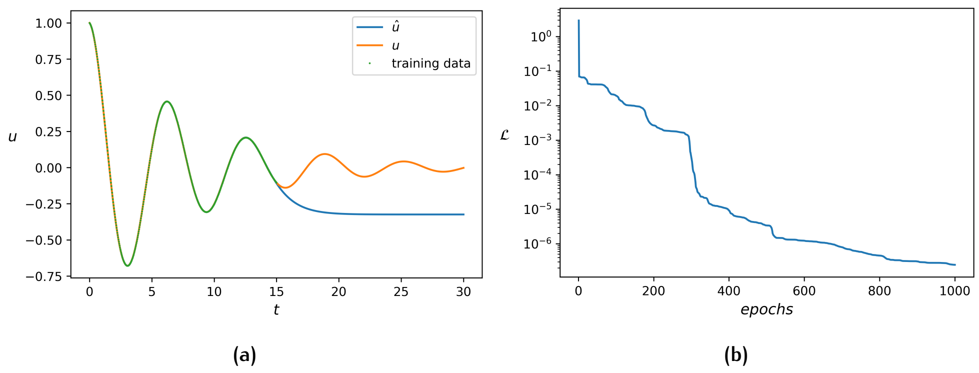


Figure 4.1: Illustration of (a) dense network on a damped harmonic oscillator with MSE loss (b). A [1,8,8,1] neuron structure was used with the tangent hyperbolic function as activation function using BFGS for 1000 iterations.

The dense network managed to achieve a mean squared error of 2.5e-7 on the training samples and a mean squared error of 2.4e-2 on the to be predicted values. As seen in Figure 4.1a, the model is perfectly capable of interpolating the solution on the training domain but, as expected, fails to extrapolate the solution correctly. As the stagnation of the log loss indicates, no matter how much the model is trained, it cannot improve its extrapolation performance.

When designing a neural network, there are many factors which play a role in its performance, such as its depth, width, or amount of training samples to name a few. Some modifications will be explored and their effect on the performance of the network will be analysed. Although at the end of the day it comes down to trial and error when designing a dense network, we can identify some key factors which substantially improve the performance.

One of the most crucial aspects of training is the optimization algorithm used to update the model parameters. Although a few popular ones were mentioned in Section 2.7, there are many which can be used, ranging from (quasi-)Newton methods to conjugate gradients and methods which do not even need a Jacobian such as Powell's method (Powell [1964]). A few optimization algorithms with identical weight initialisation were used and the results are shown in Figure 4.2.

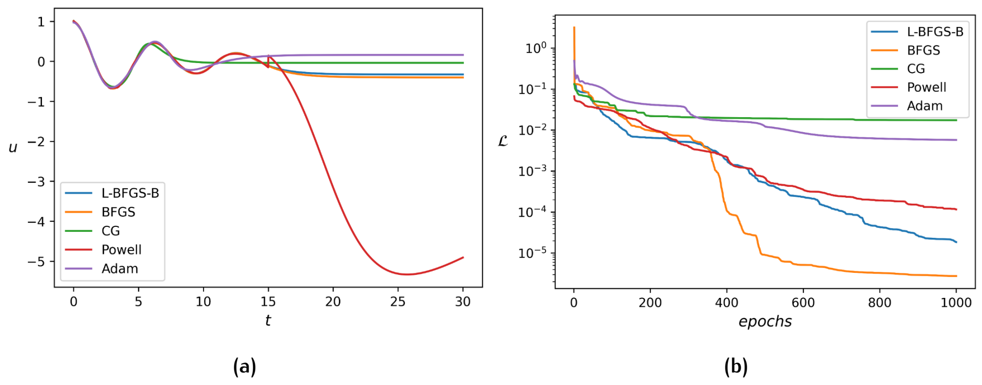


Figure 4.2: Illustration of different optimization algorithms' prediction (a) on the same oscillator problem with loss (b) during training.

Out of all optimization algorithms tested, either BFGS or L-BFGS-B seemed to achieve the fastest convergence. Surprisingly, even Powell which does not use the Jacobian beats many algorithms, though its much higher computation time ($\geq$ 1000x!) makes it worthless to use, provided the gradients are available. Furthermore, some cases with very poor extrapolation can be identified in the remaining tables and in Figure 4.2a, where one may even encounter extreme outliers such as Powell's prediction. For the remainder of experiments with a dense network, only BFGS will be used with 1000 epochs.

Another factor which plays a major role in obtaining accurate results is the architecture choice. In particular, one may expect for an increasingly wider network to yield better results. However, this is not always true as shown in Table 4.1.

Neurons / hidden layer	MSE (train)	MSE (prediction)
2	9.2e-3	5.6e-3
4	1.4e-6	2.2e-1
6	7.8e-7	6.9e-2
8	2.5e-7	2.4e-2
10	1.1e-6	8.0e-2
12	1.6e-7	5.2e-2
24	2.6e-7	1.9e-2

Table 4.1: A dense network with 2 hidden layers with altering but equal neurons per layer. Same training and to be predicted data is used as in the oscillator problem. Network is trained for 1000 epochs, and the resulting mean squared error (MSE) is displayed for both separate training and prediction datasets.

Even though using more neurons per hidden layer results in more nonlinear and thus more powerful networks which are capable of learning more complex problems, they may not necessarily give more accurate results. For example, the network with 10 neurons per layer performed substantially worse compared to the model using 8 or 6 neurons on the training samples. Also, the model with the highest amount of neurons with the highest-dimensional search space was not able to achieve the lowest loss on the training samples during training. Furthermore, increasing the amount of neurons does not improve a dense network's extrapolating capabilities as the lowest prediction MSE was achieved by the network with the least amount of neurons.

Instead of improving a network's learning capabilities by increasing the network's width, one may also consider making the network deeper. Especially the depth allows a network to learn progressively more advanced problems, though making it more prone to vanishing gradients. The question remains whether increasing a network's depth on simple problems such as the oscillator can yield more accurate results.

Layers	MSE (train)	MSE (prediction)
1	1.3e-7	2.1e-3
2	1.1e-6	8.0e-2
3	1.2e-7	2.0e-2
4	8.6e-5	1.1e-1

Table 4.2: A dense network with different amount of hidden layers with 10 neurons per hidden layer. Network is trained for 1000 epochs with BFGS, and the resulting mean squared error (MSE) is displayed for the training dataset and prediction dataset.

It turns out, using deeper models on simple regression problems such as an oscillator barely improved the loss on the training samples and even could make it worse. The model with 4 hidden layers was heavily outperformed by the 1 hidden layer model.

Finally, the most straightforward answer for a model to predict more accurately is to let the model train on bigger training datasets. As more training samples are available, the loss is evaluated in a more dense domain, which decreases the likelihood of overfitting and makes the model more generalized. The results after changing the amount of training samples are shown in Table 4.3.

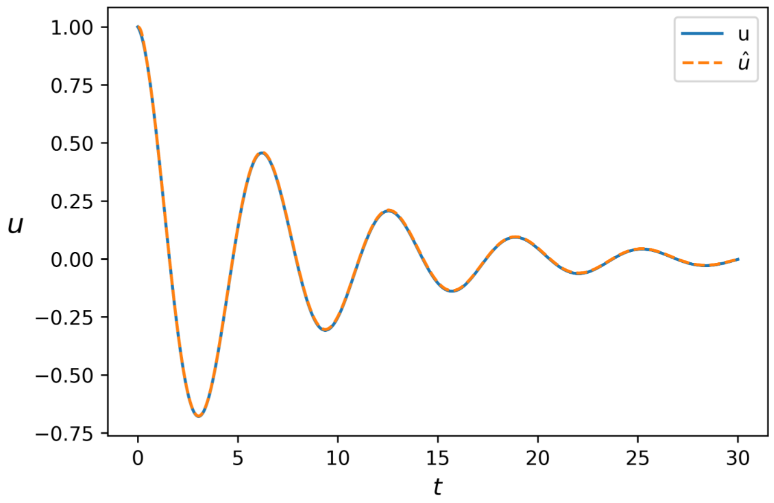
Training samples	MSE (train)	MSE (prediction)
100	1.7e-6	1.1e-1
200	7.4e-7	3.6e-2
500	1.1e-6	8.0e-2
1000	1.5e-6	2.4e-2
2000	5.6e-7	3.8e-2
5000	2.1e-8	1.2e-2
10000	1.6e-8	1.0e-2

Table 4.3: A dense [1,10,10,1] network after 1000 epochs with BFGS using different amount of training samples. The resulting mean squared error (MSE) is displayed for the training dataset and prediction dataset.

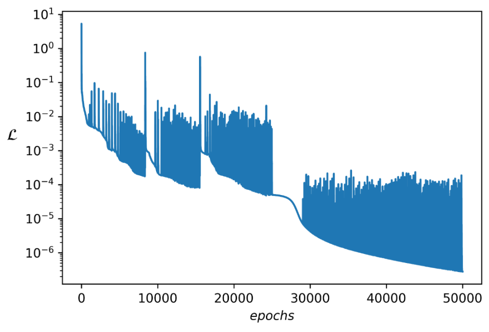
A clear improvement of the loss after training is visible with increasing training samples, with a surprising jump at 200 training samples.

4.1.2 Performance of PINN

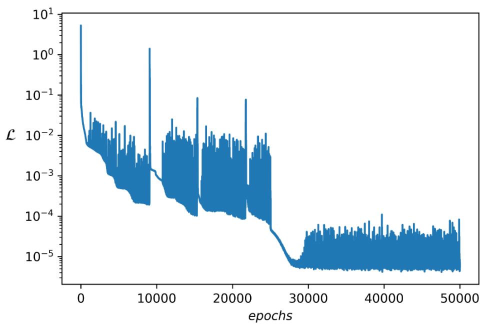
Instead of relying solely on measurements on a bounded domain, PINNs will simply evaluate the PDE at a set of collocation points and try to correctly enforce the underlying physics by minimizing the PDE loss. Suppose that only the initial condition is known of an oscillator, and 100 collocation points are used in the domain [0,30]. While a-PINNs rely on automatic differentiation, n-PINNs will approximate the partial derivative terms using central finite difference of second-order with step size ϵ = 1e-2. To greatly decrease the computation time during learning, Adam is used as optimization algorithm with an adaptive learning rate to reach faster convergence. The only type of adaptive learning rate that will be enforced is the so-called multistep learning rate, which decreases by a given factor after a threshold iteration step has been reached. Both PINNs use two hidden layers with 50 neurons in each layer with 50000 epochs. We first assume no training samples are available and thus exclude the first term in the PINN's loss as defined in Equation 3.2. All PINN experiments were performed with PyTorch using an NVIDIA Quadro P1000 graphics card.



(a)



(b)



(c)

Figure 4.3: Prediction ($\hat{u}$) of a-PINN and n-PINN on the oscillator (a) with 10 training samples on [0,15] with respective PDE loss during training (b) and (c).

After training, the mean squared error was evaluated between the prediction ($\hat{u}$) of the PINN and the analytic solution at 3000 equidistant points on [0,30]. The a-PINN and n-PINN reached a respective MSE of 1.0e-3 and 1.1e-3. Although both PINNs managed to capture the physics behaviour well, the dense neural network used for interpolation was able to yield more accurate results while containing less neurons per layer. A very easy method to drastically improve their performance is by combining PINNs with dense network's data driven approach and add a few training samples. When both PINNs were provided with as few as 10 equidistant training samples on [0,15], the MSE of the a-PINN and n-PINN reached respectively 8.6e-6 and 4.0e-6, with the prediction displayed in Figure 4.3a. Remarkably, the n-PINN managed to outperform the a-PINN, even though it approximates derivatives with numerical methods. Their differences become more apparent when tested upon a more complex problem such as the KdV equation.

