Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science

Solving Partial Differential Equations with Neural Networks

In partial fulfillment of the requirements
for the degree of

Master of Science in
Applied Mathematics

Submitted by
Remco van der Meer

Supervised by
Prof. dr. ir. Cornelis Oosterlee & dr. Anastasia Borovykh

June 2019

Abstract

Recent works have shown that neural networks can be employed to solve partial differential equations, bringing rise to the framework of physics informed neural networks.The aim of this project is to gain a deeper understanding of these novel methods, and to use these insights to further improve them. We show that solving a partial differential equation can be formulated as a multi-objective optimization problem, and use this formulation to propose several modifications to existing methods. These modifications manifest as a scaling parameter, which can improve the accuracy by orders of magnitude for certain problems when it is chosen properly. We also propose heuristic methods to approximate the optimal scaling parameter, which can be used to eliminate the need to optimize this parameter. Our proposed methods are tested on a variety of partial differential equations and compared to existing methods. These partial differential equations include the Laplace equation, which we solve in up to four dimensions, the convection-diffucsion equation and the Helmholtz equation, all of which show that our proposed modifications lead to enhanced accuracy.

Aknowledgements

First of all, I would like to thank both of my supervisors Prof. dr. ir. Cornelis Oosterlee and dr. Anastasia Borovykh. They were both tremendously helpful, and were actively involved in searching for solutions to certain problems. When at times I thought my results were lacking, they convinced me about their positive aspects, and motivated me to keep moving forward. I thank Prof. dr. ir. Cornelis Oosterlee for his patience, and for giving incredibly detailed feedback on my work. His suggestions allowed me to tackle many problems efficiently. I would also like to express my gratitude to dr. Anastasia Borovykh for our fruitful discussions and for her help in searching for relevant literature. She provided me with many relevant articles, which gave me a much better understanding of the state of the art of neural networks and the relevance of this project in a broader context. Without the support of my supervisors, this project would not have been successfull.

My gratitude goes out to Casper Borgman BSc for his mental support and for the frequent days of working on both our theses together.

I also thank Yous van Halder for meaningful discussions and suggestions on how to improve the training process.

Finally, I thank my parents, friends and family for their unconditional support.

Contents

List of Figures

List of Tables

1 Introduction

Advances in computing power and rapid growth of available data in recent years have invigorated the field of machine learning and data science. In particular, deep learning methods have achieved exceptional results in a wide range of problems, including image recognition, natural language processing, genomics and reinforcement learning. The most notable architecture within deep learning is the deep neural network. Although the theory to train such models has been available since the early 60's, only recently has it become possible to train them on commonly available hardware [2].

The potential of deep learning methods stems from the approximation capabilities of deep neural networks, and the ability of training algorithms to find network parameters with which accurate approximations are achieved. Although empirical studies have already showcased that neural networks can achieve excellent results, theoretical results are slowly catching up as researchers are starting to understand the behaviour of deep neural networks during the training process [3, 4]. These results make deep neural networks interesting tools for the development of novel methods.

Typical applications of neural networks employ these networks to recover functions that are not directly available to the user. For example in image recognition, functions that map images to the corresponding classes are vastly complex and high dimensional functions, which convolutional neural networks are able to accurately recover. Other problems where one is interested in finding unknown functions are given by partial differential equations. For more difficult partial differential equations, analytical solutions are typically unavailable, and one has to resort to numerically approximating the solution. Many state of the art numerical solvers divide the geometry of the problem into a mesh of points or simple geometric elements, and proceed to compute an approximation of the solution on this set of points or using the basis functions on these finite elements. Neural networks can also be utilized as a kind of basis function. The study of [5] shows that there is a strong connection between finite element methods and neural networks with rectified linear units, as only two hidden layers are required for such neural networks to be able to express any output of a finite element method.

The paradigm of using neural networks as a kind of basis function to solve differential equations has been explored in several other studies. Particularly, the study of [6] motivates the use of neural networks in this context by stating that neural networks are closed form expressions, and thus provide information about the approximation anywhere in the relevant domain. This information includes derivatives of the solution, as neural networks with the appropriate activation functions are differentiable functions. Furthermore, they state that training neural networks is a highly parallellizable process. The authors of

[7] argue that deep neural networks may be the key to solving high dimensional partial differential equations since these methods are meshfree.

The authors of [8] consider similar methods in a different setting, as they enhance performance in the low data regime by adding information about the underlying equations. The methods these authors develop are perhaps even more general than the methods we mentioned earlier, and the study of [9] considers these methods with a focus on solving partial differential equations.

Although most works that utilize deep neural networks to solve partial differential equations are similar in nature, these methods are still not well understood. In particular, it is poorly understood why these methods work and what kind of partial differential equations can be solved with them. The main goal of this project is to gain a deeper understanding of these methods, and to use that knowledge to improve the methods. In particular, we aim to understand why these methods work and how they interact with different problems beloning to important classes of partial differential equations, including the Poisson equation, the convection-diffusion equation and the Helmholtz equation, the latter of which has proven to be notoriously difficult [10] to solve with finite difference methods.

This thesis is structured as follows; in chapter 2 we discuss the background theory that is required to understand the results. This chapter covers theory about neural networks and gives some definitions regarding partial differential equations that we will make use of. The chapter also discusses the partial differential equations we will solve in this thesis individually. Chapter 3 covers existing methods and proposes several modifications. These existing methods are reconstructed from scratch, such that we obtain a clearer picture of the underlying mathematics. Modifications to these methods include the introduction of a scaling parameter with which the importance of different loss functions may be tuned. The last part of this chapter revolves around this scaling parameter and introduces several different methods to handle its optimization. Chapter 4 treats the optimization of hyperparameters present in the proposed methods, and discusses some general properties of these methods. This chapter is practical in nature and serves to establish the algorithms in detail. Chapter 5 presents the results of the proposed methods applied to several different partial differential equations, using the optimal hyperparameters found in the previous chapter. Finally, in chapter 6 we conclude this thesis by giving a brief summary and discussion of the results, and we give some recommendations for further research.

2 Preliminaries

In this chapter we introduce the background theory that will be needed to understand the methods used in this thesis. This theory includes partial differential equations (PDEs), which are covered in section 2.1, and artificial neural networks, which are covered in section 2.2. Section 2.2 also treats some optimization algorithms that are commonly used to train neural networks.

2.1 Partial Differential Equations

In this section we discuss some general concepts regarding PDEs and we introduce the specific PDEs we will be solving in this thesis. We start by introducing some notation and give a mathematical definition of well-posedness. Section 2.1.1 introduces the Laplace equation. Section 2.1.2 covers the convection-diffusion equation. Finally, in section 2.1.3 we discuss the Helmholtz equation. We assume that the reader is familiar with PDEs.

Let us consider a general d-dimensional PDE for the function $u(x_1, \dots, x_d)$ on the domain Ω. In this work we only consider finite domains, such that $\Omega \subset \mathbb{R}^d$. To avoid having to write out all d variables that u depends on, we will use vector notation, and write

$$u(\boldsymbol{x}) \equiv u(x_1, \dots, x_d). \tag{1}$$

PDEs are defined through some relation between the partial derivatives of u with respect to x_i, that must be satisfied everywhere on Ω. In general, such equations can be formatted as

$$N_I\left(\boldsymbol{x}; u; \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_1}; \dots\right) = 0 \text{ on } \Omega, \tag{2}$$

where we separately included the partial derivatives of u to emphasize that N_I depends on those. We normally omit these arguments, as they can be inferred from u itself, and instead write

$$N_I\left(\boldsymbol{x}, u\right) = 0 \text{ on } \Omega. \tag{3}$$

In a similar fashion, the boundary conditions of a PDE can be represented as

$$N_B(\boldsymbol{x}, u) = 0 \text{ on } \partial\Omega. \tag{4}$$

When a PDE is defined through multiple relations, and thus describes a system of equations, then N_I and N_B become vector-valued. In this thesis we restrict ourselves to the scalar case. Note that N_I and N_B may be nonlinear functions.

For many PDEs it is important that the solution depends continuously on the parameters and data present in both of these operators. For example, if the boundary conditions were to change slightly, then it is important that the solution also only changes slightly.

This property is usually referred to as well-posedness. To properly define this property, let us modify the notation used in eqs. 2 and 4 and define the PDE as

$$\hat{N}_I(\boldsymbol{x}, u) = F(bmx) \text{ in } \Omega, \tag{5}$$

$$\hat{N}_B(bmx, u) = G(bmx) \text{ on } \partial\Omega. \tag{6}$$

Here, sources and other data present in the PDE are represented by F and G. For such PDEs, well-posedness is defined as follows.

Definition 2.1. A PDE of the form given in eqs. 5 and 6 is called well-posed if for all F, G there exists a unique solution, and if for every two sets of data F_1, G_1 and F_2, G_2, the corresponding solutions u_1 and u_2 satisfy

$$\|u_1 - u_2\| \leq C\{\|F_1 - F_2\| + \|G_1 - G_2\|\} \tag{7}$$

for some fixed constant $C \in \mathbb{R}$. We will refer to such constants as the Lipschitz constant of a PDE.

In the next sections, we cover several individual PDEs which we aim to solve in this thesis.

2.1.1 Laplace Equation

The Laplace equation is the simplest elliptic PDE, and is defined as

$$\nabla^2 u = 0 \text{ on } \Omega. \tag{8}$$

This equation is a special case of many more advanced elliptic PDEs, which makes the study of this equation invaluable. The Laplace equation can be used to describe the steady state solution of the heat equation, and is well-posed for Dirichlet, Neumann and Robin boundary conditions.

The Laplace equation can be solved analytically by separating the variables. In two dimensions, this can be performed by setting

$$u(x_1, x_2) = X(x_1)Y(x_2). \tag{9}$$

Then, we obtain ordinary differential equations for X and Y, which are given by

$$\frac{X''}{X} = -\frac{Y''}{Y} = \lambda. \tag{10}$$

For $\lambda > 0$, solutions to eq. 10 are given by

$$X(x_1) = Ae^{\sqrt{\lambda x_1}} + Be^{-\sqrt{\lambda x_1}}, \tag{11}$$

$$Y(x_2) = C\sin(\sqrt{\lambda x_2}) + D\cos(\sqrt{\lambda x_2}). \tag{12}$$

For $\lambda = 0$, the solutions are given by

$$X(x_1) = A + Bx_1 \tag{13}$$
$$Y(x_2) = C + Dx_2. \tag{14}$$

Finally, for $\lambda < 0$ we have the solutions

$$X(x_1) = C\sin(-\sqrt{-\lambda x_1}) + D\cos(\sqrt{-\lambda x_1}), \tag{15}$$
$$Y(x_2) = Ae^{\sqrt{-\lambda x_2}} + Be^{-\sqrt{-\lambda x_2}}. \tag{16}$$

Thus, the analytical solution is a superposition of

$$\begin{aligned}
u(x_1,x_2) &= 1, & u(x_1,x_2) &= x,\\
u(x_1,x_2) &= y, & u(x_1,x_2) &= xy,\\
u(x_1,x_2) &= e^{\omega y}\sin\omega x, & u(x_1,x_2) &= e^{-\omega y}\sin\omega x,\\
u(x_1,x_2) &= e^{\omega x}\sin\omega y, & u(x_1,x_2) &= e^{-\omega x}\sin\omega y,\\
u(x_1,x_2) &= e^{\omega y}\cos\omega x, & u(x_1,x_2) &= e^{-\omega y}\cos\omega x,\\
u(x_1,x_2) &= e^{\omega x}\cos\omega y, & u(x_1,x_2) &= e^{-\omega x}\cos\omega y.
\end{aligned} \tag{17}$$

These functions are the eigenfunctions of the Laplace equations. For rectangular domains with Dirichlet boundary conditions, it is convenient to exploit linearity by splitting the problem into four subproblems, each having exactly one inhomogeneous boundary condition of the original problem. The solution can then be obtained by adding the four solutions together. Because the subproblems only have a single inhomogeneous boundary condition, their solutions can only be comprised of specific eigenfunctions. These may be found by constructing a Fourier series of this inhomogeneous boundary condition. We will not detail this process here.

Solving the Laplace equation numerically is straightforward, as central difference schemes are stable and generally yield adequate results.

2.1.2 Convection-Diffusion Equation

The convection-diffusion equation is given by

$$\frac{\partial u}{\partial t} = \nabla\cdot(D\nabla u) - \nabla\cdot(\boldsymbol{v}u) \tag{18}$$

and describes the time evolution of a system where physical quantities such as temperature are subject to both convection and diffusion. This equation combines both hyperbolic and parabolic parts.

The diffusion constant D and the flow velocity v are often assumed to be constant, which simplifies the equation to

$$\frac{\partial u}{\partial t} = D\nabla^2 u - \boldsymbol{v}\cdot\nabla u. \tag{19}$$

This is the variant of the equation we will consider in this thesis. Using our previous notation, it can be defined by eq. 2 by setting

$$N_I(\boldsymbol{x}, u) \equiv \frac{\partial u}{\partial t} - D\nabla^2 u + \boldsymbol{v} \cdot \nabla u. \tag{20}$$

In one dimension, eigenfunctions of this equation take the form

$$u(x, t) = e^{-\omega^2 D t} \sin(\omega(x - vt)). \tag{21}$$

Just like for the Laplace equation, a Fourier series can be used to find the analytical solution.

The Laplace equation is related to the stationary convection-diffusion equation. Under the previous simplifications, this equation is given by

$$D\nabla^2 u - \boldsymbol{v} \cdot \nabla u = 0, \tag{22}$$

with $D > 0$. The Laplace equation is a special case of this equation, as it can be recovered by choosing $\boldsymbol{v} = 0$.

Solving the convection-diffusion equation numerically is not as straightforward as solving the Laplace equation. Using a central difference method results in numerical dispersion, and thus requires very small time steps to ensure stability. A more stable method can be obtained by using a upwind differencing method. However, upwind methods introduce numerical diffusion, which results in a lower order of accuracy.

For very small diffusion coefficients $D \ll v$, the convection-diffusion equation describes a singularly perturbed problem. This means that the solution cannot be approximated by setting $D = 0$. Such small values of D cause the resulting analytical solution to have a boundary layer, which makes these problems extremely challenging to solve.

2.1.3 Helmholtz Equation

Another generalization of the Laplace equation is given by the Helmholtz equation, which is defined as

$$(\nabla^2 + k^2)u = F. \tag{23}$$

The Helmholtz equation arises when using separation of variables to solve the time-dependent wave equation,

$$\nabla^2 u - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = 0. \tag{24}$$

By demanding that the solution is of the form

$$u(\boldsymbol{x}, t) = X(\boldsymbol{x})T(t), \tag{25}$$

one obtains the relation between X and T given by

$$\frac{\nabla^2 X}{X} = \frac{1}{c^2 T}\frac{\partial^2 T}{\partial t^2} = -k^2, \tag{26}$$

where the minus sign on the right hand size was chosen for convenience. The resulting Helmholtz equation thus describes standing waves, of which the time evolution is described by $T(t)$.

Unlike the Laplace equation, the homogeneous Helmholtz equation with homogeneous Dirichlet boundary conditions may have nontrivial eigenfunctions. On the rectangular domain $[0,1]\times[0,1]$, these are given by

$$u(x,y) = \sin(k_x \pi x)\sin(k_y \pi y), \tag{27}$$

with $k_x, k_y \in \mathbb{N}$ and k satisfying

$$k_x^2\pi^2 + k_y\pi^2 = k^2. \tag{28}$$

When k is chosen such that no nontrivial eigenfunctions exist, the Helmholtz equation with Dirichlet boundary conditions is well-posed.

The Helmholtz equation is a notoriously difficult equation to solve numerically, because the matrix that follows from central difference schemes is extremely ill-conditioned and indefinite [10].

2.2 Artificial Neural Networks

In this section we introduce artificial neural networks. We first give a brief conceptual overview, after which we formally define the components of neural networks in section 2.2.1. Section 2.2.2 covers some important properties of neural networks. Section 2.2.3 treats the general concept of training neural networks. Finally, sections 2.2.4, 2.2.5 and 2.2.6 cover three popular training algorithms.

Artificial neural networks are mathematical constructs that are structured in a way that is similar to animal brains. They have been a critical component in many recent advances in machine learning. Notable examples of such advances are facial recognition, computer vision and language translation. Neural networks are expected to remain important in the near future. Nvidia recently built a new supercomputer which is meant to train neural networks involved in autonomous driving.

Neural networks are conceptually simple; they contain neurons and edges between neurons. Just like animal brains, neural networks are able to *learn* complex patterns by tuning the strength of connections between certain neurons. In essence, neural networks are large parametrized functions with some very useful properties.

2.2.1 Architecture

Neural networks define complicated functions that are parametrized by the weights of the connections between neurons, and the biases of these neurons. Let us denote these weights by W and the biases by b, and the joint set containing all parameters by $\theta = \{W, b\}$. Each of the neurons computes the weighted average of their input neurons and the corresponding weights, adds its bias to the result, and feeds the resulting value through a nonlinear activation function, which we label σ. We will discuss these activation functions in more detail later. Other than these components, all neural networks also have an input layer and an output layer. Let us denote the inputs by the vector $\boldsymbol{x}$. With this notation, a neural network defines a function $f(\boldsymbol{x}, \theta)$, where f has the same dimension as the number of neurons in the output layer, and may thus be vector-valued.

The simplest configuration of deep neural networks is the multilayer perceptron. For such neural networks, all neurons are arranged in ordered layers, and each neuron is connected to all neurons in neighbouring layers. Under this configuration, it makes sense to label the outputs of the parameters belonging to each layer separately. Using vector notation, we denote the output of the i-th layer by $\boldsymbol{f}_i$ and the biases by $\boldsymbol{b}_i$. We denote the weights connecting the $(i-1)$-th layer to the i-th layer by the matrix W_i. The input layer is assigned the index 0, and the output layer is assigned the index k. Then, the output of the i-th layer is given by

$$\boldsymbol{f}_i(\boldsymbol{x}) = \begin{cases} \boldsymbol{x}, & i = 0, \\ \boldsymbol{\sigma}(W_i \boldsymbol{f}_{i-1}(\boldsymbol{x}) + \boldsymbol{b}_i), & i = 1, \ldots, k-1, \\ W_i \boldsymbol{f}_{i-1}(\boldsymbol{x}) + \boldsymbol{b}_i, & i = k, \end{cases} \tag{29}$$

where $\boldsymbol{\sigma}$ is the vector containing the outputs of the activation function σ applied element-wise to the argument vector. The output of the network is thus given by $f(\boldsymbol{x}, \theta) = f_k(\boldsymbol{x})$. It is of critical importance that these activation functions are nonlinear, since otherwise neural networks would only be able to describe linear functions, regardless of the configuration. Common choices of activations functions are sigmoid functions such as the logistic function,

$$\sigma(x) = \frac{1}{1 + e^x}, \tag{30}$$

or the hyperbolic tangent. The rectified linear unit (ReLU) is another popular choice, most notably in classification problems, and is defined as

$$\sigma(x) = \begin{cases} 0 \text{ for } x < 0, \\ x \text{ otherwise.} \end{cases} \tag{31}$$

Although these are the most popular choices, many other functions have been successfully used as activation functions, including non monotonic functions such as gaussians and sinusoids. These nested nonlinear functions make multilayer perceptrons capable of learning highly nonlinear patterns. Note that neural networks are closed-form expressions. This means that they inherit the differentiability of the used activation functions.

There exist more advanced configurations of neural networks than the multilayer perceptron. A common modification to basic feedforward neural networks is the addition of skip connections, which connect non-neighbouring layers. The study of [11] shows that the addition of such connections can make it much easier to train neural networks.

Another notable configuration is the recurrent neural network. Such networks are commonly used to process sequences of data, such as letters in text. Recurrent neural networks maintain a hidden state which they feed themselves as an input whenever a new element of the sequence is processed. This essentially gives recurrent neural networks memory, significantly boosting their performance in many applications.

Other common variants of multilayer perceptrons are convolutional neural networks, which are well suited for processing images, or similarly structured data. Unlike multilayer perceptrons, neighbouring layers of convolutional neural networks are not fully connected. Instead, neurons are only connected to other neurons in specific regions of previous layers. This causes neurons to only perceive small parts of the data, allowing the network to exploit the strong correlations between neighbouring pixels. Stacking many layers like these allows convolutional neural networks to learn a hierarchy of features that eventually covers the entire image.

The last variant of multilayer perceptrons we discuss is the autoencoder. Autoencoders are neural networks with the characteristic that their layers get smaller in the middle of the network. By training these networks to approximate the identity operator, one obtains a neural network that can compress and uncompress data; the first half of the neural network compresses the data from the size of the input layer to the size of the middle layer, while the second half reverses this process. Although it is clearly impossible to compress arbitrary data without losing information, neural networks are often able to exploit patterns in the underlying data to perform this compression.

In this thesis, we will only use multilayer perceptrons to keep the results general.

2.2.2 Universal Approximation

As mentioned, nonlinear activation functions are a crucial component of neural networks, as these nonlinearities allow neural networks to learn complex patterns. This property was more formally explored in the study of [12], where the authors show that feedforward neural networks are able to approximate any function arbitrarily well given sufficient neurons and when using certain activation functions. This property makes neural networks very powerful tools in theory.

It has also been shown that a similar property holds for the derivatives of functions. In the study of [13], the authors show that when the activation function is sufficiently differentiable, then neural networks can simultaneously approximate the function and its derivatives arbitrarily well. This property is particularly important in the setting of solving PDEs, as these define relations between functions and their derivatives. The requirement that activation functions be sufficiently differentiable means that the popular ReLU cannot be used when one aims to approximate second derivatives; ReLU activation functions have vanishing second derivatives because they are piecewise linear.

2.2.3 Loss Functions

Although neural networks have very nice properties in theory, it is not straightforward to make use of these in practice. It has proven to be very challenging to find sets of weights with which accurate approximations are achieved. There seem to be no direct ways of finding such weights. However, by constructing a kind of distance measure between the parametrized function that a neural network defines and the function one aims to approximate, the problem of finding good weights can be transformed into a minimization problem. Such distance measures are generally referred to as *loss functions*. Loss functions do not need to be true distance measures; the only thing that matters is that their minima correspond to accurate approximations.

It is not entirely clear why this minimization process yields good results in practice; due to the highly nonlinear nature of neural networks, loss functions tend to be chaotic, and one would expect local minima to be problematic. In practice, local minima do not seem to be problematic at all [3], and empirical evidence supports this conclusion. Unfortunately, that does mean that there are few theoretical guarantees about the convergence of this optimization process. To assess the accuracy of the resulting approximations, one generally has to evaluate the network output on a test dataset. This lack of theoretical rigour should not be a reason to halt further research, however.

To give some insight in the nature of these loss functions, let us consider an example. We aim to approximate some n-dimensional scalar function $g(\boldsymbol{x})$, which is only known to us through the input-output pairs $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$. Such scenarios are common in classification problems, where the exact relation between the variables and the resulting class is not known to us. To approximate this function with a neural network, the input layer should consist of n neurons, and the output layer should consist of one. Directly constructing a loss function based on this information is not straightforward, as it is not directly clear how the parameters of this neural network affect the accuracy. It is much simpler to construct an auxiliary function that measures the similarity between the available data and another function h. An example of such a loss function can be obtained by computing the mean square error as

$$L(h) \equiv \frac{1}{N}\sum_{i=1}^{N}\left(h(\boldsymbol{x}_i) - y_i\right)^2 . \tag{32}$$

From this auxiliary function, it is straightforward to construct a loss function by substituting the parametrized function for h, i.e.

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(f(\boldsymbol{x}_i, \theta) - y_i\right)^2 . \tag{33}$$

Minimizing such a loss function is called *training.* For notational convenience, let us arrange the parameters contained in θ in a vector, denoted by $\boldsymbol{\theta}$. In the next sections we discuss three common training algorithms. Although some of these algorithms gained popularity through neural networks, they are general purpose optimization algorithms, and can be applied in other contexts as well. These three algorithms are all local optimization methods, i.e. they find local minima of the loss function. Although there also exist global training methods, these methods are typically orders of magnitude more expensive computationally. Therefore we only discuss local methods.

2.2.4 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is one of the most common training algorithms for neural networks. It is very similar to gradient descent, which is a first order gradient-based optimization method that updates the variables subject to optimization in the direction of steepest descent. Applying gradient descent to neural networks is straightforward, because neural networks define closed form expressions which are differentiable almost everywhere. Many machine learning libraries that support neural networks also provide automatic differentiation tools, making the implementation of such algorithms managable.

Despite the simplicity of implementing gradient descent, stochastic gradient descent is generally preferred over gradient descent because it is less likely to get stuck at local minima or saddle points. This stochasticity is usually introduced by computing the gradients with respect to subsets of the data. The size of these subsets is called the *batch size*, and is a hyperparameter without a clear optimal choice. Hyperparameters like these are generally optimized empirically.

SGD depends on another hyperparameter: the learning rate. This parameter determines how large the update steps are every iteration, and is often denoted by α. Using a batch size of k, an iteration of SGD when minimizing the loss function $L(\boldsymbol{\theta})$ is given by

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \alpha\nabla L(\boldsymbol{\theta}_i), \tag{34}$$

where $\hat{L}$ is the loss function evaluated on a randomly selected batch containing k samples, i.e.

$$L(\boldsymbol{\theta}) = \frac{1}{k}\sum_{j=1}^{k}\left(f(\boldsymbol{x}_{i(j)}, \boldsymbol{\theta}) - y_{i(j)}\right)^2 \tag{35}$$

with $i(j)$ an arbitrary injection between $\{1, \ldots, k\}$ and $\{1, \ldots, N\}$, where N is the total number of available data points.

2.2.5 Adam

Adam is a modification of SGD developed in the study of [1]. Just like SGD, it is a first order gradient-based method. Adam uses an exponential moving average of the gradients with respect to the network parameters to update these parameters. Furthermore, the algorithm reduces the learning rate based on an exponential moving average of the variance of these gradients, such that noisy loss functions are more carefully traversed.

