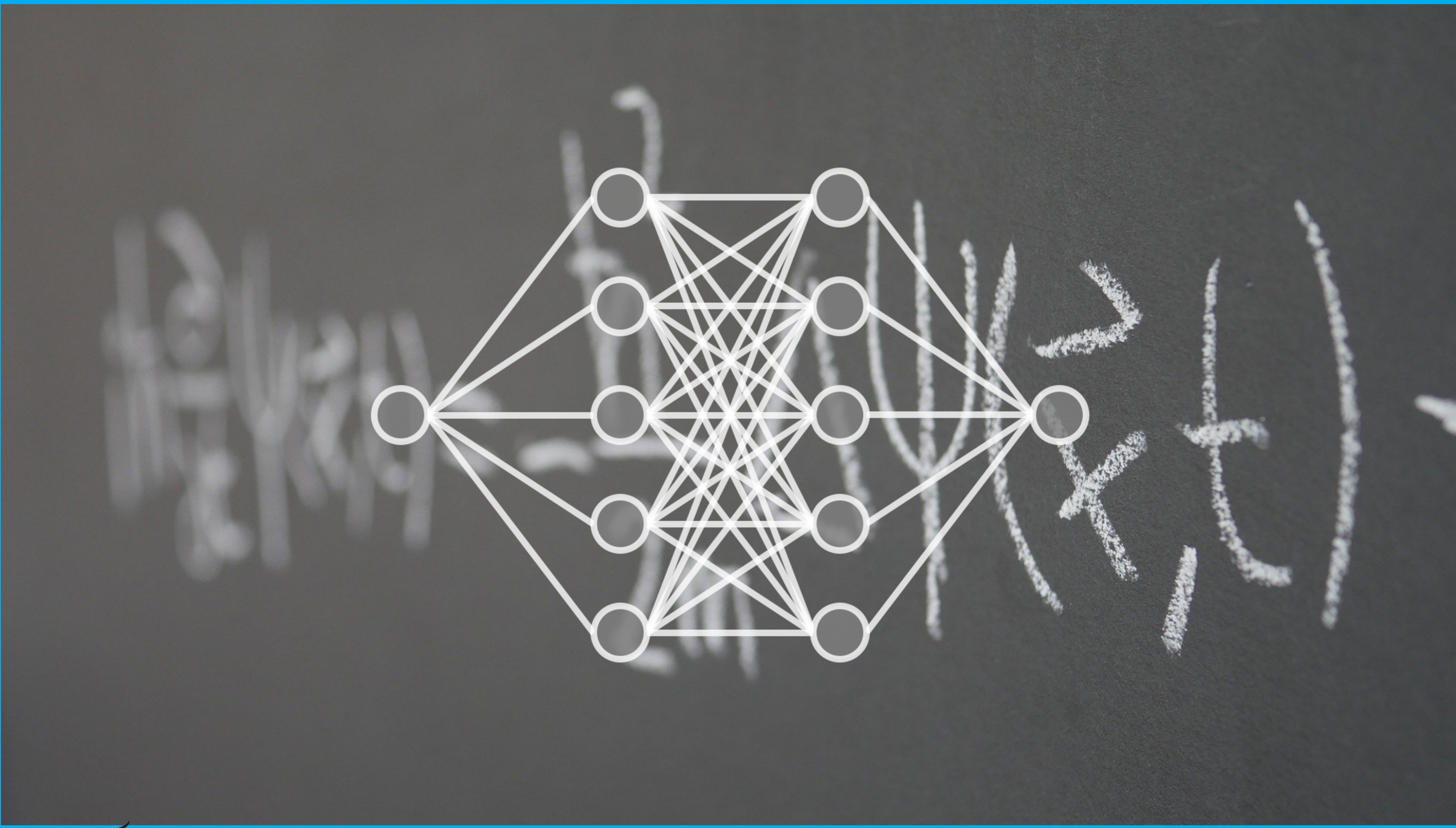
PINNs for parametrized problems

F.N.P. van Ruiten



PINNs for parametrized problems

by

F.N.P. van Ruiten

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday December 19, 2022 at 13:00 PM.

Student number:	4293339	
Project duration:	February 2020 – November 2022	
Thesis committee:	Prof. dr. D. Toshniwal,	TU Delft, supervisor
	Prof. dr. M. Möller,	TU Delft, supervisor
	Prof. dr. ir. A. W. Heemink,	TU Delft

TU Delft

Abstract

Physics Informed Neural Networks are a relatively new subject of study in the area of numerical mathematics. In this thesis, we take a look at part of the work that has been done in this area up until now, with the ultimate goal to develop a new type of PINN that improves upon the old concept. We introduce the concept of parameterized PINNs, which allow a single trained network to solve multiple partial differential equations for multiple boundary conditions and geometries by parametrizing these variables as an input for the network. Two methods are tested: one using global basis functions, and one using B-splines. The proposed methods are tested for Laplace's equation and Poisson's equation in multiple dimensions, most of which show that these methods are viable alternatives for the current style of collocation-based PINNs.

Contents

List of Figures 1

1 Introduction 5

2 Background Information 7

2.1 Partial differential equations 7

2.1.1 Definitions 7

2.1.2 Examples 8

2.2 Numerical methods for differential equations 10

2.2.1 Finite difference method 10

2.2.2 Finite element method 11

2.3 Machine learning and neural networks 12

2.3.1 Machine learning 13

2.3.2 Neural networks 13

2.3.3 Architecture 14

2.3.4 Activation functions 16

2.3.5 Loss function 16

2.3.6 Training 17

3 Physics Informed Neural Networks 19

3.1 The basic PINN 20

3.1.1 General definition 20

3.1.2 Loss functions 20

3.2 Variations on the basic PINNs 21

3.2.1 DPINNs 21

3.2.2 fPINNs 22

3.2.3 VPINNs 22

3.3 Preliminary investigation into PINN extensions 22

4 PINNs using a global basis 25

4.1 Basic idea 25

4.2 Adapting PINNs to solve parametrized problems 27

4.3 Application to 1D Laplace . . . 29
4.3.1 Initial Results . . . 29
4.3.2 Investigating the effect of varying the latent space size . . . 32
4.3.3 Investigating the effect of different training point sets . . . 34
4.4 Application to 1D Poisson with parametrized source function . . . 35
4.4.1 Hard-coded polynomial source function . . . 36
4.4.2 Polynomial source function as input parameter. . . . 37
4.4.3 Non-polynomial source functions . . . 39
4.4.4 Investigating the influence of the chosen basis . . . 40
4.5 Application to a higher dimensional problem . . . 45

5 Spline-based PINNs 47
5.1 Adapting PINNs to use splines . . . 47
5.2 Application to one dimensional cases . . . 49
5.2.1 Application to Laplace's equation . . . 49
5.2.2 Error near boundaries . . . 51
5.2.3 Application to Poisson's equation . . . 52
5.2.4 Results for non-polynomial source functions . . . 53
5.3 Two dimensional cases . . . 55
5.3.1 Adapting the method to two dimensions . . . 55
5.3.2 Application to Poisson's equation . . . 55
5.4 Incorporating a parametrized geometry map . . . 58
5.4.1 Encoding the geometry . . . 58
5.4.2 Application to Poisson's equation . . . 58

6 Conclusion 61
6.1 Summary . . . 61
6.2 Future Research . . . 62
6.2.1 Global basis PINNs . . . 62
6.2.2 Spline based PINNs. . . . 62

Bibliography 63

List of Figures

2.1 A typical example of a feed-forward neural network. Image sourced from `miro.medium.com/max/1200/1*Gh5PS4R_A5drl5ebd_gNrg@2x.png` 15

3.1 Three overlapping networks, with interactions between all three of them. The network is trying to predict the function $f(x, y) = \sin(2\pi x)\sin(\pi y)$, which it is able to do adequately. 24

4.1 The loss of a network plotted for every training epoch. While the results may differ between training attempts, they all share the same general shape: a rapid decrease in loss until $O(10^{-3})$, followed by a gradual decrease with many fluctuations. Training the network took 4.2 seconds. 29

4.2 The loss of two networks plotted for every training epoch. Both networks were trained using the same training data, the only difference being the batch size. The unbatched network too 6.2 seconds to train, while the batched network too 9.0 seconds. Note that the training time is longer due to the higher number of epochs. This was chosen because the spikes do not appear prominently before this point in the full set. 30

4.3 Three randomly generated test points, with the exact solution plotted on top of the predicted solution. Note that the points in the interior of (0,1) are only shown for visual clarity, as the predicted solutions only depend on the value of the boundary points. 30

4.4 Three randomly generated test points, with the exact solution plotted on top of the predicted solution. Like before, interior points are there for visual clarity. 31

4.5 Multiple networks trained with different sizes for the latent space. Here, n is the order of the polynomial that is generated for the solution. Training time for all networks was around 6.1 seconds. 33

4.6 The MSE loss of the network after 1000 epochs, trained using the different training sets stated before. As can be seen, the loss stays mostly the same across the different training sets. 35

4.7 The loss of a network plotted for every training epoch. The chaotic nature of the plot is explained through the larger number of batches, which introduces more randomness in the optimization. Training the network took 6.5 seconds. 36

4.8 Three random test points. The source function was taken to be $f \equiv 2$. 37

4.9 The loss of a network plotted for every training epoch. Again, the increased chaos is due to the increase of the number of batches. Training took 34 seconds. 37

4.10 The loss of a network plotted for every training epoch. With an increase in solution space size, the accuracy is starting to drop. Training took 3 minutes due to the increased input tensor size. 38

4.11 The first generated solution, denoted using small red dots, shows that the network can successfully solve the PDE with a linear source function. The second generated solution, denoted using the large red circles, shows that the network can still solve the PDE with a constant source function. The third generated solution, denoted using the red stars, shows that the network can extrapolate somewhat adequately. 38

4.12 The loss for the same network, trained with a different size n for the order of our latent space. While $n = 9$ is used later, it has been left out of this graph for clarity reasons. It slightly improved over $n = 7$. 39

4.13 The output for a random test point (dots) plotted together with the analytical solution (line). The network is able to approximate the desired solution very well. The coefficients generated solution approach the coefficients of the Taylor series of the analytical solution. 40

4.14 The loss of a network trained using Legendre polynomials, for different size latent space. As can be seen, networks with a larger latent space have a larger loss on average. Training took about 18 seconds for all latent spaces. 41

4.15 The loss of several networks trained using a Fourier basis with N terms. Compared to the polynomial networks, it is a lot slower to train, as it took 5000 epochs to reach a level which most polynomial networks could easily reach in hundreds. We see that the networks improve with more terms, which was expected. Training for every network took about 75 seconds. 43

4.16 The generated solution found by the network trained on a Fourier basis up to the 7th terms. The network was only trained with a loss calculated on $[0,1]$, where we have also placed the analytical solution it was trying to approximate. As can be seen, the it is able to approximate this solution adequately well. 44

4.17 The loss of two networks trained to solve Laplace's equation using the Fourier series as a basis. Both were generated up to the 7th terms. The 'periodic boundary' network was trained for boundary conditions for which $u(-1) = u(1) = a$, with $a \in \{0, 1, ..., 10\}$. The 'non-periodic boundary' network was trained for boundary conditions $u(-1) = a, u(1) = b$, with $a, b \in \{0, 1, ..., 10\}$ 44

4.18 The generated solution found for untrained point $c = 4.6$. The network was trained on manufactured solution 4.11 for $c \in \{0, 1, 2, ..., 10\}$. A latent space with terms up to x^5 was used to train the network on. 46

5.1 The loss of the network trained to solve Laplace's equation in 1 dimension using splines. The speed at which the network is trained is noticeably slower than when using global basis functions. Training this network took 61 seconds. 50

5.2 The generated solution for a trained input and 2 untrained inputs plotted together with the analytical solutions. Note that the input with one boundary condition close to zero is not being solved correctly. 51

5.3 Three generated solutions for inputs close to 0. This used the same network as in the previous figure. Note that the solutions start to improve for boundary values further away from 0. 51

5.4 The loss of the network trained to solve Poisson's equation in 1 dimension using splines. The figure is similar to the figure for Laplace's equation, yet is an order of magnitude worse than before. Training this network took 119 seconds. 52

5.5 Three generated solutions; one trained and two untrained inputs. Once again the network is able to solve for the trained inputs and the untrained inputs away from 0, but has problems with untrained inputs with a boundary condition close to 0. 53

5.6 The loss for two identical networks save for the number of coefficients that needed to be predicted. As can be seen, the network with more knots is slower to train than the one with fewer; yet yields better results eventually. Training took 135 seconds for both networks. 54

5.7 The generated solutions for the randomly generated input with $u(0) = 8.2$, $u(1) = 5.6$ and $f = -6.4\sin(x)$. The solutions generated by the two networks with different numbers of knots have been plotted together. As can be seen, increasing the number of knots improves the solution. 54

5.8 The loss of the network trained to solve Poisson's equation in 2 dimensions. Training this network took 532 seconds. 56

5.9 The generated solution for a trained point with boundary values 0. As can be seen, the network has trouble solving the boundary conditions even though the loss had plateaued. 56

5.10 The loss of the network trained to solve Poisson's equation in 2 dimensions, with additional weight placed on the boundary condition loss. Training the network took 529 seconds. 57

5.11 The generated result for an untrained point with boundary values 0 and source function $f \equiv -4.2$. As can be seen, the quality of the result has improved considerably over the unweighted version. 57

5.12 The figure on the left is the result of a trained point with boundary values at 0, $\theta_{max} = 1$ and $f \equiv -10$. The figure on the right is the results of a testing input with boundary value 1.4 at $\theta_{max} = 1.5$ and $f \equiv -10$. The boundary values at $r = 0.5$ and $r = 1$ were taken to increase linearly as a function of θ to ensure that the boundary values at the corners are the same. 59

1 Introduction

Numerical methods for solving differential equations have been used and extensively researched since the moment computers became strong and affordable enough to make use of these methods. Finite element methods in particular has become the gold standard in many engineering disciplines due to its robustness. All of these numerical methods have in common that they require an often very large system of linear equations to be solved.

This is where a large part of the problems that classical numerical methods can have come in. Every time a new instance of a model needs to be run, the program has to run the linear solver. This is the reason why so much time is spent researching faster and more stable linear solvers, as an efficiency increase of a small percentage can already cut costs quite a bit.

Neural networks are one of the major machine learning methods that have seen a lot of development in the past decade. Recently, the first paper on neural networks being used to solve differential equations was written. As a trained neural network can evaluate a given input extremely quickly, these can be of great interest to solve multiple very similar differential equations.

While the neural networks presented in [10] are a great start, they fail to wholly take advantage of the unique properties that neural networks possess. As such, this thesis is mainly focused on introducing multiple new methods of using neural networks to solve differential equations in a way that does take advantage of these properties.

This thesis is structured as follows: In chapter 2 we quickly go over the background knowledge that is required to understand the rest of the thesis. This chapter covers the relevant definition of partial differential equations, examples of differential equations that are used throughout this thesis, as well as ways to solve them both analytically and numerically. This chapter also contains the introduction to machine learning and neural networks. In chapter 3, we look at the basics of solving partial differential equations using neural networks. It also covers the a variant of this method, and the work that was based off this variant. Chapter 4 covers a completely new method of using neural networks to solve partial differential equations, using global basis functions. This covers both theory and results in 1 and 2 dimensions. Chapter 5 contains another different method of using neural networks to solve partial differential equations, this time using B-splines. Again, results in both 1 and 2 dimensions are provided. We also introduce a parametrized geometry with results. In chapter 6, we conclude the thesis by giving a summary of the results and give recommendations for further research.

2

Background Information

2.1. Partial differential equations

In this section, we introduce both the general concept of partial differential equations (PDEs) and some examples of PDEs used in this thesis. As the central subject of this thesis, it is assumed the reader has at least a passing familiarity with (partial) differential equations.