One final observation to remark is the evolution of the loss during training. The PDE loss during training although decreases overall, the descent is by no means monotonous and is diverging in the same manner as noise signals. Although the a-PINN reached the lowest PDE residual loss (2.8e-7 as opposed to 5.7e-6), the n-PINN obtained the most accurate result, pinpointing the fact that a lower PDE residual loss may not result in a more accurate prediction. The exploding loss turns out to be such a major issue, that during simulations taking a step in the optimization algorithm may result in a dramatic increase of the loss function. Even after tens of thousands more iterations, the loss was not able to return to the previous loss value. To prevent this unreliable behaviour, all experiments from now on will have gradient clipping enforced (see Section 2.8.1) to prevent massively diverging losses.

4.2 1D HEAT EQUATION

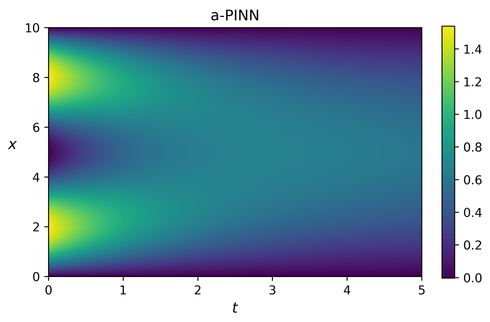
Consider a homogeneous problem where the heat conduction for a finite 1D rod with constant thermal coefficient and no heat source can be described by the following PDE, subject to the following BC and IC:

$$\begin{aligned} \frac{\partial u}{\partial t} &= k\frac{\partial^2 u}{\partial x^2} \\ u(0,t) &= u(L,t) = 0 \\ u(x,0) &= \sin(\pi x/L) + \sin(3\pi x/L). \end{aligned} \tag{4.3}$$

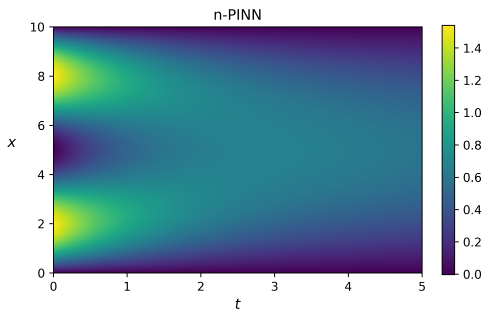
Using separation of variables, the solution to Equation 4.3 is

$$u(x,t) = \sin(\pi x/L)e^{-kt(\pi/L)^2} + \sin(3\pi x/L)e^{-kt(3\pi/L)^2}. \tag{4.4}$$

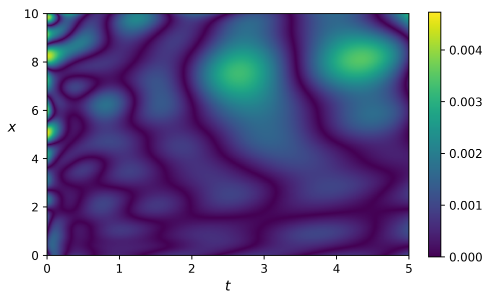
We let both PINNs train long enough to obtain accurate results, using the Adam optimizer with 10000 epochs. The domain is taken over $x \in [0,10]$ and $t \in [0,5]$, with $k = 1$. Equidistant collocation points are chosen on a 101 x 101 discretized grid and both PINNs use an adaptive learning rate with 4 hidden layers and 50 neurons in each layer. Then, both PINNs used a 501 x 501 discretized grid to predict the solution, with the results illustrated in Figure 4.4.



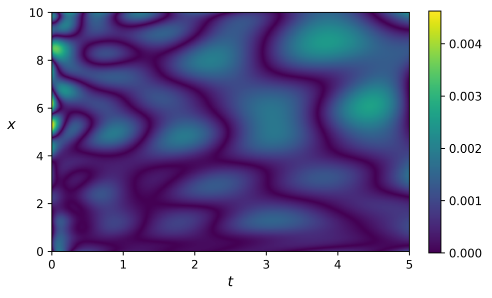
(a)



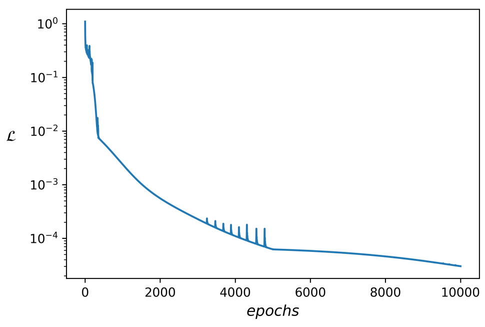
(b)



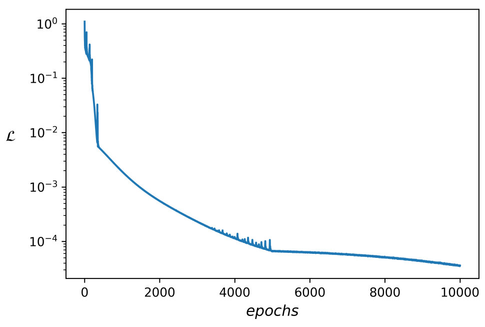
(c)



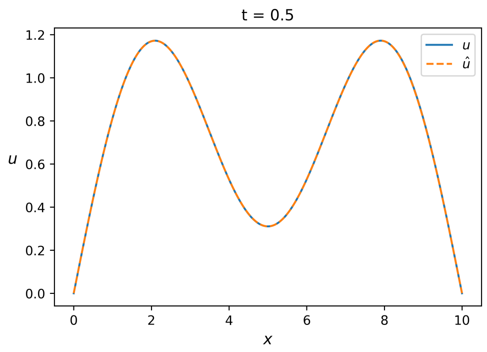
(d)



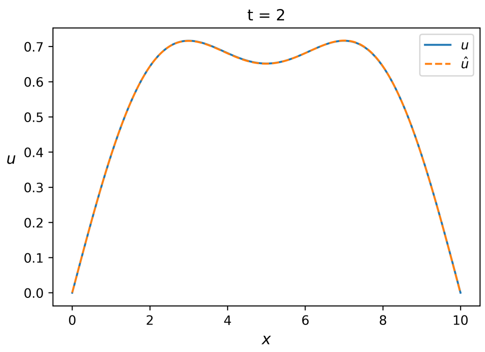
(e)



(f)



(g)



(h)

Figure 4.4: Illustration of prediction to the solution to the given heat equation by the a-PINN (a) and n-PINN (b). The respective absolute error and training loss is shown by the a-PINN (c,e) and n-PINN (d,f). Predicted outcomes at various time frames is illustrated (g,h) with no difference visible between the outcome of both PINNs.

After training both PINNs, the a-PINN finished with a 1.2e-6 MSE in 5 minutes and the n-PINN finished with a 9.6e-7 MSE in 4 minutes. In this specific example, the n-PINN had a slightly more accurate prediction than the a-PINN, while also finishing faster. However, both managed to converge to the correct solution with excellent accuracy. One method to compare both models' accuracy is to alter the amount of collocation points, which are used to evaluate the PDE during training. We initialised both models identically and compared their MSE after training for various amounts of collocation points, which is shown in Table 4.4.

	a-PINN	n-PINN
Grid size	MSE	
11 x 11	2.3e-1	2.8e-1
21 x 21	2.0e-1	2.2e-1
51 x 51	2.2e-6	2.3e-6
101 x 101	1.2e-6	9.6e-7
201 x 201	9.6e-7	9.1e-7

Table 4.4: Mean Squared Error (MSE) between the prediction and true values using the a-PINN and n-PINN for the heat equation. Amount of collocation points used during training is altered by changing the grid size.

The differences between both models are very small, and statistically insignificant to draw any major conclusion. Hence we repeat the simulations on a 51 x 51 grid with ten different weight initialisations. The a-PINN reached an average 1.1e-5 MSE and the n-PINN reached an average 3.5e-6 MSE. In this scenario, we conclude that the n-PINN reaches more accurate predictions than the a-PINN. Note, however, when too few collocation points are used, the PINNs are prone to very large errors, especially if the points are not dense near the highly nonlinear part of the solution. Although both PINNs could end up with entirely different results on different grid sizes of collocation points, we will rather focus on their accuracy on different applications.

4.3 KORTEWEG–DE VRIES EQUATION

The KdV equation is a model, which describes shallow water waves on a surface. It is given by

$$\alpha \frac{\partial u}{\partial t} + \beta u \frac{\partial u}{\partial x} + \gamma \frac{\partial^3 u}{\partial x^3} = 0, \tag{4.5}$$

where the standard form is obtained for α = 1 , β = 6, γ = 1. There are many types of solutions based on the initial condition. The solution we will be approximating is the so-called solitary wave solution, given by Anco and Willoughby [2022], in which the 1-soliton solution can be described as

$$u(x,t) = \frac{c}{2}\operatorname{sech}^2\left(\frac{\sqrt{c}}{2}(x - ct - x_0)\right) \tag{4.6}$$

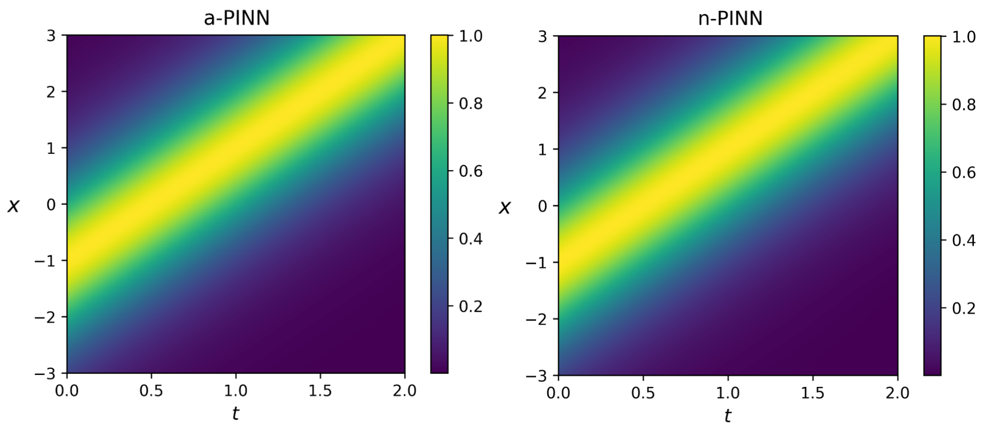
with wave velocity c and some displacement x_0. The 2-soliton solution can be described as

$$u(x,t) = \frac{2(c_1 - c_2)(c_1 \cosh^2(\sqrt{c_2}\xi_2/2) + c_2 \sinh^2(\sqrt{c_1}\xi_1/2))}{((\sqrt{c_1} - \sqrt{c_2})\cosh((\sqrt{c_1}\xi_1 + \sqrt{c_2}\xi_2)/2) + (\sqrt{c_1} + \sqrt{c_2})\cosh(\sqrt{c_1}\xi_1 - \sqrt{c_2}\xi_2)/2)^2} \tag{4.7}$$

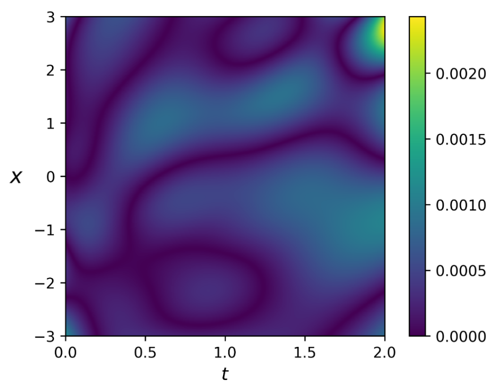
where $c_1 > c_2$ are the wave speeds and the transformation $\xi_1 = x - c_1 t - x_1$ and $\xi_2 = x - c_2 t - x_2$ have been made with initial displacements x_1 and x_2.