Let α again denote the learning rate. Furthermore, let $\boldsymbol{m}_i$ denote the moving average of the gradients, and let $\boldsymbol{v}_i$ denote the moving average of the variance of the gradients. These moving averages decay with rates β_1 and β_2 respectively. Then, the i-th iteration of Adam takes the form

$$\boldsymbol{g}_{i+1} = \nabla L(\boldsymbol{\theta}_i), \tag{36}$$

$$\boldsymbol{m}_{i+1} = \beta_1 \boldsymbol{m}_i + (1-\beta_1)\boldsymbol{g}_{i+1}, \tag{37}$$

$$\boldsymbol{v}_{i+1} = \beta_2 \boldsymbol{v}_i + (1-\beta_2)\boldsymbol{g}_{i+1} \odot \boldsymbol{g}_{i+1}, \tag{38}$$

$$\hat{\boldsymbol{m}}_{i+1} = \frac{\boldsymbol{m}_{i+1}}{\left(1-\beta_1^{i+1}\right)}, \tag{39}$$

$$\hat{\boldsymbol{v}}_{i+1} = \frac{\boldsymbol{v}_{i+1}}{\left(1-\beta_2^{i+1}\right)}, \tag{40}$$

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \alpha \hat{\boldsymbol{m}}_{i+1} \oslash \left(\sqrt{\hat{\boldsymbol{v}}_{i+1}} + \epsilon\right). \tag{41}$$

Here, $\odot$ and $\oslash$ denote Hadamard multiplication and division respectively, which are elementwise operations. The square root in eq. 41 is also applied elementwise. This algorithm tends to outperform SGD in practice.

2.2.6 L-BFGS

Another very popular training algorithm is the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS). This is a quasi-Newton method, i.e. it uses an approximation of the Hessian matrix, which we denote by B_i for the i-th iteration. The algorithm consists of three steps. First, given some approximation of the solution $\boldsymbol{\theta}_i$, the algorithm finds the solution to

$$B_i \boldsymbol{p}_i = -\nabla L(\boldsymbol{\theta}_i) \tag{42}$$

for $\boldsymbol{p}_i$. This is normally the update vector of Newtons's method. BFGS uses this update vector as a search direction to perform the line search

$$\alpha_i = \arg\min_{\alpha} L(\boldsymbol{\theta}_i + \alpha \boldsymbol{p}_i). \tag{43}$$

Then, the parameters are updated via

$$\boldsymbol{s}_i = \alpha_i \boldsymbol{p}_i, \tag{44}$$

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \boldsymbol{s}_i, \tag{45}$$

$$\boldsymbol{y}_i = \nabla L(\boldsymbol{\theta}_{i+1}) - \nabla L(\boldsymbol{\theta}_i), \tag{46}$$

$$B_{i+1} = B_i + \frac{\boldsymbol{y}_i \boldsymbol{y}_i^T}{\boldsymbol{y}_i^T \boldsymbol{s}_i} + \frac{B_i \boldsymbol{s}_i \boldsymbol{s}_i^T B_i^T}{\boldsymbol{s}_i^T B_k \boldsymbol{s}_i}. \tag{47}$$

The study of [14] proposes a limited memory version of BFGS, abbreviated to L-BFGS, which we will use instead of the regular BFGS. Without getting into the technical details of these modifications, this method does not store the full matrix B_i, but instead stores a number of the vectors $\boldsymbol{s}_i$ and $\boldsymbol{y}_i$ that are computed during the iterations, and uses these vectors to implicity represent the approximate Hessian. This results in a linear memory requirement with respect to the number of parameters, whereas the memory requirement of regular BFGS scales quadratically with the number of parameters. Since neural networks often have a very large number of parameters, these modifications are critical.

Unlike SGD and Adam, BFGS cannot be used in conjunction with batching techniques. This can be prohibitive, as it means that large portions of the available data must be processed when using this method.

3 Methods

Despite the large advances made in machine learning techniques in recent years, taking advantage of these new techniques in the context of solving PDEs has turned out to be quite challenging. So far, the most promising results involved either learning improved solvers, or directly learning the solution. Especially the latter method is conceptually very simple; the idea of directly learning the solution of a PDE is applicable to every PDE, and because it is such a general idea, it gives us a promising starting point.

Although there are different ways to learn the solution of a PDE, we will mainly refer to the work done in [8], as this work is very general and therefore easier to build on. To gain a better understanding of why these methods work, we reconstruct them from scratch in section 3.1 by showing that solving a PDE is equivalent to simultaneously optimizing several functionals. By introducing a loss weight parameter, we obtain a generalization of the original methods developed in [8]. Section 3.2 then covers some practical aspects of implementing these methods. Finally, section 3.3 treats the analysis of the loss weight parameter introduced in section 3.1. This section also covers an additional method of tackling the multi-objective optimization problem.

3.1 Physics Informed Neural Networks

So far, researchers have made several different attempts at solving PDEs by directly learning their solution. All of these attempts share the same key components: a neural network is used to approximate the solution as a function of the inputs. This network is then trained by minimizing a special loss function that is closely related to the PDE. There are, however, several fundamental differences in the ways this has been approached.

The solution of a PDE has to satisfy boundary or initial conditions, as well as the PDE itself. Essentially, these are all constraints that we somehow need to enforce upon the neural network. Key differences between the attempts that have been made so far, all lie in the treatment of these constraints. One way to enforce these constraints, is treating most them as hard constraints, and minimizing a loss function that encodes the last constraint such that all the hard constraints are satisfied. In some cases, it is fairly straightforward to transform boundary conditions into hard constraints. For example, in [6], the authors multiply their neural network with a function that is equal to 0 along the boundary, and manually find another function that satisfies the boundary conditions. Adding these two together allows the authors to only worry about satisfying the PDE itself, as the boundary conditions are satisfied for any neural network. This may not be easy to generalize, however.

Another way of enforcing hard constraints has been explored in [15]. Here, the authors come up with a completely new way of minimizing a loss function while satisfying constraints. This could become a viable approach in the future, but we believe that more advances need to be made first.

Instead of treating these constraints as hard constraints, we can also treat them as soft constraints by embedding them into loss functions, and minimizing these loss functions simultaneously. This method is used in [8] and is extremely general, and also very versatile. By modifying these loss functions, one is able to deal with different types of boundary conditions as well as different PDEs without fundamentally changing the training method.

In the next section, we will reconstruct this framework from a more mathematical perspective. We start by defining these loss functions, and analyze their properties.

3.1.1 Learning the Solution

Our first task is to construct a loss function that allows the network to find the solution of the PDE we are trying to solve. Although the loss function will be optimized with respect to the network's parameters, directly defining this function as a mapping from this parameter space into the real numbers would pose a significant challenge. We can simplify this problem by taking into account that neural networks are closed-form expressions, and instead define the loss function as a mapping from some function space into the real numbers, similar to the procedure we followed in section 2.2.3.

By treating the neural network and the training algorithms as black-box optimization methods, we are able to define the loss function without requiring any a priori knowledge about the architecture of the neural network or the optimization algorithm.

Let us start by defining the PDE that we wish to solve. Let the problem domain be given by $\Omega \subset \mathbb{R}^d$, and let $\boldsymbol{x} \in \Omega$. We define the problem for $u(\boldsymbol{x})$ by

$$N_I(\boldsymbol{x}, u) = 0 \text{ in } \Omega, \tag{48}$$

subject to the boundary conditions

$$N_B(\boldsymbol{x}, u) = 0 \text{ on } \partial\Omega, \tag{49}$$

for some scalar functions N_I, N_B.

Our goal is to recover the true solution of this PDE, which we denote by $\hat{u}(\boldsymbol{x})$, by minimizing some functional $L(u)$ over the space of k-times differentiable functions, where k depends on the derivatives present in the PDE. Since we are treating the training algorithm as a black-box minimization algorithm, there are several highly desirable properties that L should satisfy. In order to recover the solution by minimizing L, we require the following property.

Property 3.1. The minimizer of L coincides with the solution of the PDE, i.e.

$$\arg\min_{u \in C^k} L(u) = \hat{u}. \tag{50}$$

Although neural networks are able to approximate any function arbitrarily well, it is generally not true that finite neural networks can express any function. In particular, it is unreasonable to assume that the solution of any PDE can be expressed as a finite neural network. Therefore, the true minimum of L may not be reachable, and thus we have another important requirement that L should satisfy.

Property 3.2. For any $\epsilon > 0$, there exists a $\delta > 0$ such that

$$L(u) - L(\hat{u}) < \delta \tag{51}$$

implies that

$$\|u - \hat{u}\| < \epsilon. \tag{52}$$

Next, we require two properties that allow many black box optimization algorithms to successfully minimize the loss functional. These properties should be sufficient to guarantee many such methods to converge, given infinitesimal step sizes and complete freedom within the employed function space.

First, we will deal with the inability of many local optimization methods such as gradient descent to escape from local minima. To ensure that the methods do not get stuck, we will require that the loss functional only has a single, and therefore global minimum.

Property 3.3. $L(u)$ has a unique, global minimum.

The last property we consider will ensure that the optimizer cannot run off towards infinity, by requiring that the loss increases for functions far away from the true solution.

Property 3.4. For every $\epsilon > 0$, there exists a $\delta > 0$ such that

$$\|u - \hat{u}\| > \delta \implies L(u) - L(\hat{u}) > \epsilon. \tag{53}$$

Properties 3.3 and 3.4 allow us to construct bounded regions that contain the full path an optimizer might take when minimizing L. Such paths only terminate when a local minimum is reached. Because the path is contained in a bounded region, it must eventually terminate, and thus idealized local optimization methods must converge if properties 3.3 and 3.4 are satisfied.

With these properties in mind, let us construct a candidate loss functional. By assuming N_I and N_B to be scalar, we obtain an intuitive starting point for defining the loss functional: the norms of N_I and N_B, which are given by

$$\|N_I(\boldsymbol{x}, u)\|_p \equiv \left[\int_\Omega |N_I(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}\right]^{\frac{1}{p}} \tag{54}$$

and

$$\|N_B(\boldsymbol{x}, u)\|_p \equiv \left[\int_{\partial\Omega} |N_B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}\right]^{\frac{1}{p}} \tag{55}$$

for some $p \geq 1$. Minimizing these loss functionals would result in a solution that either satisfies the PDE, but not necessarily the boundary conditions, or vice versa. The function that minimizes both norms simultaneously is a solution of the PDE. Note that in order to minimize Eqs. 54 and 55, we can omit the p-th root. This way, we obtain two families of loss functionals,

$$L_I(p, u) = \int_\Omega |N_I(\boldsymbol{x}, u)|^p \, d\boldsymbol{x} \tag{56}$$

$$L_B(p, u) = \int_{\partial\Omega} |N_B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}. \tag{57}$$

In order to train the neural network, we have to construct a single loss functional. To this end, one could combine the two loss functionals defined above as is done in [8], i.e. add them together. However, we consider it unreasonable to assume that both loss functionals are always equally important. Therefore, we will combine the loss functionals in a more general fashion, by introducing the parameter $\lambda \in (0, 1)$ that represents their relative importance. The total loss functional is then given by

$$\begin{aligned} L(u, p) &= \lambda L_I(u, p) + (1 - \lambda) L_B(u, p) \\ &= \lambda \int_\Omega |N_I(\boldsymbol{x}, u)|^p \, d\boldsymbol{x} + (1 - \lambda) \int_{\partial\Omega} |N_B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}. \end{aligned} \tag{58}$$

We will investigate this loss weight in more detail in section 3.3.1.

The case $p = 2$ reduces to least squares regression, which is one of the most studied regression methods. Although there is no particular reason to assume that $p = 2$ would yield better results than any other value, it is generally more straightforward to analyze. In most of the results, we will therefore use the loss functional

$$\begin{aligned} L(u) &= \lambda L_I(u, 2) + (1 - \lambda) L_B(u, 2) \\ &= \lambda \int_\Omega N_I(\boldsymbol{x}, u)^2 d\boldsymbol{x} + (1 - \lambda) \int_{\partial\Omega} N_B(\boldsymbol{x}, u)^2 d\boldsymbol{x}. \end{aligned} \tag{59}$$

as a starting point.

3.1.2 Properties of the Basic Loss Functional

We will show that under certain conditions, the loss functional defined in eq. 58 satisfies all four properties, starting with with property 3.1.

Theorem 1. *The loss functional defined in eq. 58 attains its minimum at the solution of the PDE, $\hat{u}$.*

Proof. This property is very straightforward to prove. By definition, $\hat{u}$ satisfies the PDE, and therefore we know that for any $\boldsymbol{x} \in \Omega$, $N_I(\boldsymbol{x}, \hat{u}) = 0$ and for any $\boldsymbol{x} \in \partial\Omega$, $N_B(\boldsymbol{x}, \hat{u}) = 0$. Therefore, $L(\hat{u}, p) = 0$. Conversely, if u does not satisfy the PDE exactly, be it at the boundary or in the interior of the domain, then N_I or N_B will be nonzero. Since we only consider $u \in C^k$, this also means that the loss must be nonzero. Since $L(u, p) \geq 0$, it follows that for any $u \neq \hat{u}$, $L(\hat{u}, p) < L(u, p)$. $\square$

Next, we will examine property 3.3. We will show that this property is satisfied for linear PDEs. To exploit linearity, we rewrite the definition of the PDE as

$$\hat{N}_I(\boldsymbol{x}, u) = F(\boldsymbol{x}) \text{ in } \Omega, \tag{60}$$

$$\hat{N}_B(\boldsymbol{x}, u) = G(\boldsymbol{x}) \text{ on } \partial\Omega, \tag{61}$$

where $\hat{N}_I(\boldsymbol{x}, u)$ and $\hat{N}_B(\boldsymbol{x}, u)$ are linear in u. Then, we have the following theorem.

Theorem 2. *For linear PDEs, the loss functional defined in eq. 59 only has a single, global minimum.*

Proof. We will show that for an arbitrary $u \neq \hat{u}$, there exists a path $v(t)$ in function space from u to $\hat{u}$ for which $L(v(t))$ is monotonically decreasing.

Let us start by rewriting the loss in terms of the linear operators present in the PDE. The linear operators are related to the original definition of the PDE by

$$N_I(\boldsymbol{x}, u) = \hat{N}_I(\boldsymbol{x}, u) - F(\boldsymbol{x}), \tag{62}$$

$$N_B(\boldsymbol{x}, u) = \hat{N}_B(\boldsymbol{x}, u) - G(\boldsymbol{x}), \tag{63}$$

and thus for the losses we have

$$\begin{aligned} L_I(u, p) &= \int_\Omega |N_I(\boldsymbol{x}, u)|^p \, \boldsymbol{x} \\ &= \int_\Omega \left|\hat{N}_I(\boldsymbol{x}, u) - F(\boldsymbol{x})\right|^p d\boldsymbol{x} \end{aligned} \tag{64}$$

and

$$\begin{aligned} L_B(u, p) &= \int_{\partial\Omega} |N_B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x} \\ &= \int_{\partial\Omega} \left|\hat{N}_B(\boldsymbol{x}, u) - G(\boldsymbol{x})\right|^p d\boldsymbol{x}. \end{aligned} \tag{65}$$

Next we construct the parametrized path $v(t)$ for $t \in [0,1]$ as

$$v(t) \equiv (1-t)u + t\hat{u}. \tag{66}$$

Then, if we evaluate the loss along this parametrized path, we obtain

$$L(v(t),p) = \lambda L_I(v(t),p) + (1-\lambda) L_B(v(t),p). \tag{67}$$

For the interior loss it holds that

$$\begin{aligned} L_I(v(t),p) &= \int_\Omega \left|\hat{N}_I(\boldsymbol{x}, v(t)) - F(\boldsymbol{x})\right|^p d\boldsymbol{x} \\ &= \int_\Omega \left|\hat{N}_I(\boldsymbol{x}, (1-t)u + t\hat{u}) - F(\boldsymbol{x})\right|^p d\boldsymbol{x} \\ &= \int_\Omega \left|(1-t)\hat{N}_I(\boldsymbol{x}, u) - (1-t)F(\boldsymbol{x}) + tN(\boldsymbol{x},\hat{u}) - tF(\boldsymbol{x})\right|^p d\boldsymbol{x} \\ &= (1-t)^p \int_\Omega \left|\hat{N}_I(\boldsymbol{x}, u) - F(\boldsymbol{x})\right|^p d\boldsymbol{x} \\ &= (1-t)^p L_I(u,p). \end{aligned} \tag{68}$$

Performing the same procedure for the boundary loss yields

$$L_B(v(t),p) = (1-t)^p L_B(u,p). \tag{69}$$

After combining the results, we finally arrive at

$$L(v(t),p) = (1-t)^p L(u,p), \tag{70}$$

which is, since $L(u,p) > 0$, indeed monotonically decreasing in t. Therefore, u cannot be a local minimum point of L, and hence the minimum point $\hat{u}$ must be unique. □

Next, we will investigate the asymptotic behaviour of this loss functional, both for u close to $\hat{u}$ and for u far away. We will start with the behaviour close to the solution. The desired property in this region would be for u to approach $\hat{u}$ as $L(u)$ approaches $L(\hat{u})$. We will show that this is indeed the case for the loss functional defined in eq. 58, provided that the PDE is well-posed.

Theorem 3. *Let a well-posed PDE be given by*

$$N_I(\boldsymbol{x}, u) = F(\boldsymbol{x}) \text{ in } \Omega, \tag{71}$$

$$N_B(\boldsymbol{x}, u) = G(\boldsymbol{x}) \text{ on } \partial\Omega, \tag{72}$$

and let us denote the solution of this PDE by $\hat{u}$. Let the interior and boundary loss be given by respectively

$$L_I(u) = \int_\Omega |N_1(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}, \tag{73}$$

$$L_B(u) = \int_{\partial\Omega} |N_2(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}, \tag{74}$$

for some $p \geq 1$. Then, for any $\epsilon > 0$ there exists a $\delta > 0$ such that

$$L(u) < \delta \implies \|u - \hat{u}\| < \epsilon. \tag{75}$$

Proof. Define

$$\delta \equiv \frac{\epsilon}{C\left[\frac{1}{\lambda}|\Omega|^{1-\frac{1}{p}} + \frac{1}{1-\lambda}|\partial\Omega|^{1-\frac{1}{p}}\right]}, \tag{76}$$

where C is the Lipschitz constant of the PDE, and let u be some approximate solution of the PDE such that

$$L(u) < \delta. \tag{77}$$

Because u may not exactly satisfy the operators N_I and N_B, we have

$$N_I(\boldsymbol{x}, u) = F + \hat{F}, \tag{78}$$

$$N_B(\boldsymbol{x}, u) = G + \hat{G}. \tag{79}$$

In other words, u satisfies a perturbed version of the PDE. Because this PDE is well-posed, it follows from definition 2.1 that

$$\begin{aligned} \|u - \hat{u}\| &< C\{\|(\hat{F} + F) - F\| + \|(\hat{G} + G) - G\|\} \\ &= C\{\|\hat{F}\| + \|\hat{G}\|\}. \end{aligned} \tag{80}$$

Using Hölder's inequality, we obtain for $\|\hat{F}\|$ that

$$\begin{aligned} \|\hat{F}\| = \int_\Omega \|\hat{F}\| d\boldsymbol{x} &\le |\Omega|^{1-\frac{1}{p}} \left[\int_\Omega \|\hat{F}\|^p d\boldsymbol{x}\right]^{\frac{1}{p}} \\ &= |\Omega|^{1-\frac{1}{p}} L_I(u). \end{aligned} \tag{81}$$

Similarly, for $\|\hat{G}\|$ we find

$$\begin{aligned} \|\hat{G}\| = \int_{\partial\Omega} \|\hat{G}\| d\boldsymbol{x} &\le |\partial\Omega|^{1-\frac{1}{p}} \left[\int_{\partial\Omega} \|\hat{G}\|^p d\boldsymbol{x}\right]^{\frac{1}{p}} \\ &= |\partial\Omega|^{1-\frac{1}{p}} L_B(u). \end{aligned} \tag{82}$$

Combining these results gives us

$$\begin{aligned} \|u - \hat{u}\| &< C\{\|\hat{F}\| + \|\hat{G}\|\} \\ &\le C\{|\Omega|^{1-\frac{1}{p}} L_I(u) + |\partial\Omega|^{1-\frac{1}{p}} L_B(u)\} \\ &\le C\{|\Omega|^{1-\frac{1}{p}} \frac{1}{\lambda} L(u) + |\partial\Omega|^{1-\frac{1}{p}} \frac{1}{1-\lambda} L(u)\} \\ &= C\left(|\Omega|^{1-\frac{1}{p}} \frac{1}{\lambda} + |\partial\Omega|^{1-\frac{1}{p}} \frac{1}{1-\lambda}\right) L(u) \\ &= \frac{\epsilon}{\delta} L(u) < \epsilon. \end{aligned} \tag{83}$$

□

Finally, we prove the last property.

Theorem 4. *Let a well-posed PDE be given by*

$$N_I(\boldsymbol{x}, u) = F(\boldsymbol{x}) \text{ in } \Omega, \tag{84}$$

$$N_B(\boldsymbol{x}, u) = G(\boldsymbol{x}) \text{ on } \partial\Omega, \tag{85}$$

and let us denote the solution of this PDE by $\hat{u}$. Let the interior and boundary loss be given by respectively

$$L_I(u) = \int_\Omega |N_I(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}, \tag{86}$$

$$L_B(u) = \int_{\partial\Omega} |N_B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}, \tag{87}$$

for some $p \geq 1$. Then, for any $\epsilon > 0$ there exists a $\delta > 0$ such that

$$\|u - \hat{u}\| > \delta \implies L(u) > \epsilon. \tag{88}$$

Proof. Let $\epsilon > 0$ be arbitrary, and define

$$\delta \equiv C \left[2^{p-1} \max \left\{ \frac{1}{\lambda} |\Omega|^{p-1}, \frac{1}{1-\lambda} |\partial\Omega|^{p-1} \right\} \epsilon \right]^{\frac{1}{p}}, \tag{89}$$

where C is the Lipschitz constant of the PDE. Let furthermore u be some approximate solution of the PDE such that

$$\|u - \hat{u}\| > \delta. \tag{90}$$

Let us now compute a lower bound for the loss, which was given by

$$L(u) = \lambda L_I(u) + (1 - \lambda) L_B(u). \tag{91}$$

First, we will write this loss functional in terms of $\|\hat{F}\|$ and $\|\hat{G}\|$. For L_I we have

$$\begin{aligned} L_I(u) &= \int_\Omega |\hat{F}(\boldsymbol{x})|^p dx = \|\hat{F}\|_p^p \\ &\geq \left[\frac{\|\hat{F}\|}{|\Omega|^{1-\frac{1}{p}}} \right]^p = \|\hat{F}\|^p |\Omega|^{1-p}, \end{aligned} \tag{92}$$

and similarly for L_B,

$$\begin{aligned} L_B(u) &= \int_{\partial\Omega} |\hat{G}(\boldsymbol{x})|^p dx = \|\hat{G}\|_p^p \\ &\geq \left[\frac{\|\hat{G}\|}{|\partial\Omega|^{1-\frac{1}{p}}} \right]^p = \|\hat{G}\|^p |\partial\Omega|^{1-p}, \end{aligned} \tag{93}$$

where the inequalities follow from Hölder's inequality. Thus, the loss can be lower bounded by

$$\begin{aligned} L(u) &\geq \lambda \|\hat{F}\|^p |\Omega|^{1-p} + (1-\lambda) \|\hat{G}\|^p |\partial\Omega|^{1-p} \\ &\geq \min \left\{ \lambda |\Omega|^{1-p}, (1-\lambda) |\partial\Omega|^{1-p} \right\} \left(\|\hat{F}\|^p + \|\hat{G}\|^p \right). \end{aligned} \tag{94}$$

We can again apply Hölder's inequality on $\|\hat{F}\|^p + \|\hat{G}\|^p$ to find

$$L(u) \geq \min\left\{\lambda|\Omega|^{1-p}, (1-\lambda)|\partial\Omega|^{1-p}\right\}\left(\|\hat{F}\| + \|\hat{G}\|\right)^p 2^{1-p}. \tag{95}$$

By well-posedness, it must hold that

$$\begin{aligned} L(u) &\geq \min\left\{\lambda|\Omega|^{1-p}, (1-\lambda)|\partial\Omega|^{1-p}\right\}\left(\frac{1}{C}\|u-\hat{u}\|\right)^p 2^{1-p} \\ &> \min\left\{\lambda|\Omega|^{1-p}, (1-\lambda)|\partial\Omega|^{1-p}\right\}\left(\frac{\delta}{C}\right)^p 2^{1-p} \\ &= \epsilon, \end{aligned} \tag{96}$$

completing the proof. □

We have established a loss functional that satisfies all four of our desired properties. Idealized local optimization methods should hence be able to find decent approximations of the true solution for a large class of PDEs, under the assumption of total freedom within the space of sufficiently differentiable functions. To conclude this section, we leave one remark regarding these theoretical results.

At first glance, one might think that the bounds present in theorems 3 and 4 can be used to obtain useful information about λ and p. However, one has to be careful with this, because many of the inequalities that we used are not tight. Therefore, these bounds do not reveal the underlying interactions between λ, u and p. We will investigate these parameters more thoroughly in section 3.3.