2.1.1. Definitions

On a d-dimensional, finite domain $\Omega \subset \mathbb{R}^d$, we consider the PDE

$$\mathscr{L}(x_1, ..., x_d; u; \frac{\partial u}{\partial x_1}, ..., \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1^2}, ...) = f(x_1, ..., x_d) \text{ on } \Omega$$

which describes the relation between the solution $u(x_1, ..., x_d)$ and the source function $f(x_1, ..., x_d)$. For ease of notation, we will be writing $\mathbf{x} := (x_1, ..., x_d)$ and $\mathscr{L}(\mathbf{x}, u) = f$ from now on. Similarly, we consider the following on the boundary $\partial\Omega$:

$$\mathscr{B}(x_1, ..., x_d; u; \frac{\partial u}{\partial x_1}, ..., \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1^2}, ...) = g(x_1, ..., x_d) \text{ on } \partial\Omega$$

where once again we simplify our notation to $\mathscr{B}(\mathbf{x}, u)$. Taking both equations as a system we get a boundary value problem (BVP) defined on Ω. Since we will be interpolating solutions of various boundary value problems using neural networks later, it is important that these solutions can flow into each other continuously. For this, the problem needs to be well-posed.

For a BVP to be well-posed, we need a solution $u(\mathbf{x})$ to exist and be unique. We also need the solution's behaviour to depend continuously on $\mathscr{B}$, f and g. In this thesis, we will only be discussing time-independent problems with a Dirichlet boundary condition. As such, we have omitted dependency on the initial condition from our well-posedness. The upcoming examples are both well-posed problems [17].

2.1.2. Examples

Example 1: Laplace's Equation

Sometimes also referred to as the steady-state heat equation, Laplace's equation is defined as follows:

$$\Delta u = 0 \text{ on } \Omega \tag{2.1}$$

Laplace's equation is the simplest elliptic PDE and a special case of both Poisson's equation and the Helmholtz equation. This property combined with the simplicity of the solution makes this a very easy first step into solving more general PDEs.

Solving Laplace's equation analytically can be done quite easily. In one dimension, the entire equation can simply be integrated twice to obtain a linear equation for the interior:

$$\int\int \Delta u = \int\int \frac{d^2u}{dx^2} = c_1 x + c_2 \tag{2.2}$$

Applying the boundary conditions gives us the values of both integration constants, and thus the solution.

In two or more dimensions, we can find the solution by separation of variables. This example has been worked out for the two dimensional case on a square, but similar techniques can be used on higher dimensional problems or different geometries. Instead of solving the problem directly, it is usually easier to split up the problem into multiple separate problems first:

$$u(x,y) = u_1(x,y) + u_2(x,y) + u_3(x,y) + u_4(x,y) \tag{2.3}$$

Which takes advantage of the linearity of the differential operator to get:

$$\Delta u(x,y) = \Delta u_1(x,y) + \Delta u_2(x,y) + \Delta u_3(x,y) + \Delta u_4(x,y) = 0 \tag{2.4}$$

Here, every u_i solves the same equation on the interior of the domain (i.e. $\Delta u_i = 0$), but solves only one of the boundary values. The other boundary values are set to zero for the problem. This way, we solve the four boundary values separately. We can then simply add our u_i to obtain the actual solution u. Using separation of variables, we get

$$u_i(x,y) = X_i(x)Y_i(y) \tag{2.5}$$

Which gives us the following eigenvalue problem:

$$\frac{\partial^2 X_i}{\partial x^2} = -\lambda^2 X_i \quad \text{and} \quad \frac{\partial^2 Y_i}{\partial y^2} = \lambda^2 Y_i \tag{2.6}$$

Depending on the value of λ^2, we find different solutions.
For $\lambda^2 > 0$:

$$\begin{cases} X_i(x) = c_1 \sin(\lambda x) + c_2 \cos(\lambda x) \\ Y_i(y) = c_3 e^{\lambda y} + c_4 e^{-\lambda y} \end{cases} \tag{2.7}$$

For $\lambda = 0$:

$$\begin{cases} X_i(x) = c_1 x + c_2 \\ Y_i(y) = c_3 y + c_4 \end{cases} \tag{2.8}$$

For $\lambda^2 < 0$:

$$\begin{cases} X_i(x) = c_1 e^{\lambda x} + c_2 e^{-\lambda x} \\ Y_i(y) = c_3 \sin(\lambda y) + c_4 \cos(\lambda y) \end{cases} \tag{2.9}$$

Using the zero-valued boundary conditions, one can find the allowed values of λ, which we will call λ_n. This will leave us with constants B_n, one for every λ_n. Taking the superposition over all λ_n gives us the general form of the answer. The last remaining constants can then be found by applying the remaining boundary condition. Once this has been done for all four (or fewer, if the original problem has a zero-valued boundary) boundary equations, we can simply add the four solutions to obtain u.

Example 2: Poisson's Equation

Poisson's equation takes the following form:

$$\Delta u = f \text{ on } \Omega \tag{2.10}$$

From this, it is immediately apparent that Laplace's equation is a special case of Poisson's equation, namely when $f \equiv 0$. Much like ODEs, one has to take into account both the homogeneous and the non-homogeneous equation. The homogeneous part of Poisson's equation is Laplace's equation, so this just leaves the non-homogeneous part.

In the one dimensional case, one can simply integrate f twice to obtain the particular solution:

$$u_{part} = \int\int \frac{d^2u}{dx^2} = \int\int f \tag{2.11}$$

Which gives us the following complete solution:

$$u = u_{hom} + u_{part} = c_1 x + c_2 + \int\int f \tag{2.12}$$

Solving Poisson's equation in more than one dimension is very similar to solving it in only one. We split the equation into a homogeneous part and a particular part. The homogeneous equation is Laplace's equation, which we solved before. The particular part of the equation can be problematic, however. Techniques such as variation of parameters [5] can be used to solve this.

2.2. Numerical methods for differential equations

In this section, we will briefly go over some of the most widely used numerical methods for solving differential equations. While the methods described in this section vary quite a bit when it comes to the execution of the algorithm, the methods all share some of the same steps.

Section 2.2.1 covers the finite difference method. In section 2.2.2, we discuss the finite element method. Only a shallow understanding of the methods described here is necessary to understand the differences between these classical methods and the new methods discussed in this thesis. Therefore, no familiarity with numerical methods is needed.

2.2.1. Finite difference method

Finite difference methods (FDM) are one of the most used methods in numerical analysis, due to both its general applicability and the ease of implementation. While many variations of finite difference methods exist, all methods share the same principles in basis. An FDM can be summarised in 3 steps, these being discretisation, approximation and computing.

A grid is imposed on our domain Ω, giving us a finite number of nodes $\mathbf{x}_i$. The density of this grid affects the computing time and the accuracy of the solution; a more dense grid will give a more accurate solution, at the cost of a higher computing time.

This grid does not need to be uniform, but the accuracy of the solution on the boundary between two grids of different density is lower than the accuracy of either grid by itself. As such, unless a specific part of the problem is in need of a high density grid, it is preferable to use a uniform grid whenever possible.

Next is approximation, in which the PDE that is to be solved is locally approximated. This step is where most of the different finite difference methods vary from one another. Which, how many, and with what weight the nearby grid nodes are used to approximate the PDE change between different methods. We quickly go over one of the simpler approximations for a first and second derivative.

On every node $\mathbf{x}_i$, the derivatives are approximated using a Taylor polynomial. When working in $\mathbb{R}$, this results in the following expression for the node x_j, the so called 'central difference scheme':

$$\frac{\partial u}{\partial x}(x_j) = \frac{u(x_j+h)-u(x_j-h)}{2h} + O(h) \tag{2.13}$$

Obtaining an approximation for higher order derivatives simply requires us to use this approximation twice:

$$\frac{\partial^2 u}{\partial x^2}(x_j) = \frac{1}{h}[\frac{\partial u}{\partial x}(x_j+\frac{h}{2}) - \frac{\partial u}{\partial x}(x_j-\frac{h}{2})] + O(h^2) = \frac{1}{h^2}[u(x_j+h)-2u(x_j)+u(x_j-h)] + O(h^2)$$

When done for every node in the grid, this results in a system of equations. This leads us to the final step of the process: the computing. Simple solvers can be used to solve small systems of equations, but the same cannot be said for larger systems. Since all equations in our systems only depend on the nearest nodes, this leads to highly sparse matrices. While taking a grid twice as dense reduces the order of the error term accordingly, it also doubles the size of the number of equations. When working in $\mathbb{R}^d$ with $d > 1$, the curse of dimensionality rears its ugly head as well, increasing the number of equations by a factor of 2^d instead. This has sparked the development of multiple algorithms for solving systems of linear equations with these highly sparse matrices.

2.2.2. Finite element method

Compared to FDM, the finite element method (FEM) has the advantage when it comes to complex domain shapes. Instead of a rigid grid structure that is used in an FDM, a generic mesh is used. This allows the method to work in and around complex geometries with relative ease.

Like FDMs, this method starts by subdividing our domain Ω. While using triangles to construct the mesh is most common, the method can also be applied to differently shaped elements.

We then convert the PDE to the weak formulation by introducing a test function v. Using the boundary conditions and some common integration techniques, one can find a separate integral equality. This allows us to reduce the PDE to the following problem: Find a suitable u such that for all test functions the found equality holds. Since all of this has been done in a Sobolev space, we can choose a finite basis by simply restricting the forms both u and v take. The most common choice here is by using piecewise linear elements, but higher order elements can be used for higher accuracy.

Like in the previous method, this results in an equation for every element. By approximating the integrals where necessary, this leaves a system of linear equations. Like with the finite difference method, we can use linear solvers to find the solution for every element.

FEMs have a history of successful applications in many engineering disciplines, and as such is usually preferred over FDM in these topics. This is mainly because of two factors. Firstly, FEM has a higher degree of accuracy when compared to FDM. Secondly, because the solution is constructed using base functions that have a value on the whole element, FEM predicts the solution on every point in the domain, whereas FDM only does this for the chosen gridpoints.

2.3. Machine learning and neural networks

In this section, both the concept of machine learning and in particular neural networks are introduced. In section 2.3.1, the basic concepts surrounding machine learning are discussed. In section 2.3.2, the particular type of machine learning used in this thesis, neural networks, are introduced. It is assumed that the reader has no prior knowledge of machine learning.

2.3.1. Machine learning

As the name suggests, machine learning is a form of programming that allows a machine to ‘learn’ something from a data set. As such, it is a form of artificial intelligence (AI), and is currently seen as a stepping stone to create an AI with true intelligence comparable to humans. While the idea of artificial intelligence -intelligence invented or created by mankind- has been around since antiquity, any real progress has only come with the invention of computers that were able to perform calculations faster and cheaper than humans. As such, artificial intelligence and deep learning are for all intents and purposes, a rather new scientific subject. With computers becoming progressively faster, machine learning too has become a more useful and realistic tool to use.

Machine learning in itself comes in several different forms, which are usually divided into supervised and unsupervised learning, with semi-supervised taking parts of both techniques. The difference between supervised and unsupervised learning is in the labeling of the input data. Unsupervised learning models are supplied with unlabeled data, usually with the goal to find a structure. Finding clusters of data or hitherto unfound connections in data is one of the primary uses of unsupervised learning. Supervised learning on the other hand, uses labeled data. This means that the program is given a desired output for every data point, which allows it learn the correct response to a given input. Semi-supervised learning uses a mix of both labeled and unlabeled data, which has been found to improve the accuracy of machine learning models.

2.3.2. Neural networks

Neural networks are a prime example of supervised learning. A network is given a set of input data and the desired outcome for each data point, and over the course of (usually) thousands of iterations, will learn to see the pattern that brings the desired outcome. Neural networks are heavily inspired by the structure of animal brains (hence the name), and have been around for about a century. However, due to the lack of computational power at this time, the subject saw little improvement until the 1980’s. With the invention of Werbos’s backpropagation algorithm in 1975 [1], multi-layered networks became a viable possibility, and shortly after, in 1986, it was shown that a network was able to successfully predict the next word in a sequence [2]. From this point onward, many applications have been found for neural networks, including facial recognition, computer vision and recently, PINNs.

While neural networks have seen a surge in popularity in the last few years, it has also seen its fair share of criticism. It is often called a black box method, as we are unable to understand how a trained network works. While it is indeed possible to look under the hood and see any and all of the connections the computer has made, the sheer number of these make it too complex to understand for a human. This has sparked controversy, especially when a network is handling sensitive data, such as personal information. While neural networks are great in finding correlation, this can lead to problems if the initial training data set has a bias.

As an example, let us say that a neural network was created to try to predict potential tax fraud. If our initial data set contains only five people from a certain city, all of whom have committed tax fraud, our network might correlate tax fraud to this city very strongly. While it is possible that there is a city in which a high percent of the population commits tax fraud, the more likely explanation is that our sample size was too small for people from this city. As such, a good data set should both have a large sample size (ideally for every relevant value of every variable), and a varied data set (to prevent initial bias).

2.3.3. Architecture

Neural networks mainly contain two components: neurons, and edges, which are the connections between neurons. The neurons are arranged in layers, starting with the input layer, followed by a number of hidden layers, and ending in the output layer. The number of layers is called the width of the network, while the number of neurons in a layer is called the depth of the network. These parameters, along with any other parameters that are chosen by the creator of the network and which cannot be changed for a given network are called hyperparameters. The network has two types of variables that can be changed to affect the outcome, which are the weight W and the bias b. Other than this, each connection between neurons is filtered through an activation function ϕ. If ϕ is a linear function, the network is nothing more than a convoluted series of tensor multiplications. If ϕ is nonlinear, we call the network a deep neural network, and is in theory capable of producing more interesting results.

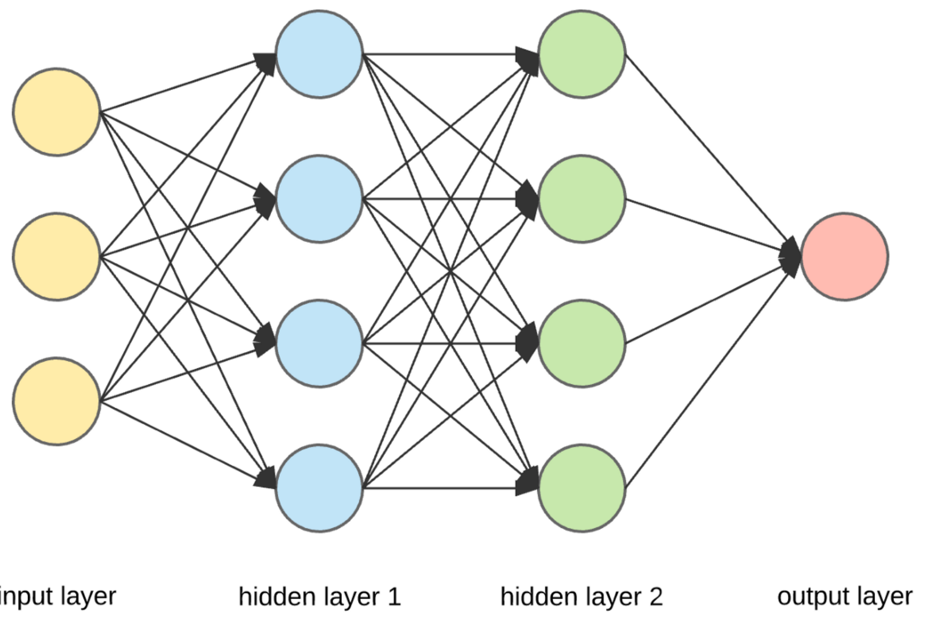


Figure 2.1: A typical example of a feed-forward neural network. Image sourced from `miro.medium.com/max/1200/1*Gh5PS4R_A5drl5ebd_gNrg@2x.png`

Figure 2.1 denotes a feed-forward network, a network in which information can only go forward into the network. Neural networks in which it is possible for information to go to a neuron in an earlier layer is called a recurrent network, but these are outside the scope of this project. For a network of width $n+1$, we denote the value of the neurons on the ith layer by f_i and the biases by b_i. The weight tensor W_i denotes the weights between layers $i-1$ and i. The input data is denoted by x_0. We can then write our network as follows:

$$f_i(x) = \begin{cases} x_0 & i = 0, \\ \phi(W_i f_{i-1}(x) + b_i) & i = 1, 2, ..., n-1, \\ W_n f_{n-1} + b_n & i = n \end{cases} \tag{2.14}$$

There are several things of note here. Firstly, it is not necessary to have a single activation function for all layers. It is possible to have multiple activation functions. Secondly, it is possible to have so called skip-connections, which connect non-neighboring layers. It has been shown that these can vastly improve the trainability of a network.[6] Both of these topics are outside the scope of this project.