4.3.1 1-soliton solution

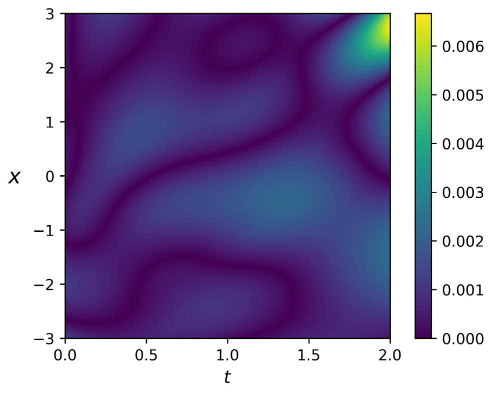
For the 1-soliton solution we chose c = 2, x_0 = -1 and domain $x \in$ [-3,3] and $t \in$ [0,2]. The equidistant collocation points were chosen on a 201 x 201 discretized grid, and the boundary and initial conditions were enforced by Equation 4.6. The Adam optimizer was used with a multistep learning rate for 10000 epochs where the PINNs contain two hidden layers with 50 neurons in each layer. No training samples were provided. After training, the PINNs were used to predict the solution on a 501 x 501 grid, with results illustrated in Figure 4.5.



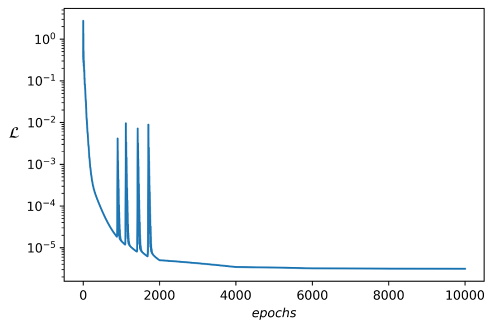
(a) (b)



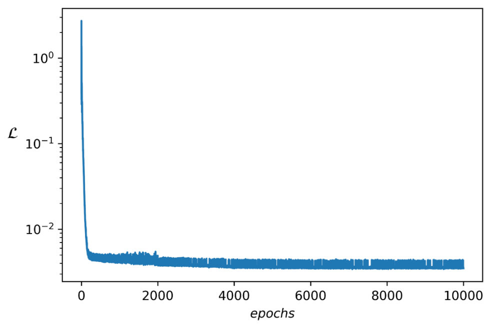
(c)



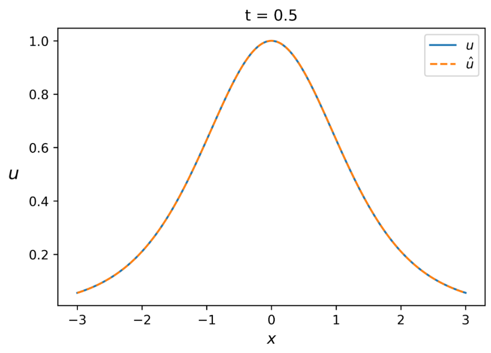
(d)



(e)



(f)



(g)

Figure 4.5: Illustration of prediction of the 1-soliton solution of the KdV equation by the a-PINN (a) and n-PINN (b). The respective absolute error is displayed in (c) and (d) with respective loss during training in (e) and (f). An illustration of the prediction of both the a-PINN and n-PINN of the soliton at $t = 0.5$ is shown in (g).

After training, the a-PINN reached a 2.0e-7 MSE and the n-PINN reached a 1.2e-6 MSE. Both PINNs were able to correctly predict the 1-soliton solution, with the a-PINN reaching more accurate results than the n-PINN, though needing more computation time to finish (16 min versus 10 min). Again, we compare their performance by altering the grid size, and comparing the resulting loss, which is shown in Table 4.5.

	a-PINN				n-PINN			
Grid size	$\mathcal{L}_{BC}$	$\mathcal{L}_{IC}$	$\mathcal{L}_I$	$\mathcal{L}$	$\mathcal{L}_{BC}$	$\mathcal{L}_{IC}$	$\mathcal{L}_I$	$\mathcal{L}$
5 x 5	2.3e-7	8.3e-10	1.9e-7	4.3e-7	9.4e-6	9.9e-7	2.7e-3	2.7e-3
11 x 11	4.7e-7	7.3e-8	2.3e-6	2.8e-6	2.3e-6	6.0e-6	4.8e-3	4.8e-3
21 x 21	3.8e-7	8.7e-8	2.6e-6	3.1e-6	1.5e-6	5.4e-7	2.8e-3	2.8e-3
51 x 51	3.7e-7	1.0e-7	2.7e-6	3.2e-6	2.5e-6	2.4e-7	4.1e-3	4.1e-3
101 x 101	3.5e-7	1.1e-7	2.9e-6	3.3e-6	1.9e-6	2.9e-7	4.0e-3	4.0e-3
201 x 201	3.3e-7	1.0e-7	2.7e-6	3.1e-6	2.1e-6	2.7e-7	3.5e-3	3.5e-3

Table 4.5: Final training loss of boundary condition ($\mathcal{L}_{BC}$), initial condition ($\mathcal{L}_{IC}$) and interior ($\mathcal{L}_I$) with $\mathcal{L} = \mathcal{L}_{BC} + \mathcal{L}_{IC} + \mathcal{L}_I$ on different grid size by the a-PINN and n-PINN for the KdV equation.

Noticeably, the n-PINN is outclassed by the a-PINN in all the different loss terms used during training, being primarily limited by the loss evaluated on the interior. In theory, this problem could be resolved by either using higher-order central differences or smaller step size ϵ in evaluating the finite differences. However, both methods failed to work during testing, where a smaller ϵ caused a very high numerical instability in the third-order partial derivative and exploded. Using a higher-order accuracy of central differences also failed to lower the loss in the interior, and only affected the n-PINN negatively as the model needed to evaluate more points to evaluate the same partial derivative term, which could lead to even higher computation times than the a-PINN. Though as earlier seen, the PINNs' accuracy is best judged by their mean squared error between their predictions and true values, which is displayed in Table 4.6.

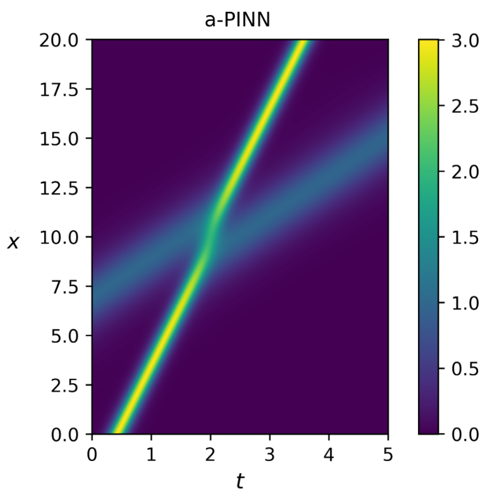
	a-PINN	n-PINN
Grid size	MSE	
5 x 5	3.4e-3	3.1e-3
11 x 11	7.6e-7	2.0e-6
21 x 21	2.9e-7	1.9e-6
51 x 51	2.1e-7	1.4e-6
101 x 101	2.1e-7	1.8e-6
201 x 201	2.0e-7	1.2e-6

Table 4.6: Mean Squared Error (MSE) for the prediction dataset for different grid size using the a-PINN and n-PINN for the KdV equation.

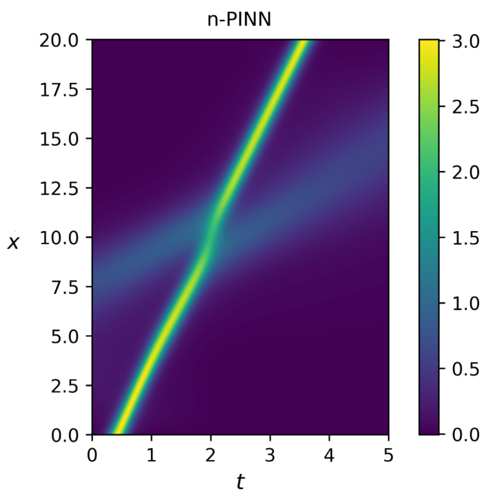
On all the different grid sizes, the a-PINN managed to largely outperform the n-PINN, with the only exception in the smallest grid. Notice in this case how the MSE is significantly smaller for smaller grid sizes compared to simulations with the heat equation. This goes to show that the amount and placement of the collocation points does significantly affect the prediction for different cases. Furthermore, both PINNs' performances seem to depend on the problem they face, as neither of the two has consistently given more accurate predictions over the other on all simulations performed until now. Although there are different ways to compare their performance such as varying the exact locations of the collocation points in the grid, or adding training samples, we will restrict ourselves here to the tests described.

4.3.2 2-soliton solution

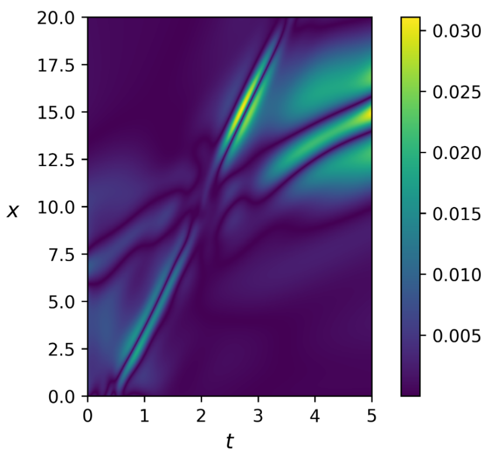
To challenge the two PINNs even more, we used them to predict the 2-soliton solution to the KdV equation. To make the problem as difficult as possible, the solitons have different wave speeds such that the solitons first merge and then split when colliding. For this problem, we took $c_1 = 6$, $c_2 = 2$, $x_1 = -2$ and $x_2 = 6$ with domain $x \in [0,20]$ and $t \in [0,5]$. The PINNs evaluated the loss in a 201 x 201 grid and used 8 hidden layers with 50 neurons in each layer. First, the PINNs were given an easier case and the IC is given at $t = 2$, exactly when the solitons merge. Afterwards, the IC is given at $t = 0$, to ensure the nonlinearity during the merge is completely unknown to the PINN. The results are shown in Figure 4.6 after 10000 epochs using the Adam optimizer with the initial condition given at $t = 2$.