3.1.3 Mixed Boundary Conditions

So far we have only considered pure boundary conditions. However, many problems have mixed boundary conditions. For such problems, we essentially have that $N_B(\boldsymbol{x}, u)$ takes a different form in different regions of the boundary. This does not mean that anything fundamental changes about the loss functionals that we have defined so far, since eq. 58 can still be evaluated. Despite this, we suggest that the integral should be split up into different parts corresponding to the different boundary conditions. Let us assume that the boundary conditions are given by

$$N_B(\boldsymbol{x}, u) = \begin{cases} N_{B,1}(\boldsymbol{x}, u), & \boldsymbol{x} \in \partial\Omega_1 \\ \dots \\ N_{B,k}(\boldsymbol{x}, u), & \boldsymbol{x} \in \partial\Omega_k. \end{cases} \tag{97}$$

Without making any changes to our original definition of the loss functional, we write

$$\begin{aligned} L_B(u) &= \int_{\partial\Omega} |N_B(\boldsymbol{x}, u)|^p \, d\boldsymbol{x} \\ &= \sum_i \int_{\partial\Omega_i} |N_{B,i}(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}. \end{aligned} \tag{98}$$

Here, it becomes clear why this loss functional may not be ideal when we are dealing with mixed boundary conditions. Similar as before, it is not at all clear whether the different loss functionals present in eq. 98 are all equally important and equally easy to optimize. It makes much more sense to add a weight to each of these loss terms, so that their importance may be fine tuned to the problem of interest. We will denote these weight parameters by γ_i, and redefine the boundary loss as

$$L_B(u) = \sum_i \gamma_i \int_{\partial\Omega_i} |N_{B,i}(\boldsymbol{x}, u)|^p \, d\boldsymbol{x}. \tag{99}$$

3.1.4 Complex-Valued PDEs

So far we have assumed the PDE operators to be scalar and real-valued. In this section we briefly discuss complex-valued PDEs.

In order for a neural network to approximate a complex function, we have two choices: we can either convert the entire network to be complex valued, or we can let the network approximate the real and imaginary parts of the function separately by doubling the dimension of the network's output. The second option has many computational advantages, as the number of parameters in the network does not change significantly. Furthermore, many algorithms are not directly suited to perform complex optimization, so changing the data type of the neural network usually requires significant modifications to the training algorithms.

3.2 Numerical Tractability

In this section we cover the transition from the theoretical, function space methods we developed in section 3.1 to practical algorithms that train neural networks to approximate the solution of a wide variety of PDEs. This transition consists of two parts. First of all, we have to change the space over which we perform the optimization from function space to weight space. Secondly, we need to consider the training algorithms that can perform this optimization. In the last part of this section we discuss the initialization of the network parameters.

3.2.1 Weight Space Optimization

Let us start by discussing the transition from function space to weight space. So far our methods rely entirely on optimizing the loss functional over the space of k times differentiable functions. In order to express our solution with a neural network, we instead have to perform this optimization over the parameters of the network $\boldsymbol{\theta} \in \mathbb{R}^N$. In principle, this is a straightforward transition to make. Essentially, neural networks define parametrized functions $u(\boldsymbol{\theta})$. By directly substituting these parametrized functions into the methods we have obtained so far, the transition is essentially complete.

Thus, in order to transform our methods to the weight space corresponding to the neural network, we redefine the loss functional $L(u, p)$ as a loss function $L(\boldsymbol{\theta}, p)$. Sticking to the general family of loss functionals given in eq. 58, we have

$$\begin{aligned} L(\boldsymbol{\theta}, p) &= L(u(\boldsymbol{\theta}), p) \\ &= \lambda \int_\Omega |N_I(\boldsymbol{x}, u(\boldsymbol{\theta}))|^p \, d\boldsymbol{x} + (1 - \lambda) \int_{\partial\Omega} |N_B(\boldsymbol{x}, u(\boldsymbol{\theta}))|^p \, d\boldsymbol{x}. \end{aligned} \tag{100}$$

Unfortunately, as a result of this process, the properties that we have examined so carefully are no longer necessarily satisfied, and there is not much we can do to remedy this. Although much is known about neural networks and the relation between their weights and the corresponding loss surfaces, theoretical guarantees for convergence seem to be evasive. This is a fundamental problem that neural networks suffer from, and it is not within the scope of this project to adress these issues. However, empirical evidence indicates that optimization asymptotically obtains the degrees of freedom that would make this transition valid. As more theoretical results are obtained, we hope that it becomes possible to prove that in the limit of $N \to \infty$, the properties we derived earlier become valid again.

Now that we have defined the loss function in terms of the neural network's weights, we still have to define the optimization algorithms. In the preliminaries we already discussed several common methods of training neural networks. We will discuss these methods while keeping in mind that our goal is to solve PDEs.

3.2.2 Normalized SGD

In the preliminaries, we mentioned three different training algorithms that are commonly used. Although Adam and L-BFGS are more sophisticated and generally lead to better performance, SGD is much simpler and therefore easier to modify. It has, however, one flaw that we would like to address first: unlike Adam or L-BFGS, SGD does nothing to regulate the learning rate. When using SGD, one usually has to choose the learning rate manually, often through trial and error, such that the process converges at all and within reasonable time. A learning rate that is too large may cause gradients to vanish or to blow up, ultimately resulting in values that can no longer be represented by floating point numbers. On the other hand, learning rates that are too small can raise the number of iterations required to reach a decent approximation to infeasible values. Since the range of suitable learning rates strongly depends on the network configurations, every time the network architecture is changed, be it by choosing a different activation function, or a different number of layers or neurons, a new learning rate has to be searched for.

Although it is difficult to eliminate this issue entirely, we can alleviate it to a decent degree by normalizing the gradients. Normalization means that the step sizes no longer depend on the loss function in any way, and thus we obtain full control. Furthermore, the same step sizes should work comparably well for different network configurations, since one expects the individual weights to be in a similarly small range regardless of the activation function used, and regardless of the number of neurons.

As mentioned in section 2.2.4, the learning rate is generally denoted by α. Recall that a SGD iteration is given by

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha \nabla L(\boldsymbol{\theta}_n). \tag{101}$$

By normalizing the gradient $\nabla L(\boldsymbol{\theta}_n)$, we obtain normalized SGD steps with magnitude equal to the learning rate, i.e.

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha \frac{\nabla L(\boldsymbol{\theta}_n)}{\|\nabla L(\boldsymbol{\theta}_n)\|}. \tag{102}$$

We will refer to this method as normalized SGD, which we abbreviate to NSGD.

No normalization is needed for Adam or BFGS, since both of these methods are scale-invariant, as mentioned in sections 2.2.5 and 2.2.6. Note that this modification to SGD makes this method scale-invariant as well, since multiplying the loss function by a constant no longer affects the results. With this modification in place, it is safe to scale the loss functions for mathematical convenience.

3.2.3 Monte Carlo Methods

The three methods we will be using have one thing in common: they rely on their ability to compute the gradients of the loss function with respect to the weights. Due to the integrals present in the loss functions, these gradients are usually intractable, and we have to resort to approximations.

Monte Carlo integration is a well known technique that allows us to deal with these difficult integrals. It has the added benefit that its performance remains quite good even as the dimension of the problem increases. This property is quite useful, since high-dimensional problems are particularly interesting to solve using neural networks, which are known to deal well with those.

Using Monte Carlo integration, integrals can be approximated via

$$\int_\Omega d\boldsymbol{x} \approx \|\Omega\| \frac{1}{n} \sum_{i=1}^{n} H(\boldsymbol{x}_i), \tag{103}$$

where $\boldsymbol{x}_i$ are sampled uniformly over Ω. Applying this to the family of loss functions defined in eq. 100, we obtain a family of approximated loss functions.

$$\begin{aligned} L(\boldsymbol{\theta}, p) &= \lambda \int_{\Omega} |N_I(\boldsymbol{x}, u(\boldsymbol{\theta}))|^p \, d\boldsymbol{x} + (1-\lambda) \int_{\partial\Omega} |N_B(\boldsymbol{x}, u(\boldsymbol{\theta}))|^p \, d\boldsymbol{x} \\ &\approx \lambda \|\Omega\| \frac{1}{n_I} \sum_{i=1}^{n_I} |N_I(\boldsymbol{x}_{I,i}, u(\boldsymbol{\theta}))|^p + (1-\lambda) \|\partial\Omega\| \frac{1}{n_B} \sum_{i=1}^{n_B} |N_B(\boldsymbol{x}_{B,i}, u(\boldsymbol{\theta}))|^p. \end{aligned} \tag{104}$$

We will label the approximated losses $\hat{L}$. As mentioned in section 3.2.2, the three methods we will be using are scale-invariant. Since we have not yet specified λ, we can incorporate the constants $\|\Omega\|$ and $\|\partial\Omega\|$ into λ without changing the results. Therefore, eq. 104 can be simplified to

$$\begin{aligned} \hat{L}(\boldsymbol{\theta}, p) &= \lambda \hat{L}_I(u(\boldsymbol{\theta}), p) + (1-\lambda) \hat{L}_B(u(\boldsymbol{\theta}), p) \\ &= \lambda \frac{1}{n_I} \sum_{i=1}^{n_I} |N_I(\boldsymbol{x}_{I,i}, u(\boldsymbol{\theta}))|^p + (1-\lambda) \frac{1}{n_B} \sum_{i=1}^{n_B} |N_B(\boldsymbol{x}_{B,i}, u(\boldsymbol{\theta}))|^p. \end{aligned} \tag{105}$$

We call $\boldsymbol{x}_{I,i}$ and $\boldsymbol{x}_{B,i}$ collocation points.

3.2.4 Weight Initialization

There has been a lot of research on the initialization of the weights of neural networks. In particular the work done in [16] has proved to be very useful for networks with high numbers of layers. This particular initialization scheme preserves gradients as layers are added, and hence one needs not worry about vanishing or exploding gradients, which is an issue that commonly prevents training large neural networks.

As powerful as large neural networks can be, we will focus mainly on using smaller neural networks. In particular, if a neural network is able to approximate a solution at all, then the same network should be able to learn the solution via the methods we are developing. For these smaller networks, it is not very important that the gradients are preserved between layers. Instead, there is a different property that we would like to exploit, which will give us a different initialization scheme: neural networks obtain most of their approximative power from the nonlinear activation function. Therefore, we suspect that making sure the inputs of most neurons cover the nonlinear range will yield the best results.

Unfortunately, the resulting initialization scheme will depend on the activation function. We will only construct such an initialization scheme for the hyperbolic tangent function, since that is the activation function we will use in most cases in this work.

Like the authors of [16], we only initialize the network weights to nonzero values, and ignore the biases. Glorot initialization is based on preserving the variance of layers. By assuming that most activations are still in their linear regimes, and by assuming that the derivative of the activation function in this linear regime is approximately equal to 1, they obtain the rule

$$n\mathrm{Var}(\boldsymbol{\theta}) = 1, \tag{106}$$

where n is the number of neurons per layer. Combined with the condition that

$$(n+1)\mathrm{Var}(\boldsymbol{\theta}) = 1, \tag{107}$$

this results in the uniform distribution of the weights given by

$$\boldsymbol{\theta} \sim \Big[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\Big], \tag{108}$$

where n_i is the number of neurons in the i-th layer.

We will mimic this procedure. In order to exploit nonlinearity, which for the hyperbolic tangent function is mostly concentrated around in the region $[-2, 2]$, we choose the variance such that

$$n\mathrm{Var}(\boldsymbol{\theta}) = \frac{2}{\tanh(2)}. \tag{109}$$

This results in the weights

$$\boldsymbol{\theta} \sim \Big[-\frac{\sqrt{6}}{\sqrt{n_i \tanh 2}}, \frac{\sqrt{6}}{\sqrt{n_i \tanh 2}}\Big]. \tag{110}$$

3.3 Multi-Objective Optimization

We have encountered several problems that require optimizing multiple loss functions simultaneously. To recap, we chose to implement the constraints defined by initial and boundary conditions as a loss function, i.e. we solve PDEs by minimizing the boundary loss and the interior loss simultaneously. Furthermore, when boundary conditions are mixed, we end up with additional loss terms. Up until now, we dealt with this by multiplying each loss function by weights, λ for the interior loss, $1-\lambda$ for the total boundary loss, and γ_i for the different parts of the boundary losses. In this section we address the choice of these weights.

In section 3.3.1, we start by treating the weights as constants which need to be chosen a priori. Then, we discuss ways of programmatically choosing λ while still treating it as a constant during optimization. In section 3.3.2, we modify the loss functions to incorporate the weights. Finally, in section 3.3.3 we investigate some of the theoretical properties of these methods. Finally, in section 3.3.5, we discuss an additional way of dealing with multi-objective optimization.

3.3.1 Optimizing Loss Weights

In this section we treat λ and γ_i as constants, and aim to find their optimal values for certain PDEs. Since most of these ideas are generalizable, we will focus primarily on PDEs with pure boundary conditions, such that we only have to consider λ.

Let us start by examining an example. Consider the two-dimensional Laplace equation, given by

$$\begin{aligned} \nabla^2 u &= 0 \text{ in } \Omega, \\ u &= G \text{ on } \partial\Omega. \end{aligned} \tag{111}$$

We will not use the approximations required to implement the loss functions yet. Using the family of loss functionals defined in eq. 58, we obtain a total loss functional given by

$$\begin{aligned} L(u,p) &= \lambda L_I(u,p) + (1-\lambda) L_B(u,p) \\ &= \lambda \int_\Omega \left|\nabla^2 u(\boldsymbol{x})\right|^p d\boldsymbol{x} + (1-\lambda)\int_{\partial\Omega} |u(\boldsymbol{x}) - G(bmx)|\, d\boldsymbol{x}. \end{aligned} \tag{112}$$

This equation is well known to be well-posed, and because the problem is also linear the default loss functional satisfies all properties we derived in section 3.1. One of these properties already gives us some insight in the relation between different loss terms: property 3.2 tells us that as the losses approach 0, so does the error of the solution. However, in order to use this information to determine how the different loss terms are related, one has to find the Lipschitz constant of the PDE.

The worst case scenario that this Lipschitz constant accounts for is often given by the error being of a particular shape. In practice, and especially when using neural networks, this shape does not at all coincide with the actual error. When we are training the neural networks to minimize these functionals, which are based on the derivatives, the error tends to attain a similar shape as the true solution. Highly oscillating solutions will often lead to highly oscillating errors, and the result is that large portions of error cancel each other out. This suggests that when we expect the solution to have large derivatives, we can be more lenient in the error of these derivatives. Also note that not every PDE admits a Lipschitz constant; these only exist for well-posed PDEs.

For these reasons, instead of considering the absolute error, we will consider the relative error, both of the solution itself and of the derivatives. We will call a candidate solution ϵ-close to the true solution if it satisfies

$$\left|\frac{\partial^n u}{\partial x_i^n} - \frac{\partial^n \hat{u}}{\partial x_i^n}\right| \leq \epsilon \frac{\partial^n \hat{u}}{\partial x_i^n} \tag{113}$$

for all $n \geq 0$ and $i \in \{1, \ldots, d\}$, where d is the dimension of the problem. By searching for solutions that are ϵ-close to the true solution, rather than just asking for a low absolute error, we can make much stronger statements about the loss functions.

Let us examine the loss of a solution u that is ϵ-close to $\hat{u}$. For the interior loss, we obtain for $p \geq 1$,

$$\begin{aligned} L_I(u,p) &= \int_\Omega |u_{xx}(\boldsymbol{x}) + u_{yy}(\boldsymbol{x})|^p \, d\boldsymbol{x} \\ &= \int_\Omega |u_{xx}(\boldsymbol{x}) - \hat{u}_{xx}(\boldsymbol{x}) + \hat{u}_{xx}(\boldsymbol{x}) + u_{yy}(\boldsymbol{x}) - \hat{u}_{yy}(\boldsymbol{x}) + \hat{u}_{yy}(\boldsymbol{x})|^p \, d\boldsymbol{x} \\ &\leq \int_\Omega (|u_{xx}(\boldsymbol{x}) - \hat{u}_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x}) - \hat{u}_{yy}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x}) + \hat{u}_{xx}(\boldsymbol{x})|)^p \, d\boldsymbol{x} \\ &\leq \int_\Omega (\epsilon \, |\hat{u}_{xx}(\boldsymbol{x})| + \epsilon \, |\hat{u}_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x} \\ &= \epsilon^p \int_\Omega (|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x}. \end{aligned} \tag{114}$$

For the boundary loss, we find in a similar fashion that

$$\begin{aligned} L_B(u,p) &= \int_{\partial\Omega} |u(bmx) - G(\boldsymbol{x})|^p \, d\boldsymbol{x} \\ &\leq \int_{\partial\Omega} |\epsilon \hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x} \\ &\leq \epsilon^p \int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x}. \end{aligned} \tag{115}$$

Thus, we obtain a total loss bounded by

$$L(u,p) \leq \epsilon^p \left[\lambda \epsilon^p \int_\Omega (|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x} + (1-\lambda) \int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x} \right]. \tag{116}$$

These inequalities define necessary conditions for a solution to be ϵ-close to the true solution. It is straightforward to see that they are not sufficient; by increasing the variance of the error, it is possible to satisfy the integrals without satisfying eq. 113 everywhere. This ties in with the reasoning we used earlier; by assuming the error to be similarly shaped to the true solution, higher losses are allowed.

This assumption has several important implications. Firstly, the expression given in eq. 116 depends mainly on the true solution, as opposed to any expression we might obtain from the proof of property 3.2. This dependency is something we will examine thoroughly. It does make intuitive sense, however; if we are trying to learn the solution of a PDE with very difficult boundary conditions, then we expect a naturally higher boundary loss. Similarly, if the shape of the solution is very complex in the interior of the domain, then we expect an interior loss that is naturally higher. Demanding both losses to be equally small in general is nonsensical.

In fact, eq. 116 can be used to optimize λ in some sense; it is desirable that necessary conditions are as easy to satisfy as possible. Given some value of $L(u,p)$, we can optimize

λ to minimize the value of ϵ for which both of the necessary conditions given in eq. 116 and 114 are satisfied. Of course, it is not clear that neural networks are able to achieve the same loss for different loss weights, but we suspect that by sacrificing one loss term to improve the other, neural networks can come close to achieving this.

To perform the optimization, let us fixate the value of $L(u,p)$. Then, we can derive lower bounds for L_I and L_B if $\lambda \in (0,1)$. These bounds follow from

$$L(u,p) = \lambda L_I(u,p) + (1-\lambda) L_B(u,p), \tag{117}$$

and we find for the interior loss

$$\begin{aligned} L_I(u,p) &= \frac{1}{\lambda}\left[L(u,p) - (1-\lambda)L_B(u,p)\right] \\ &\leq= \frac{L(u,p)}{\lambda}, \end{aligned} \tag{118}$$

and for the boundary loss

$$\begin{aligned} L_B(u,p) &= \frac{1}{1-\lambda}\left[L(u,p) - \lambda L_I(u,p)\right] \\ &\leq= \frac{L(u,p)}{1-\lambda}. \end{aligned} \tag{119}$$

For both of these bounds, we can compute for which ϵ the necessary conditions are always satisfied. For the interior loss, the condition is satisfied if

$$\frac{L(u,p)}{\lambda} \leq \epsilon^p \int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x}. \tag{120}$$

this gives us the upper bound for ϵ,

$$\epsilon^p \geq \frac{L(u,p)}{\lambda}\left[\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x}\right]^{-1}. \tag{121}$$

Repeating the same procedure for the boundary loss yields

$$\epsilon^p \geq \frac{L(u,p)}{1-\lambda}\left[\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x}\right]^{-1}. \tag{122}$$

Finally, we simplify these expressions to obtain

$$\begin{aligned} \epsilon &\geq \left[\frac{L(u,p)}{\lambda}\left[\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x}\right]^{-1}\right]^{\frac{1}{p}} \\ &= \left[\frac{\lambda}{L(u,p)}\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x}\right]^{-\frac{1}{p}} \end{aligned} \tag{123}$$

and

$$\begin{aligned} \epsilon &\geq \frac{L(u,p)}{1-\lambda}\left[\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x}\right]^{-\frac{1}{p}} \\ &= \left[\frac{1-\lambda}{L(u,p)}\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x}\right]^{-\frac{1}{p}}. \end{aligned} \tag{124}$$

The smallest ϵ for which both necessary conditions are satisfied, given the value of $L(u,p)$, is thus given by

$$\begin{aligned}\epsilon &= \max\left\{\left[\frac{\lambda}{L(u,p)}\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x}\right]^{-\frac{1}{p}}, \left[\frac{1-\lambda}{L(u,p)}\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p d\boldsymbol{x}\right]^{-\frac{1}{p}}\right\} \\ &= \left(\min\left\{\frac{\lambda}{L(u,p)}\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x}, \frac{1-\lambda}{L(u,p)}\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p d\boldsymbol{x}\right\}\right)^{-\frac{1}{p}}.\end{aligned} \tag{125}$$

This is the expression we want to minimize with respect to λ; the smaller ϵ can be while still satisfying the necessary conditions, the more accurate our solution might be. The smallest maximum is attained when both terms in the minimization in eq. 125 are equal, and thus we obtain for λ,

$$\frac{\lambda}{L(u,p)}\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x} = \frac{1-\lambda}{L(u,p)}\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p d\boldsymbol{x}, \tag{126}$$

which has as solution

$$\lambda = \frac{\int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p d\boldsymbol{x}}{\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})|\right)^p d\boldsymbol{x} + \int_{\partial\Omega} |\hat{u}(\boldsymbol{x})|^p d\boldsymbol{x}}. \tag{127}$$

This solution can be generalized to work for other PDEs by using the terms present in their operators.

Although this gives us more insight in what a proper choice for λ might look like, we rely on knowledge about the true solution. Even when the PDE is known, it is very difficult to estimate the correct values a priori. We can, however, construct iterative methods that approximate these parameters based on the approximated solution. While still treating them as constants, we can update λ after each iteration to reflect our new knowledge. For example, we may define λ at the $i+1$-th iteration as

$$\lambda^{i+1} = \frac{\int_{\partial\Omega} |u^i(\boldsymbol{x})|^p d\boldsymbol{x}}{\int_\Omega \left(|u^i_{xx}(\boldsymbol{x})| + \left|u^i_{yy}(\boldsymbol{x})\right|\right)^p d\boldsymbol{x} + \int_{\partial\Omega} |u^i(\boldsymbol{x})|^p d\boldsymbol{x}}. \tag{128}$$

The full method can be obtained by applying Monte Carlo integration to eq. 128, as described in section 3.2.3. This gives us the approximation for λ,

$$\lambda^{i+1} = \frac{\|\partial\Omega\| \frac{1}{n_B}\sum_{i=1}^{n_B} |u^i(\boldsymbol{x}_{B,i})|^p}{\|\Omega\| \frac{1}{n_I}\sum_{i=1}^{n_I} \left(|u^i_{xx}(\boldsymbol{x}_{I,i})| + \left|u^i_{yy}(\boldsymbol{x}_{I,i})\right|\right)^p + \|\partial\Omega\| \frac{1}{n_B}\sum_{i=1}^{n_B} |u^i(\boldsymbol{x}_{B,i})|^p}. \tag{129}$$

Unfortunately, this method may interact poorly with problems where the network may simultaneously reduce the interior or boundary loss and the corresponding scale factor. The Laplace equation is an example of such a problem, since the zero function solves the interior part of the problem. Simultaneously, the zero function results in a scale factor of zero, which causes the loss weight of eq. 129 to prioritize the interior loss even more. This could create feedback loops, and hence the resulting method may be unstable. This is empirically investigated in section 4.3.3.

3.3.2 Magnitude Normalization

We have defined numerical methods that iteratively update the loss weights based on generalizations of eq. 128. Given constant loss weights, the total loss functional satisfies all of the properties we derived in section 3.1. However, neural networks are well known for their ability to optimize difficult functions. As such, there is no particular reason to keep the loss weights constant. In this section, we will explore the possibilities that open up if we allow the loss weights to be functions.

To this end, let us again consider the Laplace equation given in eq. 111 as an example. For this PDE, equation 128 gives us a rule by which we can choose λ, based on the previous approximation of the solution. Instead of using the previous approximation, we may also use the current approximation. The resulting, scaled loss functional is given by

$$L(u,p) = \frac{L_I(u,p)\int_{\partial\Omega} |u(\boldsymbol{x})|^p \, d\boldsymbol{x} + L_B(u,p)\int_{\Omega} (|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x}}{\int_{\Omega} (|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x} + \int_{\partial\Omega} |u(\boldsymbol{x})|^p \, d\boldsymbol{x}}. \tag{130}$$

Let us analyze some of the properties of this method. There are two different regimes that are of particular interest: $u \approx \hat{u}$, and u far away from $\hat{u}$. To analyze the behavior of eq. 130, we label the terms present in this equation as

$$S_I(u,p) = \int_{\Omega} (|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x} \tag{131}$$

and

$$S_B(u,p) \equiv \int_{\partial\Omega} |u(\boldsymbol{x})|^p \, d\boldsymbol{x}. \tag{132}$$

Then, eq. 130 can be written as

$$L(u,p) = \frac{L_I(u,p)S_B(u,p) + L_B(u,p)S_I(u,p)}{S_I(u,p) + S_B(u,p)}. \tag{133}$$

Starting with the case $u \approx \hat{u}$, it is clear that the scale factors barely change, and are approximately given by

$$S_I(u,p) \approx S_I(\hat{u},p) \tag{134}$$

and

$$S_B(u,p) \approx S_B(\hat{u},p). \tag{135}$$

Thus, in this scenario the loss functional behaves as intended, as the correct loss weight is simulated by these scale factors.

Now consider the case where u is significantly different from $\hat{u}$. In this case, the scale factors cannot be treated as constants. Instead, we will treat them as variables that the network is able to change independently from the loss functionals. This reveals a dangerous property: if there exists a function for which one of the loss terms and its corresponding scale factor are 0, then the entire loss can be reduced to 0 without solving the PDE. For the 2-dimensional Laplace equation, such examples are easy to find: take for example any linear function,

$$u = ax + by. \tag{136}$$

Then, the scale factors and the losses evaluate to

$$S_I(u,p) = \int_\Omega (|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x} = 0, \tag{137}$$

$$L_I(u,p) = \int_\Omega \left|\nabla^2 u(\boldsymbol{x})\right|^p d\boldsymbol{x} = 0, \tag{138}$$

$$S_B(u,p) = \int_{\partial\Omega} |u(\boldsymbol{x})|^p \, d\boldsymbol{x} > 0, \tag{139}$$

$$L_B(u,p) = \int_{\partial\Omega} |u(\boldsymbol{x}) - G(\boldsymbol{x})|^p \, d\boldsymbol{x} > 0, \tag{140}$$

and the loss becomes

$$\begin{aligned} L(u,p) &= \frac{L_I(u,p)S_B(u,p) + L_B(u,p)S_I(u,p)}{S_I(u,p) + S_B(u,p)} \\ &= 0. \end{aligned} \tag{141}$$

Clearly, this function does not satisfy the nonlinear boundary conditions, and thus the PDE is not solved.