2.3.4. Activation functions

As stated before, activation functions add the non-linearity to the network, which makes it capable of producing interesting results. Unfortunately, the non-linearity also makes it very hard to analyze neural networks, but more on that later. In this section, we list a few of the most commonly used activation functions, and their primary use.

Sigmoid function The Sigmoid function is defined as $\phi(x) = \frac{1}{1+e^{-x}}$. Since the gradients are 0 for very small and very large values, it is susceptible to the vanishing gradients problem. Since the amount of change in weights is determined by the value of the gradients, this problem can stop a network from optimizing. It does however see use in the final layer of classification-type networks. The Sigmoid function can be further generalized to the so called Softmax function.

ReLU The rectified linear unit function or ReLU, is defined as $\phi(x) = \max(0, x)$. It is commonly used as the activation function for hidden layers. A large drawback of ReLU is that it has gradient 0 for $x < 0$, which causes nodes to get stuck. An improved version of ReLU was created, called Leaky ReLU. Leaky ReLU is defined as $\phi(x) = \max(0, x) + \alpha \min(0, x)$, for some $0 < \alpha << 1$. Since the gradient is now no longer 0, it can no longer get stuck like ReLU.

2.3.5. Loss function

In order for a network to find the optimal variables, it needs to be able to evaluate the results. This is done via the so-called loss function (sometimes referred to as the cost function or error function). During training, the network computes the outcome of the input data, and compares it to the desired outcome via the loss function. Which loss function is used depends on what the network is used for.

Classification-type networks usually use maximum likelihood estimation or cross-entropy loss functions. Non-classification type networks, like the one in this thesis, generally use the mean square error as a loss function. This is defined as

$$L(f) = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2$$

2.3.6. Training

Once the hyperparameters and the loss function have been chosen, the network can be trained. During training, the network will iterate thousands of times. Every time the data is run through the entirety of the input data, it is called an epoch. A subset of the input data called a batch will be fed as input to the network. The output is compared to the desired output via the loss function, and some of the variables are changed to improve the results. The network will repeat this until loss is as low as desired, or the maximum number of epochs is reached. While the batch size is a hyperparameter, it is currently unclear what the optimal value is for the batch size, or whether an optimal value exists at all.

The way the network knows which weights and biases to change is via gradient descent. As the name suggests, the algorithm looks at the direction the gradient is steepest, and changes the variables in this direction. The hyperparameter that determines how large this change is, is called the learning rate of the algorithm. Having a learning rate that is too small results in a very slow algorithm, while having a learning rate that is too large might result in overshooting the optimal solution.

It is possible for the network to reach a local minimum which it can get stuck in using gradient descent, but fortunately slight alterations to the algorithm make this less likely. Alternatives include stochastic gradient descent [3], which is gradient descent which requires a batch size smaller than the whole input data. This adds an element of randomness into the direction of the gradient, which means it is less likely to get stuck in a local minimum. An algorithm that builds further upon SGD is Adam [4], which changes the learning rate depending on the moving average of the variance in the gradients. Throughout this thesis, the Adam optimizer has been used for all neural networks.

3 Physics Informed Neural Networks

The idea behind using neural networks to solve PDEs instead of more traditional methods is quite simple. When implemented correctly, this would theoretically allow us to train a single network to quickly solve a PDE for multiple different boundary conditions, or perhaps multiple PDEs. As an example, consider modelling the airflow around the wing of an aeroplane. For every design that is to be tested, the program has to run again. The appeal behind using a neural network is that it only needs to be trained one single time. Once trained, the network can be evaluated very quickly, potentially cutting down heavily on the time needed when compared to running a classical method for every design that needs to be tested. While these networks can be applied to any generic, well-defined PDE, they are commonly referred to as physics informed neural networks (PINNs).

Unfortunately, as neural networks themselves are a relatively young subject of study, so are PINNs. Most research in the last five years has been based on [10]. As such, this chapter will introduce the concept of PINNs as was created by Karniadakis et al. The method described in section 3.1 gives a general idea for what the structure of a basic PINN should look like. Several variants on the basic PINN have been researched in the last few years, some of them are discussed in section 3.2.

3.1. The basic PINN

3.1.1. General definition

The idea proposed in [10] and the works that have built on top of this all share the same method of attempting to solve a PDE using a neural network. The idea is to try to let the network learn the solution directly. This is achieved by giving the network a set of points in the work domain as an input, and letting it generate the solution for every point separately as an output. Theoretically, this would allow one to input a hence unforseen point in the domain into the PINN, and get the solution at that point as an output.

The difficulty here lies in how to let the network learn the solution of the PDE. In one way or another, the network will have to fulfill both the boundary conditions on the boundary, and the PDE in the interior. For time-dependent problems, an initial condition is present as well. These properties which the network must fulfill can be seen as constraints, and multiple ways of implementing these restraints have been attempted [11]. A distinction is made between so-called hard and soft constraints.

Hard constraints are defined as constraints that have to be fulfilled in order for the predicted solution to be a valid one. At a first glance, this is the best way to describe (part) of the constraints we are dealing with. The boundary conditions especially are a natural fit for the hard boundary conditions. However, while research into using hard constraints is currently being done, it is currently not at a stage in which it is easy to implement or execute.

3.1.2. Loss functions

Soft constraints are simply defined as non-hard constraints. Predicted solutions do not necessarily have to fulfill these conditions, but rather be as close as possible to the constraint without affecting the solution as a whole negatively. A great way to do this is to simply include a term for every soft constraint in the loss function. This means that using soft constraints is an very easy and general method of adding additional constraints to a network when compared to hard constraints. Unfortunately, adding a term to the loss function causes the network to train slower in the best case and be overall worse in the worst case. As such, it is preferable to use hard constraints wherever possible. Since the development of general-use hard constraints is still in an early stage, this thesis was made solely using soft constraints.

The most commonly used loss function and the one used by Karniadakis in PINNs is the mean squared error (MSE), which is an example of soft constraints and the method used in this thesis. We define the loss by:

$$l = l_{PDE} + l_{BC} = \frac{1}{N_\Omega}\sum_{i=1}^{N_\Omega} |f(\mathbf{x}_i) - \mathscr{L}(u_p)(\mathbf{x}_i)|^2 + \frac{1}{N_{\partial\Omega}}\sum_{i=1}^{N_{\partial\Omega}} |\mathscr{B}(u)(\mathbf{x}_i) - u_p(\mathbf{x}_i)|^2$$

Where u_p is the predicted solution and $\mathbf{x}_i \in \Omega$, which we call the collocation points. It is not immediately clear that this is a good soft constraint, or good loss function in general. There are certain properties that are desirable for a good loss function. These properties, together with an in-depth analysis on what makes a good loss function can be found [12].

3.2. Variations on the basic PINNs

As researchers have been looking at PINNs, forms different from the basic PINN have emerged. By changing different parts of the network, these new forms attempt to improve the basic PINN in one or multiple ways. In this section, we will quickly discuss some of the results found by researchers in the field of PINNs that served as inspiration for the initial part of this thesis.

3.2.1. DPINNs

Distributed PINNs, or DPINNs [13] for short, take inspiration from the finite volume method. Whereas the regular method creates a singular PINN to solve the PDE on the entire domain Ω, DPINNs partition Ω into non-overlapping, smaller cells. Now a separate PINN is created for every cell.

As the PINNs have to flow smoothly into one another, additional boundary restraints are created between the cells. For two neighboring cells Ω_i and Ω_j, a new term is added to the loss function:

$$l = l_{BC} + l_{PDE} + l_{IC} + l_{interface} \tag{3.1}$$

l_{BC} and l_{PDE} are familiar, but due care has to be taken with the fact that not all cells lie on the boundary of the domain, and hence not all cells have this term. While l_{IC} has not come up in this thesis, it is a term similar in form to the boundary condition term. It is for the initial condition, as the authors of [13] are solving time-dependent problems. As this thesis will not be solving any time-dependent problems, this term is not relevant to this thesis.

The interface term is the main topic of the paper. The interface term is itself the sum of three new terms:

$$l_{interface} = l_{C_x^0} + l_{C_t^0} + l_{C_x^1} \tag{3.2}$$

These terms are loss in continuity in x ($l_{C_x^0}$), loss in continuity in t ($l_{C_t^0}$) and loss in differentiability in x ($l_{C_x^1}$). Every term by itself is again similar in form to the boundary condition term. It is however of note that these terms are symmetrical across Ω_i and Ω_j.

The cells and this new term both help stabilize the network, and make the network more data-efficient than a single PINN. DPINNs were able to accurately solve multiple equation, including the advection equation and the Burgers' equation.

3.2.2. fPINNs

fPINNs [14] are a good example of a specialization within PINNs, as this variant has been specifically developed to solve fractional advection-diffusion equations very efficiently. One of the shortcomings of using automatic differentiation is that it does not work for fractional derivatives. As such, the authors of [14] include numerical discretization for the fractional operators that the automatic differentiation has trouble with. They were able to solve fractional advection-diffusion equations in 1, 2 and 3 dimensions with relative errors of $O(10^{-3})$ in the solution.

3.2.3. VPINNs

Whereas fPINNs were made for a very specific application, Variational PINNs (VPINNs) [15] are an attempt at improving the core PINN for general cases. By developing a Petrov-Galerkin version of the PINN, they improve both the speed and accuracy of the classical PINN. This is done by incorporating the variational form of the PDE into the loss function and by integrating this by parts, lowering the order of the differential operators. Results were shown in both analysis of very shallow networks and empirically for more complex networks.

3.3. Preliminary investigation into PINN extensions

The initial goal of this project was to develop a new method building on the work of Karniadakis. While the final product has strayed quite far from this, I felt it nevertheless right to include the preliminary results obtained from this direction of research in this thesis.

There are many existing libraries that can be used to implement artificial neural networks. One of the more popular libraries for Python, Tensorflow, is used throughout this thesis. In particular for this section of the thesis, DeepXDE [16] was used, which is a library for PINNs built on top of Tensorflow. Using this library comes highly recommended for anyone wishing to try out PINNs for a first time. By default, the program comes with the ability to solve multiple different PDEs and on multiple geometries. While the programming done for this chapter was done in DeepXDE, this is not the case for later chapters of this thesis. This is mainly because rewriting parts of DeepXDE would take longer than simply writing new code in this instance.

As stated before, this part was inspired by DPINNs. As it has been shown that it is possible to 'glue together' multiple PINNs, it would be interesting to see what else can be done with this notion. As such, we will be discussing a variation on DPINNs: Overlapping DPINNs

For overlapping DPINNs, we look at what might happen if we have $\Omega_i \cap \Omega_j \neq \emptyset$, for some $i \neq j$. In this case, we look at the cases in which all Ω_i have at least some part that is unique to that particular Ω_i. That is to say, we have $\bigcup\limits_{i\neq k} \Omega_i \subsetneq \Omega_k$ for all k in our index set.

In DPINNs, the boundary between two neighbouring sections created natural boundary conditions for both of the sections. In our case, we have a large overlapping area in which we can add new boundary conditions for our network. Initial tests showed that taking every point in $\Omega_i \cap \Omega_j$ to be a Dirichlet boundary condition for both networks decreased the performance of both networks. This effect occurred because the sheer number of boundary points kept the value in the overlapping region very stable. Unfortunately, the value it took on in this region was roughly equal to the average of the values created in the initial epoch, which were essentially random.

Two things were changed to fix this problem. Firstly, the networks were allowed to run separately for a while before the new boundary points were introduced. This allowed the networks to get at least somewhat close to the desired solution. Secondly, the collocation points in the overlapping area were weighted. A collocation point further away from $\Omega_i \setminus \Omega_j$ was weighted less by Ω_i. This allowed a smooth transition from weaker to stronger weighted boundary conditions. The result of one such collection of networks can be found in figure 3.1:

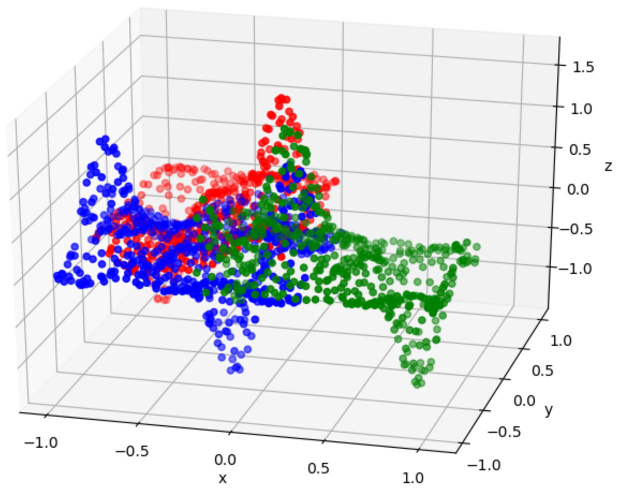


Figure 3.1: Three overlapping networks, with interactions between all three of them. The network is trying to predict the function $f(x, y) = \sin(2\pi x)\sin(\pi y)$, which it is able to do adequately.

4
PINNs using a global basis

All of the methods discussed in this thesis up until now work well for solving a single PDE. While this is certainly the ultimate goal for any numerical solver and a great first step into PINNs, they fail to use some of the unique advantages of neural networks.

4.1. Basic idea

Let us assume that we have a PINN N and a set of training points T_{train} taken from the set T of all possible inputs. Training the network on T_{train} gives us a set S_{train} of outputs, which is an approximation of the desired output. S_{train} lives in the solution space S (often also called the latent space) and defines the nonlinear map N which maps T onto S, with $N(T_{train}) = S_{train}$. While S is defined by T and the training of N, this unfortunately does not define the shape of S. As neural networks essentially interpolate between the training points when evaluating testing points, a poorly shaped S can lead to undesirable results.

This leads us to the following question: When is S 'nice' enough for a network to interpolate correctly? The answer to this can be found in Variational Autoencoders (VAE). Autoencoders are neural networks that take a set of data and encode it to reduce dimensionality, in order for it to take up less space. A second neural network can later be used as a decoder, to regain (most of) the original data. But what were to happen if one was to take a random point in the latent space created by the encoder and use this as input for the decoder network? Without a 'nice' enough latent space, the result would most likely be meaningless. With a 'nice' enough latent space however, one could potentially generate new, useful data [7].

In VAEs the 'niceness' of a latent space is called the *regularity* of the latent space. Going forward, this is alsot the term that will be used in this thesis. There is no real way to measure regularity, other than checking the results of the network(s) and see how well it performs. We do however know, which two properties a highly regular latent space should have.

First of all, we want similar solutions to be close to each other in S. A small change in the input should translate to a small change in the solution space. Without this property, an interpolation between two points could be influence by a third, unrelated point in the solution space. Secondly, we would like there to be as few as possible meaningless elements in our solution space (ideally none at all).

As S is defined by N, we can fulfill the first condition very easily by needing N to be a continuous map. The second condition is problematic however. In the case of PINNs, 'meaningless' elements in our solution space are outputs that are not a solution to the problem we are trying to solve. These mainly take the form of functions that either do not solve the PDE or the boundary conditions (or neither). Eliminating these elements from our latent space would require us to either implement hard constraints, or would require a priori knowledge of the form that a solution should take. The implementation of the hard constraints falls outside the scope of this project, but using a priori knowledge of the solution form is something that can be used.

The ability to interpolate inbetween the trained points is exactly why neural networks were designed in the first place. When testing a potentially very large number of similar problems, there can be situation in which it is much more expensive to implement and run a classic numerical method many times versus creating a PINN. It has to be noted however, that classic numerical methods give a guarantee of accuracy, whereas PINNs do not. In order to use this idea, we have to look at the basic PINN and change the way inputs and outputs are handled to accommodate this property.