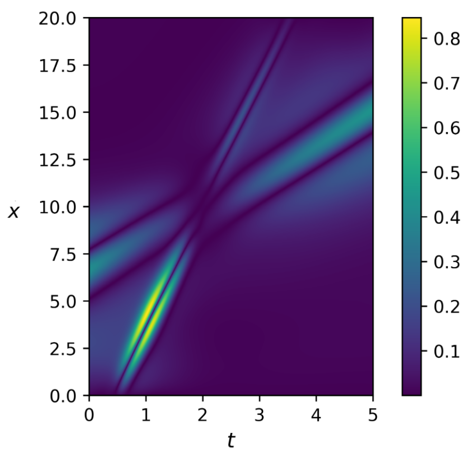
(a)



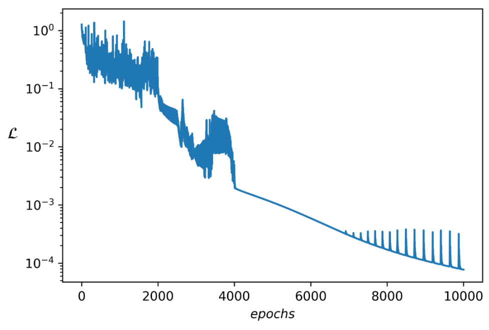
(b)



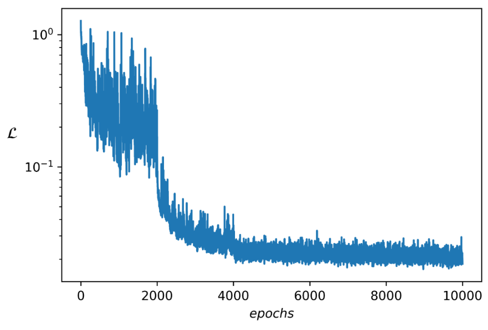
(c)



(d)



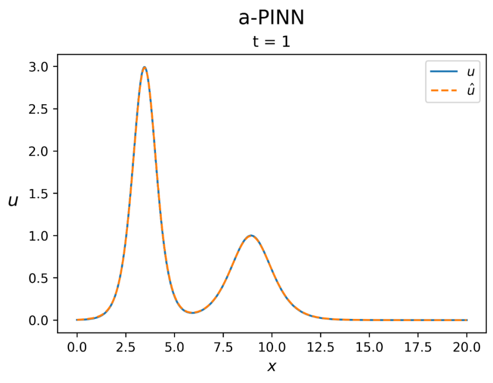
(e)



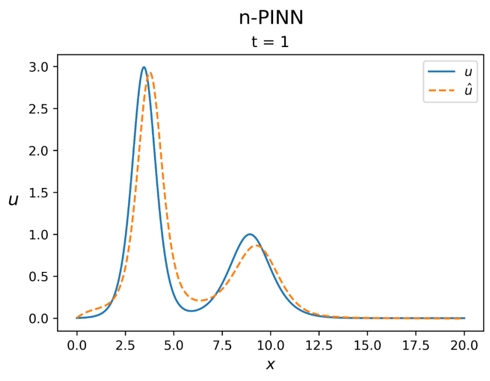
(f)

Figure 4.6: Illustration of prediction of the 2-soliton solution of the KdV equation by the a-PINN (a) and n-PINN (b) with given IC at $t = 2$. The respective absolute error is displayed in (c) and (d) with respective loss during training in (e) and (f).

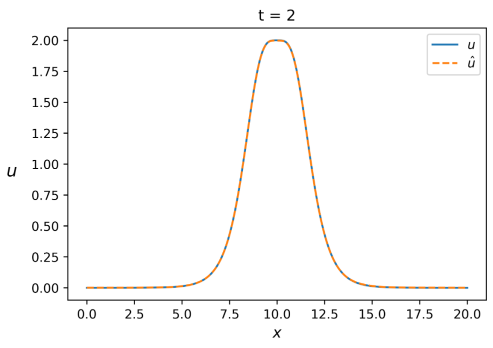
At the end, the a-PINN reached a 6.6e-4 MSE and the n-PINN reached a 1.2e-2 MSE, making the a-PINN a substantially better candidate. Although the n-PINN managed to some extent predict two solitons, it had trouble into finetuning the slower soliton. Increasing the amount of epochs, width or depth of the network did not resolve this issue and failed to reach the same order of accuracy as the a-PINN did. In fact, as seen in Figure 4.6e, after 10000 epochs the loss still does not stagnate as opposed to the n-PINN, and the a-PINN may even reach higher-order of accuracy when trained longer. Choosing a smaller step size ϵ in the numerical approximation of the derivative terms led to very high unstability when calculating the third-order derivative. Furthermore, using central differences of fourth-order accuracy made no visible improvement during training, other than increasing the computation time. To further illustrate the large gap in accuracy between the two PINNs, Figure 4.7 illustrates the predicted solitons, which are evaluated at $t = 1$, $t = 2$ and $t = 3$.



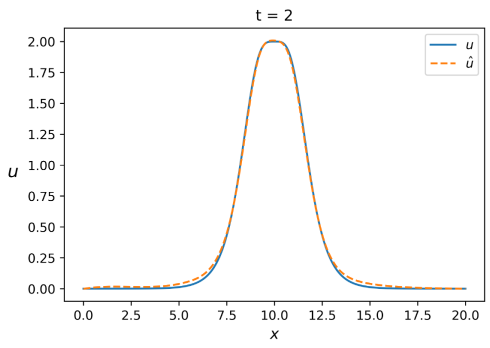
(a)



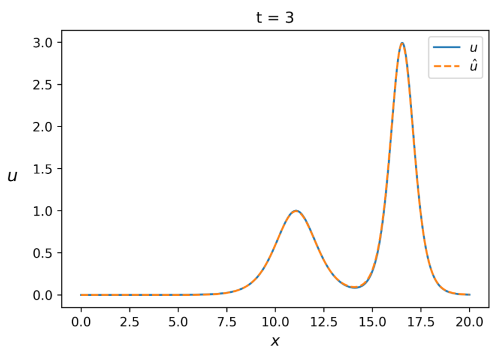
(b)



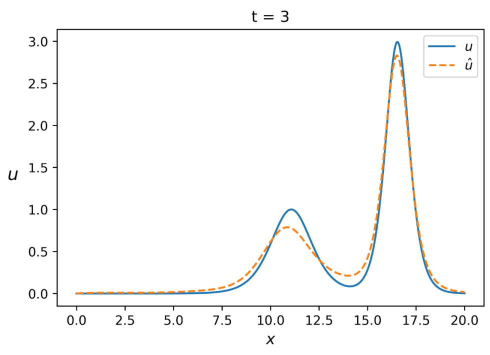
(c)



(d)



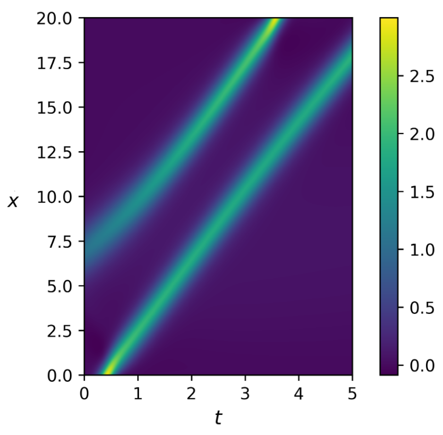
(e)



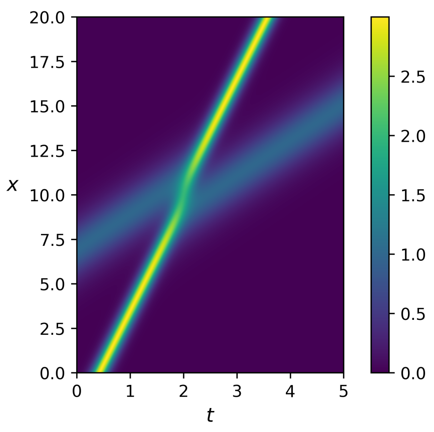
(f)

Figure 4.7: Illustration of the predicted solitons at various times by the a-PINN (a,c,e) and n-PINN (b,d,f).

Additionally, PINNs seem to be very sensitive to the given IC and BC if no training samples are present, which may result in erroneous convergence. If the initial condition were to be enforced at $t = 0$, both PINNs attempted to approximate a different solution to the KdV equation: two non-interfering solitons, as illustrated in Figure 4.8.



(a)



(b)

Figure 4.8: Result after training with an a-PINN with initial condition enforced at $t = 0$, instead at $t = 2$ is illustrated in (a) with real solution in (b).

Even using a large weight for the interior loss term made no difference. In fact, the outcome is a consequence of not adding any form of training samples for the model to train upon. Even if the PINN fulfills the initial and boundary conditions, a unique convergence is not guaranteed. To still see whether the a-PINN manages to converge to the correct unique solution with given IC at $t = 0$, a small modification is made in the initial displacement. We now choose x_1 = -2 instead of x_1 = 2, such that both solitons are visible at $t = 0$ to the PINN.

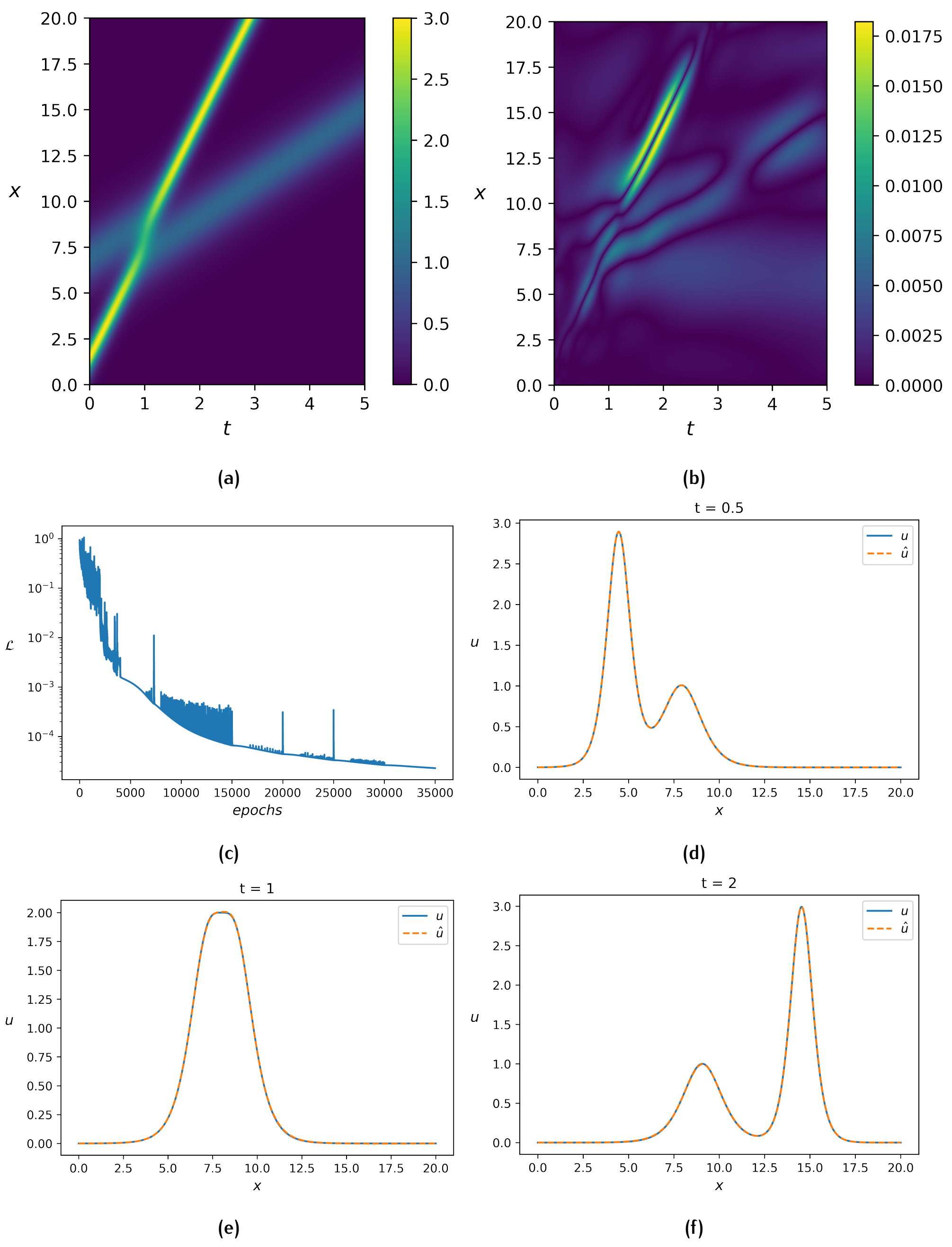


Figure 4.9: Illustration of the predicted solitons by the a-PINN (a) with absolute error (b). The total loss during training is shown in (c). The real solution and prediction at various times by the a-PINN are shown in (d,e,f).