We will solve this issue by multiplying the loss functional with another functional that only depends on the scale factors. This should ensure that the nice behaviour close to $\hat{u}$ is preserved, as in this region the scale factors are approximately constant. We are looking for a rescale factor that blows up when either of the scale factors decrease to 0. An example of such a rescale factor would be

$$R(S_I, S_B) = \frac{1}{S_I S_B}. \tag{142}$$

On the other hand, the rescale factor may not decrease too fast when the scale factors increase. With eq. 142, the denominator of the loss functional becomes $S_I S_B(S_I + S_B)$, which has a third order dependency on the scale factors. The numerator has a second order dependency on the scale factors and the loss terms, and hence it becomes possible to reduce the loss asymptotically to 0 by blowing up the solution.

A rescale factor that satisfies both properties is given by

$$R(S_I, S_B) = \frac{S_I + S_B}{S_I S_B}. \tag{143}$$

This rescale factor results in a total loss given by

$$\begin{aligned}
L(u,p) &= \frac{S_I(u,p)+S_B(u,p)}{S_I(u,p)S_B(u,p)}\frac{L_I(u,p)S_B(u,p)+L_B(u,p)S_I(u,p)}{S_I(u,p)+S_B(u,p)} \\
&= \frac{L_I(u,p)S_B(u,p)+L_B(u,p)S_I(u,p)}{S_I(u,p)S_B(u,p)} \\
&= \frac{L_I(u,p)}{S_I(u,p)}+\frac{L_B(u,p)}{S_B(u,p)} \\
&= \frac{\int_\Omega |\nabla^2 u(\boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_\Omega (|u_{xx}(\boldsymbol{x})|+|u_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x}} + \frac{\int_{\partial\Omega} |u(\boldsymbol{x})-G(\boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_{\partial\Omega} |u(\boldsymbol{x})|^p \, d\boldsymbol{x}}.
\end{aligned} \tag{144}$$

This functional has the added benefit that it completely removes all cross terms present in eq. 133, preventing the network from eliminating loss terms by blowing up the scale factors.

The resulting loss functional has a very intuitive interpretation: the terms comprising it represent the relative error. Essentially, the loss terms are normalized by the magnitudes of their components. Therefore, we label this method *magnitude normalization*

In order to convert these loss functionals to loss functions that can be optimized by the algorithms we mentioned before, we again apply Monte Carlo integration. This gives us the scaled loss terms

$$\begin{aligned}
\hat{L}_I(\boldsymbol{\theta},p) &= \frac{\|\Omega\|\frac{1}{n_I}\sum_{i=1}^{n_I} |\nabla^2 u(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|^p}{\|\Omega\|\frac{1}{n_I}\sum_{i=1}^{n_I} (|u_{xx}(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|+|u_{yy}(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|)^p} \\
&= \frac{\sum_{i=1}^{n_I} |\nabla^2 u(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|^p}{\sum_{i=1}^{n_I} (|u_{xx}(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|+|u_{yy}(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|)^p}
\end{aligned} \tag{145}$$

and

$$\begin{aligned}
\hat{L}_B(\boldsymbol{\theta},p) &= \frac{\|\partial\Omega\|\frac{1}{n_B}\sum_{i=1}^{n_B} |u(\boldsymbol{\theta},\boldsymbol{x}_{B,i})-G(\boldsymbol{x}_{B,i})|^p}{\|\partial\Omega\|\frac{1}{n_B}\sum_{i=1}^{n_B} |u(\boldsymbol{\theta},\boldsymbol{x}_{B,i})|^p} \\
&= \frac{\sum_{i=1}^{n_B} |u(\boldsymbol{\theta},\boldsymbol{x}_{B,i})-G(\boldsymbol{x}_{B,i})|^p}{\sum_{i=1}^{n_B} |u(\boldsymbol{\theta},\boldsymbol{x}_{B,i})|^p}.
\end{aligned} \tag{146}$$

Thus, we obtain the family of loss functions

$$\hat{L}(\boldsymbol{\theta},p) = \frac{\sum_{i=1}^{n_I} |\nabla^2 u(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|^p}{\sum_{i=1}^{n_I} (|u_{xx}(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|+|u_{yy}(\boldsymbol{\theta},\boldsymbol{x}_{I,i})|)^p} + \frac{\sum_{i=1}^{n_B} |u(\boldsymbol{\theta},\boldsymbol{x}_{B,i})-G(\boldsymbol{x}_{B,i})|^p}{\sum_{i=1}^{n_B} |u(\boldsymbol{\theta},\boldsymbol{x}_{B,i})|^p}. \tag{147}$$

The next section covers some theoretical properties of magnitude normalized losses.

3.3.3 Properties of Magnitude Normalized Losses

The loss functional defined in eq. 144 serves as a different class of loss functionals, much like the family defined in eq. 58. However, unlike the latter family, the scaled family was not designed with the four properties we discussed in section 3.1.1 in mind. In this section we investigate whether the new functionals satisfy some of those properties, starting with property 3.3.

It is straightforward to see that the magnitude normalized loss functionals satisfy property 3.3 if scalefactors are nonzero; in these cases, the normalized losses evaluate to 0 if and only if u solves the PDE. Thus, the normalized losses have a unique global minimum.

The other property we consider is property 3.4. This property is, unfortunately, straightforward to violate; we will give a counterexample for the boundary value problem stated in eq. 111. Consider the parametrized function

$$u(t, \boldsymbol{x}) = t\hat{u}(\boldsymbol{x}), \tag{148}$$

where $\hat{u}(\boldsymbol{x})$ is the analytical solution of the problem. Clearly, for $t > 1$, this is not a solution of the PDE. However, the loss functional evaluates to

$$\begin{aligned} L(u(t)) &\equiv \frac{\int_\Omega |\nabla^2 u(n, \boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_\Omega (|u_{xx}(n, \boldsymbol{x})| + |u_{yy}(n, \boldsymbol{x})|)^p \, d\boldsymbol{x}} + \frac{\int_{\partial\Omega} |u(n, \boldsymbol{x}) - G(\boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_{\partial\Omega} |u(n, \boldsymbol{x})|^p \, d\boldsymbol{x}} \\ &= \frac{\int_{\partial\Omega} |t\hat{u}(\boldsymbol{x}) - \hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_{\partial\Omega} |t\hat{u}(\boldsymbol{x})|^p \, d\boldsymbol{x}} \\ &= \frac{|t-1|^p}{t^p}, \end{aligned} \tag{149}$$

showing that a functions that lie very far away from the true function may still have a reasonably low loss.

Thus, although magnitude normalized losses still have a unique minimizer that corresponds to the true solution, algorithms using these losses are no longer guaranteed to converge to this minimizer. Therefore, care must be taken when using these functionals instead of the original.

3.3.4 Variance Normalization

It is not straightforward to alter the magnitude normalized loss functions in such a way that they satisfy properties 3.2 and 3.4. One of the implications of the losses not satisfying property 3.4 is that the network may drift away from the true solution to asymptotically reduce the loss function. This may not be a rare occurrence either; when the absolute value of the network output is present in the scale factor, it only takes a single network parameter to blow up the scale factor, as the bias of the output layer serves as an offset to the entire solution. This may result in instabilities.

To address this, one could slightly change the loss function to not depend on the magnitude of the network output, but to instead depend on the variance of the network output.

For the Laplace equation, this results in the family of loss functionals given by

$$L(u,p) = \frac{\int_\Omega |\nabla^2 u(\boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_\Omega (|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})|)^p \, d\boldsymbol{x}} + \frac{\int_{\partial\Omega} |u(\boldsymbol{x}) - G(\boldsymbol{x})|^p \, d\boldsymbol{x}}{\int_{\partial\Omega} |u(\boldsymbol{x}) - \overline{u}|^p \, d\boldsymbol{x}}, \tag{150}$$

where $\overline{u}$ is the mean of u over the corresponding domain, which in this case is given by

$$\overline{u} = \frac{1}{\|\partial\Omega\|} \int_{\partial\Omega} u(\boldsymbol{x}) d\boldsymbol{x}. \tag{151}$$

we label this method *variance normalization.* The resulting loss function is given by

$$\hat{L}(\boldsymbol{\theta},p) = \frac{\sum_{i=1}^{n_I} |\nabla^2 u(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})|^p}{\sum_{i=1}^{n_I} (|u_{xx}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})| + |u_{yy}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})|)^p} + \frac{\sum_{i=1}^{n_B} |u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - G(\boldsymbol{x}_{B,i})|^p}{\sum_{i=1}^{n_B} \left| u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - \frac{1}{n_B} \sum_{i=1}^{n_B} u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) \right|^p}. \tag{152}$$

3.3.5 Gradients Normalization

We have discussed several ways of stimulating the neural network to simultaneously optimize different losses by tuning loss weights or redefining the loss function entirely. The information we have exploited in order to achieve good multi-objective optimization algorithms consists mainly of the features present in the output of the neural network. In particular, we have used the value of this approximation across the domain and its derivatives with respect to the input parameters. We have not used any information regarding the neural network itself. In this section we attempt to use some properties of the neural network itself to obtain different multi-objective optimization methods. We do this with NSGD in mind; the simplicity of this method gives us freedom to use other information available to us.

The methods we derived so far rely on the ability of the neural network to optimize different loss functions equally well. For loss functions involving multiple loss terms, we implicitly made the assumption that a neural network would arrive at a solution where all loss terms involved had similar magnitudes. However, it is not unreasonable to assume that neural networks are not able to learn all functions equally easily. What if it turns out that the boundary conditions of a certain problem are much easier to learn than the solution in the interior of the domain? In cases like this, the loss terms involved may no longer remain similarly sized, in which case our results no longer apply. This suggests that the ability of neural networks to learn certain functions is a useful property to investigate.

Unfortunately, this property itself is not straightforward to analyze. One way of finding out how easily a neural network can learn different functions, is looking at the gradients of the loss of these functions with respect to the network's parameters. If these gradients are very large, then the network is able to achieve significant improvements with small changes in the weights. Conversely, small gradients mean that changes in the network's weights barely improve the approximation. Although the gradients do not entirely correspond to the ability of the network to learn functions, they can give us a decent approximation.

With this in mind we aim to find a method that achieves decent multi-objective optimization based on these gradients. To start, we will consider a PDE with pure boundary conditions, with a loss function that belongs to the family of loss functions given in eq. 58. We will again treat λ, and later γ_i as constants that may be updated in between iterations. Let α denote the learning rate. Recall that SGD is linear in the loss function, i.e. the update vector of the sum of losses is the same as the sum of the update vectors of the individual losses. Therefore, we can express the update vector $\Delta\boldsymbol{\theta}$ as given by NSGD as

$$\begin{aligned}\Delta\boldsymbol{\theta} &= -\alpha \left\|\nabla L\right\|^{-1} \nabla L \\ &= -\alpha \left\|\lambda\nabla L_I + (1-\lambda)\nabla L_B\right\|^{-1} \left(\lambda\nabla L_I + (1-\lambda)\nabla L_B\right) \\ &\equiv -\hat{\alpha} \left(\lambda\nabla L_I + (1-\lambda)\nabla L_B\right),\end{aligned} \tag{153}$$

where $\hat{\alpha}$ is an adjusted learning rate. However, ∇L_I and ∇L_B may point in opposing directions, i.e.

$$\nabla L_I \cdot \nabla L_B < 0. \tag{154}$$

This is exactly the case where problems may arise, since it becomes possible that

$$\Delta\boldsymbol{\theta} \cdot \nabla L_I > 0 \tag{155}$$

or

$$\Delta\boldsymbol{\theta} \cdot \nabla L_B > 0, \tag{156}$$

meaning that one loss is sacrificed to improve the other. However, it turns out that we can choose λ such that both losses are optimize simultaneously. In fact, it is possible to choose λ such that

$$\Delta\boldsymbol{\theta} \cdot \nabla L_I \left\|\nabla L_I\right\|^{-1} = \Delta\boldsymbol{\theta} \cdot \nabla L_B \left\|\nabla L_B\right\|^{-1}, \tag{157}$$

i.e. the weight update takes both loss terms equally into account.

Let us work out the value of λ for which this is the case. By eq. 157, it must hold that

$$\Delta\boldsymbol{\theta} \cdot \nabla L_I \left\|\nabla L_B\right\| = \Delta\boldsymbol{\theta} \cdot \nabla L_B \left\|\nabla L_I\right\|. \tag{158}$$

This means that

$$\left(\lambda\nabla L_I + (1-\lambda)\nabla L_B\right) \cdot \nabla L_I \left\|\nabla L_B\right\| = \left(\lambda\nabla L_I + (1-\lambda)\nabla L_B\right) \cdot \nabla L_B \left\|\nabla L_I\right\|. \tag{159}$$

After performing some basic arithmetic operations, we obtain

$$\lambda\nabla L_I \cdot (\nabla L_I \left\|\nabla L_B\right\| - \nabla L_B \left\|\nabla L_I\right\|) = (1-\lambda)\nabla L_B \cdot (\nabla L_B \left\|\nabla L_I\right\| - \nabla L_I \left\|\nabla L_B\right\|), \tag{160}$$

which has as solution

$$\begin{aligned}\lambda &= \frac{\nabla L_B \cdot (\nabla L_B \|\nabla L_I\| - \nabla L_I \|\nabla L_B\|)}{\nabla L_B \cdot (\nabla L_B \|\nabla L_I\| - \nabla L_I \|\nabla L_B\|) + \nabla L_I \cdot (\nabla L_I \|\nabla L_B\| - \nabla L_B \|\nabla L_I\|)} \\ &= \frac{\|\nabla L_B\| (\|\nabla L_I\| \|\nabla L_B\| - \nabla L_I \cdot \nabla L_B)}{\|\nabla L_B\| (\|\nabla L_I\| \|\nabla L_B\| - \nabla L_I \cdot \nabla L_B) + \|\nabla L_I\| (\|\nabla L_I\| \|\nabla L_B\| - \nabla L_I \cdot \nabla L_B)} \\ &= \frac{\|\nabla L_B\|}{\|\nabla L_B\| + \|\nabla L_I\|}. \end{aligned} \tag{161}$$

Thus, we obtain an update vector given by

$$\begin{aligned}\Delta\boldsymbol{\theta} &= -\hat{\alpha}\left(\frac{\|\nabla L_B\|\nabla L_I}{\|\nabla L_B\| + \|\nabla L_I\|} + \frac{\|\nabla L_I\|\nabla L_B}{\|\nabla L_B\| + \|\nabla L_I\|}\right) \\ &= -\alpha \left\| \frac{\|\nabla L_B\|\nabla L_I + \|\nabla L_I\|\nabla L_B}{\|\nabla L_B\| + \|\nabla L_I\|} \right\|^{-1} \frac{\|\nabla L_B\|\nabla L_I + \|\nabla L_I\|\nabla L_B}{\|\nabla L_B\| + \|\nabla L_I\|} \\ &= -\alpha \left\| \|\nabla L_B\| \nabla L_I + \|\nabla L_I\| \nabla L_B \right\|^{-1} (\|\nabla L_B\| \nabla L_I + \|\nabla L_I\| \nabla L_B)\,. \end{aligned} \tag{162}$$

We call this method *Gradients Normalized SGD*, which we abbreviate to GNSGD.

It is important to note that this update step does not always exist: it turns out that if the gradients ∇L_I and ∇L_B are pointing in the exact opposite direction, i.e.

$$\nabla L_I \cdot \nabla L_B = -\|\nabla L_I\| \|\nabla L_B\|\,, \tag{163}$$

then the normalization step will fail, since the vector that needs to be normalized has magnitude 0. To see this, consider the normalization term of eq. 162, which we will label M. We have

$$\begin{aligned} M &= \left\| \|\nabla L_B\| \nabla L_I + \|\nabla L_I\| \nabla L_B \right\| \\ &= \sqrt{(\|\nabla L_B\| \nabla L_I + \|\nabla L_I\| \nabla L_B) \cdot (\|\nabla L_B\| \nabla L_I + \|\nabla L_I\| \nabla L_B)} \\ &= \sqrt{2\|\nabla L_B\|^2 \|\nabla L_I\|^2 + 2\|\nabla L_I\| \|\nabla L_B\| \nabla L_I \cdot \nabla L_B} \\ &= 0. \end{aligned} \tag{164}$$

In this case, it is not possible to improve either of the loss terms any further without sacrificing the other. Cases like these define global extrema or saddle points, which are problematic for regular SGD as well. However, given the sheer number of parameters present in neural networks, these cases should be incredibly rare. Therefore, we will not address this problem in detail.

Now that we have a method that, in theory, is able to optimize two loss functions simultaneously, the obvious question is whether it is possible to generalize this to n loss functions. More formally, given a loss function of the form

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{k} \gamma_i L_i(u), \tag{165}$$

we wish to find a vector $\Delta\boldsymbol{\theta}$ such that

$$\Delta\boldsymbol{\theta} \cdot \nabla L_i \left\|\nabla L_i\right\|^{-1} = \Delta\boldsymbol{\theta} \cdot \nabla L_j \left\|\nabla L_j\right\|^{-1} < 0 \text{ for } i, j \in \{1, \dots, k\}. \tag{166}$$

It turns out that this is not always possible without making assumptions. Problems arise for cases that are similar to the scenario described in eq. 163. By assuming that the vectors ∇L_i are independent, scenarios like these can be avoided entirely. We again stress that such assumptions are not unreasonable to make as long as the number of different loss functions k is much smaller than the dimension of $\boldsymbol{\theta}$, n.

Under this assumption, it does become possible to find such a vector $\Delta\boldsymbol{\theta}$. There is, however, one other property that $\Delta\boldsymbol{\theta}$ should satisfy: it should be a linear combination of the vectors $\{\nabla L_i\}$. The reason this property is important, is that it ensures that the resulting vector $\Delta\boldsymbol{\theta}$ does not contain any components that are orthogonal to all the gradient vectors. In other words, $\Delta\boldsymbol{\theta}$ contains no redundant parts.

By restricting the vector $\Delta\boldsymbol{\theta}$ to be a linear combination of $\{\nabla L_i\}$, i.e.

$$\Delta\boldsymbol{\theta} = \sum_{i=1}^{k} a_i \nabla L_i, \tag{167}$$

we are able to obtain a unique solution for which eq. 166 is satisfied. To this end, let us solve eq. 166 for $\{a_i\}$. Without loss of generality, we may scale $\Delta\boldsymbol{\theta}$ such that eq. 166 becomes

$$\Delta\boldsymbol{\theta} \cdot \nabla L_i \left\|\nabla L_i\right\|^{-1} = -1 \text{ for } i \in \{1, \dots, k\}. \tag{168}$$

Let us label the gradients as

$$\boldsymbol{v}_i \equiv \nabla L_i. \tag{169}$$

With this notation, eqs. 167 and 168 can be transformed into matrix-vector products. This yields

$$\Delta\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_k \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix} \tag{170}$$

and

$$\begin{bmatrix} \|\boldsymbol{v}_1\|^{-1} & 0 & \dots \\ 0 & \ddots & 0 \\ \dots & 0 & \|\boldsymbol{v}_k\|^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^T \\ \vdots \\ \boldsymbol{v}_k^T \end{bmatrix} \Delta\boldsymbol{\theta} = \begin{bmatrix} -1 \\ \vdots \\ -1 \end{bmatrix}. \tag{171}$$

Then, for a_i it holds that

$$\left(\begin{bmatrix} \|\boldsymbol{v}_1\|^{-1} & 0 & \dots \\ 0 & \ddots & 0 \\ \dots & 0 & \|\boldsymbol{v}_k\|^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1^T \\ \vdots \\ \boldsymbol{v}_k^T \end{bmatrix} \right) \left(\begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_k \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix} \right) = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}. \tag{172}$$

Solving this system yields

$$\begin{aligned}
\begin{bmatrix} a_1 \\ \vdots \\ a_k \end{bmatrix} &= \left(\begin{bmatrix} \|\boldsymbol{v}_1\|^{-1} & 0 & \dots \\ 0 & \ddots & 0 \\ \dots & 0 & \|\boldsymbol{v}_k\|^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_k \end{bmatrix}^T \begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_k \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \\
&= \left(\begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_k \end{bmatrix}^T \begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_k \end{bmatrix} \right)^{-1} \begin{bmatrix} \|\boldsymbol{v}_1\| & 0 & \dots \\ 0 & \ddots & 0 \\ \dots & 0 & \|\boldsymbol{v}_k\| \end{bmatrix} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}.
\end{aligned} \tag{173}$$

Note that this inverse exists, because we assumed $\boldsymbol{v}_i$ to be independent. To verify that these results line up with the 2-dimensional case, we plug in two gradients $\boldsymbol{v}_i$ to obtain

$$\begin{aligned}
\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} &= \left(\begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{v}_1 & \boldsymbol{v}_2 \end{bmatrix} \right)^{-1} \begin{bmatrix} \|\boldsymbol{v}_1\| & 0 \\ 0 & \|\boldsymbol{v}_k\| \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \|\boldsymbol{v}_1\|^2 & \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \\ \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 & \|\boldsymbol{v}_2\|^2 \end{bmatrix}^{-1} \begin{bmatrix} \|\boldsymbol{v}_1\| \\ \|\boldsymbol{v}_2\| \end{bmatrix} \\
&= c \begin{bmatrix} \|\boldsymbol{v}_2\|^2 & -\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \\ -\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 & \|\boldsymbol{v}_1\|^2 \end{bmatrix} \begin{bmatrix} \|\boldsymbol{v}_1\| \\ \|\boldsymbol{v}_2\| \end{bmatrix} \\
&= c \begin{bmatrix} \|\boldsymbol{v}_1\|\|\boldsymbol{v}_2\|^2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \|\boldsymbol{v}_2\| \\ \|\boldsymbol{v}_1\|^2\|\boldsymbol{v}_2\| - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \|\boldsymbol{v}_1\| \end{bmatrix}.
\end{aligned} \tag{174}$$

Here, we have introduced the constant c to simplify the result. Because the final result needs to be normalized, the vector $\boldsymbol{a}$ can be scaled without affecting the results, and thus the exact value of c does not matter. With this in mind, we can incorporate additional terms into the constant c to find

$$\begin{aligned}
\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} &= c \begin{bmatrix} \|\boldsymbol{v}_1\|\|\boldsymbol{v}_2\|^2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \|\boldsymbol{v}_2\| \\ \|\boldsymbol{v}_1\|^2\|\boldsymbol{v}_2\| - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \|\boldsymbol{v}_1\| \end{bmatrix} \\
&= c \begin{bmatrix} \|\boldsymbol{v}_2\| \left(\|\boldsymbol{v}_1\|\|\boldsymbol{v}_2\| - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \right) \\ \|\boldsymbol{v}_1\| \left(\|\boldsymbol{v}_1\|\|\boldsymbol{v}_2\| - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 \right) \end{bmatrix} \\
&= c \begin{bmatrix} \|\boldsymbol{v}_2\| \\ \|\boldsymbol{v}_1\| \end{bmatrix},
\end{aligned} \tag{175}$$

confirming that both methods yield the same results if the gradients are independent.

4 Empirical Method Analysis

This chapter covers the empirical analysis of the methods developed in the study of [8] and the modifications proposed in chapter 3. Throughout this chapter, we solve problems that belong to a family of boundary value problems of the Laplace equation. This family of problems is described in section 4.1. The analysis of the methods covers two aspects; in section 4.2, we analyze the hyperparameters present in these methods, with the aim of finding appropriate parameters for solving PDEs in general. Then, in section 4.3, we analyze the behaviour of the different methods. Here we show that the original methods can yield very poor results, even on relatively simple problems. We also show that our proposed modifications can significantly improve these results.

4.1 A Family of Test Problems

This section discusses a family of test problems for the Laplace equation. There are several reasons why the Laplace equation is particularly useful to perform this initial analysis. As already mentioned, the Laplace equation is well-posed and linear, and therefore it should have the theoretical properties that guarantee that the methods work under idealized circumstances. Besides this, the Laplace equation is one of the simples PDEs, forming the foundation of many more complicated PDEs. Because the eigenfunctions of the Laplace equation span $L_2(\Omega)$, this PDE gives us freedom to alter the problems to make them more difficult. Problems can also be made more difficult by increasing the dimensionality, which is a topic we investigate in section 5.1.

Because the one-dimensional Laplace equation only has linear solutions, we will consider two-dimensional problems. We restrict ourselves to the rectangular domain given by $(x, y) \in [0, 1] \times [0, 1]$. Furthermore, we will only be using pure Dirichlet boundary conditions. Thus, the test problems are of the form

$$\begin{aligned} \nabla^2 u(x,y) &= 0 && \text{for } (x,y) \in [0,1]\times[0,1], \\ u(x,y) &= G(x,y) && \text{on } \partial\left([0,1]\times[0,1]\right). \end{aligned} \tag{176}$$

To simplify the mathematics involved in analyzing the results, we will focus on boundary value problems corresponding to the eigenfunctions of the Laplace equation. To recap, these eigenfunctions are given by

$$\begin{aligned} u(x,y) &= 1, & u(x,y) &= x, \\ u(x,y) &= y, & u(x,y) &= xy, \\ u(x,y) &= e^{\omega y}\sin\omega x, & u(x,y) &= e^{-\omega y}\sin\omega x, \\ u(x,y) &= e^{\omega x}\sin\omega y, & u(x,y) &= e^{-\omega x}\sin\omega y, \\ u(x,y) &= e^{\omega y}\cos\omega x, & u(x,y) &= e^{-\omega y}\cos\omega x, \\ u(x,y) &= e^{\omega x}\cos\omega y, & u(x,y) &= e^{-\omega x}\cos\omega y. \end{aligned} \tag{177}$$

The derivation of these eigenfunctions can be found in section 2.1.1. The eigenfunctions $1, x, y$ and xy are very simple, and should pose no challenge for neural networks to find. The sine and cosine terms, however, may turn out to be much harder, especially if their frequency ω is increased.