Currently, most PINNs use the set of collocation points $(x_1, ..., x_n)$ as the input for the neural network, and choose $(u_p(x_1), ..., u_p(x_n))$ as the predicted solutions at these points as the output. While this seems like a logical choice to make for a PINN, it fails to fully take advantage of the interpolation capabilities of the network. It is possible to see this input as one of two ways.

The most logical way to interpret this input is as n training points, each consisting of the coordinates of the collocation point in question. This allows the user to input a single set of coordinates into the trained network to obtain the predicted solution for this point. While this does technically fulfill the stated properties, it does so in a manner that is not very useful.

There is not a lot of use for the ability to find the approximated solution to an untrained point. Collocation points are chosen in such a way that are uniformly or close to uniformly distributed amongst areas with similar importance. This is in order to make sure that any given untrained point is always relatively close to a trained point. But if the collocation points are almost uniformly distributed, then a simple (multi-)linear interpolation between two or more trained points should also be a relatively accurate approximation.

The other way one can interpret this input is as a single input vector (consisting of n sets of coordinates). Since we are working with a single training point, the idea behind the nice properties is lost here. If we were to add more training points, we would have to add multiple complete sets of collocation points. Should we successfully train a network using this input space, the result would be a network that is able to generate a solution for a generic set of points in our input space, which does not need to be ordered, but has to be of size n.

This is somewhat more useful than the other interpretation. This would allow us to input a set of n collocation points in any order and location, which can help if we quickly want to generate a solution with a certain number of collocation points in a specific area. Unfortunately, doing things this way gives us a different problem: the training itself. If we have a network with n collocation points and we wish every collocation point to train on m different places, we need n^m input points to train for every possibility. This exponential increase means this way is rather impractical.

4.2. Adapting PINNs to solve parametrized problems

While training on multiple different sets sounds promising, it is impractical with the current implementation. As such, it is necessary to change the structure of the neural network. Due to its simplicity, we take Laplace's equation in 1D as a starting point for this. We will generalize this later on once we have a solid idea of what the PINN should look like.

We wish to design a PINN that is capable of solving the following boundary value problem for various a, b:

$$\begin{cases} \frac{d^2u}{dx^2} = 0 \text{ on } (0,1) \\ u(0) = a, u(1) = b \end{cases} \tag{4.1}$$

If we look at the structure of the network, we can immediately see which parts are not in need of changing for this purpose. While the choice of optimizing algorithm, the number of nodes and layers, and the activation function are all important hyperparameters for a neural network, they are not relevant for changing what the network does. This leaves us with three things to change: the input tensor, the output tensor, and the loss function.

Since we wish for the network to work for any (a, b) we give it, (a, b) need to be included in the input by necessity. We also want the network not to rely on coordinates as input parameter, since this restricts us to what has been done previously. The exact solution to the problem is $u(x) = (b-a)x + a$, so the solution space contains exactly all line segments between a at $x = 0$ and b at $x = 1$ for all $a, b \in \mathbb{R}$. In other words, the output tensor needs to be such line segment or be able to somehow generate it. Lastly, this also gives us an idea for the loss function: since we know the solution for any training pair (a, b), we can easily compare it to the generated solution.

As we need to be able to generate multiple different functions as our output, we are working in a function space, and must choose a basis. Since we know that all possible solutions to the above Laplace's equation are linear functions, choosing the polynomial basis up to first degree is a natural choice. This way, we have no meaningless elements in our solution space. We will explore what effect the choice of basis has later. This gives us the following structure:

$$(a, b) \xrightarrow{N} (c_0, c_1) \rightarrow u_p(x) = c_1 x + c_0 \tag{4.2}$$

The only thing we need now is a loss function. What doesn't change from the collocation based method is that we need to evaluate the predicted solution both on the boundaries and on the interior. Since our solution is a polynomial, we can simply take the derivative twice instead of doing a numerical approximation. However, due to the simplicity of our problem and the choice of our solution space, the second derivative will automatically be 0. Therefore, we only need to find the loss on the boundaries:

$$l = l_{BC} + l_{PDE} = \frac{1}{2}[(u_p(0) - a)^2 + (u_p(1) - b)^2] + 0 = \frac{1}{2}[(c_0 - a)^2 + (c_1 + c_0 - b)^2] \tag{4.3}$$

Once we generalise the problem to Poisson's equation, there will be a loss term for the interior. Options and complexities that might arise are discussed in section 4.4.

4.3. Application to 1D Laplace

4.3.1. Initial Results

This section discusses the results for solving Laplace's equation using the basis method in one dimension. As this is a very simple problem with a linear solution, we would expect the network to have no trouble solving this problem.

A small network was created consisting of 4 layers, each with 50 neurons. A total of 121 training points were generated for the T_{train}, consisting of the points (m, n), for $m, n \in \{0, 1, 2, ..., 10\}$. Here the point (a, b) is the BVP with a Dirichlet boundary at $x = 0$ with value a and a Dirichlet boundary at $x = 1$ with value b. $loss_{PDE}$ was evaluated in the interior on 25 uniformly distributed points.

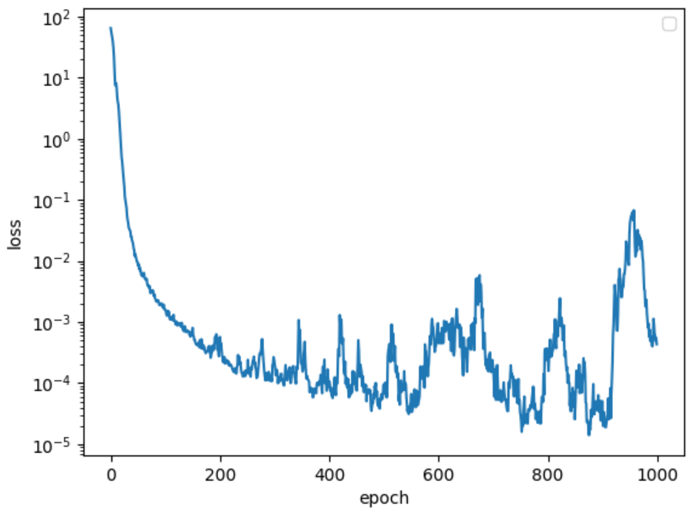


Figure 4.1: The loss of a network plotted for every training epoch. While the results may differ between training attempts, they all share the same general shape: a rapid decrease in loss until $O(10^{-3})$, followed by a gradual decrease with many fluctuations. Training the network took 4.2 seconds.

After training for 1000 epochs, the lowest loss in the training is $O(10^{-5})$, as can be seen in figure 4.1. Multiple training attempts have shown that the network can reach this level of loss very quickly, but stagnates after this. Once the network has a loss of around $O(10^{-4})$, it tends to either improve very slowly or make a mistake which sends the loss back up.

The reason that the loss suddenly increases by multiple orders of magnitude is due to the network overshooting a local minimum during optimization. While the Adam algorithm is great at diminishing this effect, it isn't perfect. The reason the network has so much trouble recovering from these spikes is due to batching. This effect can be seen very clearly in figure 4.2:

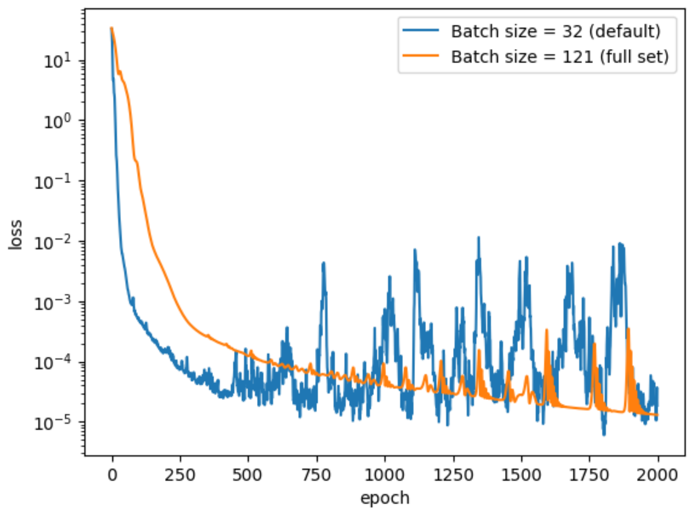


Figure 4.2: The loss of two networks plotted for every training epoch. Both networks were trained using the same training data, the only difference being the batch size. The unbatched network too 6.2 seconds to train, while the batched network too 9.0 seconds. Note that the training time is longer due to the higher number of epochs. This was chosen because the spikes do not appear prominently before this point in the full set.

While using the full set is more stable, experiments using these types of PINNs have consistently shown that the loss reached by an unbatched set is outperformed by a batched set when it comes to minimum loss reached and the number of epochs needed to reach this minimum. It is also of note that unbatched sets might run into memory issues.

Test points (a_{test}, b_{test}) were generated by randomly choosing $0 < a_{test}, b_{test} < 10$. The trained network is able to accurately predict the coefficients for the polynomial, with an MSE of $O(10^{-5})$ in the coefficient, which also gives an MSE of $O(10^{-5})$ in the solution.

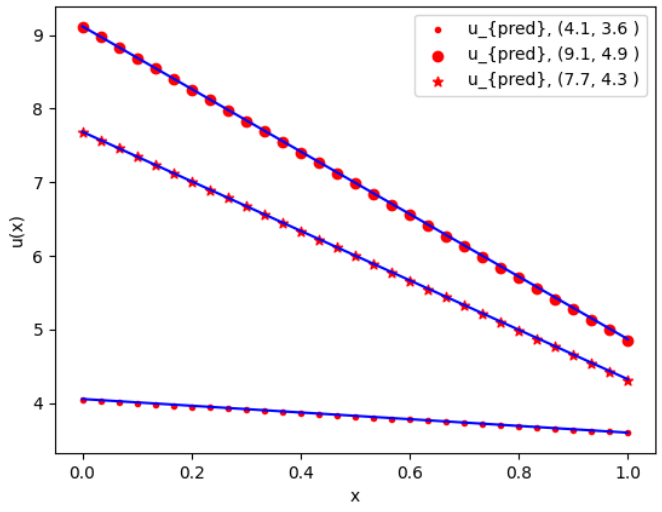


Figure 4.3: Three randomly generated test points, with the exact solution plotted on top of the predicted solution. Note that the points in the interior of (0,1) are only shown for visual clarity, as the predicted solutions only depend on the value of the boundary points.

We can conclude that this PINN is able to solve Laplace's equation with Dirichlet boundary conditions rather accurately. This doesn't come as a surprise, as we know from the exact solution that $c_1 = b - a$ and $c_0 = a$. This means the network has to simulate the following:

$$N(\begin{bmatrix} a \\ b \end{bmatrix}) = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

If we were to simplify the network to a single neuron and remove the activation function, the network can accurately predict this exact matrix. When this network is trained with the same training data as before, the resulting network is able to reach losses as small as $O(10^{-10})$ and extrapolate outside of the training data very accurately:

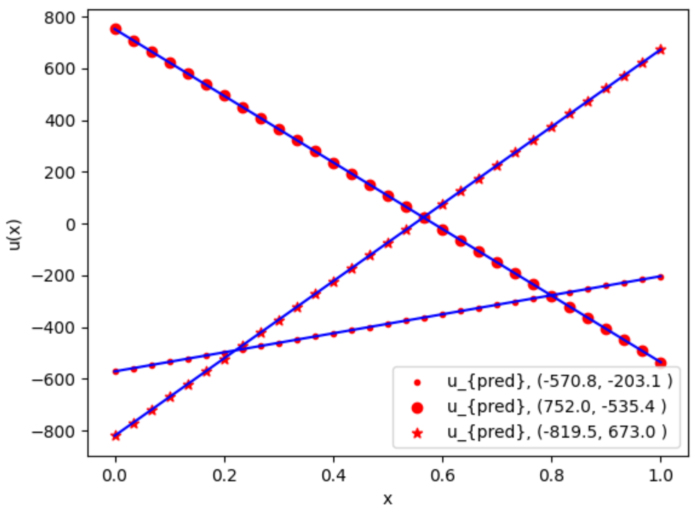


Figure 4.4: Three randomly generated test points, with the exact solution plotted on top of the predicted solution. Like before, interior points are there for visual clarity.

For figure 4.4, testing points (a_{test}, b_{test}) were generated by randomly choosing $-1000 < a_{test}, b_{test} < 1000$. Do note that while the results are very accurate, they aren't exactly useful since the non-linearity introduced by the activation function is what makes neural networks capable of interesting behaviour.

Due to the straight-forwardness of the problem and the solution, there was ample space to experiment with various hyperparameters. During the testing of this particular neural network, the size of the latent space and the choice of training points were varied.

4.3.2. Investigating the effect of varying the latent space size

As discussed in section 4.1, it is necessary to control our latent space to prevent our neural network from producing undesirable solutions. The latent space used in the previous network was $span\{1,x\}$, since our solutions are all linear. Once we get to non-linear solutions with Poisson's equation however, the latent space of our network increases by necessity. Choosing a large enough size for the solution space is necessary, but what if the size of the latent space isn't known? How will exactly will too large of a latent space affect the solution?

We will amend our predicted function (and hence loss function) to allow for a latent space of arbitrary size:

$$u_p = \sum_{i=0}^{N} c_i x^i \tag{4.4}$$

Since latent spaces that are too small are unable to produce the correct solution at all, the only spaces of interest are those too large. Since our predicted solution can now contain terms of order 2 and higher, the second derivative on the interior is no longer automatically 0. As such, we must amend our loss function to take this into account.

There are multiple ways to evaluate the interior loss. As we are working with polynomials, we can directly compute the derivatives needed to compare the PDE to the appropriate derivatives of the predicted solution. This would allow one to take more advanced and/or accurate methods of comparing two functions, such as function norms, or approximations of these norms. In this thesis, we simply opted to compare the two functions pointwise, on a predefined set of control points $\{x_1,...,x_n\}$. Unless mentioned otherwise, every such set of interior control points consists of 25 uniformly distributed points. This led to the following loss function:

$$l_{PDE} = \frac{1}{n}\sum_{i=1}^{n}(\frac{d^2 u_{pred}(x_i)}{dx^2} - 0)^2 \tag{4.5}$$

In figure 4.5, the results from different sizes of latent spaces can be found. Aside from the latent space size, the hyperparameters are exactly the same as the previous network.

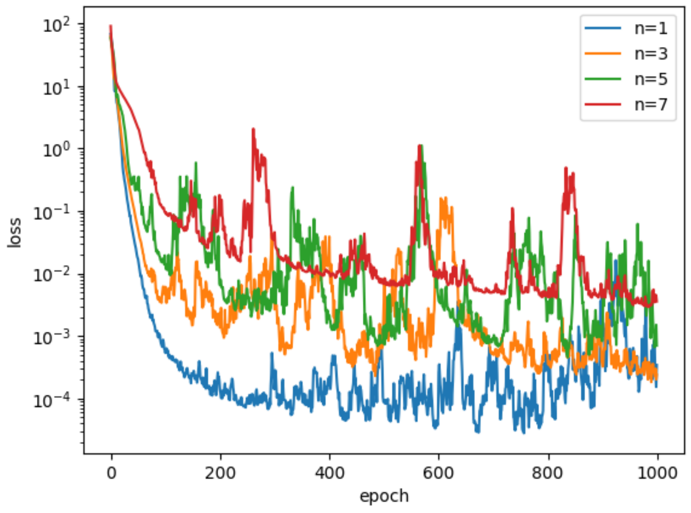


Figure 4.5: Multiple networks trained with different sizes for the latent space. Here, n is the order of the polynomial that is generated for the solution. Training time for all networks was around 6.1 seconds.

As can be clearly seen from the figure, the networks with larger latent spaces tend to be slower when compared to networks with smaller latent spaces. This doesn't come as a surprise; if the network tries to improve by adjusting a higher order term to nonzero, it can take a while to resolve this. We also see that the loss plateaus a bit higher than networks with only lower order terms. That being said, various trials with this setup have shown that the loss will improve over time, albeit a lot slower than networks with just lower order terms.