To reach a high prediction accuracy, 35000 epochs were used, which resulted in a computation time of almost 6 hours on the used GPU with a final 7.5e-6 MSE. The evolution of the MSE is summarised in Table 4.7.

Epochs	MSE
10000	1.4e-3
15000	1.0e-4
20000	2.6e-5
25000	1.3e-5
30000	9.2e-6
35000	7.5e-6

Table 4.7: Evolution of the prediction MSE during training of an a-PINN on the 2-soliton solution to the KdV equation with given IC at t = 0.

From Figure 4.9 we make two important observations. Firstly, the a-PINN yielded very accurate results when trained sufficiently, and could be fine-tuned better when trained longer. Also, the a-PINN managed to capture the nonlinearity of two solitons merging very well. Secondly, we observe how drastically the a-PINN's prediction has changed when used on an equally sized but slightly shifted grid (see Figure 4.8b and Figure 4.9a). A huge improvement is made whenever both solitons are initially seen at $t = 0$ as opposed to split across two borders of the grid, even when using as many as just under 40000 collocation points.

4.3.3 Train of solitons

Up until now, PINNs have shown accurate results when trained sufficiently. One may ask themselves how PINNs fare against numerical methods. Hence we introduce a highly nonlinear problem to the KdV equation, given as

$$\begin{gathered}\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \delta^2 \frac{\partial^3 u}{\partial x^3} = 0 \\ u(x,0) = \cos(\pi x).\end{gathered} \tag{4.8}$$

The proposed problem was introduced and studied by Zabusky and Kruskal [1965], in which the solution is a train of solitons interacting with one another. To cause many solitons to interact in a small grid, we let δ^2 = 1e-3. Equation 4.8 can be solved numerically with a spectral method, see Appendix A.1 for details. The highly nonlinear solution was numerically computed in less than seven seconds, and is illustrated in Figure 4.10 with $x \in$ [-2,2] and $t \in$ [0,4].

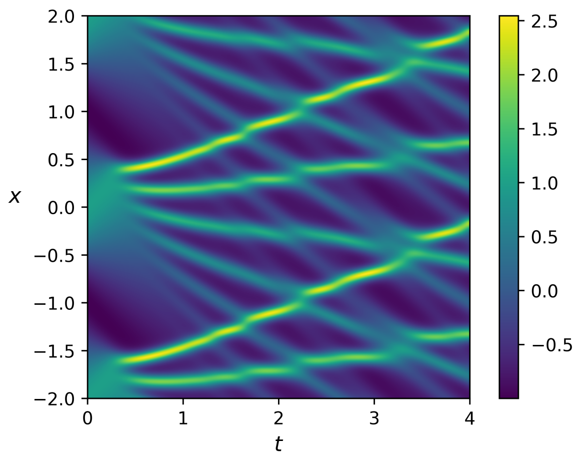
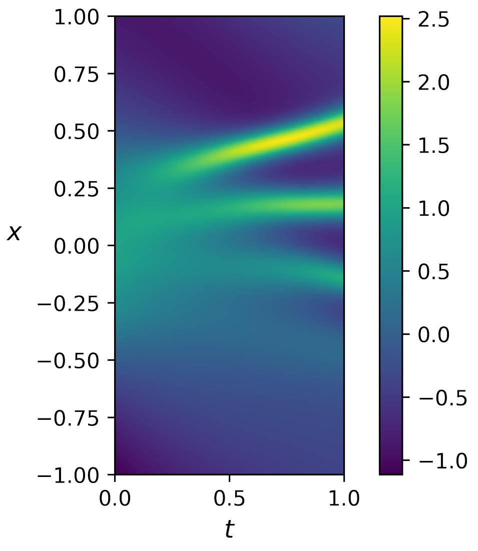
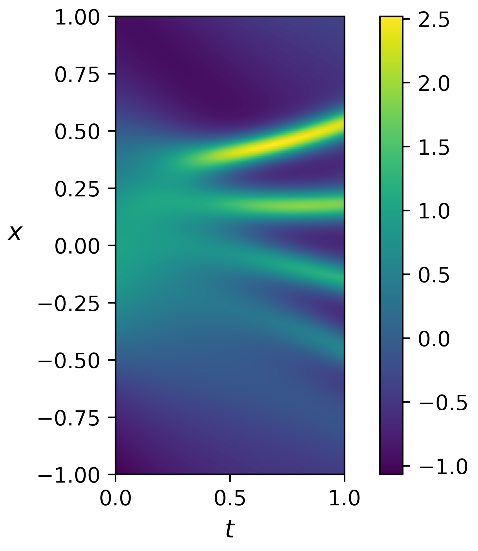


Figure 4.10: Illustration of the numerical solution of Equation 4.8.

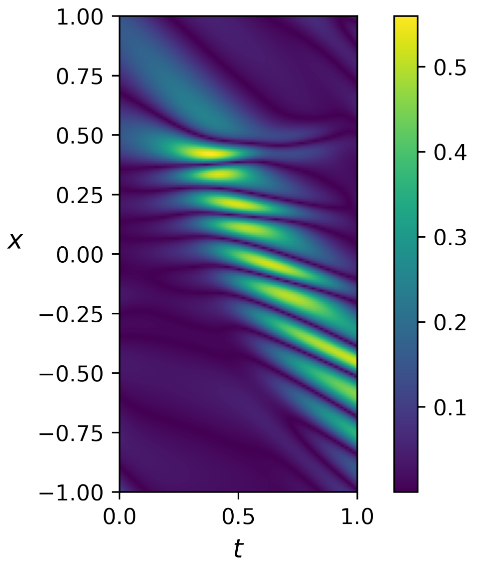
We start by first considering the domain $x \in$ [-1,1] and $t \in$ [0,1] to omit the train of solitons interfering with one other, and provide training samples at t = 0 and at the final time $t = 1$. Again, 40000 equidistant collocation points are chosen to evaluate the PDE, and we let the a-PINN train long enough with an adaptive learning rate. In the first simulation, we used no training samples on the interior, and in the second simulation we randomly generated 500 training samples on the interior using the spectral method. Nine hidden layers were used, each containing 50 neurons and the hyperbolic tangent is used as activation function. The results of the first simulation are shown on the left, and the results of the second simulation are shown on the right of Figure 4.11.



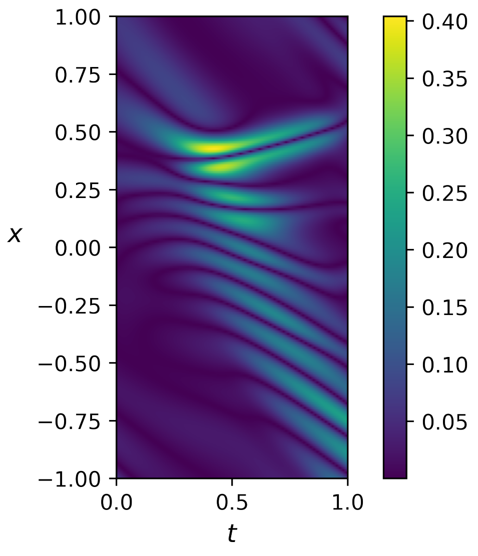
(a)



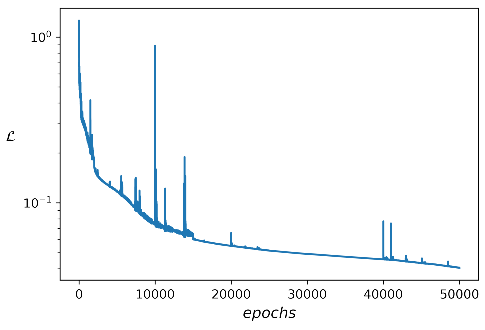
(b)



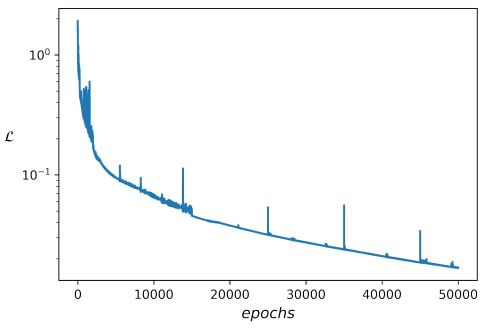
(c)



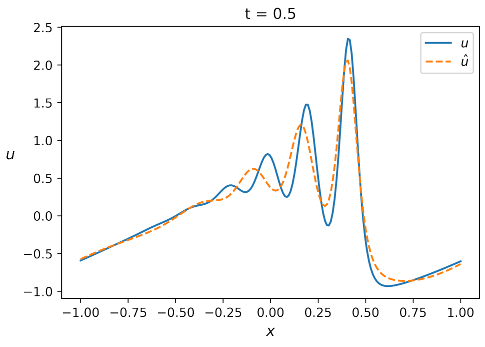
(d)



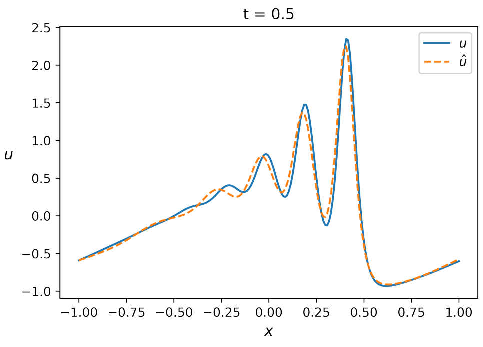
(e)



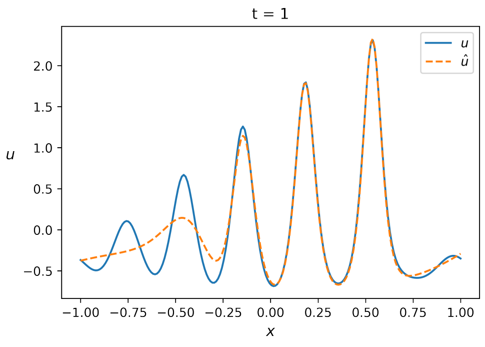
(f)



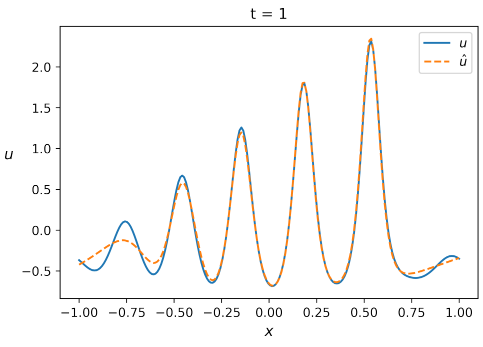
(g)



(h)



(i)