Many of these eigenfunctions can be mapped to one another by applying a linear transformation to the input coordinates. Since neural networks can perform such transformations by changing their weights, it should suffice to examine only a single one of these eigenfunctions. In the remainder of this section, we will therefore focus on problems corresponding to the eigenfunctions

$$u(x, y) = \sin(\omega y)e^{-\omega x}. \tag{178}$$

The eigenfunction with frequency $\omega = 6\pi$ is depicted in fig. 1.

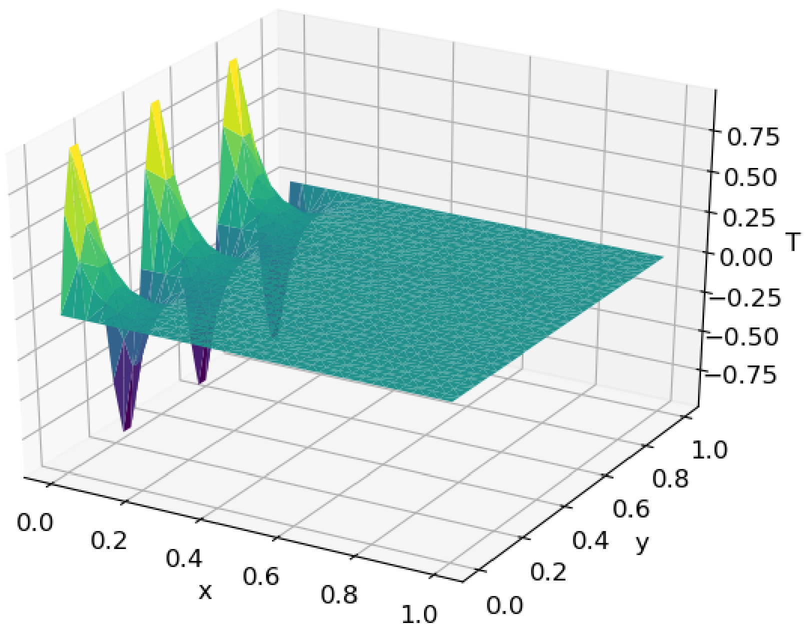


Figure 1: One of the eigenfunctions that we will be using throughout this chapter to analyze the methods. The frequency can be easily adjusted. Frequencies that are a multiple of π have three homogeneous boundary conditions, and frequencies that are multiples of 2π have zero mean.

Note that frequencies that are multiples of 2π are particularly convenient to use, as these result in three homogeneous boundaries and one inhomogeneous boundary with mean zero. We will mainly use the frequencies $\omega = 2\pi$, $\omega = 4\pi$ and $\omega = 6\pi$ to test our methods. These frequencies should cover a decent range of difficulty.

4.2 Hyperparameter Optimization

This section covers the optimization of the hyperparameters present in the methods. Because the number of hyperparameters is large, it is infeasible to optimize all of them simultaneously. Therefore, we will treat several groups of hyperparameters separately.

To this end, we start by optimizing the hyperparameters related to the neural network architecture in section 4.2.1. These hyperparameters include the number of layers, the number of neurons, the weight initialization and the activation function. This optimization is done by the training neural networks directly to approximate the solutions of the test problems. Once the networks are established, we consider the individual optimization algorithms and optimize their parameters. Section 4.2.2 covers the hyperparameters present in L-BFGS, while section 4.2.3 covers the hyperparameters of NSGD. Section 4.2.4 covers the hyperparameters of Adam. Finally, section 4.2.5 summarizes the sets of hyperparameters that we will use by default for each method.

4.2.1 Network Configuration

We start by determining suitable network configurations for our purposes. There are several different aspects that need to be considered. The reason the network configuration is important, is that neural networks only have finite capacity in practice. Although our methods rely heavily on the universal approximation theorems discussed in section 2.2.2, neural networks with a fixed size cannot achieve arbitrary accuracy. Thus, the networks we use need to have sufficient capacity. On the other hand, the computational cost of training neural networks increases with their size, so we need to find a good compromise between capacity and speed.

As mentioned, this chapter focuses entirely on the eigenfunctions of the Laplace equation. Thus, the networks must have sufficient capacity to learn these eigenfunctions. Unfortunately, the only way to find out whether a network has sufficient capacity to learn a certain function, is to train the network to approximate this function and measure the accuracy. Thus, the hyperparameters related to the network configuration are inherently coupled to the hyperparameters of the training algorithms.

To address this coupling, we will use L-BFGS to perform the initial training. This method only depends on a single hyperparameter, namely the number of collocation points. For L-BFGS this parameter is important, as this method does not allow the collocation points to be changed mid training, since that would invalidate the approximation of the Hessian. Hence, the collocation points chosen should be representative of the full solution. Choosing too few points will lead to overfitting, while choosing too many will slow the algorithm down without improving the results significantly. This parameter seems to be strongly related to Nyquist's sampling criterion. For our initial training, performance is not all that important, as the aim of this section is to probe the capabilities of the neural networks we are training. By using a very large number of collocation points, results should become independent of the collocation points, and only reflect the capabilities of the networks.

To probe the capabilities of the neural networks, given that the solutions of the test problems we will use are the eigenfunctions given in eq. 178, we will train the networks to directly approximate these eigenfunctions. We do this by employing the loss function

$$\hat{L}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(u(\boldsymbol{\theta}, x_i, y_i) - \hat{u}(x_i, y_i)\right)^2, \tag{179}$$

where $\hat{u}$ is the function we aim to approximate, given by

$$\hat{u}(x, y) = \sin(\omega y)e^{-\omega x} \tag{180}$$

for some ω. The collocation points are sampled uniformly on Ω. To investigate whether there exists a relationship between network size and maximum frequency, we will train networks with different configurations to approximate the eigenfunctions corresponding to $\omega = 2\pi$, $\omega = 4\pi$ and $\omega = 6\pi$.

For a given frequency ω, the spatial resolution in one dimension requires at least 2ω uniform sample points to be able to uniquely resolve the underlying continuous function. Since we are dealing with a two-dimensional problem, the number of points required to satisfy this lower bound is given by

$$n \geq (2\omega)^2. \tag{181}$$

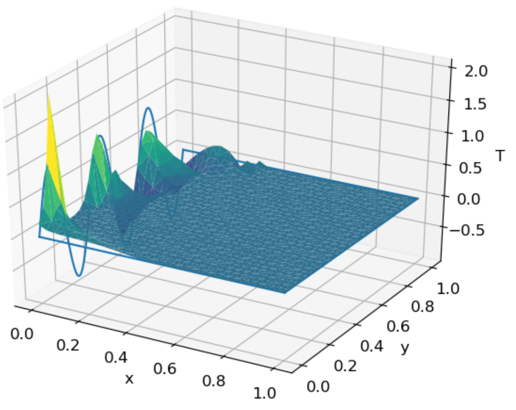
For the frequencies of interest, that means at least 1421 collocation points are needed to learn the solutions. Because we use random instead of uniformly spaced collocation points, we will stay clear from this lower bound and use 8192 collocation points. We will test networks with hidden layer counts ranging between 2 and 4, and neuron counts ranging between 5 and 50. We will use the hyperbolic tangent activation function, because this is a popular sigmoid activation function that is sufficiently differentiable, and therefore a good candidate activation function for learning the solution of PDEs. We initialize the network weights using the well established Glorot initialization. The resulting mean square errors are given in table 1.

Layers	Neurons $\diagdown\,\omega$	2π	4π	6π
2	5	1.57e-5	9.39e-5	8.95e-3
	15	2.18e-6	1.26e-4	4.14e-3
	50	5.50e-6	4.26e-6	5.83e-5
3	5	7.15e-6	1.00e-5	5.12e-3
	15	5.79e-6	2.03e-5	1.24e-5
	50	4.42e-6	7.37e-6	4.90e-6
4	5	2.41e-6	6.62e-5	4.76e-5
	15	3.02e-6	8.23e-6	2.82e-5
	50	5.98e-6	1.20e-5	4.00e-6

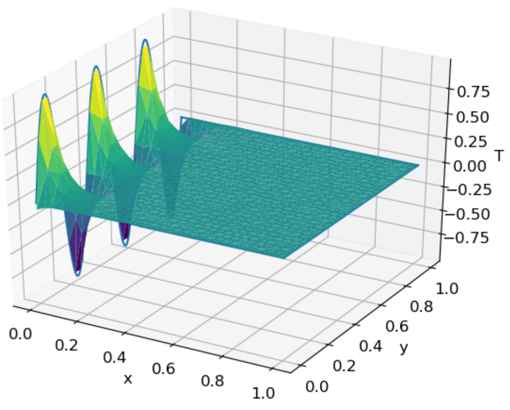
Table 1: Mean square error after training a network with L-BFGS with 8192 collocation points for different eigenfrequencies. These results show that even very small networks are able to accurately learn low frequency solutions. To learn higher frequencies, larger networks are required.

These results establish that the larger networks we tested have plenty of capacity to learn these eigenfunctions, and thus later results should not be limited by the network capacity. The results also show that in general, larger networks achieve better performance. However, the accuracy shows asymptotic behaviour; poor approximations can be significantly improved by increasing the network size, but once a mean square error around 10^{-5} has been achieved, adding additional neurons or layers does not seem to affect the results. Essentially, networks have to be sufficiently large to learn a certain function, and once they are sufficiently large their exact size does no longer matter. Determining this lower bound remains a challenge.

To give some more insight in the capabilities of these networks, we plotted the best and worst approximations of the solution with frequency $\omega = 6\pi$. The approximation achieved by the network with 2 hidden layers of 5 neurons is depicted in fig. 2a, and for the network with 4 layers of 50 neurons the approximation is depicted in fig. 2b. The approximation of the smaller network deviates visibly from the true solution, while the approximation of the larger network appears almost fully accurate. In general, mean square errors of the order 10^{-2} indicate that the network was unable to learn the characteristics of the solution, while errors smaller than 10^{-4} indicate that the network did manage to learn the general shape of the solution.



(a)



(b)

Figure 2: Comparison between the accuracy after training a network with 2 layers of 5 neurons, depicted in fig. 2a, and after training a network with 4 layers of 50 neurons, depicted in fig. 2b. The smaller network did not capture all the peaks that are present in the eigenfunction, which is depicted in fig. 1. On the other hand, the approximation by the larger network does not deviate visibly from the true solution.

Next, we investigate the activation functions. To this end, we again learn the eigenfunction with frequency $\omega = 6\pi$ directly using L-BFGS, this time only using networks with 4 layers of 50 neurons. We specifically compare hyperbolic tangent, standard logistic, sine, cosine and arctangent activation functions. The mean square errors and the number of iterations required to train the networks are given in table 2.

Activation function	Hyperbolic tangent	Arctangent	Logistic	Sine	Cosine
MSE	4.00e-6	6.25e-6	1.32e-2	4.84e-6	5.49e-6

Table 2: Comparison of the accuracy for hyperbolic tangent, standard logistic, sine, cosine and arctangent activation functions. The logistic function paired with Glorot initialization was unable to create the pronounced curves that are present in this eigenfunction, resulting in an approximation close to the zero function. All other activation functions resulted in much more accurate results.

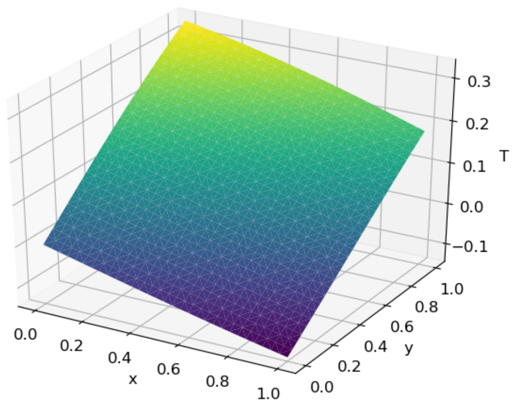
Although the logistic function resulted in very poor approximations, the other activation functions all yielded much better results. Since none of them stood out as better than the others, we will stick to using the hyperbolic tangent function because of its popularity in the literature.

To conclude this section, we investigate the final hyperparameter: the weight initialization scheme. Specifically, we compare the initialization scheme derived in section 3.2.4 to the well known Glorot initialization. One would hope that the initialization does not affect the accuracy at all, but instead affects the number of iterations required to train the networks. We compare both values for the three eigenfunctions in table 3.

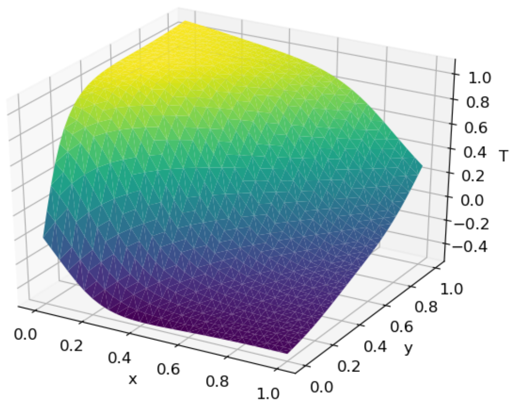
Initialization	ω	2π	4π	6π
Glorot	MSE	5.98e-6	1.20e-5	4.00e-6
	Iterations	1085	2189	2655
Proposed	MSE	5.09e-6	2.19e-6	3.49e-6
	Iterations	344	852	1767

Table 3: Comparison between the proposed initialization scheme and Glorot initialization. Both schemes result in similar accuracy, but the proposed scheme requires significantly fewer iterations.

To understand where this difference in the required iterations comes from, we compare the network outputs for the different schemes in fig. 3. This shows that our proposed method does cover the nonlinear range of the activation function, while Glorot initialization does not.



(a)



(b)

Figure 3: Comparison between two networks initialized using the different schemes. Fig. 3a shows Glorot initialization, while fig. 3b shows the scheme proposed in section 3.2.4. The nonlinearity of the proposed method is clearly visible, while Glorot initialization is approximately linear.

This section fully establishes the network configuration; networks with 4 layers of 50 neurons using hyperbolic tangent activation functions have the capacity that is required to further analyze the methods. Although our initialization method is promising, we will use Glorot initialization unless specified otherwise. The reason for this is that we aim to obtain results that are as general as possible. Glorot initialization, albeit suboptimal for the small neural networks that we used in this thesis, has been developed to initialize networks of any size.

4.2.2 L-BFGS

In this section we treat the hyperparameters related to L-BFGS. As mentioned in the previous section, one hyperparameter is given by the number of collocation points. The only other hyperparameters involved in L-BFGS is given by the number of iterations that are kept in memory. This parameter is tricky to optimize because its optimum depends on the nature of the Hessian; in the literature, this number is usually chosen larger than 10. We will use the number 50 to ensure that the approximation of the Hessian is accurate.

That leaves only the number of collocation points to be optimized. Increasing the number of collocation points improves the Monte Carlo approximation of the loss functional, and simultaneously increases the computational cost. This computational cost scales linearly with the number of collocation points, although iterations also contain computations that do not depend on the number of collocation points.

Because the computational cost depends on this hyperparameter, our aim is to choose the number of collocation points as low as possible without significantly decreasing the accuracy. To this end, we again directly approximate the eigenfunctions with frequency $\omega = 2\pi$, $\omega = 4\pi$ by minimizing the loss function given in eq. 179. One would expect that choosing too few collocation points not only leads to a worse approximation, but also leads to overfitting, i.e. a much lower loss at the collocation points in comparison to the rest of the domain. To measure how much the network overfits, we will also compute the ratio between the mean square error and the mean loss at the collocation points. When this ratio is close to 1, the network is not overfitting. The mean square errors and this overfitting measure are given in table 4.

ω	2π		4π		6π	
Batch Size	MSE	Overfitting	MSE	Overfitting	MSE	Overfitting
32	2.18e-1	1.12e4	9.51e-2	1.66e4	3.94	7.67e6
128	2.97e-5	2.66	1.33e-2	3.69e3	7.16e-3	2.80e3
512	1.43e-5	2.14	3.60e-5	1.90	1.09e-2	6.34e1
2048	1.66e-5	1.44	1.49e-5	2.05	1.33e-5	2.09
8192	5.98e-6	1.79	1.20e-5	1.79	4.00e-6	1.45

Table 4: Results of training a neural network with 4 layers of 50 neurons with L-BFGS for different batch sizes and for different problem frequencies. It is clear that higher frequencies require more data; too few points results in severe overfitting, which we measured by the ratio between the MSE and the training loss.

These results show that it is crucial that the number of collocation points is high enough. The Nyquist rate does not exactly line up with our results, but does seem to give a rough estimate for the minimum number of points required. Unfortunately, computing this rate requires knowledge about the true solution, which may not be available when one is solving a PDE.

We remark that for the configurations tested, the majority of the computational cost of L-BFGS came from the approximation of the Hessian; increasing the number of collocation points from 512 total points to 2048 points did not significantly increase the time required per iteration. When using L-BFGS, we will therefore generally use very large batch sizes, usually consisting of at least 2048 collocation points.

4.2.3 NSGD

Next, let us consider NSGD. As mentioned, this method has more hyperparameters that need to be considered, namely the number of iterations, batch sizes and the learning rate. To analyze the influence of these parameters on the results, we will again train neural networks to approximate the eigenfunctions of the Laplace equation given in eq. 180. We will use the same neural network as we did when analyzing L-BFGS, i.e. a network consisting of 4 hidden layers of 50 neurons, with the hyperbolic tangent as the activation function. Furthermore, we will use Glorot initialization.

Before optimizing the hyperparameters, we note that there exists a strong relation between the learning rate and the batch size; for SGD, doubling the batch size is essentially equivalent to performing two iterations with half the learning rate. Due to the normalization in NSGD, this equivalence does no longer hold exactly, but the same idea applies. Thus, we expect similar behaviour from increasing the batch size and decreasing the learning rate, allowing us to effectively eliminate one of the three hyperparameters. The remaining hyperparameters are the learning rate and the number of iterations, both of which are described by the learning rate profile.

We will consider learning rate profiles that decay exponentially. These profiles have been well established in the literature, and intuitively make sense; as the neural network learns the target function, more accurate representations of the true solution are needed to make further improvements, and this in turn can be implemented by reducing the learning rate. For completeness, a learning rate profile with initial learning rate α_0 and final learning rate $\alpha_1 \leq \alpha_0$ that decays exponentially over k iterations is given by

$$\alpha(n) = \alpha_0 \left(\frac{\alpha_1}{\alpha_0}\right)^{n/k}. \tag{182}$$

We will perform anywhere between 5,000 and 100,000 iterations, with learning rates in the range $[0.001, 0.1]$, and 128 collocation points per batch, which we update every iteration. The resulting mean square errors for the frequency $\omega = 2\pi$ are given in table 5, and for

$\omega = 6\pi$ in table 6. For the latter table, we included a column with 200,000 iterations, since a more difficult problem warrants using more iterations. This was not needed for the problem with $\omega = 2\pi$, since some profiles reached results that are comparable to the results of L-BFGS.

Profile \ Iterations	5,000	20,000	50,000	100,000
$\alpha \in [0.1, 0.1]$	4.94e-3	2.95e-3	4.35e-3	2.19e-3
$\alpha \in [0.1, 0.01]$	5.73e-4	2.13e-4	7.42e-5	4.53e-5
$\alpha \in [0.1, 0.001]$	4.71e-4	6.99e-5	1.25e-5	4.45e-6
$\alpha \in [0.01, 0.01]$	1.66e-2	8.80e-4	5.41e-4	2.73e-4
$\alpha \in [0.01, 0.001]$	1.81e-2	2.23e-4	1.09e-4	5.05e-5
$\alpha \in [0.001, 0.001]$	2.19e-2	1.64e-2	1.72e-4	8.59e-5

Table 5: Mean square errors for $\omega = 2\pi$. Results clearly show that performing more iterations tends to yield better results, with improvements being more significant when there is large learning rate decay. This makes sense, since small learning rates require more iterations to cover the same distance in weight space.

Profile \ Iterations	5,000	20,000	50,000	100,000	200,000
$\alpha \in [0.1, 0.1]$	1.59e-2	1.52e-2	1.34e-2	1.27e-2	2.88e-3
$\alpha \in [0.1, 0.01]$	1.32e-2	1.23e-2	9.82e-3	4.43e-4	8.42e-5
$\alpha \in [0.1, 0.001]$	1.31e-2	1.27e-2	1.17e-2	1.61e-3	3.66e-5
$\alpha \in [0.01, 0.01]$	1.33e-2	1.30e-2	1.20e-2	1.35e-3	4.07e-4
$\alpha \in [0.01, 0.001]$	1.29e-2	1.28e-2	1.24e-2	1.06e-2	4.49e-4
$\alpha \in [0.001, 0.001]$	1.30e-2	1.29e-2	1.30e-2	1.26e-2	1.19e-2

Table 6: Mean square errors for $\omega = 6\pi$. Although results are generally worse than for $\omega = 2\pi$, the same learning rate profiles perform well. This higher frequency solution has a higher Nyquist rate, and hence requires more data points, which translates to a lower learning rate. This is likely the reason why large learning rate profiles show the biggest performance differences when compared to the results in table 5.

We notice a strong trend that suggests that stretching learning rate profiles over more iterations improves the results. The strength of this trend seems to be correlated to the decay of the learning rate, i.e. if the learning rate decays significantly, then one should also use many iterations. This phenomenon can likely be explained by looking at the loss surface with respect to the weight space: the optimal set of weights may be located far away from the initial weights, and thus a large distance may need to be covered. Starting with a learning rate that is too small will prevent the weights from reaching this optimum. Once the local optimum has been reached, however, smaller steps may be needed to refine the solution; this would correspond to a narrow minimum in weight space. A narrow minimum that lies far away from the initial weights would fully explain the results in both table 5 and table 6.

4.2.4 Adam

This section covers the optimization of the hyperparameters of Adam. There are essentially four hyperparameters: the batch size, the learning rate, and two exponential decay parameters that govern the moving average of the first and second moments of the gradients, labeled by β_1 and β_2 respectively. However, just like for NSGD, larger batches are equivalent to smaller learning rates. Therefore, instead of optimizing the batch size and learning rate separately, we use a fixed 128 collocation points and optimize the learning rate.

This leaves three parameters to be considered. The authors of [1] recommend the default settings $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We will use these parameters as a starting point to optimize the learning rate. Then, we proceed to optimize the parameters β_1 and β_2 using the optimized learning rate. To optimize the learing rate, we again learn the eigenfunction with frequency $\omega = 6\pi$ by optimizing eq. 179. Since Adam performs some sort of learning rate annealing by itself, there is no need to decay the learning rate any further. Thus, we only consider constant learning rates. The results are given in table 7.

Learning Rate \ Iterations	5,000	20,000	50,000	100,000
$\alpha = 0.1$	3.55e-2	6.42e-2	3.23e-2	2.22e-2
$\alpha = 0.05$	1.20e-2	1.36e-2	1.84e-2	1.42e-2
$\alpha = 0.01$	1.31e-2	1.21e-2	1.43e-2	1.35e-2
$\alpha = 0.005$	1.31e-2	1.09e-3	9.50e-5	2.83e-5
$\alpha = 0.001$	1.07e-2	1.92e-4	3.12e-5	1.23e-5
$\alpha = 0.0005$	1.16e-2	1.25e-4	9.89e-6	1.91e-5

Table 7: Mean square errors for $\omega = 6\pi$ using $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Note that the results are very sensitive to the learning rate; learning rates that are slightly too large can result in mean square errors that are orders of magnitude larger.

Clearly, Adam requires much smaller learning rates than SGD. This likely indicates that the momentum that this method uses requires somewhat consistent updates. The learning rate α = 5e-4 outperformed the other learning rates for any number of iterations. This is the learning rate that we will use.

Next, we consider the decay parameters β_1 and β_2. To optimize these parameters, we use the learning rate $\alpha = 10^{-3}$. The optimization is done by solving the same problem as before. The results are given in table 8. These results show that Adam is not very sensitive to these parameters. Because the default parameters defined in [1] resulted in the most accurate results, we will stick to these values, i.e. use $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

β_1 \ β_2	0.9	0.99	0.999	0.9999
0.5	1.10e-4	1.88e-4	1.34e-4	1.00e-4
0.9	1.57e-5	1.85e-5	9.89e-6	1.87e-5
0.95	1.94e-5	2.31e-5	1.91e-5	1.28e-5
0.99	1.06e-3	3.55e-5	9.99e-6	1.10e-5
0.999	3.37e-2	5.27e-5	2.63e-5	1.61e-5

Table 8: Mean square errors for $\omega = 6\pi$ using the learning rate α =5e-4 for various β_1 and β_2. The default parameters defined by the authors of [1] yielded the best results.

These results establish default parameters for Adam. The next section combines all optimal hyperparameters for each method into labeled sets of default parameters.

4.2.5 Default Hyperparameters

We have optimized the hyperparameters related to the network configuration in section 4.2.1, and the hyperparameters of the three training algorithms in sections 4.2.2, 4.2.3 and 4.2.4. This section recaps these results and labels default sets of hyperparameters for each of the methods, allowing us to specify the test setups used in later tests more concisely.

To this end, we start by labeling the default network configuration we will use; section 4.2.1 has shown that larger networks generally outperform smaller networks. The default network architecture we will use consists of 4 layers of 50 neurons and uses hyperbolic tangent activation functions and Glorot initialization.

The default set of hyperparameters for L-BFGS, which we label as *default L-BFGS*, contains this default network configuration and uses 8192 interior collocation points. For problems that also have boundary loss terms, default L-BFGS uses 1024 boundary points.

For NSGD, we label the set of default hyperparameters as *default NSGD*. This set uses the default network configuration, and uses batches containing 256 interior collocation points and 256 boundary collocation points. Default NSGD uses learning rates that decay exponentially from $\alpha = 10^{-1}$ to $\alpha = 10^{-3}$ over the course of 200,000 iterations.