It has to be noted however that while the loss in is relatively low, the error in the coefficients isn't. The network has essentially found a solution which in which the higher order terms cancel each other out relatively well on the control points in the interior. As an example, the network with $n = 7$ produced the following solution for the randomly chosen training point $(2.1, 6.2)$:

$$u_p(x) = -0.26x^7 + 0.89x^6 - 1.17x^5 + 0.70x^4 - 0.20x^3 + 0.03x^2 + 4.09x + 2.11$$

When checked on the control points, this solution has an MSE of $O(10^{-4})$. Yet as can be seen quite clearly, this is not the correct solution. Further training does decrease this phenomenon, and we can conclude that the slower decrease in the loss is likely because of this. A different method of evaluating l_{PDE} could decrease this phenomenon, but this has not been tested in this thesis.

From this we can see that a larger than necessary latent space is not useful. Not only is it slower to train, but it also runs the risk of having a 'wrongly' trained network that compensates higher order terms with each other instead of setting them to 0.

4.3.3. Investigating the effect of different training point sets

In figure 4.4, we saw that the simplified network trained on very few points were able to accurately extrapolate quite a bit before the boundary conditions were no longer satisfied. While it is unlikely that a single network needs to work properly on such different orders of magnitude, it is nonetheless interesting to see how the choice of training points influences the accuracy of the solution. Ultimately, this can give us insight into how the training data should be chosen for any PINN based on this method.

While it is possible to increase the accuracy of a network by simply training on a larger training set, doing this also increases the training time. As such, optimizing the distribution of the existing training points is a more interesting way of increasing the accuracy of the network. The following four sets will be compared to one another:

$$T_1\colon\ \{0,1,2,3,4,5,6,7,8,9,10\}$$

$$T_2\colon\ \{0,\tfrac{1}{10},\tfrac{4}{10},\tfrac{9}{10},\tfrac{16}{10},\tfrac{25}{10},\tfrac{36}{10},\tfrac{49}{10},\tfrac{64}{10},\tfrac{81}{10},10\}$$

$$T_3\colon\ \{0,\tfrac{19}{10},\tfrac{36}{10},\tfrac{51}{10},\tfrac{64}{10},\tfrac{75}{10},\tfrac{84}{10},\tfrac{91}{10},\tfrac{96}{10},\tfrac{99}{10},10\}$$

$$T_4\colon\ \{0,\tfrac{1}{9},\tfrac{2}{9},\tfrac{3}{9},\tfrac{4}{9},\tfrac{5}{9},\tfrac{6}{9},\tfrac{7}{9},\tfrac{8}{9},1,10\}$$

T_1 is our previous set of training points. We have seen before that networks trained on this set were able to reach a loss of $O(10^{-5})$ with no problem. T_2 and T_3 were chosen as sets leaning more heavily in training point density towards the lower and upper bound of the training data set respectively. T_4 was chosen to see the effect a very poorly distributed test set might have.

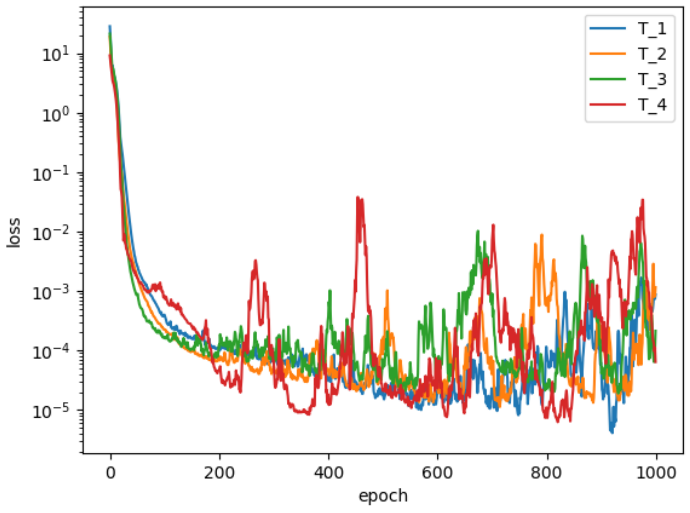


Figure 4.6: The MSE loss of the network after 1000 epochs, trained using the different training sets stated before. As can be seen, the loss stays mostly the same across the different training sets.

As can be seen in figure 4.6, the loss for all four training sets was very similar. In terms of the quality of the test sets, the first three networks also performed similarly. The last network did perform worse than the other three sets when it came to points further away from the cluster of training points, but only marginally so. As such, wherever possible, a uniform training density is preferred (and often easier to implement).

4.4. Application to 1D Poisson with parametrized source function

This section discusses solving Poisson's equation using the basis method in one dimension. While working on Laplace's equation was an easy first step, it does not give us enough data to say that this method can be useful. It could be possible that the network is unable to optimize the PDE loss, something we didn't have to worry much about previously (since solving the boundary conditions also solved the problem). Therefore implementing Poisson's equation is arguably a lot more useful than Laplace's equation.

Since we already have a working network that can solve Laplace's equation, we only have to implement the source function. We can go about doing this 2 ways. The first option is to hard-code the source term into the loss function. The second option is to include the source term in our input somehow. Both options have their potential advantages and disadvantages. The obvious advantage of the second option is that one could potentially have a network capable of solving many different equations. By necessity, this network would need to use a larger basis, and as we already saw, this has a negative effect on the accuracy of the network as a whole. We will be covering both options in this chapter.

4.4.1. Hard-coded polynomial source function

By amending our interior loss function, we can very easily add a source function f to our network:

$$l_{PDE} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d^2 u_{pred}(x_i)}{dx^2} - f\right)^2 \tag{4.6}$$

We start off by training and testing the network for a very simple source term, namely $f \equiv c \neq 0$. This means our solution will be of order 2. Once again, we train the network for 1000 epochs, which we can see in figure 4.7:

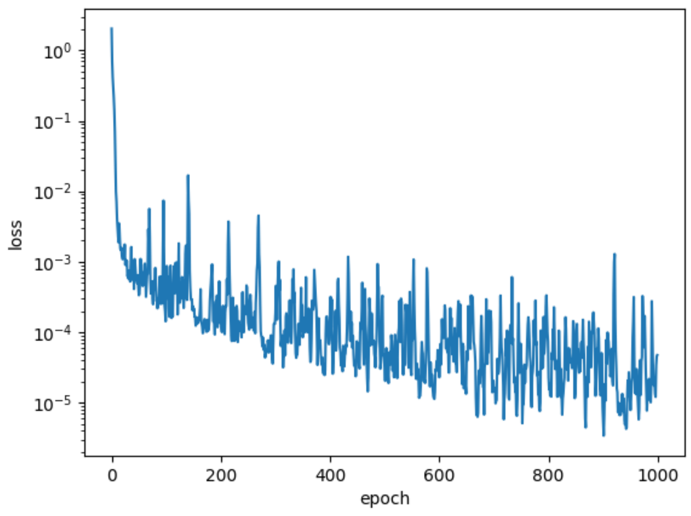


Figure 4.7: The loss of a network plotted for every training epoch. The chaotic nature of the plot is explained through the larger number of batches, which introduces more randomness in the optimization. Training the network took 6.5 seconds.

Much like the case in which $f \equiv 0$, the network is able to reach a loss of $O(10^{-5})$ quite comfortably. The loss has become more chaotic when compared to Laplace's equation. This is however a product of the number of batches. This network trained with the same number of batches as Laplace's equation (4 batches) gives a qualitatively similar loss plot.

This also gives us accurate results when testing, with an MSE of $O(10^{-4})$. This can be seen in figure 4.8.

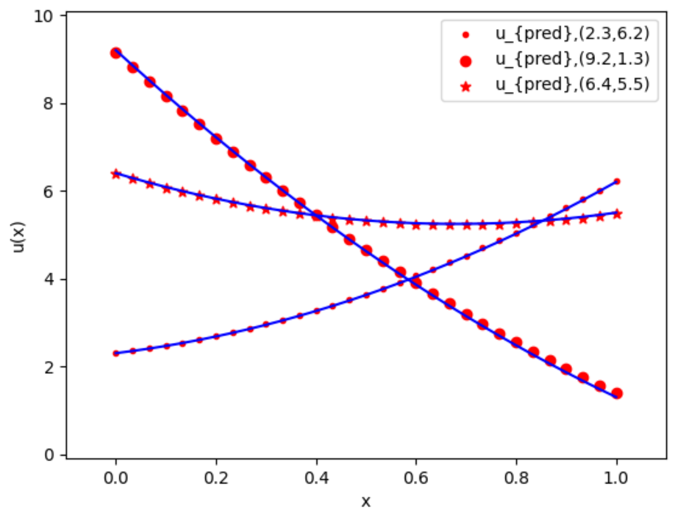


Figure 4.8: Three random test points. The source function was taken to be $f \equiv 2$.

We saw in section 4.3.2 that our network began to have training problems when the size of the solution space increases. As such, if f is taken to be a polynomial, the performance of the network decreases depending on the order of the polynomial, as this would require a larger latent space. We will take a closer look at both polynomial and non-polynomial f in the upcoming sections.

4.4.2. Polynomial source function as input parameter

As stated before at the start of this chapter, we can include the source function in the input and train the network on a family of source functions. Once again, we start simple by using constants as our source functions. We add variable k_0 to our input, which will be our source function.

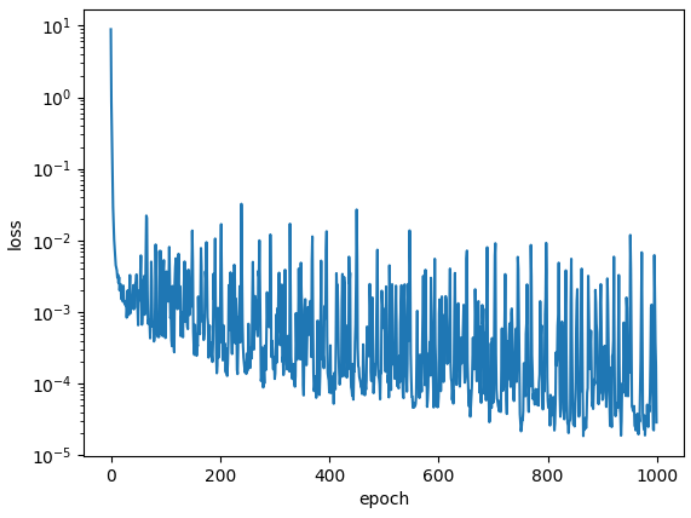


Figure 4.9: The loss of a network plotted for every training epoch. Again, the increased chaos is due to the increase of the number of batches. Training took 34 seconds.

The order of the source function can easily be increased by adding more variables to our input. We add $\{k_0,...,k_{n-2}\}$ instead of k_0, for the polynomial $\sum_i^{n-2} k_i x^i$. This means our solution space should be of order n, and all solutions are of this order or lower (when the last few terms are 0). A network was trained with $n = 3$ and the same parameters as in the previous cases; except k_1 was taken to be $\{1,2,...,50\}$, to allow the linear source function to be more visible in the plot.

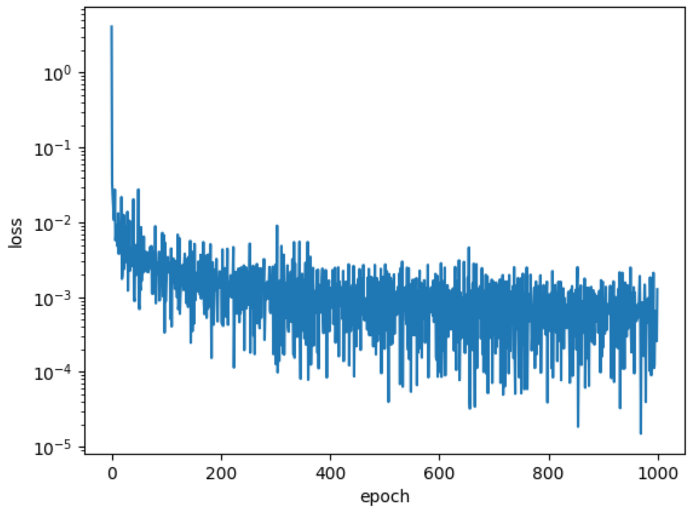


Figure 4.10: The loss of a network plotted for every training epoch. With an increase in solution space size, the accuracy is starting to drop. Training took 3 minutes due to the increased input tensor size.

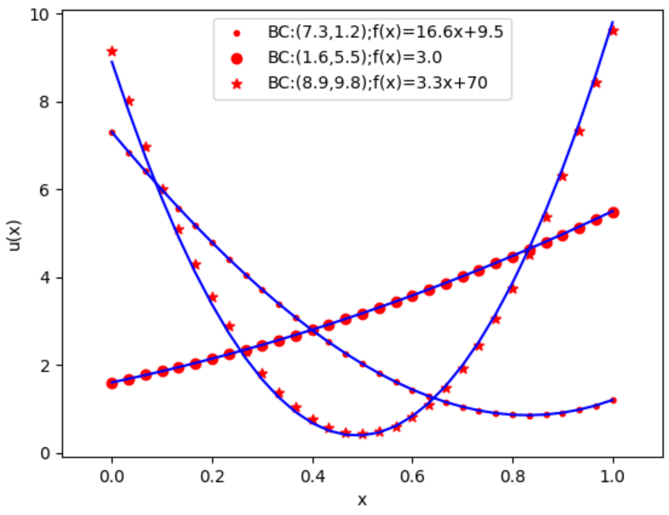


Figure 4.11: The first generated solution, denoted using small red dots, shows that the network can successfully solve the PDE with a linear source function. The second generated solution, denoted using the large red circles, shows that the network can still solve the PDE with a constant source function. The third generated solution, denoted using the red stars, shows that the network can extrapolate somewhat adequately.

As can be seen in figure 4.11, the network is able to accurately solve problems with a source function of up to degree 1. The network can also extrapolate; k_0 was only trained up to 10, but results are still rather accurate above this value. Only once the value is taken too far (above 50 in this case), does the network start to have problems solving the equation.

4.4.3. Non-polynomial source functions

While the polynomial source functions were convenient for testing purposes, it would be much more interesting to look at some source functions that give solutions that are not in the span of our basis. With polynomials, we can easily encode the source function into our input via the coefficients. We can do the same with any other continuous function. We theorize that a network with the same structure as before should be able to approximate any continuous function, since the polynomials lie dense in the continuous functions. This does mean that the network will have to be trained with a large enough latent space in order to approximate the source function correctly. As an example, a network was trained with $f(x) = k_0 \sin(2\pi x)$ as source functions:

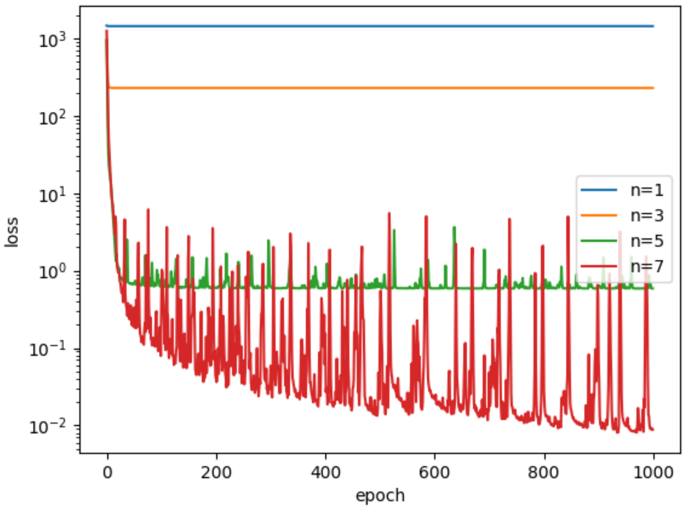


Figure 4.12: The loss for the same network, trained with a different size n for the order of our latent space. While $n = 9$ is used later, it has been left out of this graph for clarity reasons. It slightly improved over $n = 7$.