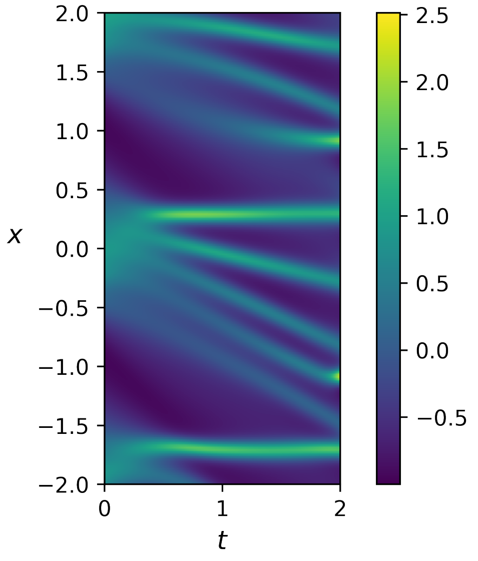


(j)

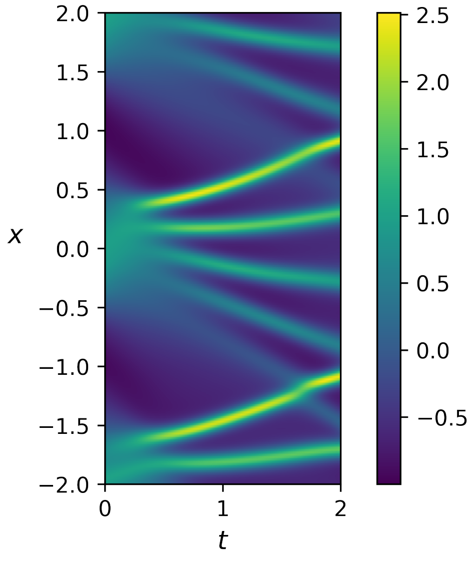
Figure 4.11: Illustration of the predicted train of solitons for $x \in$ [-1,1] and $t \in$ [0,1] by the a-PINN with no training samples on the interior (a), with absolute error (c), total training loss (e), and prediction at various times (g,i). The simulation with 500 training samples on the interior is shown in (b,d,f,h,j) respectively.

We compare the a-PINN's performance by comparing the prediction with values generated with the spectral method. After 50000 epochs, the a-PINN with no training samples on the interior reached a 2.3e-2 MSE, and the a-PINN with training samples on the interior reached a 6.9e-3 MSE. After approximately 9 hours of training, we observe from Figure 4.11e and 4.11f that further training would lead to very little progress, especially for the case where no training samples were used on the interior. Note that faster solitons are qualitatively well captured in both simulations. Adding training samples on the interior did help the performance, as the slower solitons are at least visible, which is not the case in the first simulation. Although it may be possible to reach more accurate results with an a-PINN with training samples on the interior, we will omit further training as it could take days to reach accurate results.

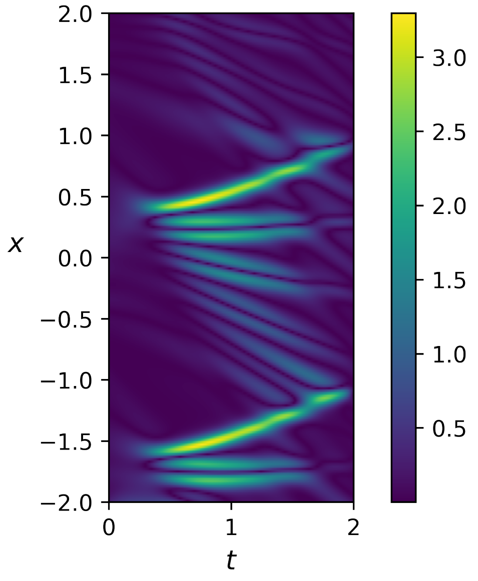
We repeated both simulations, but now considered the domain $x \in [-2,2]$ and $t \in [0,2]$. This leads to trains of solitons interfering with each other. We used an identical setup, but used an additional 500 training samples for the second simulation, totalling up to 1000 training samples on the interior.



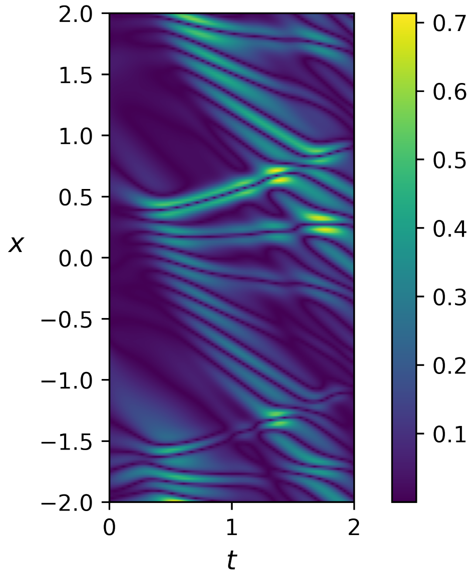
(a)



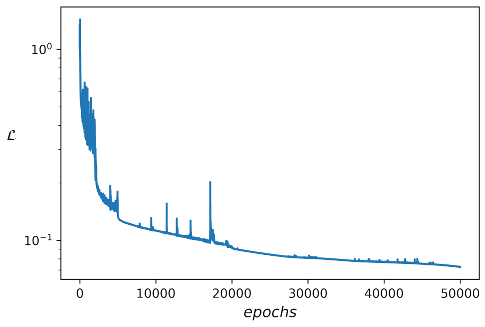
(b)



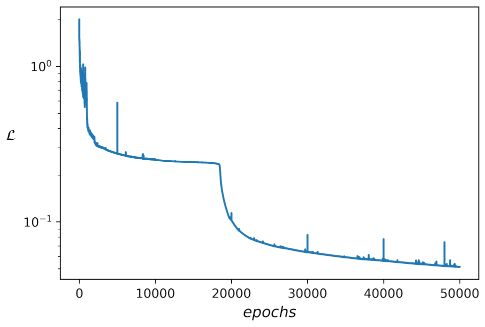
(c)



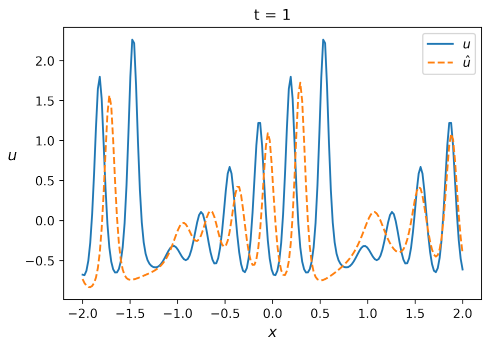
(d)



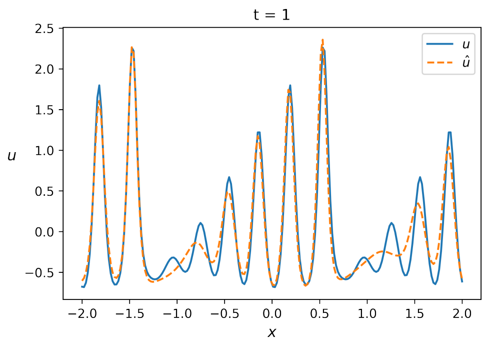
(e)



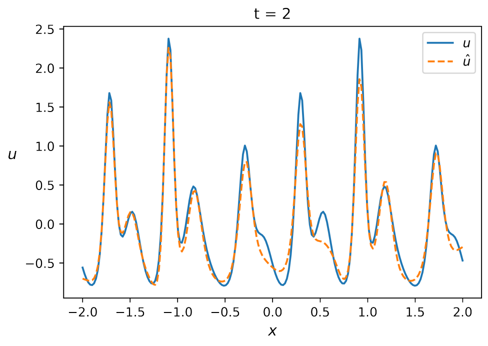
(f)



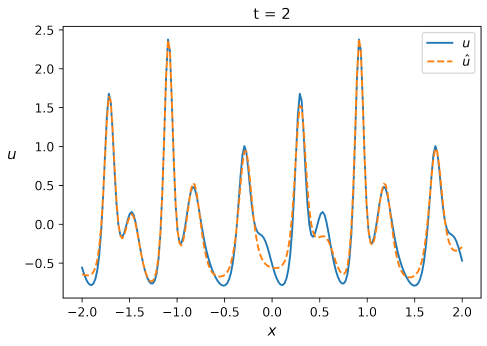
(g)



(h)



(i)

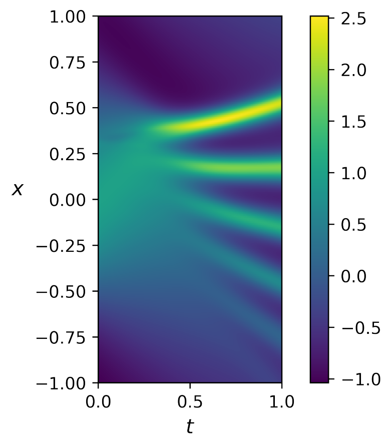


(j)

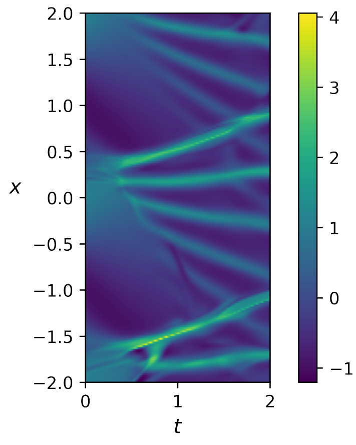
Figure 4.12: Illustration of the predicted interfering trains of solitons for $x \in$ [-2,2] and $t \in$ [0,2] by the a-PINN with no training samples on the interior (a), with absolute error (c), total training loss (e), and prediction at various times (g,i). The simulation with 1000 training samples on the interior is shown in (b,d,f,h,j) respectively.

From Figure 4.12a and 4.12d we observe that only providing training samples at the initial and final time yields very poor results. Furthermore, the training loss in Figure 4.12e stagnates, which again indicates it is very likely further training will not improve the performance. After 50000 epochs, the first simulation yielded a 4.6e-1 MSE, and the second simulation yielded a 2.7e-2 MSE. This example illustrates that although PINNs seem to be very promising in solving PDEs, substantially more effort is required to find a better network configuration to accurately predict complex solutions to highly nonlinear PDE problems.

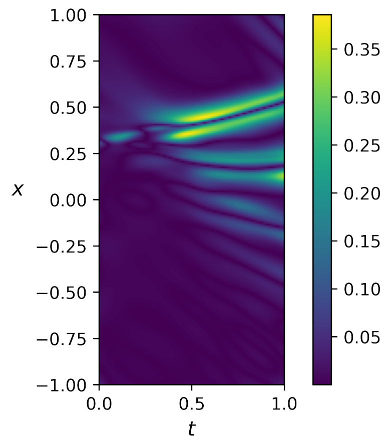
For example, one problem that basic PINNs may face is how the model at each iteration step tries to optimize on the whole space time domain, which may be too difficult. Multiple models have been developed to improve PINNs, such as a backward compatible PINN (bc-PINN) proposed by Mattey and Ghosh [2022], which solves a PDE sequentially over multiple time segments. The bc-PINN is designed to satisfy previous time segments by penalizing the model on the previous time segments over the already obtained prediction over the previously trained grid. Alternatively, Wang et al. [2021b] showed that PINNs with a spatio-temporal multi-scale Fourier feature architecture are superior to regular PINNs for solutions with a high-frequency behaviour. For future research, it may be worthwhile to investigate if more advanced versions of the regular PINN can solve trains of interfering solitons from the KdV equation accurately. In training an a-PINN to predict trains of solitons, training samples were used on the interior. In particular, on the domain $x \in$ [-1,1] and $t \in$ [0,1] we generated 500 training samples and on $x \in$ [-2,2] and $t \in$ [0,2] we generated 1000 training samples. In this scenario, instead of using PINNs, one could also use the same training samples to train a dense neural network. Hence we use both domains and an identical setup to compare a DNN with an a-PINN.