Finally, We label the default set of hyperparameters for Adam as *default Adam*. This set of hyperparameters also includes the default network setup, and uses batches containing 256 interior collocation points and 256 boundary collocation points. Furthermore, default Adam uses the learning rate $\alpha = 5\text{e-}4$, and the decay parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Default Adam performs 100,000 iterations.

4.3 Physics Informed Neural Networks

In this section we consider the family of test problems defined in eq. 176, which we aim to solve without providing the network with the analytical solution as we have done in the previous section. The aim of this section is to investigate the behaviour of the methods developed in [8] and the proposed modifications to gain a better understanding of their strengths and weaknesses. Since the family of test problems contains a parameter by which the difficulty can be increased, we will use this parameter to probe the capabilities of the mentioned methods. We will make heavy use of the results of the previous section, as these results give us a performance benchmark for neural networks which we can use to assess the accuracy of both the original methods and our proposed modifications.

This section is structured as follows; in section 4.3.1, we analyze the original physics informed neural networks as defined in [8]. Section 4.3.2 treats the optimization of the loss weight, following the methods derived in section 3.3.1. Section 4.3.3 briefly covers an iterative method that approximates this optimum. Section 4.3.4 covers the heuristic methods that approximate the optimum based on the magnitudes and variances of the network output. Finally, in section 4.3.5 we discuss the method of gradients normalization.

4.3.1 Default Loss Weight

We start by analyzing the original physics informed neural networks. As mentioned earlier, these methods can be obtained from our generalizion in eq. 105 by setting $\lambda = \frac{1}{2}$ and using $p = 2$. Applying this generalization to the family of test problems yields the family of loss functions given by

$$\begin{aligned}\hat{L}(\boldsymbol{\theta}, p) &= \lambda \frac{1}{n_I} \sum_{i=1}^{n_I} |N_I(\boldsymbol{x}_{I,i}, u(\boldsymbol{\theta}))|^p + (1-\lambda) \frac{1}{n_B} \sum_{i=1}^{n_B} |N_B(\boldsymbol{x}_{B,i}, u(\boldsymbol{\theta}))|^p \\ &= \frac{\lambda}{n_I} \sum_{i=1}^{n_I} |u_{xx}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i}) + u_{yy}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})|^p + \frac{1\ \lambda}{n_B} \sum_{i=1}^{n_B} |u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - \hat{u}(\boldsymbol{x}_{B,i})|^p.\end{aligned} \tag{183}$$

Because we aim to minimize the mean square error of the solution, it makes sense to minimize the loss function for $p = 2$, and thus we will use the loss function

$$\hat{L}(\boldsymbol{\theta}) = \frac{\lambda}{n_I} \sum_{i=1}^{n_I} \left(u_{xx}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i}) + u_{yy}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})\right)^2 + \frac{1-\lambda}{n_B} \sum_{i=1}^{n_B} \left(u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - \hat{u}(\boldsymbol{x}_{B,i})\right)^2. \tag{184}$$

Substituting $\lambda = \frac{1}{2}$ into this loss function yields the loss function that corresponds to the original physics informed neural networks, given by

$$\hat{L}(\boldsymbol{\theta}) = \frac{1}{2n_I}\sum_{i=1}^{n_I}\left(u_{xx}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i}) + u_{yy}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})\right)^2 + \frac{1}{2n_B}\sum_{i=1}^{n_B}\left(u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - \hat{u}(\boldsymbol{x}_{B,i})\right)^2. \quad (185)$$

We will examine the results that this default choice would give; we stress that we are attempting to solve one of the simplest PDEs available, and thus one would expect decent results. In order to exploit the results from section 4.2, we will focus on testing the method for the frequencies $\omega = 2\pi$, $\omega = 4\pi$ and $\omega = 6\pi$. Since L-BFGS achieved the best results for these particular eigenfunctions, and since L-BFGS has the smallest number of hyperparameters that could negatively affect the results, we will use default L-BFGS so that we are best able to compare the results of a physics informed neural network to the results of the same networks directly learning the solution. We will consider NSGD and Adam in later sections. The mean square errors for the different frequencies after training the physics informed neural network are given in table 9.

ω	Interior Loss	Boundary Loss	MSE
2π	6.20e-5	1.07e-5	1.62e-6
4π	2.87e-3	3.08e-3	5.26e-4
6π	4.72e-3	9.72e-2	1.05e-2

Table 9: Mean square errors and losses after training a physics informed network to minimize the loss function given in eq. 184 for different frequencies. The mean square errors grow rapidly as the frequency increases.

These results do not line up well with observations from section 4.2. Although there we also observed that increasing the eigenfrequency made it harder for networks to learn the solution, here we see a much faster degradation of the performance when the frequency is increased. In particular, the approximated solution for $\omega = 6\pi$ deviates significantly from the true solution, even though the network has sufficient capacity, as we showed in section 4.2.

There are two possible explanations for this phenomenon: the network's capacity may not be leveraged properly, i.e. there are ways to further reduce the loss, or the network's capacity might be used for the wrong purposes, i.e. the local minimum of the loss function aligns poorly with our goals. Since we used the default choice $\lambda = \frac{1}{2}$, one would expect that the latter explanation applies here. This does, however, not rule out the former. Therefore, we will examine both explanations, starting with the former.

Examining whether the network's capacity is being leveraged properly is somewhat tricky; the main tool we have to investigate this, is evaluating the loss function on networks that were trained differently. This way, confirming that the capacity is not being used to its full extend is straightforward, but showing that the capacity is indeed fully leveraged is difficult. However, if we can show that it is possible to achieve good results without demanding much more from the neural network, then that should be sufficient evidence that the training algorithm itself is not the issue, and that a poor correspondence between the local minimum and our goals is the only cause of these poor results.

To this end, let us take a closer look at the outputs of the physics informed neural networks compared to the outputs of neural networks that were trained to directly approximate the analytical solution. If the problem at hand is related to a poor utilization of the network capacity, then one would expect that the loss of eq. 185 evaluates lower for neural networks trained to directly approximate the solution. Table 10 gives the values of these losses evaluated on the neural networks trained in the previous section using L-BFGS. We only compare networks with 4 layers of 50 neurons.

ω	Interior Loss	Boundary Loss	MSE
2π	1.04e1	5.30e-5	5.98e-6
4π	2.75e2	1.63e-4	1.20e-5
6π	1.80e2	7.74e-5	4.00e-6

Table 10: Mean square error and losses given in eq. 184 evaluated after training a neural network to minimize the loss function given in eq. 179.

Table 10 reveals what is going on. Although the direct approximation is very accurate, this accuracy is achieved in ways that are impossible for the physics informed neural network to replicate, since the physics informed neural network is forced to spend its resources on optimizing the interior loss.

The obvious next step is to investigate what happens if we again train a network on the analytical solution, but this time also provide the network with information about the derivatives. In particular, since the PDE is defined by the second derivative, using a loss function of the form

$$\begin{aligned}\hat{L}(\boldsymbol{\theta}) &= \frac{\lambda}{n}\sum_{i=1}^{n}\left(u(\boldsymbol{\theta}, x_i, y_i) - \hat{u}(\boldsymbol{x}_i, y_i)\right)^2 \\ &+ \frac{1-\lambda}{n}\sum_{i=1}^{n}\left(u_{xx}(\boldsymbol{\theta}, x_i, y_i) - \hat{u}_{xx}(x_i, y_i)\right)^2 + \left(u_{yy}(\boldsymbol{\theta}, x_i, y_i) - \hat{u}_{yy}(x_i, y_i)\right)^2,\end{aligned} \tag{186}$$

with λ a constant in $[0, 1]$ might yield fruitful results. However, here we run into the same issues again; choosing the proper value for λ is not straightforward. By using the default choice $\lambda = \frac{1}{2}$, we obtained the results given in table 11. This table shows not only the mean square errors, but also the loss defined in eq. 185 evaluated on the trained networks.

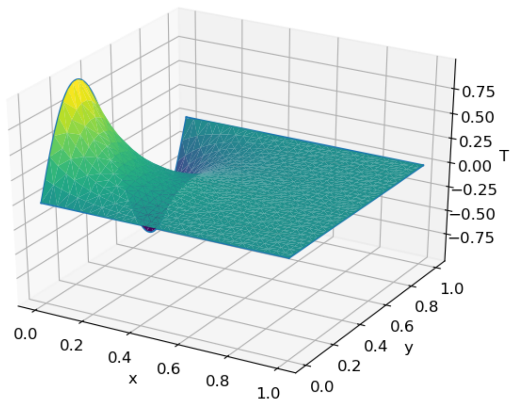
ω	Interior Loss	Boundary Loss	MSE
2π	9.33e-6	2.42e-7	8.21e-8
4π	1.14e-3	2.56e-5	1.06e-5
6π	1.59e-2	1.59e-4	8.25e-5

Table 11: Mean square error and losses given in eq. 184 evaluated after training a neural network to minimize the loss function given in eq. 186. Although the mean square errors are exceptionally low, the losses of eq. 184 are not.

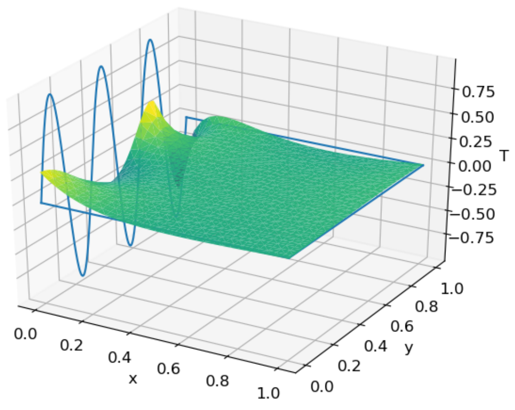
Comparing the results in table 11 to the results in table 9 shows that after training the network to approximate the true solution and its second derivatives, the losses are not much lower than after training to minimize the PDE losses, hinting that the network capacity is not the issue here. More importantly, even when we provide the network with the correct second derivatives, the PDE loss does not reduce below the PDE loss of a network that was trained to minimize the PDE loss. This implies that the network capacity not being fully leveraged cannot be the main cause of these problems.

We remark that the approximation for the frequency $\omega = 2\pi$ is more accurate than the approximation after directly learning the solution. Apparently, letting the network approximate the solution and its derivatives simultaneously can have benefits for the overall accuracy. This phenomenon is further explored in the study of [17].

Now that we have ruled out a poor utilization of the network capacity as a possible cause for the poor results, let us examine whether the performance degradation may be caused by a weak correspondence between the local minimum of the loss function and the analytical solution. To gain some insight in the relation between the network output and the analytical solution, we plotted the outputs of the physics informed neural networks for $\omega = 2\pi$ and $\omega = 6\pi$ in figs. 4a and 4b respectively.



(a)



(b)

Figure 4: The outputs of physics informed networks after training them to minimize the loss function given in eq. 184. The boundary conditions are drawn in blue. For $\omega = 2\pi$, the results are quite accurate, and comparable to directly learning the solution. On the other hand, for $\omega = 6\pi$, the output does not even resemble the analytical solution.

Visually, it appears that for the network that was trained to solve the problem with frequency $\omega = 6\pi$, depicted in fig. 4b, the bulk of the error originates from the boundary; the output of the network closely resembles the solution of a different boundary value problem.

This implies that the loss weight is indeed causing our problems. With the default loss weights, higher frequency problems cause the network to prioritize accurately satisfying the operator of the PDE over satisfying boundary conditions. On the other hand, the default loss weight works quite well for low frequency problems. We suspect that the problems that were used to test the methods in [8] all work relatively well with the default loss weight; in practice, the loss weight deserves more attention.

4.3.2 Loss Weight Optimization

In section 4.3.1, we showed that the basic physics informed network with the default loss weight choice, $\lambda = \frac{1}{2}$, yields poor results for higher frequency problems, even when the neural networks used have sufficient capacity to learn the solution. In this section, we investigate whether alternate choices for the loss weight could solve these issues. To this end, we will empirically optimize the loss weight, and compare the results to the theoretical optimum that we derived in section 3.3.1. We will again employ the Laplace eigenfunction problems to perform our tests. We emphasize that throughout this section, we will treat the loss weights as constants.

Recall that the theoretical optimum of the loss weight is given in eq. 127. This equation only depends on the true solution, and hence one would expect that the optimal value is dependent on the eigenfrequency. Therefore, we will perform tests for all three eigenfrequences we have discussed so far. We will use the same networks as we did in section 4.3.1, and compare the mean square error after training for various λ using default L-BFGS. The results are given in table 12.

λ \ ω	2π	4π	6π
5e-1	1.62e-6	2.56e-3	1.05e-2
5e-2	9.08e-8	6.51e-6	1.36e-4
5e-3	5.76e-8	4.76e-6	1.30e-4
5e-4	1.06e-6	2.71e-7	2.68e-7
5e-5	3.62e-6	2.85e-7	6.04e-7
5e-6	2.44e-4	4.42e-5	3.28e-6
5e-7	3.37e-4	1.38e-4	5.47e-5

Table 12: The mean square error after training with different loss weights, for different frequencies. Note that the optimal loss weight decreases as the frequency increases. also note that the best accuracy achieved by the physics informed network exceeds the accuracy that is achieved by regular networks directly learning the solution.

These results show that physics informed networks are able to achieve highly accurate results; with the correct loss weight, they may even outperform neural networks that are trained to directly approximate the analytical solution. Note that higher frequency problems seem to become more sensitive to the loss weight.

To compare how these results relate to the theoretical optimum, we first evaluate eq. 127 by simply filling in the analytical solution. For the sake of simplicity, we will assume that $\omega = k\pi$ with $k \in \mathbb{N}$. Recall that the analytical solution is given by

$$\hat{u}(x,y) = \sin(\omega x)e^{-\omega y}. \tag{187}$$

After applying eq. 127 to the test problem, we find

$$\begin{aligned}
\int_{\partial\Omega} (\hat{u}(\boldsymbol{x}))^2 \, d\boldsymbol{x} &= \int_0^1 (\hat{u}(x,0))^2 + (\hat{u}(x,1))^2 \, dx + \int_0^1 (\hat{u}(0,y))^2 + (\hat{u}(1,y))^2 \, dy \\
&= \int_0^1 (\sin(\omega x))^2 + \left(\sin(\omega x)e^{-\omega}\right)^2 dx \\
&= \left(1 + e^{-2\omega}\right) \int_0^1 (\sin(\omega x))^2 \, dx \\
&= \frac{1}{2}\left(1 + e^{-2\omega}\right) \approx \frac{1}{2}
\end{aligned} \tag{188}$$

for the boundary, and

$$\begin{aligned}\int_{\Omega}\left(|\hat{u}_{xx}(\boldsymbol{x})|+|\hat{u}_{yy}(\boldsymbol{x})|\right)^2 d\boldsymbol{x} &= \int_0^1\int_0^1 \left(|\omega^2\sin(\omega x)e^{-\omega y}|+|\omega^2\sin(\omega x)e^{-\omega y}|\right)^2 dxdy \\ &= \int_0^1 4\omega^4 e^{-2\omega y}\int_0^1 (\sin(\omega x))^2\, dxdy \\ &= 2\omega^4\int_0^1 e^{-2\omega y}dy \\ &= 2\omega^4\frac{1-e^{-2\omega}}{2\omega} \\ &= \omega^3\left(1-e^{-2\omega}\right)\approx\omega^3\end{aligned} \tag{189}$$

for the interior. Combining these results gives us

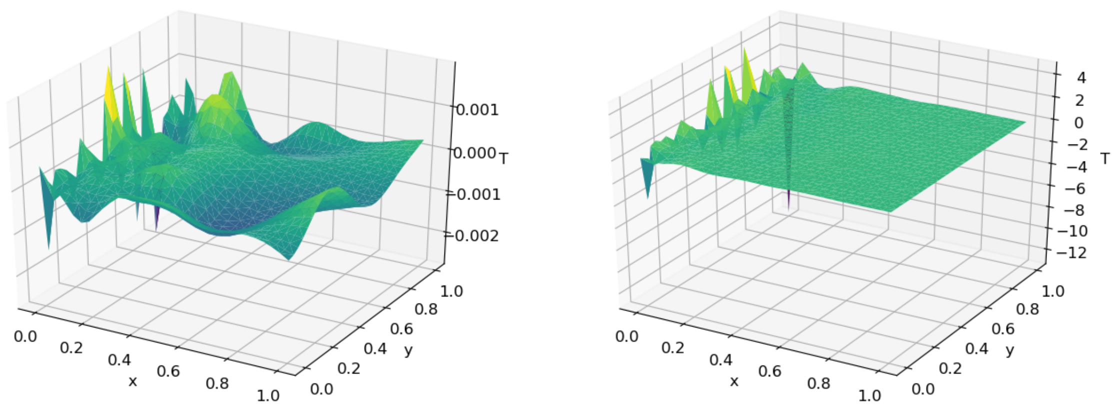
$$\begin{aligned}\lambda &= \frac{\frac{1}{2}\left(1+e^{-2\omega}\right)}{\omega^3\left(1-e^{-2\omega}\right)+\frac{1}{2}\left(1+e^{-2\omega}\right)} \\ &= \frac{\left(1+e^{-2\omega}\right)}{2\omega^3\left(1-e^{-2\omega}\right)+\left(1+e^{-2\omega}\right)}\approx\frac{1}{2\omega^3+1}.\end{aligned} \tag{190}$$

As expected, this equation does indeed strongly depend on ω. In table 13, we give the theoretical optimum and the empirical optimum for different frequencies.

ω	2π	4π	6π
Empirical Optimum	5.0e-3	5.0e-4	5.0e-4
Theoretical Optimum	2.0e-3	2.5e-4	7.5e-5

Table 13: The empirically and theoretically optimal loss weights for different frequencies of the Laplace eigenfunction problems. The theory lines up nicely with the empirical results.

We remark that the derivation of the theoretical optimum of the loss weight depended on certain assumptions. In particular, we assumed that the error of a neural network ends up similarly shaped to the true solution. The results lining up so well with the theory suggests that these assumptions hold up. We will examine this more closely by training a physics informed network with the theoretically optimal loss weight, using default L-BFGS. In fig. 5a, we plotted the absolute error, and in fig. 5b, we plotted the second derivative of this error with respect to x.



(a) (b)

Figure 5: The absolute error and the second derivative of the error to x. The magnitude of the error of the second derivative is highly correlated with the magnitude of the second derivative of the analytical solution; the error is almost negligible far away from the boundary at $x = 0$. The absolute error seems much more evenly distributed and uncorrelated to the analytical solution.

Note that the derivative of the error is equivalent to the error of the derivative. Clearly, neither the absolute error nor the second derivative of the error with respect to x are shaped similarly to the analytical solution. Thus, our assumptions do not hold up. However, there seems to be a strong connection between the shape of the second derivative of the analytical solution and the distribution of the error of the second derivative: this error is concentrated precisely in the area where the second derivative of the analytical solution is also large, namely along the boundary at $x = 0$. We suspect that this means that there exists a set of assumptions which do not contradict our observations, that yields optimal loss weights similar to the ones given in eq. 190.

With the optimal loss weights for L-BFGS established, let us examine the effect of these weights on the results for L-BFGS, NSGD and Adam. Using default NSGD and default Adam, we obtained the results given in table 14.

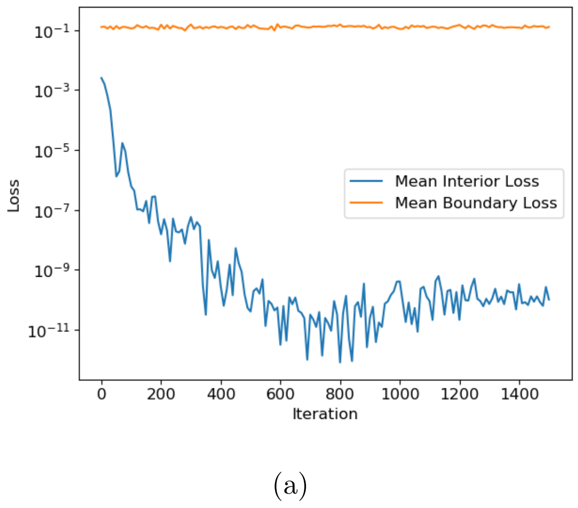
Method \ ω	2π	4π	6π
L-BFGS	3.38e-8	7.36e-7	3.18e-6
NSGD	9.73e-7	9.34e-7	6.81e-6
Adam	2.33e-6	7.61e-7	1.01e-6

Table 14: Mean square errors after training the physics informed neural networks with optimal loss weights for L-BFGS, NSGD and Adam.

It is clear that choosing the proper loss weight can improve the results by orders of magnitude. Unfortunately, determining the optimum the way we did in this section requires explicit knowledge about the analytical solution, which is generally not available when one is interested in using numerical methods to solve PDEs. In the next sections, we consider methods that do not rely on this information.

4.3.3 Iterative Loss Weight Updates

In the previous sections we verified that the theory regarding the loss weight as derived in section 3.3.1 behaves as expected. In this section we move from constant loss weights to loss weights that can be modified in between iterations. We test the method developed in the second half of section 3.3.1, and iteratively update the loss weight using eq. 129. Since L-BFGS does not allow the loss function to be modified during training, iterative loss weights can only be used with NSGD and Adam. The behaviour of the interior and boundary loss when using default NSGD is depicted in fig. 6, as well as the ratio $\frac{\lambda}{1-\lambda}$.



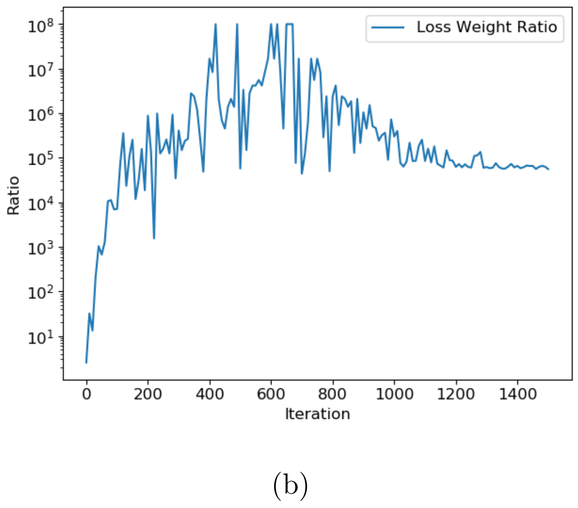


Figure 6: The behaviour of the interior and boundary loss when using iterative loss weight updates is depicted in fig. 6a. Fig. 6b shows the ratio between λ and $1-\lambda$, i.e. the ratio between the interior and boundary weights. It takes roughly 1500 iterations for the gradients to completely vanish, halting the training process.

The results in fig. 6 show that the method is unstable. One way of reducing the interior loss when solving the Laplace equation is to flatten the solution. However, this also results in a very small interior magnitude, which results in large loss weights. In turn, this causes the network to focus less on the boundary and more on the interior loss, creating a feedback loop. After about 1500 iterations, the network output became so flat that the gradients could no longer accurately be computed, effectively stagnating the training process.

We suspect that this scheme is not unstable for every PDE. However, when a loss function and the corresponding magnitude present in the update rule can be minimized simultaneously, instabilities do seem likely to arise.

4.3.4 Magnitude and Variance Normalization

In this section, we analyze the methods we derived in sections 3.3.2 and 3.3.4. In these methods we no longer treat the loss weight as a constant, but instead define the loss weight as a function of the network output, and optimize the total loss function including the loss weight. Our theoretical results suggest that there are two methods we should consider here: magnitude normalized and variance normalized. Since these two methods are very similar, we will start by employing both of them to solve the Laplace eigenfunction problem, so that we can compare their performance. Next, we will investigate whether variance normalization is indeed more stable than magnitude normalization, since the instability of magnitude normalization is the sole reason we constructed variance normalization. Lastly, we examine the behaviour of the neural networks as they optimize these normalized loss functions.

Let us first construct the loss function for these two methods. We will again use the norm $p = 2$. Substituting this into eq. 147 gives us the magnitude normalized loss function

$$\hat{L}(\boldsymbol{\theta}) = \frac{\sum_{i=1}^{n_I} \left(\nabla^2 u(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})\right)^2}{\sum_{i=1}^{n_I} \left(|u_{xx}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})| + |u_{yy}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})|\right)^2} + \frac{\sum_{i=1}^{n_B} \left(u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - G(\boldsymbol{x}_{B,i})\right)^2}{\sum_{i=1}^{n_B} \left(u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i})\right)^2} \tag{191}$$

Similarly, substituting $p = 2$ into eq. 152 yields the variance normalized loss function

$$\begin{aligned}\hat{L}(\boldsymbol{\theta}) = &\frac{\sum_{i=1}^{n_I} \left(\nabla^2 u(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})\right)^2}{\sum_{i=1}^{n_I} \left(|u_{xx}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})| + |u_{yy}(\boldsymbol{\theta}, \boldsymbol{x}_{I,i})|\right)^2} \\ &+ \frac{\sum_{i=1}^{n_B} \left(u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - G(\boldsymbol{x}_{B,i})\right)^2}{\sum_{i=1}^{n_B} \left|u(\boldsymbol{\theta}, \boldsymbol{x}_{B,i}) - \sum_{j=1}^{n_I} u(\boldsymbol{\theta}, \boldsymbol{x}_{B,j})\right|^2}.\end{aligned} \tag{192}$$

We again start our analysis using default L-BFGS. The mean square errors resulting from methods for the Laplace eigenvalue problems are given in table 15 for $\omega = 2\pi$, $\omega = 4\pi$ and $\omega = 6\pi$.

Method \ ω	2π	4π	6π
Magnitude Normalized	2.10e3	1.06e-6	1.56e-6
Variance Normalized	1.71e-8	3.63e-6	7.94e-7

Table 15: Mean square errors after minimizing magnitude normalized and variance normalized loss functions using default L-BFGS for different frequencies. Note that for $\omega = 2\pi$, the magnitude normalized loss yielded the worst approximation we have encountered so far; this has to do with the instability that we mentioned earlier. Besides this instability, both method achieved highly accurate results.