The need for a large enough latent space is confirmed in figure 4.12. The network barely improves beyond $n = 9$ for this case, the reason for which is two-fold. The first reason for this is that the network is inherently as least as complex as in section 4.3.2. The performance will therefore be worse than that particular case, which means that the performance for 1000 epochs will be unlikely to improve beyond another half an order of magnitude. Secondly, the polynomial approximation of the sine is equal to the Taylor series around 0. The Taylor series up to order 9 approximates $\sin(2\pi x)$ very well on $[0,1]$. The result can be seen in figure 4.13:

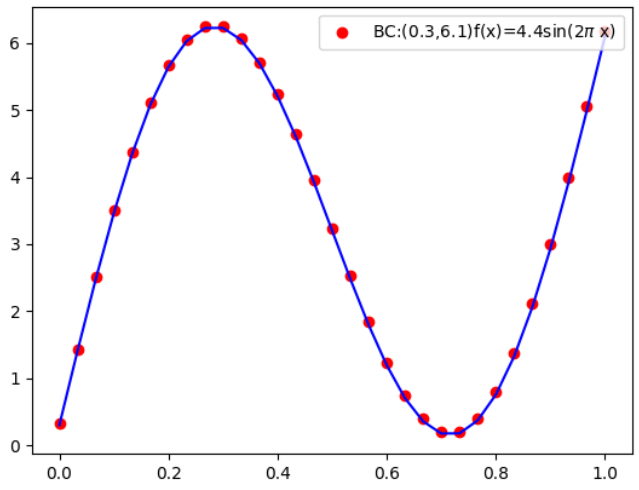


Figure 4.13: The output for a random test point (dots) plotted together with the analytical solution (line). The network is able to approximate the desired solution very well. The coefficients generated solution approach the coefficients of the Taylor series of the analytical solution.

We can conclude that a network generated this way and trained properly can solve most continuous functions, as long as the latent dimension is large enough to allow for an accurate approximation of the solution.

4.4.4. Investigating the influence of the chosen basis

Legendre polynomials

In all of our previous examples, a monomial basis was used for the solution space. More advanced polynomial bases, such as Legendre polynomials can also be used instead. Since the span of such bases is the same as the span of the monomials for every order, we can expect these bases to work without any trouble.

Using the same network architecture as before, the only thing that needs changing is the way the network calculates the function it is approximating. Using Rodrigues' formula [8] for the Legendre polynomials, we can calculate the derivatives necessary for the interior loss very easily:

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}(x^2-1)^n$$
$$\frac{d^2P_n(x)}{dx^2} = \frac{1}{2^n n!}\frac{d^{n+2}}{dx^{n+2}}(x^2-1)^n$$

Also of note is that the way the boundary loss is calculated also changes, as the Legendre polynomials of even order have a non-zero value at $x = 0$. We run the network for a range of constant source functions.

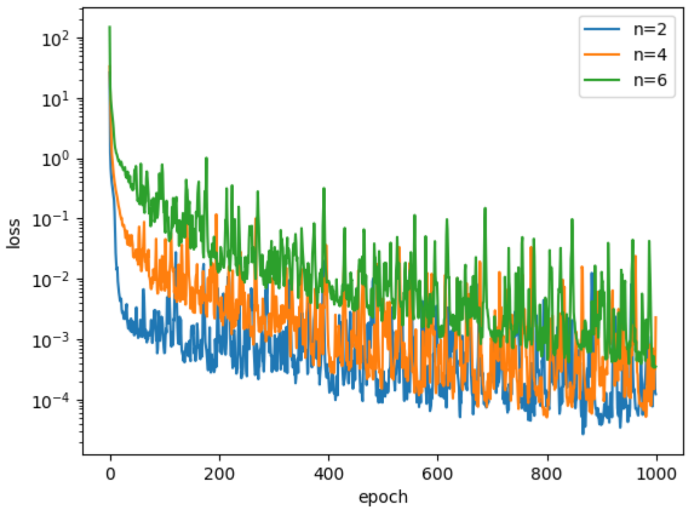


Figure 4.14: The loss of a network trained using Legendre polynomials, for different size latent space. As can be seen, networks with a larger latent space have a larger loss on average. Training took about 18 seconds for all latent spaces.

As can be seen in figure 4.14, the network performs similar to the monomial basis for $n = 2$, reaching a loss of 10^{-4} in 1000 epochs. Like before, increasing the size of the latent space does increase the loss that the network produces. However, when compared to figure 4.5, we can see that this increase is far lower than before. The reason for this is that the higher order terms of the Legendre polynomials have lower order terms. This means a network with this basis can (eventually) correct wrong higher order terms.

Fourier series

If we take a look at a non-polynomial basis, one of the most often used is of course the Fourier basis. This basis is obviously very useful when working with sinusoidal functions, but can be used to approximate any functions via the Fourier series. Unfortunately, there are two problems when it comes to the Fourier basis for general functions.

The first problem is one that follows from the network's difficulty in dealing with a large number of outputs. As an accurate approximation of a general function requires a relatively large number of terms, we would expect our network to run into the same problems as before in section 4.3.2.

The second problem involves the interaction between the PDE loss and the Fourier series. In order to compute l_{PDE}, we need to be able to differentiate u_p. This is problematic however, as one cannot simply differentiate an infinite sum term by term and expect it to work. As an example, consider the following Fourier series with period 2:

$$x = \frac{2}{\pi} \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} \sin(n\pi x) \tag{4.7}$$

If we were to differentiate this term by term, we would get:

$$2 \sum_{n=1}^{\infty} (-1)^{n+1} \cos(n\pi x) \tag{4.8}$$

Which is clearly not equal to 1. Even if we were to consider the partial sum (which we are working with in our networks), we could still not get close to 1. This isn't to say that any polynomial involving x as a term cannot be differentiated term by term; $x^3 - x$ can be correctly differentiated in this way. If f is a piecewise smooth function and continuous on $[-1,1]$, then the Fourier series of f can be differentiated term by term is $f(-1) = f(1)$.

While functions that do not satisfy this property are not excluded from being term by term differentiable by default, it does give us an idea of which type of functions could be problematic. Since we are working with Poisson's equation, we will also periodic differentiability, as the first derivative must also be term by term differentiable.

We will use the network to predict the coefficients of the sine-cosine form:

$$u_p(x) = a_0 + \sum_{n=1}^{N} a_n \cos(n\pi x) + b_n \sin(n\pi x) \tag{4.9}$$

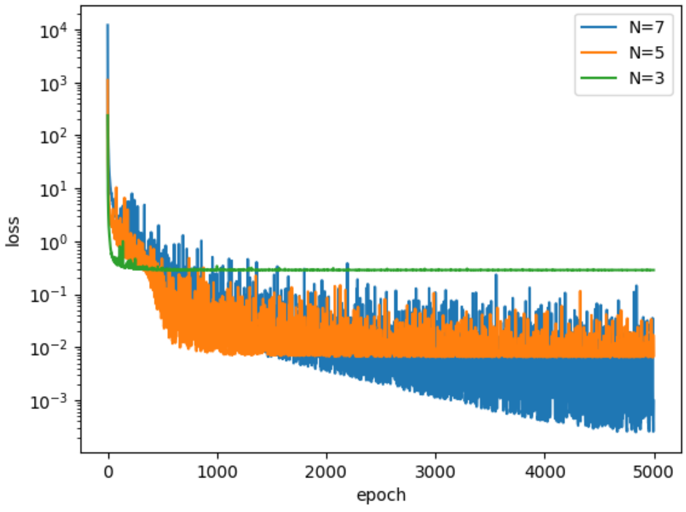


Figure 4.15: The loss of several networks trained using a Fourier basis with N terms. Compared to the polynomial networks, it is a lot slower to train, as it took 5000 epochs to reach a level which most polynomial networks could easily reach in hundreds. We see that the networks improve with more terms, which was expected. Training for every network took about 75 seconds.

A network was trained for constant source terms $\{0, 1, ..., 10\}$. As can be seen in figure 4.15, the accuracy of the network increases as more terms are added to the base. This does come at the cost of an increasingly chaotic loss as well as a slower overall decrease. While the polynomial based networks were able to reach a loss of $O(10^-4)$ very easily, the Fourier network took significantly more epochs, as well as more base terms.

While the network was able to solve Poisson's equation, it is important to note that this did not result in the Fourier series of whichever function the solution was trying to approximate. This is due to fact that the loss was only measured on $[0, 1]$, which makes the network to instead try and approximate a function which minimizes the loss on $[0, 1]$, and attempts to solve the periodic continuity and differentiability on $[-1, 0]$. This can be seen when we look at the generated solution for the PDE with solution $u(x) = x$:

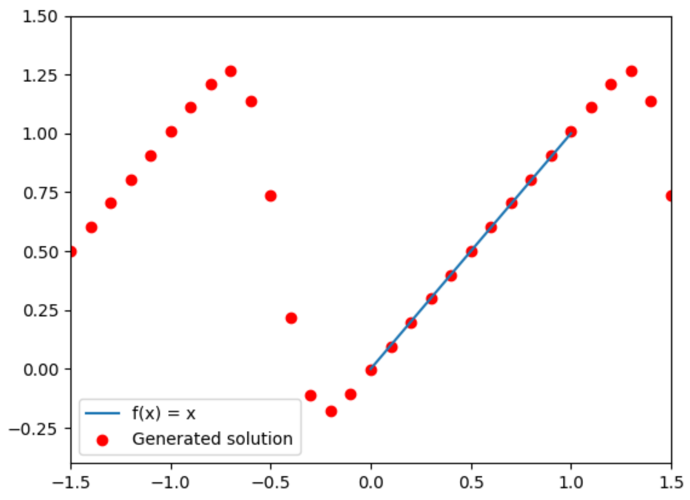


Figure 4.16: The generated solution found by the network trained on a Fourier basis up to the 7th terms. The network was only trained with a loss calculated on $[0, 1]$, where we have also placed the analytical solution it was trying to approximate. As can be seen, the it is able to approximate this solution adequately well.

An attempt was made to train a network that did calculate the loss over $[-1, 1]$ instead. The results can be seen in figure 4.17. When the network was only trained on PDE's with a differentiable periodic boundary ($u(x) \equiv c$ in this case), the network was able to correctly find the solutions, albeit very slowly. When the network was trained on linear solutions instead, it was unable to find the correct solutions.

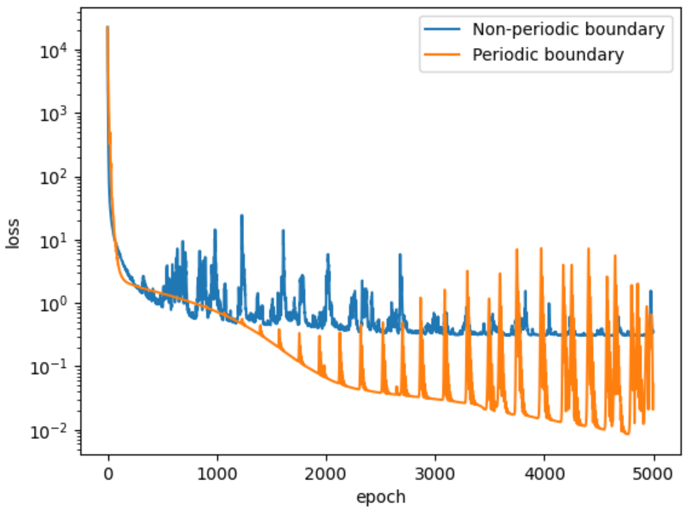


Figure 4.17: The loss of two networks trained to solve Laplace's equation using the Fourier series as a basis. Both were generated up to the 7th terms. The 'periodic boundary' network was trained for boundary conditions for which $u(-1) = u(1) = a$, with $a \in \{0, 1, ..., 10\}$. The 'non-periodic boundary' network was trained for boundary conditions $u(-1) = a, u(1) = b$, with $a, b \in \{0, 1, ..., 10\}$

4.5. Application to a higher dimensional problem

We previously saw that our network was able to solve Poisson's equation in one dimension for multiple different boundary conditions and source functions. If we try extending this into two (or more) dimensions, we run into some problems. This is primarily due to the form the solution to a generic 2D problem takes. If we look at the following BVP:

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \text{ on } (0,1) \times (0,1) \\ u(x,0) = 0;\ u(x,1) = 0 \\ u(0,y) = ay(y-1);\ u(1,y) = 0 \end{cases} \tag{4.10}$$

Solving this BVP analytically gives us the following solution:

$$\begin{aligned} u(x,y) &= 2a \sum_{n=1}^{\infty} \sin(n\pi y) \frac{\sinh(n\pi(x-1))}{\sinh(-n\pi)} \int_0^1 y(y-1)\sin(n\pi y)dy \\ &= \frac{2a}{\pi^3} \sum_{n=1}^{\infty} \sin(n\pi y) \frac{\sinh(n\pi(x-1))}{\sinh(-n\pi)} \frac{2\cos(n\pi)-2}{n^3} \end{aligned}$$

From this solution, we can immediately see a problem in using polynomials to approximate it. Since solution is an infinite sum, we would need to calculate multiple terms to approximate our solution to a decent degree. Unfortunately, as n increases, we see sine terms with higher and higher frequency; and as we saw before, we need a relatively high order polynomial to approximate these terms. As the order of polynomials needed increases, our accuracy decreases. This was reinforced by trials; networks with polynomials of order 9 had a loss of $O(10^3)$ and hence were unable to even get close to finding the solution.

An attempt was made to train a network using a manufactured solution instead of a generic PDE. The network was trained to solve the PDE corresponding to the following solution:

$$u(x,y) = x(1-x)y(1-y) + c \quad \text{on} \quad [0,1] \times [0,1] \tag{4.11}$$

When trained using the smallest latent space that contained this solution, the network was able to solve the equation and correctly predict the solution for various untrained values of c with a loss of $O(10^{-3})$. When trained using a larger latent space however, the network was not able to correctly predict the solution for untrained values of c, as can be seen in figure 4.18:

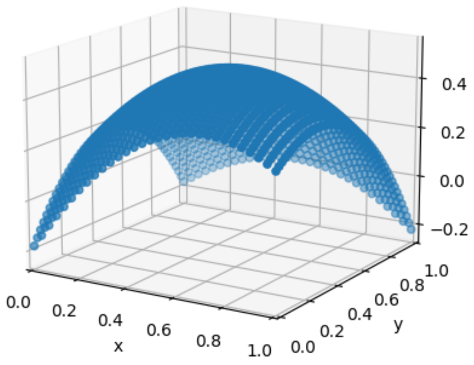


Figure 4.18: The generated solution found for untrained point $c = 4.6$. The network was trained on manufactured solution 4.11 for $c \in \{0, 1, 2, ..., 10\}$. A latent space with terms up to x^5 was used to train the network on.

We see that this network was unable to fulfill either the boundary conditions or the PDE on the interior for untrained values of c. While we had seen before that these types of networks have problems with latent spaces that are larger than necessary in 1 dimension, it seems that this problem is vastly exacerbated in higher dimensions.

Without an incredibly large network, solving 2D problems using a polynomial basis is concluded to be nearly impossible due to the nature of the solutions that need to be approximated. A network using a different basis could be better suited to solving these higher dimensional problems, mainly owing to the fact that these have also shown promise in 1 dimension. This is, however outside the scope of this project and could be an interesting subject for future research.

5
Spline-based PINNs

While the basis method initially showed a lot of promise regarding the ability to solve multiple different PDE's, it quickly became clear that the method began running into problems for higher dimensional problems. The complexity of the solutions for the higher dimensional problems caused the interpolation abilities that worked so well in the one dimensional case to fail. Fortunately, we can apply a similar technique to predict the coefficients of splines. In section 5.1, we look at how the algorithm will be adapted to splines instead of a polynomial basis. In sections 5.2 and 5.3, we show the results of this algorithm in one dimensional problems and two dimensional problems respectively. Finally, in section 5.4, we take a look at the possibility of a problem with a different geometry than the standard square.