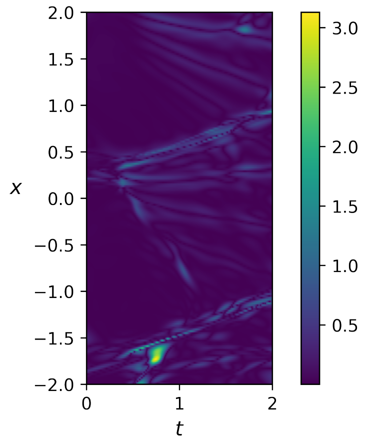
(a)



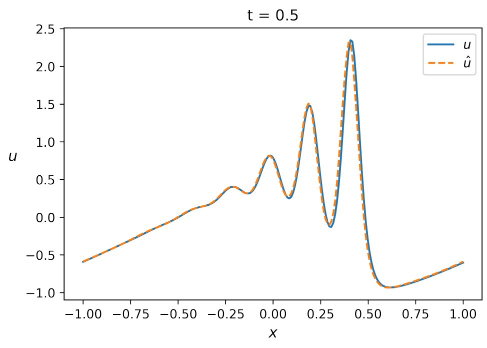
(b)



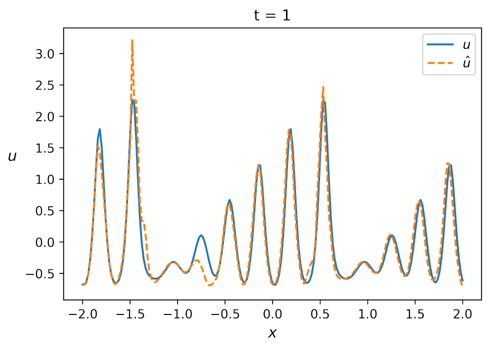
(c)



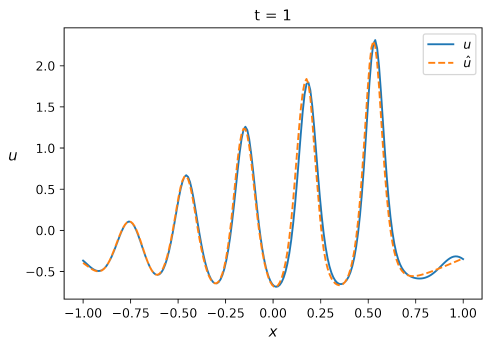
(d)



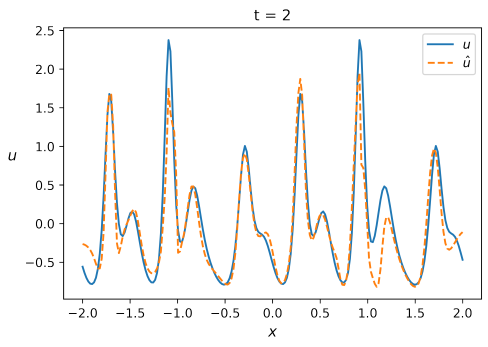
(e)



(f)



(g)



(h)

Figure 4.13: Illustration of prediction by a DNN with 500 training samples on the domain $x \in$ [-1,1] and $t \in$ [0,1] (a) and a DNN with 1000 training samples on the domain $x \in$ [-2,2] and $t \in$ [0,2] (b). Their respective absolute error and prediction at various times, is shown in (c,e,g) and (d,f,h).

Noticeably, although the PINN enforced the PDE as extra information compared to the DNN, it did not reach more accurate predictions on both grids. On the smaller grid, the a-PINN reached a 6.9e-3 MSE compared to the DNN that reached a 5.1e-3 MSE. On the larger grid, the a-PINN reached a 2.7e-2 MSE and the DNN reached a 5.1e-2 MSE. Additionally, both DNNs finished in under 10 minutes, which is substantially faster than a-PINNs' computation time of approximately 9 hours. If enough training samples are available and a small training computation time is preferred, one may be better off using a DNN. Although DNNs will fail to extrapolate. As of now, given a fixed PDE with fixed ICs and BCs, regular PINNs cannot outperform numerical methods in accuracy, which have been developed for the last couple of decades, especially since both require a PDE. However, even if a given problem can be solved numerically, there are scenarios in which PINNs may be more helpful. For example, given a PDE, PINNs can be designed to incorporate different types of inputs, such as different scalars in the PDE, a different geometry of the grid, or different BCs or ICs. And although training such a PINN could take exceptionally long, one forward pass through the network will almost always be faster than using classical numerical methods, which always require a different simulation for any change to the PDE problem. For example, this could be very useful in evaluating financial products, which derive their value from the Black–Scholes equation, and real-time data is required to constantly re-adjust a portfolio as fast as possible. Such PINNs are in very early development, and some real-time control applications have been studied by Antonelo et al. [2021] for example.

All in all, the a-PINN and n-PINN employing central differences have shown an overall good performance for the applications they have been tested on, though the a-PINN has shown to be the most consistent.

4.4 USING PINNS FOR INVERSE PROBLEMS

Up until now, PINNs have used the PDE in the loss function as a form of regularization to correctly predict the underlying solutions. To show the great utility and flexibility of PINNs, we will illustrate how PINNs can be used for inverse problem. Given enough training samples, can a PINN correctly identify the PDE which yielded the given solution? Interestingly, models have been used by for example Wang et al. [2021a] and Lu et al. [2019], where Physics-Informed Deep Operator Networks were used to correctly predict nonlinear operators of a PDE. Alternatively, Rudy et al. [2017] proposed a sparse regression method to efficiently select nonlinear and partial derivative terms to correctly identify popular PDEs using (noisy) data, including the KdV equation. We will not explore the theoretical background behind them but rather illustrate PINNs' accuracy in predicting an unknown coefficient in a PDE. This is done by using the unknown coefficient as a model parameter directly in the loss function, which is updated after each iteration step similarly to how model weights and biases are updated. Given (noisy) data, it may be of great interest to determine fundamental physical constants which are not directly measurable, such as the friction constant of the harmonic oscillator or Reynolds number in the Navier-Stokes equations. Another useful application is testing to what extent certain partial derivative terms contribute to the measurements and test whether higher order terms are negligible. This might be of great interest when a forward PINN is set up, but the PDE is not fully known.

In particular, empirical tests are performed to predict the friction constant μ from equation 4.1 and β from the KdV equation 4.5. The problem can be parameterized as follows:

$$\begin{aligned} &\mathcal{N}(u,\lambda) = 0, \quad \mathbf{x} \in \Omega, \quad t \in [0,T] \\ &\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(u_i - \hat{u}_i)^2 + \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(\hat{u}(\mathbf{x}_i,t_i),\lambda)^2, \end{aligned} \tag{4.9}$$

where λ is the unknown constant to be approximated. Notice that the collocation points are identical to the given training samples and no initial or boundary information is used.

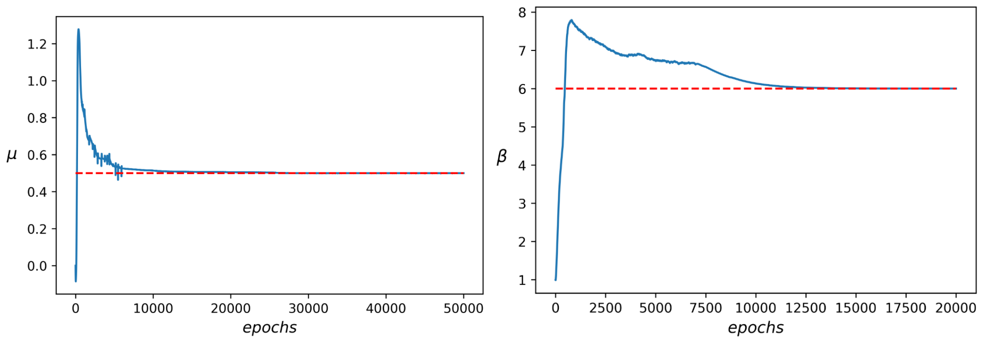
The PINNs used identical configurations as in the original problems, where 1000 training samples on the whole domain were used for the oscillator, and a 100 x 100 grid was used from the 2-soliton solution of the KdV equation. The notable differences were the loss function, and a different adaptive multistep learning rate in the Adam optimizer was used to yield faster convergence. The algorithm started with the initial guesses $\beta_0 = 1$ and $\mu_0 = 0.01$ with respective true values of 6 and 0.5. The results with noiseless data is shown in Table 4.8. Only the a-PINN was used to solve inverse problems.

Parameter	Prediction	True value	Relative error
μ	0.50007	0.5	0.015 %
β	6.002	6	0.035%

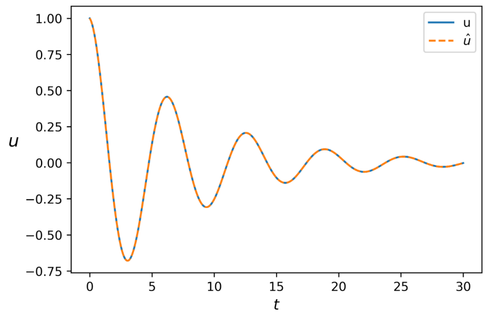
Table 4.8: Estimation of the unknown coefficient μ in the oscillator and β in the KdV equation, using noiseless training samples.

Although the initial guesses were very far off the true values, the PINN managed to estimate both unknown coefficients with extraordinary precision, with both relative errors far below 0.1%.

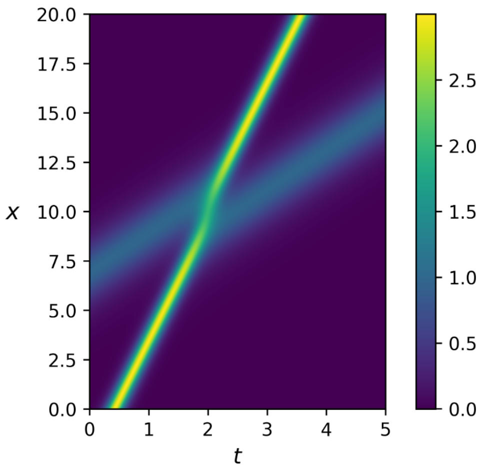
In practice, measurements are always prone to a certain level of noise. To see how accurate the unknown coefficient is estimated and the model is able to predict the correct solution, noisy data is generated by a uniform distribution. The time evolution of λ during training and the PINN's output after training with training samples with as much as 10% noise is shown in Figure 4.14.



(a) (b)



(c)



(d)

Figure 4.14: Illustration of μ (a) and β (b) during training at each iteration step with true value indicated by a dashed horizontal line. Model was given training samples with 10% noise, and predicted the oscillator (c) and the 2-soliton solution of the KdV equation (d).

Even with a high amount of noise, the model was able to correctly and simultaneously predict both solutions and a correct estimation of λ in both cases. This example greatly highlights the precision and flexibility when PINN models are data driven by training samples. The massive overshoot in Figure 4.14a and 4.14b is a result of setting the learning rate too high, but manages to correct itself and for long enough epochs the model is converging to the true value. The final estimated value of λ for various noise levels is shown in Table 4.9.