The results given in table 15 are very accurate, except for the instability that occurred. Especially the variance normalized losses lead to an accuracy that was similar to the accuracy we achieved by optimizing the loss weight. In fact, the accuracy these methods achieved surpasses the accuracy we obtained after directly learning the solution, given in table 4. We consider these to be very positive results. Although optimizing these normalized losses is computationally more demanding than optimizing the regular PDE losses with constant loss weights due to the more expensive loss functions, the need to optimize the loss weight has been completely eliminated. Optimizing the loss weight programatically likely requires training several neural networks using different loss weights, which is a process that rapidly outpaces normalized training in terms of computational cost, since normalized training seems to be about twice as costly as regular training.

The results are not entirely positive, however. For $\omega = 2\pi$, magnitude normalization resulted in the solution drifting away from the analytical solution in order to increase the scale factors. Although variance normalization did not suffer from this problem, it still deserves some attention. To gain some insight in why this drifting occurred, we plotted the output of the trained neural network for $\omega = 2\pi$ in fig. 7.

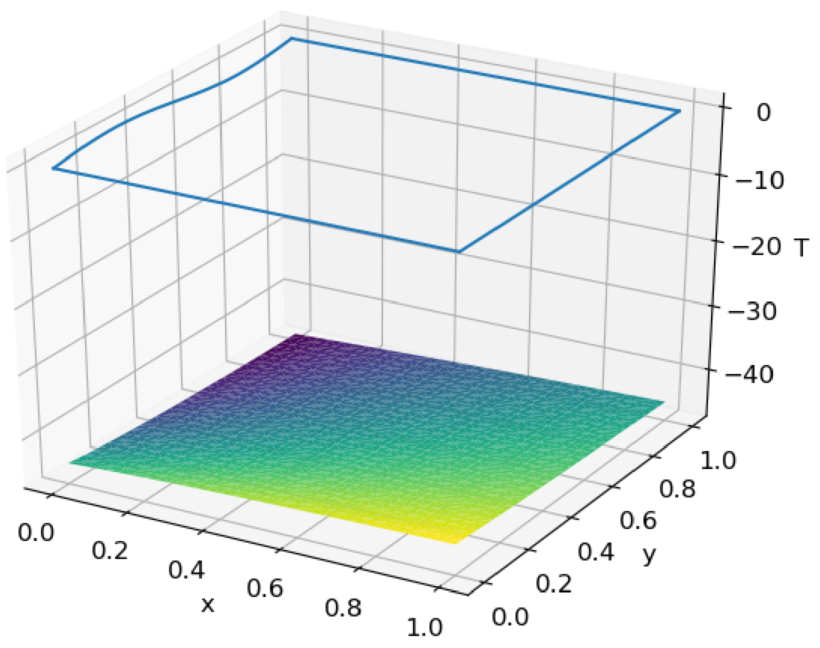


Figure 7: The output after minimizing magnitude normalized losses for $\omega = 2\pi$. In an attempt to reduce the scale factor faster than the losses would be increased, the solution has drifted away from the origin.

With the random seed we used to initialize the weights, it seems that the output of the neural network was largely on the opposite side of the origin compared to the boundary conditions. The normalized boundary loss therefore exceeded 1. Because of this, in order to approximate the boundary conditions accurately, the output of the neural network would have to cross the boundary, which would result in a very small denominator of eq. 191. Thus, the only viable way to minimize the loss function was drifting away from the origin. In the process, the loss asymptotically dropped to 1.

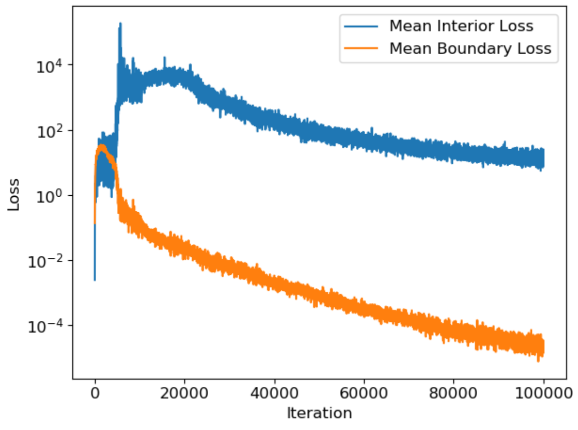
It is fairly straightforward to replicate the mathematics going on behind this process. Consider the parametrized function $u(t, \boldsymbol{x}) = -t$. This function is constant for any t, and for $t = -45$ it is similarly shaped to the output of the neural network after training. For this parametrized function, the boundary loss is given by

$$\begin{aligned} L_B(u(t)) &= \frac{\int_{\partial\Omega} \left(u(t,\boldsymbol{x}) - G(\boldsymbol{x})\right)^2 d\boldsymbol{x}}{\int_{\partial\Omega} \left(u(t,\boldsymbol{x})\right)^2 d\boldsymbol{x}} \\ &= \frac{\int_0^1 \left(-t - \sin(2\pi\boldsymbol{x})\right)^2 d\boldsymbol{x}}{\int_{\partial\Omega} t^2 d\boldsymbol{x}} \\ &= \frac{\frac{1}{2} + 4t^2}{4t^2} \\ &= 1 + \frac{1}{8t^2}. \end{aligned} \tag{193}$$

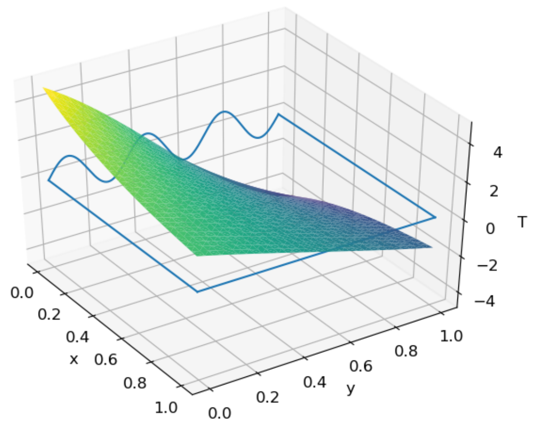
It is clear that this function approaches 1 as t increases. We note that the scenario that this parametrized function illustrates is not unlikely to occur; neural networks are generally able to replicate this parametrization almost perfectly, as the biases in the last layer serve as an offset for the entire output. This shows why the method is so unstable. If the output of a network is, on average, on the opposite side of the origin compared to the boundary conditions, then all it takes to reduce the loss is drifting away from the origin, which only requires modifying a single bias.

We stress that although we encountered an unstable initialization, instabilities do not always occur for $\omega = 2\pi$. After changing the random seed, the method managed to converge to a much more accurate result, reaching a loss similar to variance normalization.

To conclude this section, we investigate the behaviour of variance normalization during training. To this end, we measured the interior and boundary loss functions given in eq. 184 while training a neural network to minimize eq. 192 using default NSGD. These results are given in fig. 8a. The network output after 2,000 iterations is depicted in fig. 8b.



(a)



(b)

Figure 8: The unscaled interior and boundary losses evaluated during the minimization of the variance normalized losses using NSGD are depicted in fig. 8a. Fig. 8b shows the network output after 2,000 iterations, i.e. when the interior and boundary losses peaked.

This shows that variance normalization leads to some interesting behaviour; where the other methods all lead to network outputs that slowly transform from their initial shape to the shape of the solution, variance normalization causes the outputs to blow up initially. This happens in part because variance normalization penalizes the relative error, i.e. larger outputs are not necessarily bad. However, the network output overshoots the boundary conditions by a large margin, and this likely happens because property 3.4 can be violated. Fortunately, the asymptotic nature of this violation eventually steers the solution into the right direction.

4.3.5 Gradients Normalization

So far we have only considered modifying the loss function to deal with the multi-objective optimization problem. These modifications all depended entirely on the loss function itself. In this section, we analyze the method we developed in section 3.3.5, which also takes the neural network into consideration. This method considers the gradients of the loss terms with respect to the network parameters, and combines the update steps belonging to each of the loss terms in such a way that all losses are optimized equally. We will investigate whether this property is attainable in practice, and we will examine the accuracy of the resulting network output. These methods are incompatible with L-BFGS and Adam, and hence we will only consider NSGD in this section.

Since the eigenfunction problems we have considered so far consist of multiple loss terms, they are well suited problems to perform the analysis on. We only consider the eigenfrequency $\omega = 6\pi$ in this section. To verify that both loss terms are indeed optimized simultaneously, we will run the algorithm with a single batch of training data, and plot the evolution of these loss terms. If the learning rate is small enough, the graphs of both loss terms should be monotonic. On the other hand, leaning rates that are too large might result in losses that increase. Using 128 interior and 128 boundary points and a constant learning rate given by $\alpha = 10^{-4}$ resulted in the losses depicted in fig. 9

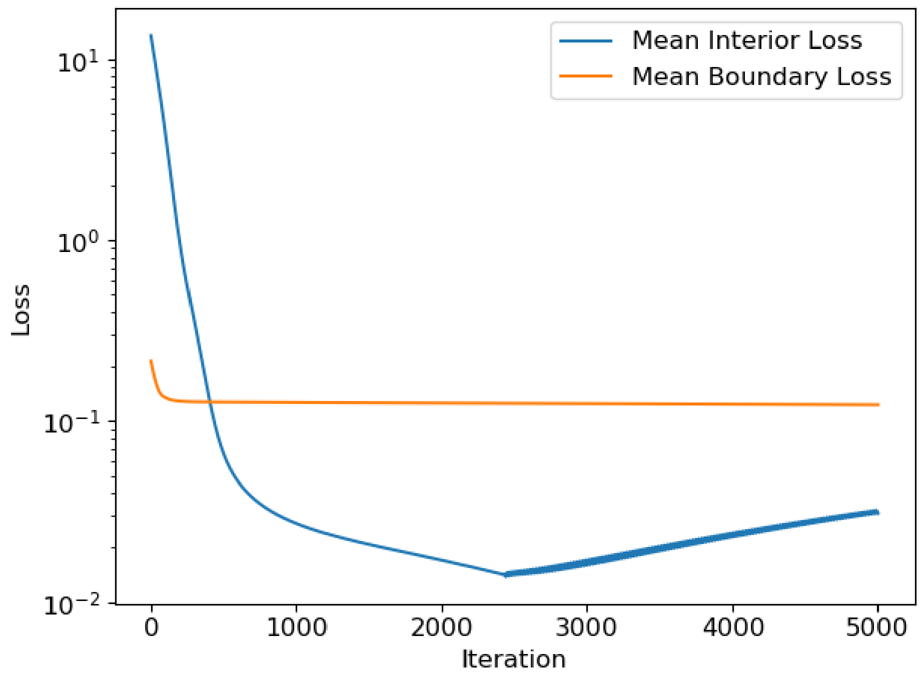


Figure 9: The interior and boundary loss during training with GNSGD, using a single batch and a constant learning rate given by $\alpha = 10^{-4}$. Although both losses showed monotonic behaviour initially, even this very small learning rate was not small enough to ensure monotonicity for the first 2,500 iterations.

This clearly shows that using this algorithm to ensure monotonicity of all involved loss terms is highly impractical, as it would require the learning rate to be far too small. That does not rule out the usefulness of this algorithm, however.

Since this algorithm only works in theory if the batches are not refreshed, we will solve the boundary value problem with different batch update rates, using more reasonable learning rates that decay exponentially from 5e-2 to 5e-4 over the course of 100,000 iterations. The results are given in table 16.

Iterations Per Batch	1	10	100	1,000
MSE	1.10e-2	8.98e-3	2.08e-5	4.81e-5
Interior Loss	8.09e-3	9.42e-1	2.50	1.11e1
Boundary Loss	8.63e-2	7.70e-2	2.81e-5	1.50e-5

Table 16: Results of GNSGD for different batch refresh rates. This parameter greatly affects the results for this method, unlike the methods previously tested.

The behaviour of the interior and the boundary loss when using 100 iterations per batch is depicted in fig. 10. Even though both losses are prioritized equally by the algorithm, the interior loss grows larger during the training process. This might indicate that the loss surface of the interior loss is more chaotic than the loss surface of the boundary loss,

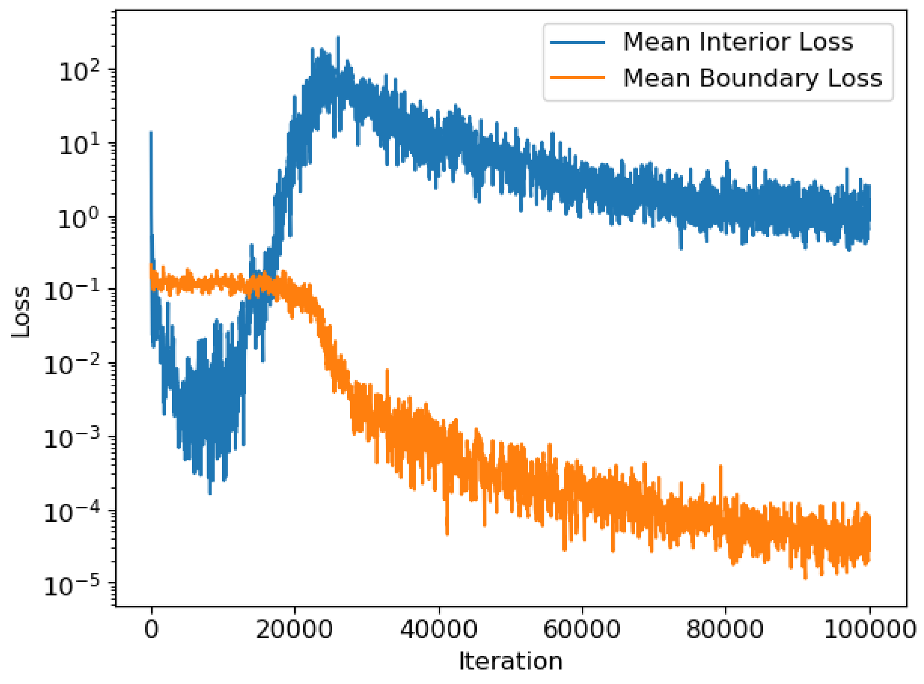


Figure 10: The interior and boundary loss during training with GNSGD when performing 100 iterations per batch, and using learning rates that decay exponentially from 5e-2 to 5e-4. It is unclear why these losses behave differently under equal prioritization.

5 Results

This chapter covers the results of the proposed methods applied to more challenging problems. The aim of this section is to gain a better understanding of the interaction between the proposed methods and various PDEs. The only method that is not included in this chapter is GNSGD, because that method was too sensitive to the choice of hyperparameters, and therefore impractical to apply to these more difficult problems.

This chapter will make heavy use of the results of chapter 4. Using those results eliminates the need to perform another hyperparameter optimization for each of the methods. However, because this chapter covers far more challenging problems than chapter 4, we use slightly modified versions of the default configurations defined in section 4.2.5. Firstly, we use neural networks with one additional layer, i.e. five total layers of 50 neurons each. For NSGD, these modifications consist of larger batches, containing 512 interior and 512 boundary points. The learning rate is also slightly reduced, with the modified learning rates decaying from 5e-2 down to 5e-4. For Adam, we also use batches containing 512 interior and 512 boundary points. For L-BFGS, we use 8192 interior points and 2048 boundary points.

This chapter is structured as follows; section 5.1 covers the Laplace equation. Although we already studied this equation extensively, this chapter considers the equation in four dimensions. Section 5.2 covers the convection-diffusion equation. Finally, section 5.3 covers the Helmholtz equation. All the PDEs covered in this chapter have been discussed in section 2.1.

5.1 Four-Dimensional Laplace Equation

Section 4 has extensively covered the two-dimensional Laplace equation. All of the problems we solved in that chapter were relatively simple, since that chapter focused entirely on analyzing the behaviour of the methods. In this section we solve a much more difficult problem, namely the four dimensional Laplace equation.

Let us start by labeling the four variables that the solution depends on by $(x, y, z, w) \in \mathbb{R}^4$. We consider the problem on the domain given by the unit hypercube $\Omega = [0,1]^4$. The boundary value problem is given by

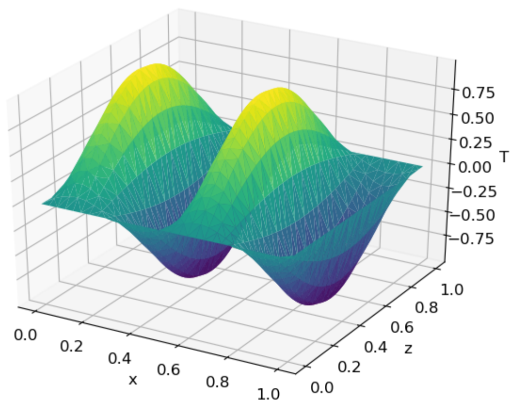
$$\begin{aligned} \nabla^2 u &= 0 && \text{for } \boldsymbol{x} \in [0,1]^4, \\ u(\boldsymbol{x}) &= G(\boldsymbol{x}) && \text{for } \boldsymbol{x} \in \partial\left([0,1]^4\right), \end{aligned} \tag{194}$$

with G yet to be specified. Note that because this is a four-dimensional problem, the boundary $\partial\left([0,1]^4\right)$ consists of eight three-dimensional cubes.

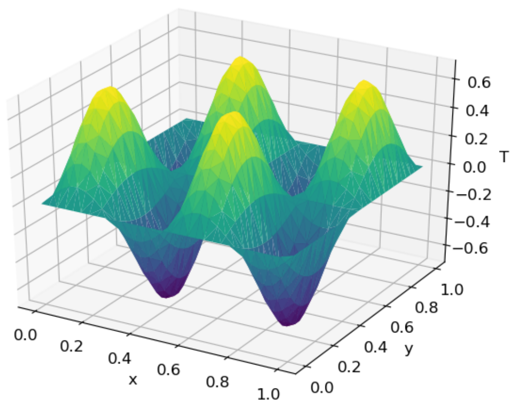
To ensure that we are able to measure the accuracy of the network output, we will again use boundary conditions that correspond to the eigenfunctions of the Laplace equation. In four dimensions, one of these eigenfunctions is given by

$$u(x,y,z,w) = \sin\left(n_x \pi x\right) \sin\left(n_y \pi y\right) \sin\left(n_z \pi z\right) e^{-\left(n_x^2+n_y^2+n_z^2\right)^{\frac{1}{2}} \pi w}. \tag{195}$$

For convenience, we set $n = \sqrt{n_x^2 + n_y^2 + n_z^2}$. When n_x, n_y and n_z are integers, this eigenfunction is nonzero on exactly one of the boundary cubes, namely the one defined by $[0,1]^3 \times \{0\}$. We will choose the eigenfunction corresponding to $n_x = 4$, $n_y = 2$, $n_z = 1$. Fig. 11 depicts the resulting boundary condition on two cross sections of this cube.



(a)



(b)

Figure 11: The boundary condition of the four-dimensional problem given in eq. 194. Fig. 11a depicts the cross section given by $y = \frac{1}{4}, w = 0$, and fig. 11b depicts the cross section given by $z = \frac{1}{4}, w = 0$.

Let us calculate the optimal loss weight for this problem. Substituting the analytical solution into eq. 127 results in

$$\begin{aligned}\int_{\partial\Omega} \left(\hat{u}(\boldsymbol{x})\right)^2 d\boldsymbol{x} &= \int_0^1 \int_0^1 \int_0^1 \left(\hat{u}(x,y,z,0)\right)^2 + \left(\hat{u}(x,y,z,1)\right)^2 dxdydz \\ &= \left(1 + e^{-2n\pi}\right) \int_0^1 \int_0^1 \int_0^1 \left(\sin(n_x \pi x)\right)^2 \left(\sin(n_y \pi y)\right)^2 \left(\sin(n_z \pi z)\right)^2 dxdydz \\ &= \frac{1}{8}\left(1 + e^{-2n\pi}\right) \approx \frac{1}{8}. \end{aligned} \tag{196}$$

for the boundary, and

$$\begin{aligned}
&\int_\Omega \left(|\hat{u}_{xx}(\boldsymbol{x})| + |\hat{u}_{yy}(\boldsymbol{x})| + |\hat{u}_{zz}(\boldsymbol{x})| + |\hat{u}_{ww}(\boldsymbol{x})|\right)^2 d\boldsymbol{x} \\
&= \int_0^1\int_0^1\int_0^1\int_0^1 \left(\left|n_x^2\pi^2\hat{u}\right| + \left|n_y^2\pi^2\hat{u}\right| + \left|n_z^2\pi^2\hat{u}\right| + \left|n^2\pi^2\hat{u}\right|\right)^2 dxdydzdw \\
&= \left[2n^2\pi^2\right]^2 \int_0^1\int_0^1\int_0^1\int_0^1 \left(\sin(n_x\pi x)\sin(n_y\pi y)\sin(n_z\pi z)e^{-n\pi w}\right)^2 dxdydzdw \\
&= 4n^4\pi^4 \frac{1}{8}\frac{1-e^{-2n\pi}}{2n\pi} \\
&= \frac{1}{4}n^3\pi^3\left(1-e^{-2n\pi}\right) \approx \frac{1}{4}n^3\pi^3
\end{aligned} \tag{197}$$

for the interior. Combining these results yields

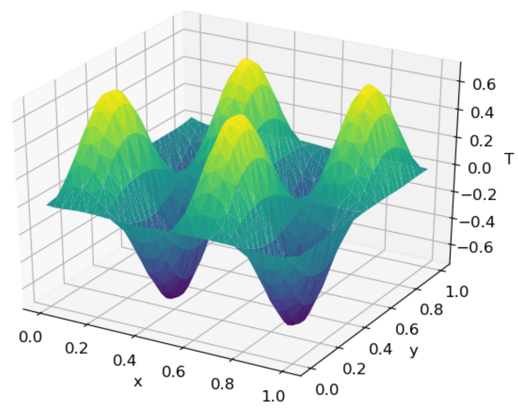
$$\begin{aligned}
\lambda &= \frac{\frac{1}{8}\left(1+e^{-2n\pi}\right)}{\frac{1}{4}n^3\pi^3\left(1-e^{-2n\pi}\right) + \frac{1}{8}\left(1+e^{-2n\pi}\right)} \\
&= \frac{\left(1+e^{-2n\pi}\right)}{2n^3\pi^3\left(1-e^{-2n\pi}\right) + \left(1+e^{-2n\pi}\right)} \approx \frac{1}{2n^3\pi^3+1}.
\end{aligned} \tag{198}$$

Perhaps unsurprisingly, this is exactly the same optimal loss weight as for the two-dimensional case, given in eq. 190. For the problem we aim to solve, the resulting optimal loss weight evaluates to 1.6e-4. Solving this problem with the methods discussed in this work yields the results given in table 17. Here, we used the configurations listed at the start of this chapter, and we initialized the neural networks using our proposed method, described in section 3.2.4.

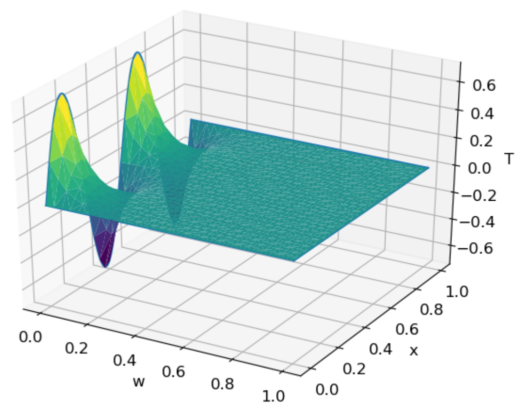
Method	NSGD	Adam	BFGS
Default Lossweight	3.76e-3	3.17e-3	1.49e-3
Optimal Lossweight	6.36e-5	2.50e-5	1.65e-5
Variance Normalized	3.99e-3	6.87e-6	5.52e-6

Table 17: Results of the different methods for the four-dimensional Laplace equation. Both variance normalization and optimal loss weights yielded accurate approximations.

These results suggest that the neural network based methods are well suited for higher dimensional problems. To highlight how accurate some of these approximations were, we depicted the network output on some cross sections of the domain after training with variance normalization in fig. 12.



(a)



(b)

Figure 12: Cross section of the output of a neural network which was trained using variance normalized Adam. Fig. 12a shows the cross section given by $z = \frac{1}{4}, w = 0$, and fig. 12b shows the cross section given by $y = \frac{1}{4}, z = \frac{1}{4}$. These results highlight how accurate the approximation is; no deviations from the boundary conditions are visible.

We remark that the number of parameters present in the networks used to solve this problem was fairly low, since each layer only contained 50 neurons. This results in a total of slightly more than 10,000 parameters. A finite difference method with this number of parameters would only have roughly 10 evenly spaced points per dimension. This suggests that neural network based methods may be very memory efficient for higher dimensional PDEs.

5.2 Stationary Convection-Diffusion Equation

In this section we consider the 1-dimensional stationary convection-diffusion equation. Specifically, we consider constant diffusion coefficients, such that the problem is given by

$$\begin{aligned} Du_{xx} - vu_x &= 0 && \text{for } x \in [0,1], \\ u(x) &= G(x) && \text{for } x \in \{0,1\}, \end{aligned} \tag{199}$$

with G defined as

$$G(x) = \begin{cases} 1, & x = 0, \\ 0, & x = 1. \end{cases} \tag{200}$$

The analytical solutions of this problem contain boundary layers, which makes the problem particularly difficult to solve for small D. For mathematical convenience, we will use negative flow velocities such that this boundary layer shows up near $x = 0$. It is possible to eliminate one parameter in this PDE by defining $w \equiv -\frac{D}{v}$, such that the problem can be written as

$$\begin{aligned} wu_{xx} + u_x &= 0 && \text{for } x \in [0,1], \\ u(x) &= G(x) && \text{for } x \in \{0,1\}. \end{aligned} \tag{201}$$

The analytical solution of this problem is given by

$$u(x) = \frac{e^{-\frac{x}{w}} - e^{-\frac{1}{w}}}{1 - e^{-\frac{1}{w}}}. \tag{202}$$

This solution is depicted in fig. 13 for $w = 0.05$, showing a rather wide boundary layer. For smaller w, the boundary layer gets proportionally smaller. Because the solution is so flat in large regions, one would expect variance normalization to yield very poor results here; flat solutions result in variance normalization inflating the error. Variance normalization typically only works if at least part of the solution has high variance; however, finding that region is precisely the hard part of solving the convection-diffusion equation, and variance normalization is not expected to make that easier.