5.1. Adapting PINNs to use splines

Adapting our existing algorithm to use B-splines instead of polynomials is not very difficult, as the only thing we let the network do is calculate coefficients. Since we can also define a B-spline by its coefficients, we can let the network predict these coefficients instead. For this to work, we need to change the number of coefficients generated by the network, as well as the loss function. In this new loss function, we calculate the spline generated by the output coefficients and compare it to our constraints. Since B-splines are differentiable, we also should have no problems calculating the derivatives for the interior constraints.

We can express any spline S as a linear combination of B-splines:

$$S_{n,\mathbf{t}}(x) = \sum_i c_i B_{i,n}(x) \tag{5.1}$$

Where n is the order of the spline, $\mathbf{t}$ is our set of m non-descending ordered knots, and $B_{i,n}$ are the B-splines. The coefficients c_i are the ones that the neural network will predict. We can see that this is incredibly similar to the polynomial-based PINNs in chapter 5.

Using Cox-De Boor's recursion algorithm[9], we can calculate the B-splines:

$$B_{i,0}(x) := \begin{cases} 1 & \text{if } t_i \leq x < t_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

$$B_{i,k}(x) := \frac{x - t_i}{t_{i+k} - t_i} B_{i,k-1}(x) + \frac{t_{i+k+1} - x}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x) \tag{5.3}$$

If $t_p < x < t_{p+1}$, then we know from the recursion formula that only B-splines with $i \in \{p-n, ..., p\}$ will be non-zero for the knot interval in which x lies. As we can leave out all other terms for this interval, we can reduce our spline to:

$$S_{n,\mathbf{t}}(x) = \sum_{i=p-n}^{p} c_i B_{i,n}(x) \tag{5.4}$$

As we have m knot intervals in $\mathbf{t}$, we have $m+n$ coefficients that need to be computed. In order for our program to run without too much hassle, we add a number of repeated knots both in front of and behind our list of knots. Including these repeated knots, our list of knots then becomes $\{t_0, ..., t_0, t_1, ..., t_{m-1}, t_m, ... t_m\}$. The number of repeated knots is n on both sides.

We can now simply write $S_{n,\mathbf{t}}$ in the form $S = cB$ (doing away with the subscripts for ease of notation), where c is the vector with all coefficients, and B is the matrix containing all entries of $B_{i,n}$. And with this, we have $u_p(x) = S(x)$.

As the spline is a known function with derivatives, we can simply calculate them, much like before when using polynomials. The derivative of one of the base functions is:

$$\frac{dB_{i,k}(x)}{dx} = \frac{k}{t_{i+k} - t_i} B_{i,k-1}(x) - \frac{k}{t_{i+k+1} - t_{i+1}} B_{i+1,k-1}(x) \tag{5.5}$$

Which can be applied twice to obtain the necessary form for Laplace's and Poisson's equations:

$$
\begin{aligned}
\frac{d^2B_{i,k}(x)}{dx^2} = \; & \frac{(k-1)k}{(t_{i+k}-t_i)(t_{i+k-1}-t_i)}B_{i,k-2}(x) \\
& - \frac{k}{t_{i+k}-t_{i+1}}\left(\frac{k-1}{t_{i+k}-t_i}+\frac{k-1}{t_{i+k+1}-t_{i+1}}\right)B_{i+1,k-2}(x) \\
& + \frac{(k-1)k}{(t_{i+k+1}-t_{i+1})(t_{i+k+1}-t_{i+2})}B_{i+2,k-2}(x)
\end{aligned}
\tag{5.6}
$$

5.2. Application to one dimensional cases

In this section, we take a look at the results of the spline based PINN in one dimension. We previously saw that the polynomial based approach was able to solve this case relatively well for low order polynomials, becoming progressively worse as the order increased.

5.2.1. Application to Laplace's equation

A network was generated with 4 hidden layers of 50 neurons each. As a relatively smooth solution was desired, a spline of order 3 was used. Knots were chosen on the points $\{0, 0.25, 0.5, 0.75, 1\}$, which meant the our network had to predict 7 coefficients. The first network generated this way was trained for Laplace's equation, using 121 training points. These training points consisted of the set $\{0, 1, 2, ..., 9, 10\}^2$, which were the values for the Dirichlet boundary conditions corresponding to $u(0)$ and $u(1)$. On the interior, 50 uniformly distributed control points were used to compute the interior loss.

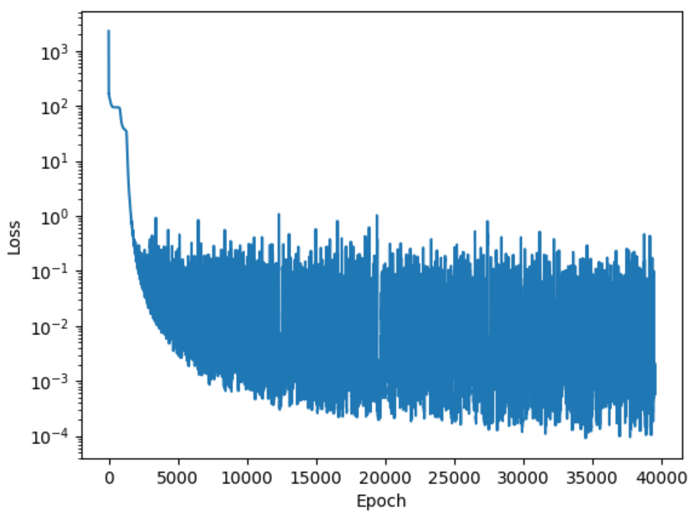


Figure 5.1: The loss of the network trained to solve Laplace's equation in 1 dimension using splines. The speed at which the network is trained is noticeably slower than when using global basis functions. Training this network took 61 seconds.

As can be seen in figure 5.1, the network needed to be trained for many more epochs compared to the polynomial based PINN. Not only that, but even after this many epochs, the loss was only $O(10^{-4})$. Furthermore, a weight had to be introduced for the boundary loss term. Without this weight, the network was unable to get a loss below $O(10^{-1})$. While not much time was spent on research into why this weight is necessary or what the optimal value is, quick tests have shown that a weight factor of 10^3 worked adequately.

The loss seemingly plateaus around $O(10^2)$, but quickly resumes a rapid decrease in loss. Only once it dips below $O(10^0)$ do we start to see a significant increase in the number of spikes. After going below $O(10^{-3})$, the loss is so chaotic that improvements are only made every few hundred epochs (if at all).

The network was then tested at three different testing points, the results of which can be seen in figure 5.2.

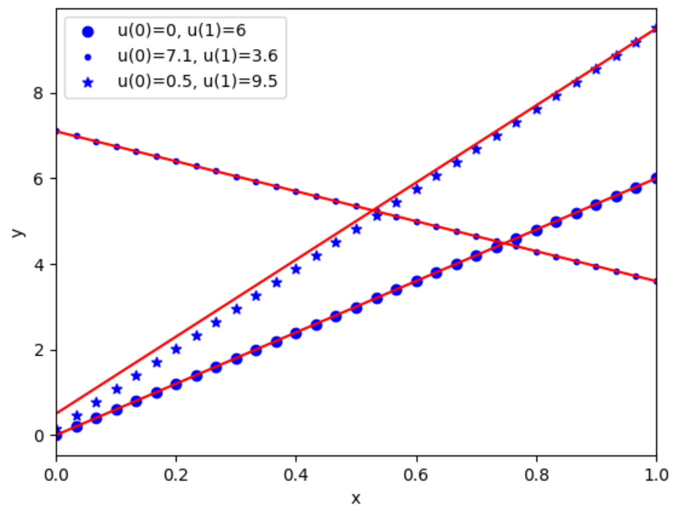


Figure 5.2: The generated solution for a trained input and 2 untrained inputs plotted together with the analytical solutions. Note that the input with one boundary condition close to zero is not being solved correctly.

As the network was trained to a loss of $O(10^{-4})$, we expect it to be able to predict the trained points well. As can be seen in figure 5.2, the network has no problem with the training points. If we take a look at a random testing point well in the interior of the training set (the small circles), the network is also able to accurately predict the solution. When we look at a testing point with one boundary value close to 0, we see that the network starts to make mistakes. We take a closer look at this phenomenon in section 5.2.2.

5.2.2. Error near boundaries

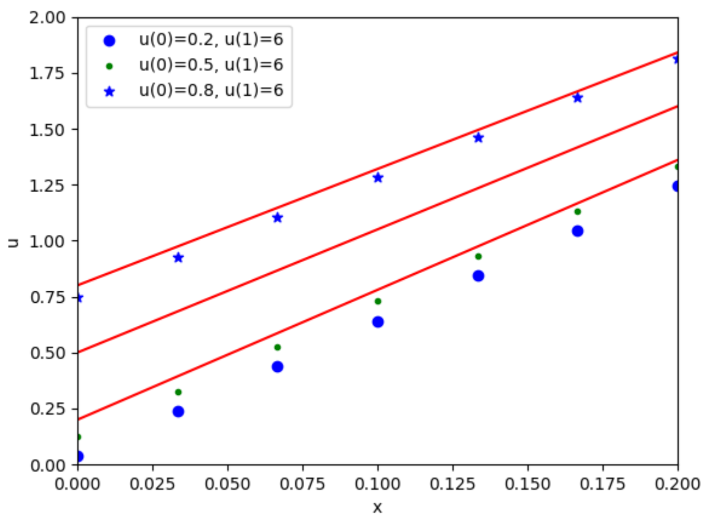


Figure 5.3: Three generated solutions for inputs close to 0. This used the same network as in the previous figure. Note that the solutions start to improve for boundary values further away from 0.

In figure 5.3, we can clearly see that the network is currently unable to correctly solve the problem for values close to the lower boundary of our training points. This behaviour has appeared in multiple networks, occurring on both the $x = 0$ and $x = 1$ boundary, with boundary values close to zero.

It is currently unknown why this behaviour persists. Multiple attempts were made to discover why this occurs, yet no conclusive results were found. However, several observations were made:

- The effect only occurs around boundary values close to 0, values close to the upper bound of the training points seem completely unaffected.
- The effect persists when the training set includes values below 0
- The effect can be somewhat mitigated by adding more training points specifically in the affected region

5.2.3. Application to Poisson's equation

The second network generated this way was created to solve Poisson's equation. The training points for the source function were taken to be $\{-10, -9, ..., -1, 0\}$. The loss and results can be seen in figures 5.4 and 5.5 respectively:

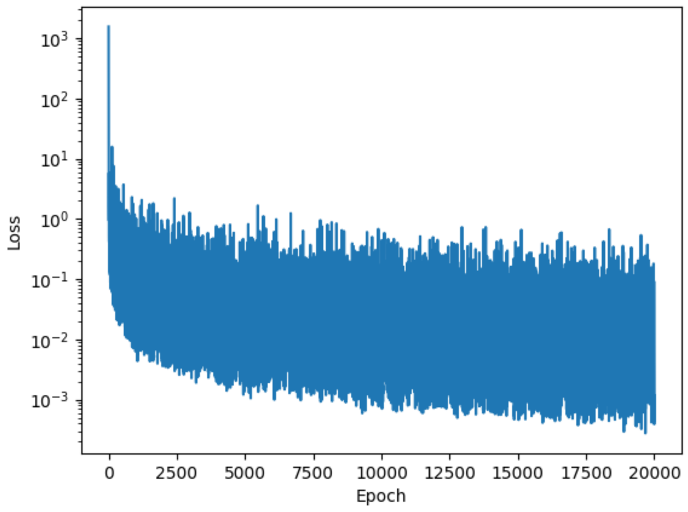


Figure 5.4: The loss of the network trained to solve Poisson's equation in 1 dimension using splines. The figure is similar to the figure for Laplace's equation, yet is an order of magnitude worse than before. Training this network took 119 seconds.

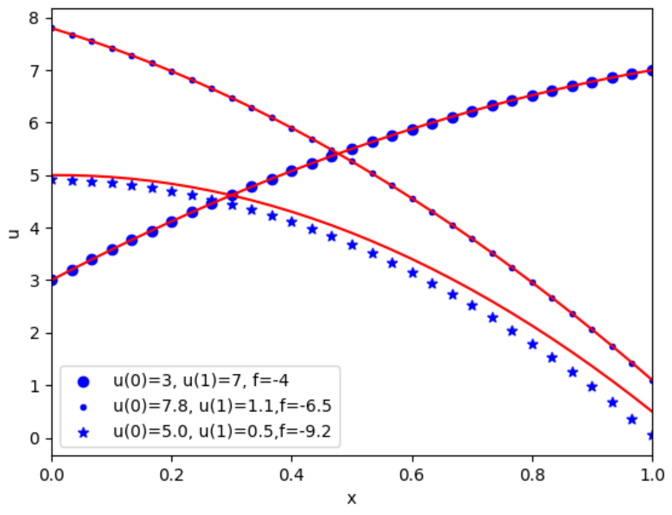


Figure 5.5: Three generated solutions; one trained and two untrained inputs. Once again the network is able to solve for the trained inputs and the untrained inputs away from 0, but has problems with untrained inputs with a boundary condition close to 0.

As can be seen in 5.4, the loss over time was qualitatively the same as in figure 5.1. As such, the results mirror the results found in the network trained on Laplace's equation. Training points and testing points away from boundary value 0 work perfectly fine, yet testing points with a boundary value close to 0 are once again problematic.

5.2.4. Results for non-polynomial source functions

Next, networks were trained to solve Poisson's equation with source function $-4\pi^2 k_0 \sin(2\pi x)$. As the discrepancy between the interior PDE values and boundary values grew larger ($O(10^2)$ compared to $O(10^0)$), the choice was made to use the relative loss for the interior:

$$l_{PDE} = \frac{1}{n}\sum_{i=1}^{n} \frac{\left(\frac{d^2 u_{pred}}{dx^2} - f\right)^2}{f} \tag{5.7}$$

This slightly increased the performance of the network (by less than one order of magnitude). The results can be found in figures 5.6 and 5.7:

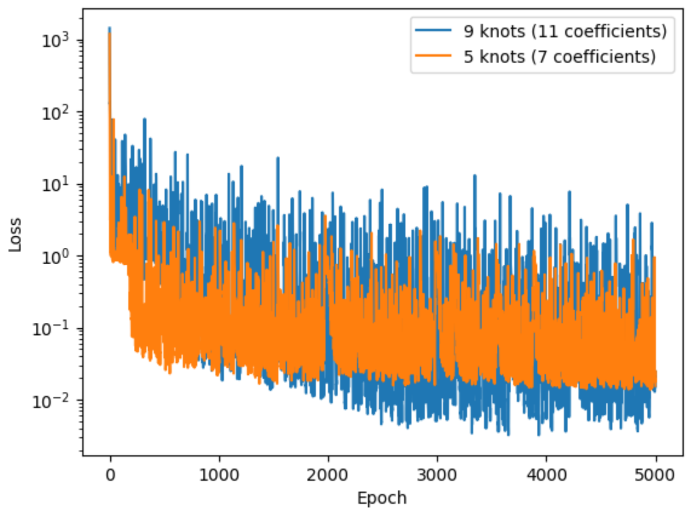


Figure 5.6: The loss for two identical networks save for the number of coefficients that needed to be predicted. As can be seen, the network with more knots is slower to train than the one with fewer; yet yields better results eventually. Training took 135 seconds for both networks.

As the initial results using 5 knots weren't great, a second attempt was made using 9 knots. This improved the loss by an order of magnitude, as can be seen in figure 5.6. This is also reflected in the quality of the solution, as the network with more knots is able to more accurately predict the solution to the PDE.