Noise	μ	Relative error μ	β	Relative error β
1%	0.4997	0.059%	6.002	0.027%
2%	0.502	0.32%	6.003	0.050%
5%	0.496	0.72%	6.01	0.17%
10%	0.508	1.66%	6.04	0.62%

Table 4.9: Estimation of the unknown coefficient μ in the oscillator and β in the KdV equation with different noise levels. Noise percentages represent maximum deviations from true value of observations and were drawn with a uniform distribution.

Even with as much noise as 10% in the training samples, the model manages to accurately predict both μ and β, with respective relative errors 1.7% and 0.6%. During multiple simulations, we observed that predictions using training samples with the highest level of noise had the biggest standard deviation. Hence to be more precise, one would need many simulations to make a more meaningful estimate of λ, for example by using the central limit theorem and constructing a confidence interval. However, this may be very time consuming to do depending on the problem, as a simulation to obtain one β took as long as 40 min on the GPU used.

Suppose that from the PDE an analytic solution can be derived with an unknown coefficient. Then if there are training samples available, one can also find the coefficient with a non-linear least squares method, such as the Levenberg-Marquardt algorithm, which is described by Levenberg [1944]. We use Levenberg's algorithm on the same noiseless training samples, and on the samples with identical noise on the oscillator to predict μ. The results are summarised in Table 4.10.

Noise	μ	s.d.	Relative error
0%	0.5	0	0%
1%	0.50002	0.0001	0.0037%
2%	0.50004	0.0003	0.0076%
5%	0.50009	0.0007	0.019%
10%	0.5002	0.001	0.038%

Table 4.10: Estimation of the unknown coefficient μ and standard deviation (s.d.) of the oscillator with different noise levels using the Levenberg-Marquardt algorithm. Noise percentages represent maximum deviations from true value of observations and were drawn with a uniform distribution.

For all noise levels, least squares estimation provides much more accurate results than inverse PINNs, while being computationally much more efficient. To put into perspective, after 7 minutes the inverse PINN reached a relative error of 1.7% with 10% noise, while the Levenberg-Marquardt algorithm finished with a 0.04 % relative error in under 3 ms.

5 CONCLUSION

The goal of this research is twofold. The first part provides a framework of the workings of a dense neural network by recollecting various literature studies to develop a theoretical background behind the simplest but complex type of neural network. We encoded a dense neural network ourselves in Python without machine learning libraries, and used that code to study the effect of changing its configuration on its performance. In particular, we have used the solution to the equation of motion of a damped harmonic oscillator at different points in space and time to predict its time evolution. Given enough training samples, a dense neural network is excellent in interpolation, but fails to extrapolate.

In an attempt to develop a model to accurately predict the solution of PDEs over the whole domain with as little training samples as possible, Raissi et al. [2019] came up with a Physics-Informed Neural Network (PINN). In the second part, we used a PINN to solve the same oscillator, the 1D heat equation and the 1-soliton and 2-soliton solution of the KdV equation. In particular, in our study we differentiated between two types of PINNs, namely the a-PINN and n-PINN, and compared their performance. Given a set of points on a discretized grid, PINNs use a form of regularization in the loss function to enforce that the prediction adheres to the underlying physics from PDEs, though without a guarantee that PINNs converge to the unique solution. When trained, both PINNs managed to approximate the solution to the oscillator, heat equation and 1-soliton with great accuracy. After different configurations, only when the PINNs were prompted with the 2-soliton solution, there was a large gap visible in their performance. The a-PINN managed to approximate the 2-soliton solution very well, while the n-PINN only managed to capture the overall physics behaviour, which showed no sign of improving performance when trying different network settings such as longer training. Another important difference between the a-PINN and n-PINN was their computation time. Furthermore, it became evident that PINNs' convergence to a solution is highly sensitive to the initial and boundary condition provided, provided there are no training samples to use on the interior. Additionally, PINNs are inaccurate when they are faced with a highly nonlinear problem such as a train of solitons. For such complex problems, training samples on the interior are required to come close to a qualitative approximation. All in all, the a-PINN has proven to be more reliable than the n-PINN and a fruitful tool in solving complex PDEs, with a promising potential future alternative, but no replacement, of the use of popular numerical methods for solving PDEs.

As a final note, PINNs were used to solve inverse problems, where the correct solution was provided and an unknown coefficient in the PDE was approximated. Even when using heavily noisy data (10%), PINNs were able to accurately predict the correct coefficient with a relative error of less than 2%. Simultaneously, the PINN managed to interpolate the solution accurately, showing the great versatility of combining PINNs with training samples, despite being trained upon noisy data. However, there is a much better alternative when predicting an unknown coefficient in PDEs when the analytic solution is known, such as the Levenberg-Marquardt algorithm, which has shown to be far more computationally efficient than inverse PINNs.

There are still many important elements that are worth to investigate for further research. In particular, despite the high complexity of dense networks, it may be worthwhile to develop a stronger theoretical background as it is common in the field of machine learning, to select and test models by trial and error. Although an attempt was made to dissect the mysterious black box, many questions remain unanswered such as:

- Why is an n-PINN able to effortlessly solve the 1-soliton solution, but fails to accurately describe the 2-soliton solution, even when both solutions come from the same PDE?
- Given a PDE with initial and boundary conditions, what requirement must be fulfilled such that PINNs will guarantee to converge to the unique solution?
- Why are data driven PINNs so succesfull with noisy data?
- Given a predetermined problem, does there exist an underlying criterion which would prove a particular configuration to be most successful, such as but not limited to the depth, width, activation function, loss function or weight initialisation scheme?
- Given a PDE, does there exist an amount and, in particular, a pattern of collocation points which would yield the most optimal results?
- Given a PDE, what is the minimum amount of training samples needed on either the interior, boundary or on some predetermined initial time, for an optimal PINN to yield desirable results?
- Does there exist a method to preemptively capture the uncertainty in the prediction, given a certain configuration of a PINN?

Although many questions are left unanswered, PINNs have shown to be a useful tool alongside numerical methods in solving PDEs. Although regular PINNs are as of now still inferior to numerical methods, which have had decades to develop, multiple extensions are being developed to improve PINNs. Nevertheless, it is conspicuous how Raissi et al.'s promising work brings a very bright future in combining machine learning with classical computational physics to come one step closer to potential breakthroughs in science and technology.

BIBLIOGRAPHY

Aggarwal, C. C. (2018). *Neural networks and deep learning*. Springer International Publishing, Cham, Switzerland, 1 edition.

Alipanahi, B., Delong, A., Weirauch, M. T., and Frey, B. J. (2015). Predicting the sequence specificities of dna- and rna-binding proteins by deep learning. *Nature Biotechnology*, 33(8):831–838.

Anco, S. and Willoughby, M. R. (2022). Kdv solitons.

Antonelo, E. A., Camponogara, E., Seman, L. O., de Souza, E. R., Jordanou, J. P., and Hubner, J. F. (2021). Physics-informed neural nets-based control. *CoRR*, abs/2104.02556.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43.

Bengio, Y. (2016). *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, London, England.

Cauchy, A. et al. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538.

Chiu, P.-H., Wong, J. C., Ooi, C., Dao, M. H., and Ong, Y.-S. (2022). Can-pinn: A fast physics-informed neural network based on coupled-automatic–numerical differentiation method. *Computer Methods in Applied Mechanics and Engineering*, 395:114909.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2014). The loss surfaces of multilayer networks.

Dauphin, Y. N., Pascanu, R., Gülçehre, Ç., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572.

Dennis, J. E. and Schnabel, R. B. (1983). *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice-Hall series in computational mathematics. Prentice Hall, Old Tappan, NJ.

Freund, Y. and Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168.

Lu, L., Jin, P., and Karniadakis, G. E. (2019). Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *CoRR*, abs/1910.03193.

Masters, D. and Luschi, C. (2018). Revisiting small batch training for deep neural networks.

Mattey, R. and Ghosh, S. (2022). A novel sequential method to train physics informed neural networks for allen cahn and cahn hilliard equations. *Computer Methods in Applied Mechanics and Engineering*, 390:114474.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Icml*.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA. PMLR.

Powell, M. J. (1964). An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162.

Raissi, M., Perdikaris, P., and Karniadakis, G. (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.

Reed, R. and Marks, I I, R. J. (1999). *Neural smithing*. A Bradford Book. Bradford Books, Cambridge, MA.

Rudy, S. H., Brunton, S. L., Proctor, J. L., and Kutz, J. N. (2017). Data-driven discovery of partial differential equations. *Science advances*, 3(4):e1602614.

Saad, D., editor (1999). *Publications of the newton institute: On-line learning in neural networks series number 17*. Publications of the Newton Institute. Cambridge University Press, Cambridge, England.

Simard, P., Steinkraus, D., and Platt, J. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 958–963.

Theil, H. (1971). *Principles of Econometrics*. John Wiley & Sons, Nashville, TN.

van der Meer, R., Oosterlee, C., and Borovykh, A. (2020). Optimally weighted loss functions for solving pdes with neural networks.

Wang, S., Wang, H., and Perdikaris, P. (2021a). Learning the solution operator of parametric partial differential equations with physics-informed deeponets. *CoRR*, abs/2103.10974.

Wang, S., Wang, H., and Perdikaris, P. (2021b). On the eigenvector bias of fourier feature networks: From regression to solving multi-scale pdes with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 384:113938.

Wolfe, P. (1969). Convergence conditions for ascent methods. *SIAM Review*, 11(2):226–235.

Wolpert, D. and Macready, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Woodworth, B., Patel, K. K., Stich, S., Dai, Z., Bullins, B., Mcmahan, B., Shamir, O., and Srebro, N. (2020). Is local SGD better than minibatch SGD? In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10334–10343. PMLR.

Zabusky, N. J. and Kruskal, M. D. (1965). Interaction of "solitons" in a collisionless plasma and the recurrence of initial states. *Phys. Rev. Lett.*, 15:240–243.

A APPENDIX

A.1 SOLVING KDV EQUATION WITH A SPECTRAL METHOD

Define the Fourier and inverse Fourier transformation as

$$F(w) = \int_{-\infty}^{\infty} f(t)e^{-iwt}dt$$
$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(w)e^{iwt}dw.$$

We can rewrite the KdV Equation 4.8 as

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right) + \delta^2\frac{\partial^3 u}{\partial x^3} = 0. \tag{A.1}$$

To apply the spatial Fourier operator $\mathcal{F}_x(\cdot)$ to Equation A.1, we use linearity and the following properties:

$$\mathcal{F}_x\left(\frac{\partial u}{\partial t}\right) = \frac{\partial}{\partial t}\left(\mathcal{F}_x(u)\right)$$

and

$$\mathcal{F}_x\left(\frac{\partial^n u}{\partial x^n}\right) = (iw)^n\mathcal{F}_x(u),$$

which can be derived using integration by parts. Applying $\mathcal{F}_x(\cdot)$ to Equation A.1 yields

$$\frac{\partial}{\partial t}\left(\mathcal{F}_x(u)\right) + \frac{1}{2}iw\mathcal{F}_x(u^2) - iw^3\delta^2\mathcal{F}_x(u) = 0. \tag{A.2}$$

After multiplying by $e^{-iw^3\delta^2 t}$, we may rewrite Equation A.2 as

$$\frac{\partial}{\partial t}\left(\mathcal{F}_x(u)e^{-iw^3\delta^2 t}\right) = -\frac{1}{2}iw\mathcal{F}_x(u^2)e^{-iw^3\delta^2 t}. \tag{A.3}$$

After discretizing the problem on a finite grid and using Fast Fourier Transformation, we can solve Equation A.3 using the fourth-order Runge-Kutta method with stepsize h = 1e-3.