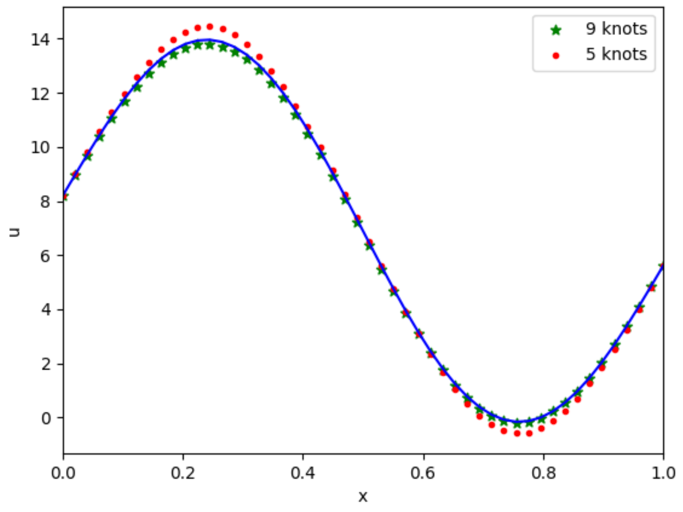


Figure 5.7: The generated solutions for the randomly generated input with $u(0) = 8.2, u(1) = 5.6$ and $f = -6.4\sin(x)$. The solutions generated by the two networks with different numbers of knots have been plotted together. As can be seen, increasing the number of knots improves the solution.

5.3. Two dimensional cases

In this section, we take a look at the results of the spline based PINN in the two dimensional cases. Where the polynomial based approach failed due to the lack of a global basis that could decently approximate our solution in finite time, we expect this to work better since we are creating localized smooth functions.

5.3.1. Adapting the method to two dimensions

Adapting this method to two (or more) dimensions was relatively easy. Since we can simply generate a spline in the x direction and one in the y direction and multiply them, we find that the formula for our spline is:

$$S(x,y) = \sum_{j=l-m}^{l} \sum_{i=k-n}^{k} c_{ij} B_{i,n}(x) B_{j,m}(y) \tag{5.8}$$

As we are once again working with an explicit formula for our generated solution, we can derive an explicit formula for the derivatives needed to calculate the PDE part of our loss.

5.3.2. Application to Poisson's equation

A network was generated with 5×5 uniformly spaced knots, which means the network had to predict 49 coefficients. A set of 11×11 uniform points were used to evaluate the generated solution and calculate the loss. Using this network to solve Poisson's equation gave us the following result:

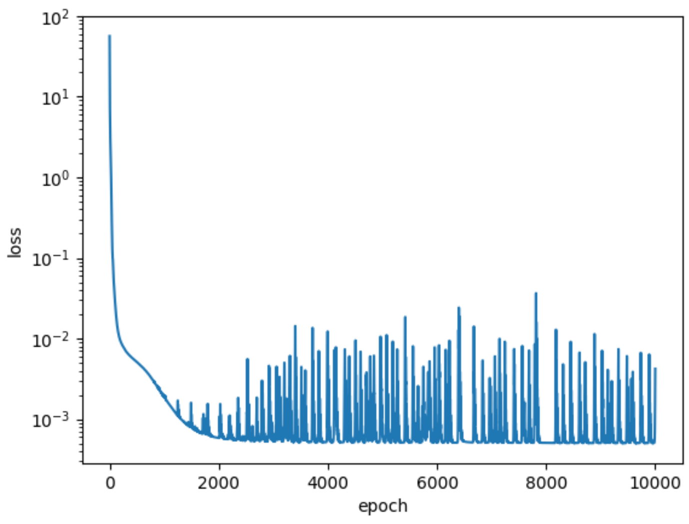


Figure 5.8: The loss of the network trained to solve Poisson's equation in 2 dimensions. Training this network took 532 seconds.

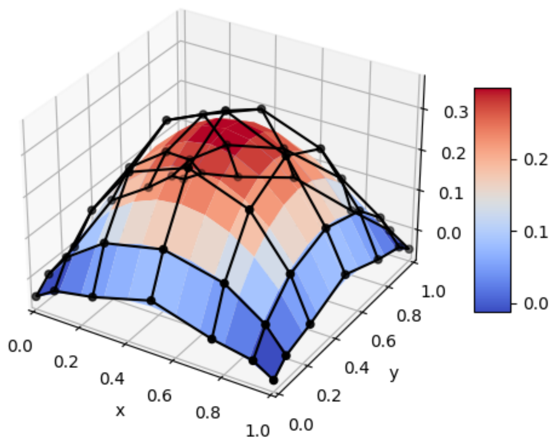


Figure 5.9: The generated solution for a trained point with boundary values 0. As can be seen, the network has trouble solving the boundary conditions even though the loss had plateaued.

While the network was able to adequately solve the differential equation in the interior, it was not able to fulfill the boundary conditions. This can be remedied by artificially increasing the weight that is given to the boundary loss term. It is possible to weigh the boundary loss in this way with a negligible effect on the training time. Weighing the boundary with a factor 10^3 in this way gave the following results:

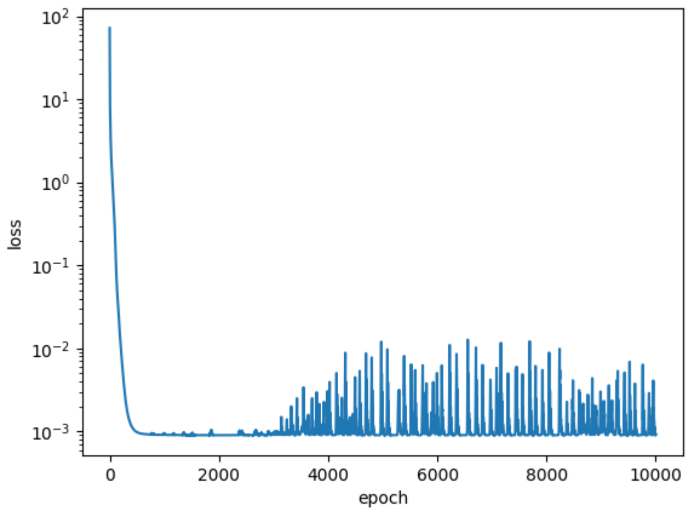


Figure 5.10: The loss of the network trained to solve Poisson's equation in 2 dimensions, with additional weight placed on the boundary condition loss. Training the network took 529 seconds.

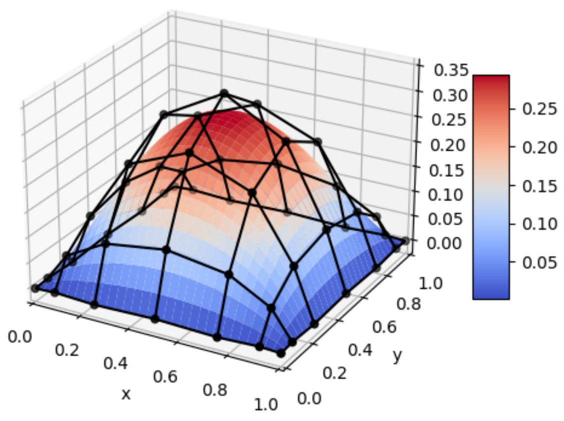


Figure 5.11: The generated result for an untrained point with boundary values 0 and source function $f \equiv -4.2$. As can be seen, the quality of the result has improved considerably over the unweighted version.

As can be seen, the network is able to satisfy the boundary condition and still solve the PDE in the interior. While the loss came only to a relatively large $O(10^{-2})$, it should be noted that this is easily improved by increasing the number of interior evaluation points, or by increasing the number of knots.

5.4. Incorporating a parametrized geometry map

Now that we have seen that it is possible to solve Poisson's equation in two dimensions using spline-based PINNs, we can take it a step further. We have seen that it is possible to encode both the boundary conditions and the source function into the input vector. Next we ask ourselves whether it is possible to create a working spline-based PINN with multiple geometries.

5.4.1. Encoding the geometry

Up until now we have been working on the unit square $[0,1]^2$. While we could hard-code a different domain, it would be a lot easier to do via a coordinate transformation. The example we are going to be using is a partial annulus, and we will be working in polar coordinates. We do this via the following map:

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} r \\ \theta \end{pmatrix} = \begin{pmatrix} (0.5x+0.5)\cos(\theta_{max}y) \\ (0.5x+0.5)\sin(\theta_{max}y) \end{pmatrix} \tag{5.9}$$

Which describes the partial annulus with radius $0.5 \leq r \leq 1$ and angle $0 \leq \theta \leq \theta_{max}$. By including θ_{max} in our input, we will be able to train a network for multiple different geometries. As our map is known and we are working in polar coordinates, we can rewrite our PDE in polar coordinates:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} \tag{5.10}$$

If our map is not known or not nice enough for a relatively clean PDE, automatic differentiation will have to be used to find the derivatives needed to compute the loss.

5.4.2. Application to Poisson's equation

A network was generated with 5×5 uniformly spaced knots and a set of 11×11 uniform points were used to evaluate the generated solution and calculate the loss. The network was trained for $\theta_{max} \in \{1,2,3\}$, the boundary value at $\theta = \theta_{max}$ was taken to be in $\{0,1,2,...,10\}$ and the source function was trained for values in $\{-10,-9,...,0\}$. The results are as follows:

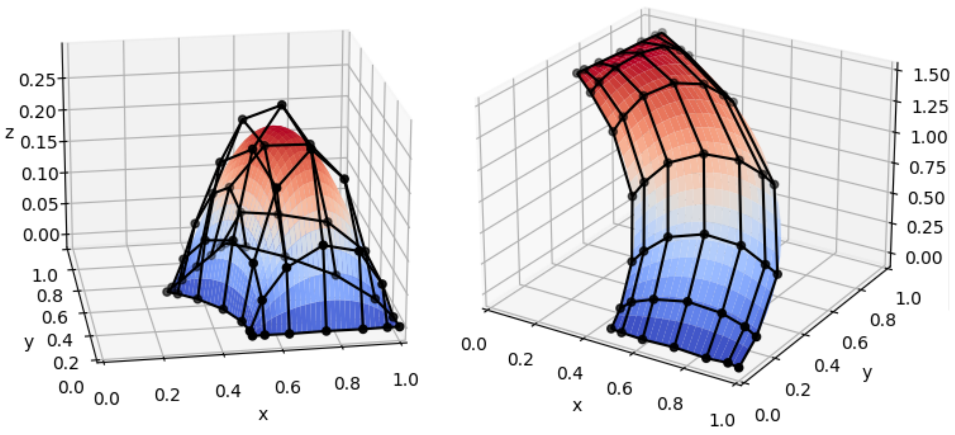


Figure 5.12: The figure on the left is the result of a trained point with boundary values at 0, $\theta_{max} = 1$ and $f \equiv -10$. The figure on the right is the results of a testing input with boundary value 1.4 at $\theta_{max} = 1.5$ and $f \equiv -10$. The boundary values at $r = 0.5$ and $r = 1$ were taken to increase linearly as a function of θ to ensure that the boundary values at the corners are the same.

As can be seen in the above figures, the network was successful in solving the boundary value problem for different input parameters. While the MSE loss was only $O(10^{-2})$, this can be improved by increasing the number of knots as we have seen before.

6 Conclusion

6.1. Summary

While the subject of solving partial differential equations is relatively young, it has seen many publications in recent years. What almost all of these publications have in common is that they are all based on the work of Karniadakis. This has led to the development of many different types of PINNs, most of them specialized towards improving the network for specific (families of) equations. And while the original intent of this thesis was to build on this concept, an alternative method of using PINNs was found and developed separate from Karniadakis' method. A short summary of some of these variant PINNs as well as the original direction of the research was seen in chapter 3.

In chapter 4 we introduced a new type of PINN based on global basis functions. Instead of letting the network directly predict the solution as output for a given set of coordinates, we let the network predict the coefficients for the basis functions. This allowed us to take greater advantage of one of the core principles of neural networks, this being the ability to interpolate between trained points. As the input for our networks did not rely upon coordinates, we were able to let a single network solve multiple PDEs for multiple boundary conditions. In this way, untrained points are essentially new BVPs that our network is able to solve.

We investigated multiple facets of these new PINNs empirically. We saw how the network was able to solve Poisson's equation for both polynomial and non-polynomial source terms and for different boundary value boundary conditions. We also saw how the training speed for the networks became increasingly slow as it had to predict more coefficients.

We've also seen how different bases can affect the training speed and overall quality of the predicted solutions. An attempt was made to lift this method to a 2-dimensional problem, but this was unsuccessful due to high number of coefficients required an the network's inability to deal with these properly.

A second type of PINN was created in chapter 5, based on splines instead of global basis functions. While these networks required a significantly longer training time and have an as of yet unexplainable error close to 0, they are capable of solving more complex problems than their global basis counterpart. This is most visible in higher dimensions, as the spline based PINNs can solve these equations with ease. The final and most promising feature of the spline based PINNs are that it was possible to encode a change in geometry as well, allowing a network to solve a PDE for multiple different boundary conditions on multiple geometries.

6.2. Future Research

While multiple new types of PINNs were introduced in this thesis and some facets of these explored, there are still many parts for which an in-depth analysis could be very beneficial.

6.2.1. Global basis PINNs

The global basis PINNs were successful at solving simple PDEs in one dimension, yet a quick foray into higher dimensional problems gave few answers. A good step for those who wish to continue with these types of PINNs could optimize the methods that were introduced or try and build a network that can solve higher dimensional problems by using a different basis or a more robust version of what was tried in this thesis.

6.2.2. Spline based PINNs

Whereas the global basis PINNs had difficulties with higher dimensional problems, the spline based PINNs had difficulties with the lower dimensional problems. Both the rather high training time and the phenomenon that occurred near $u = 0$ are great start for those looking to improve upon this method. The prime target for expanding on what was done in this thesis would be to continue developing the geometry-based PINNs. Implementing automatic differentiation would potentially allow a network to solve PDEs on an arbitrary set of geometries that can flow continuously over into one another.

Bibliography

[1] Paul Werbos (1994), *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*, New York: John Wiley & Sons, ISBN: 978-0-471-59897-6

[2] David Rumelhart, Geoffrey Hinton & Ronald Williams, *Learning representations by back-propagating errors*, Nature **323**, 533-536 (1986)

[3] Léon Bottou (1998), *Online Learning and Neural Networks*, Cambridge University Press, ISBN 978-0-521-65263-6

[4] Diederik Kingma & Jimmy Ba, *Adam: A method for Stochastic Optimization*, arXiv:1412.6980 (2014)

[5] C. de Koster & P. Habets, *On the Method of Lower and Upper Solutions in the Study of Boundary Value Problems*, Lecture Notes, [Online: accessed 12/12/2022] URL: `https://fa.ewi.tudelft.nl/ sweers/papers/CosterHabets.pdf`

[6] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer & Tom Goldstein, *Visualizing the loss landscape of neural nets*, Advances in Neural Information Processing Systems, 6389-6399 (2018)

[7] Diederik Kingma & Max Welling, *An Introduction to Variational Autoencoders*, arXiv:1906.02691 (2019)

[8] Olinde Rodrigues, *Mémoire sur l'attraction des spheroides*, Correspondence sur l'Ecole Polytechnique **3**, 361-385 (1815)

[9] C. de Boor, *Subroutine package for calculating with B-splines*, Techn. Rep. LA-4718-MS, Los Alamos Science Lab NM, p. 109, 121 (1971)

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

[11] Pablo Márquez-Neila, Mathieu Salzmann, Pascal Fua, *Imposing Hard Constraints on Deep Networks: Promises and Limitations*, arXiv: 1706.02025 (2017)

[12] Remco van der Meer, *Solving Partial Differential Equations with Neural Networks* (2019)

[13] Vikas Dwivedi, Nishant Parashar & Balaji Srinivasan, *Distributed physics informed neural network for data-efficient solution to partial differential equations*, arXiv: 1907.08967 (2019

[14] Guofei Pan, Lu Lu & George Karniadakis, *fPINNs: Fractional Physics-Informed Neural Networks*, arXiv: 1811.08967 (2018)

[15] Ehsan Kharazmi, Zhongqiang Zhang & George Karniadakis, *VPINNs: Variational Physics-Informed Neural Networks for Solving Partial Differential Equations*, arXiv:1912.00873 (2019)

[16] DeepXDE Library: deepxde.readthedocs.io/en/latest [Online: accessed 12/12/2022]

[17] Johannes Schmidt, *Well-posedness of Poisson problems*, Lecture Notes, [Online: accessed 12/12/2022] URL: `https://www.math.tu-berlin.de/fileadmin/i26_ng-schmidt/Vorlesungen/ScienComput_WS14_15/tutorial5.pdf`