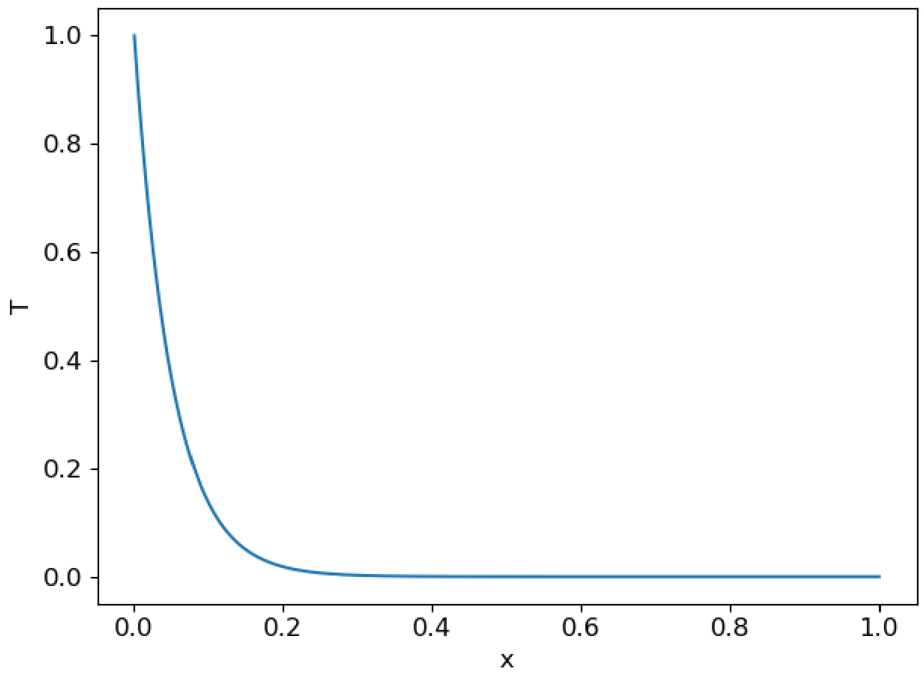


Figure 13: The analytical solution of the stationary convection-diffusion problem given in eq. 201 with $w = 0.05$.

To solve this problem, let us start by defining the loss functional. Using eq. 58 with $p = 2$, the interior loss is given by

$$L_I(u) = \int_\Omega \left[w u_{xx}(x) + u_x(x)\right]^2 dx. \tag{203}$$

Similarly, the boundary loss is given by

$$L_B(u) = \int_{\partial\Omega} \left(u(x) - G(x)\right)^2 dx. \tag{204}$$

For this loss function, the optimal loss weight of this problem is given by

$$\begin{aligned}\lambda &= \frac{\int_{\partial\Omega}(G(x))^2dx}{\int_\Omega \left(|wu_{xx}(x)| + |u_x(x)|\right)^2 dx + \int_{\partial\Omega}(G(x))^2dx} \\ &= \frac{1}{\int_\Omega \left(\frac{2}{w\left(1-e^{-\frac{1}{w}}\right)}e^{-\frac{x}{w}}\right)^2 dx + 1} \\ &= \frac{w^2\left(1-e^{-\frac{1}{w}}\right)^2}{4\left(\frac{w}{2}-\frac{w}{2}e^{-\frac{2}{w}}\right)+w^2\left(1-e^{-\frac{1}{w}}\right)^2} \\ &\approx \frac{w^2}{2w+w^2}.\end{aligned} \tag{205}$$

Next, let us solve the problem with $w = 0.05$. Solving this problem should be no issue, as the boundary layer is still relatively large for this problem. The optimal loss weight for $w = 0.05$ evaluates to $\lambda = 0.024$. The results for all methods using the configurations described at the start of this chapter are given in table 18. Surprisingly, L-BFGS was among the worst methods.

Method	NSGD	Adam	L-BFGS
Default Lossweight	2.57e-2	3.75e-9	7.36e-10
Optimal Lossweight	6.85e-7	1.96e-8	5.79e-2
Variance Normalized	3.81e-7	1.83e-8	1.61e-8

Table 18: Results for the convection-diffusion equation with $w = 0.05$. Although most methods were able to identify the boundary layer, two methods were not. This had dramatic impact on the resulting accuracy.

Naturally, reducing w further makes it more challenging for the methods to identify the boundary layer, and hence one would expect worse results. We repeated the experiments for $w = 10^{-3}$. The results are given in table 19. Clearly, with the much smaller value of w, the results were significantly worse. The most common network output for this challenging problem is depicted in fig. 14. Because the networks failed to learn the boundary layer, the best remaining way to satisfy the boundary conditions was to interpolate them.

Method	NSGD	Adam	L-BFGS
Default Lossweight	2.52e-1	2.53e-1	2.53e-1
Optimal Lossweight	3.30e-1	3.21e-1	3.31e-1
Variance Normalized	5.21e2	8.67e3	9.59e4

Table 19: Results for the convection-diffusion equation with $w = 10^{-3}$. None of the methods were able to learn the boundary layer.

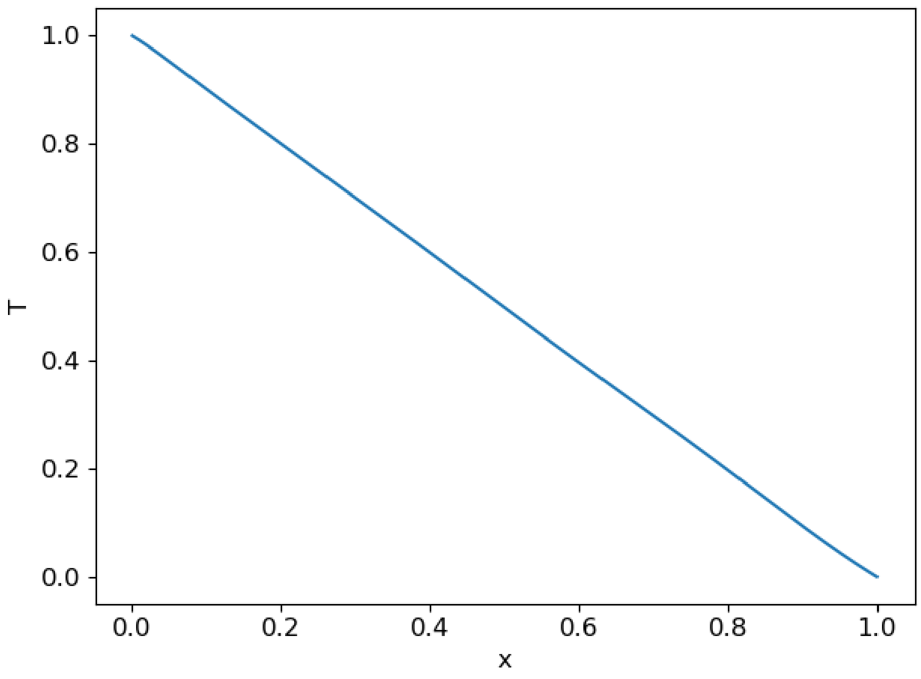


Figure 14: The most common network output for the convection-diffusion problem with $w = 10^{-3}$. Although this output satisfies the boundary conditions, it does not satisfy the PDE.

To address these issues, we will combine some simpler and more difficult problems together by adding w as an extra dimension to the network input. More specifically, we solve

$$\begin{aligned} wu_{xx} + u_x &= 0 \qquad && \text{for } (x, w) \in [0,1] \times [0.001, 0.2], \\ u(x) &= G(x) \qquad && \text{for } x \in \{0, 1\}. \end{aligned} \tag{206}$$

where $G(x)$ is still defined by eq. 200. This procedure allows the neural network to use the solution to the simpler problems to make finding the solution to the harder problem somewhat easier. We remark that our aim is still to solve the original problem for $w = 10^{-3}$. Using this method of continuation, we obtained the results given in table 20. The approximation achieved by L-BFGS at $w = 10^{-3}$ is depicted in fig. 15. Although it is still not very accurate, it at least resembles the overall shape of the analytical solution.

Method	NSGD	Adam	L-BFGS
Default Lossweight	2.80e-1	2.60e-1	4.33e-3
Optimal Lossweight	2.97e-1	3.12e-1	2.46e-2
Variance Normalized	5.23e2	8.77e3	4.64e5

Table 20: Mean square errors of the methods applied to the parametrized convection-diffusion problem, evaluated for $w = 10^{-3}$. The results were generally better than they were without this parametrization. However, variance normalization still resulted in very poor accuracy.

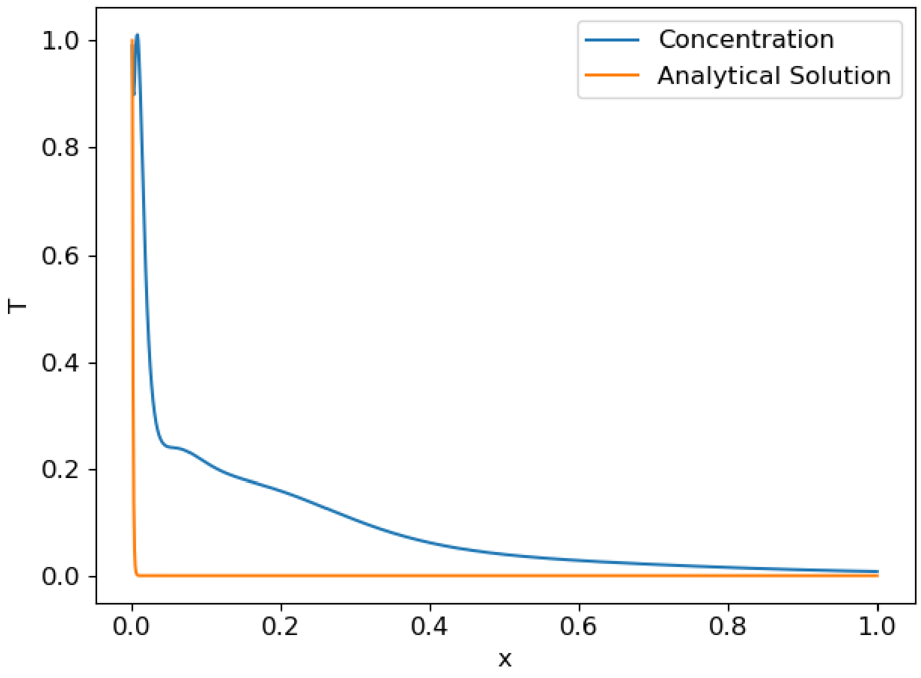


Figure 15: The approximation achieved by L-BFGS for the parametrized problem, evaluated at $w = 10^{-3}$, drawn together with analytical solution. This approximation is significantly more accurate than the ones achieved without continuation.

5.3 Helmholtz Equation

In this section we use the proposed methods and the original method to solve the Helmholtz equation. We first construct a family of model problems with Dirichlet boundary conditions in section 5.3.1. Then, in section 5.3.2 we calculate the optimal loss weight for this family of problems. Section 5.3.3 covers the results of the methods on this family of test problems. Finally, section 5.3.4 covers the calculation of eigenfunctions of the Helmholtz equation.

5.3.1 Model Problems

To investigate the capabilities of the proposed methods, we use these methods to solve the Helmholtz equation. This section introduces a family of model problems that can be used for this purpose.

Consider the inhomogeneous Helmholtz equation on the rectangular domain given by $(x, y) \in [0,1] \times [0,1]$ with Dirichlet boundary conditions. A boundary value problem for this equation is given by

$$\begin{aligned} \left(\nabla^2 + k^2\right) u(x,y) &= F(x,y) \qquad && \text{for } (x,y) \in [0,1] \times [0,1], \\ u(x,y) &= G(x,y) \qquad && \text{on } \partial\left([0,1] \times [0,1]\right), \end{aligned} \tag{207}$$

with k, F and G yet to be specified. The loss functional of physics informed neural networks can be defined without explicit knowledge of F and G. Using eq. 58 with $p = 2$, the interior loss is given by

$$L_I(u) = \int_\Omega \left[u_{xx}(\boldsymbol{x}) + u_{yy}(\boldsymbol{x}) + k^2 u(\boldsymbol{x}) - F(\boldsymbol{x})\right]^2 d\boldsymbol{x}. \tag{208}$$

Similarly, the boundary loss is given by

$$L_B(u) = \int_{\partial\Omega} \left(u(\boldsymbol{x}) - G(\boldsymbol{x})\right)^2 d\boldsymbol{x}. \tag{209}$$

As mentioned in section 2.1.3, the Helmholtz equation with homogeneous Dirichlet boundary conditions and without source functions has nontrivial eigenfunctions for specific wave numbers. This complicates the analysis of the results, since the existence of nontrivial eigenfunctions means analytical solutions are not unique. Therefore, we will choose the wavenumber such that the problem does not admit nontrivial eigenfunctions. Recall that the wavenumbers for which nontrivial eigenfunctions exist satisfy

$$k^2 = \left(k_x^2 + k_y^2\right) \pi^2, \tag{210}$$

with k_x and k_y integers. Thus, as long as the wavenumber is not a multiple of π^2, there are no nontrivial eigenfunctions. For such values, the boundary value problems defined in eq. 207 are well posed, and thus we expect decent results.

Common test problems for the Helmholtz equation use a dirac delta function as a source, since for such source functions the analytical solution is given by the Green's function, which in two dimensions is given by a Hankel function. However, we use Monte Carlo integration to approximate the integrals present in the loss functions; dirac delta functions cannot be approximated with such methods without using some form of importance sampling, and thus a different source function is required. To analyze the results, it is vital that we have access to the analytical solution. To this end, we will impose a known function as the analytical solution, and compute the corresponding source function.

To mimic the more common problems with dirac delta source functions, we will use a smooth source function that decays rapidly when moving away from the origin. The resulting analytical solution appears very similar to the Hankel function that normally describes the analytical solutions. The solution we impose is given by

$$u(r) = \begin{cases} \frac{1}{kr}\sin(kr), & r > 0, \\ 1, & r = 0, \end{cases} \tag{211}$$

with

$$r = \sqrt{x^2 + y^2}. \tag{212}$$

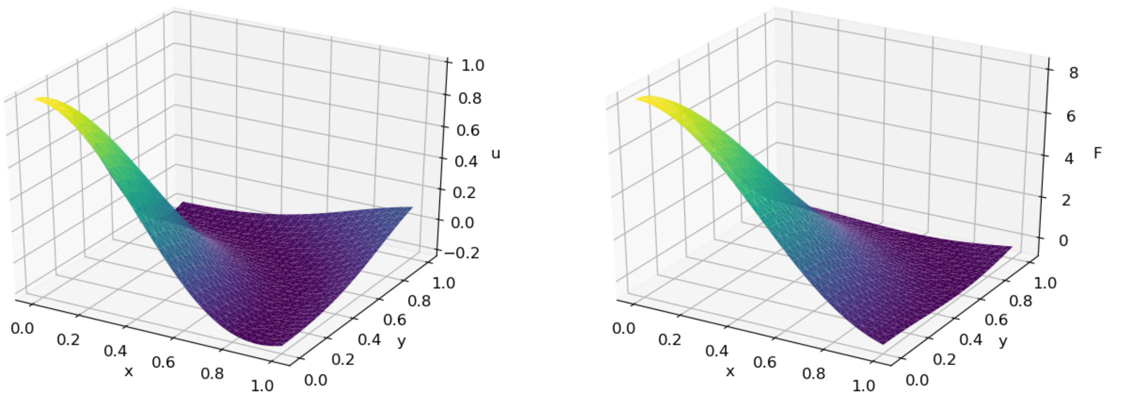
The next step is to calculate the source function that corresponds to this solution. To exploit the symmetry of the solution, it is convenient to use a coordinate transformation. In polar coordinates, the Helmholtz equation is given by

$$u_{rr} + \frac{1}{r}u_r + \frac{1}{r^2}u_{\theta\theta} + k^2 u = F(r,\theta) \tag{213}$$

Plugging the analytical solution into eq. 213 results in the source function

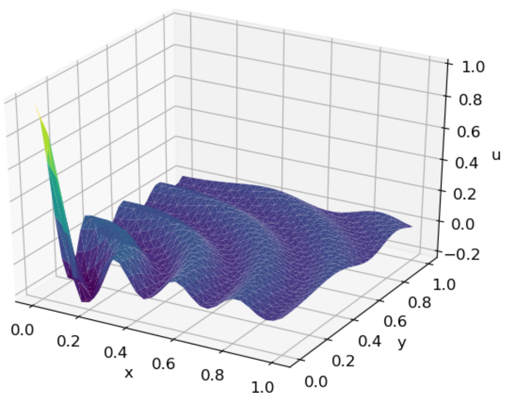
$$\begin{aligned} F(r,\theta) &= u_{rr} + \frac{1}{r}u_r + \frac{1}{r^2}u_{\theta\theta} + k^2 u \\ &= \frac{(2-k^2r^2)\sin(kr) - 2kr\cos(kr)}{kr^3} + \frac{kr\cos(kr) - \sin(kr)}{kr^3} + k\frac{\sin(kr)}{r} \\ &= \frac{1}{kr^3}\left(\sin(kr) - kr\cos(kr)\right). \end{aligned} \tag{214}$$

The solution and corresponding source functions are depicted in fig 16 for $k = 5$ and in fig. 17 for $k = 25$.

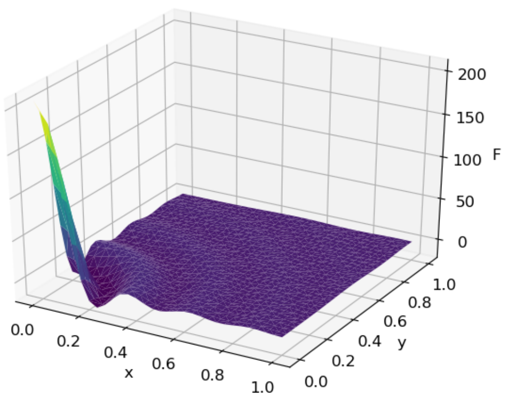


(a) (b)

Figure 16: The analytical solution (16a) and the source function (16b) of the model problem with $k = 5$.



(a)



(b)

Figure 17: The analytical solution (17a) and the source function (17b) of the model problem with $k = 25$. Note that the source function decays much faster than the analytical solution, leading to a solution that strongly resembles the analytical solution of problems with dirac delta source functions.

This family of model problems has, like all problems we discussed so far, a single parameter that can be used to control the difficulty; high frequency solutions are naturally more difficult to learn for neural networks because of their complex shapes. The problems with $k = 5$ and $k = 25$ are representative of respectively low and high frequency problems.

5.3.2 Optimal Loss Weight

As seen previously, the boundary and interior loss functions generally do not have the same dependency on the frequency of the solution. In this section we investigate the impact of the frequency on these loss functions, and we investigate how that affects the optimal loss weight.

Using the same methods as in eq. 127, the optimal λ for this problem is given by

$$\lambda = \frac{\int_{\partial\Omega} \left(u(\boldsymbol{x})\right)^2 d\boldsymbol{x}}{\int_{\Omega} \left[(u_{xx}(\boldsymbol{x}))^2 + (u_{yy}(\boldsymbol{x}))^2 + (k^2 u(\boldsymbol{x}))^2\right] d\boldsymbol{x} + \int_{\partial\Omega} \left(u(\boldsymbol{x})\right)^2 d\boldsymbol{x}}. \tag{215}$$

To evaluate this integral, we use Monte Carlo integration.

To gain some insight in the precise relation between the boundary and interior losses, we consider these loss functionals individually, starting with the boundary integral. This integral is given by

$$\begin{aligned}\int_{\partial\Omega} (u(\boldsymbol{x}))^2 \, d\boldsymbol{x} &= \int_{\partial\Omega} \left(\frac{1}{kr(\boldsymbol{x})} \sin(kr(\boldsymbol{x})) \right)^2 d\boldsymbol{x} \\ &= 2 \int_0^1 \left(\frac{1}{kx} \sin(kx) \right)^2 dx + 2 \int_0^1 \left(\frac{1}{k\sqrt{1+y^2}} \sin(k\sqrt{1+y^2}) \right)^2 dy,\end{aligned} \tag{216}$$

where the last equality follows from the symmetry of the solution. The integral is plotted in fig. 18a as a function of k. Note that the second integral is bounded by

$$\int_0^1 \left(\frac{1}{k\sqrt{1+y^2}} \sin(k\sqrt{1+y^2}) \right)^2 dy < \frac{1}{k^2}, \tag{217}$$

while the first integral approaches $\frac{\pi}{2k}$ for large k. Thus, it holds that

$$\int_{\partial\Omega} (u(\boldsymbol{x}))^2 \, d\boldsymbol{x} \to \frac{\pi}{k} \text{ as } k \to \infty. \tag{218}$$

Next, consider the interior integral of eq. 215. It is convenient to use polar coordinates to evaluate this integral, since the analytical solution is symmetrical around the origin. The derivatives in polar coordinates are given by

$$\frac{\partial}{\partial x} = \cos(\phi) \frac{\partial}{\partial r} - \frac{1}{r} \sin(\phi) \frac{\partial}{\partial \phi} \tag{219}$$

and

$$\frac{\partial}{\partial y} = \sin(\phi) \frac{\partial}{\partial r} + \frac{1}{r} \cos(\phi) \frac{\partial}{\partial \phi}, \tag{220}$$

such that

$$\begin{aligned} u_{xx} &= \frac{\partial}{\partial x} \left[\frac{\partial}{\partial x} u \right] \\ &= \frac{\partial}{\partial x} \left[\cos(\phi) u_r \right] \\ &= \sin^2(\phi) u_{rr} + \frac{1}{r} \sin^2(\phi) u_r \\ &= \sin^2(\phi) \left[u_{rr} + \frac{1}{r} u_r \right] \end{aligned} \tag{221}$$

and

$$\begin{aligned} u_{yy} &= \frac{\partial}{\partial y} \left[\frac{\partial}{\partial y} u \right] \\ &= \frac{\partial}{\partial y} \left[\sin(\phi) u_r \right] \\ &= \cos^2(\phi) u_{rr} + \frac{1}{r} \cos^2(\phi) u_r \\ &= \cos^2(\phi) \left[u_{rr} + \frac{1}{r} u_r \right]. \end{aligned} \tag{222}$$

Combining the two results gives

$$\begin{aligned}(u_{xx})^2 + (u_{yy})^2 &= \left[\sin^4(\phi) + \cos^4(\phi)\right] \left(u_{rr} + \frac{1}{r} u_r \right)^2 \\ &= \left[\sin^4(\phi) + \cos^4(\phi)\right] \left(\frac{1}{kr^3} \left[(1 - k^2r^2) \sin(kr) - kr\cos(kr) \right] \right)^2 .\end{aligned} \tag{223}$$

The total interior integral is thus given by

$$\int_\Omega \left[(u_{xx}(\boldsymbol{x}))^2 + (u_{yy}\boldsymbol{x})^2 + \left(k^2 u(\boldsymbol{x})\right)^2 \right] d\boldsymbol{x} \tag{224}$$

$$= \int_\Omega \left[\left[\sin^4(\phi) + \cos^4(\phi)\right] \left(\frac{1}{kr^3} \left[(1 - k^2r^2) \sin(kr) - kr\cos(kr) \right] \right)^2 + (u(\boldsymbol{x}))^2 \right] d\boldsymbol{x}.$$

In the asymptotic regime for large k, this integral scales with approximately $k^2 \log k$. Thus, the optimal loss weight becomes of the order $\frac{1}{k^3 \log k}$. The result of the Monte Carlo approximation of the interior integral is plotted in fig. 18b. Notice that the result becomes noisier for large wavenumbers. The reason for this is that the variance of the solution also increases with the wavenumber. This corresponds to an increase in difficulty for neural networks to learn the solution.

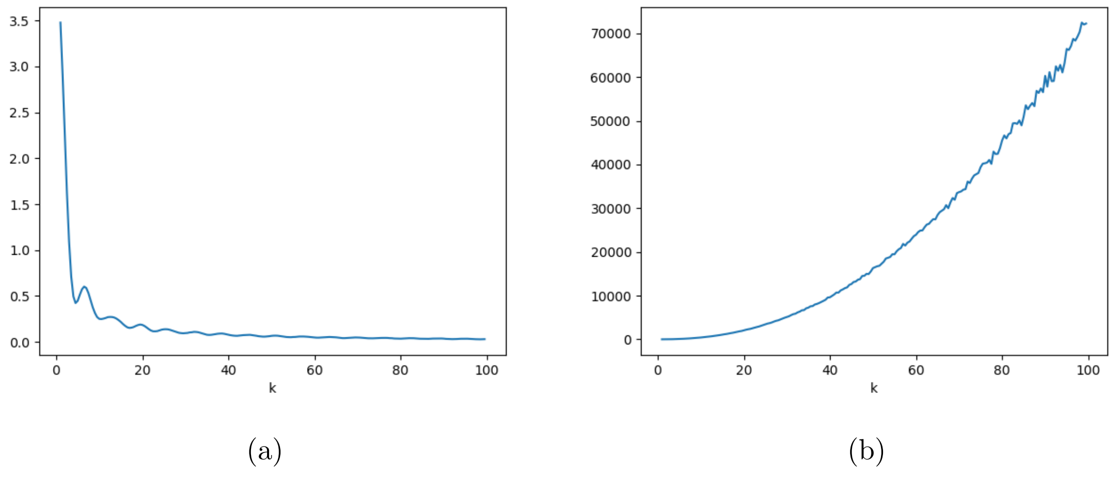


Figure 18: The boundary (18a) and interior (18b) integrals defined in eq. 215 as a function of k. The boundary integral decays at a rate of approximately $\frac{1}{k}$, while the interior integral is approximately proportional to k^3.

Using eq. 215 we obtain the optimal loss weights as depicted in fig. 19. Note that the optimum decays rapidly as k increases. Based on this relation between k and λ, one would expect the quality of the results of basic physics informed neural networks to degrade for large k.

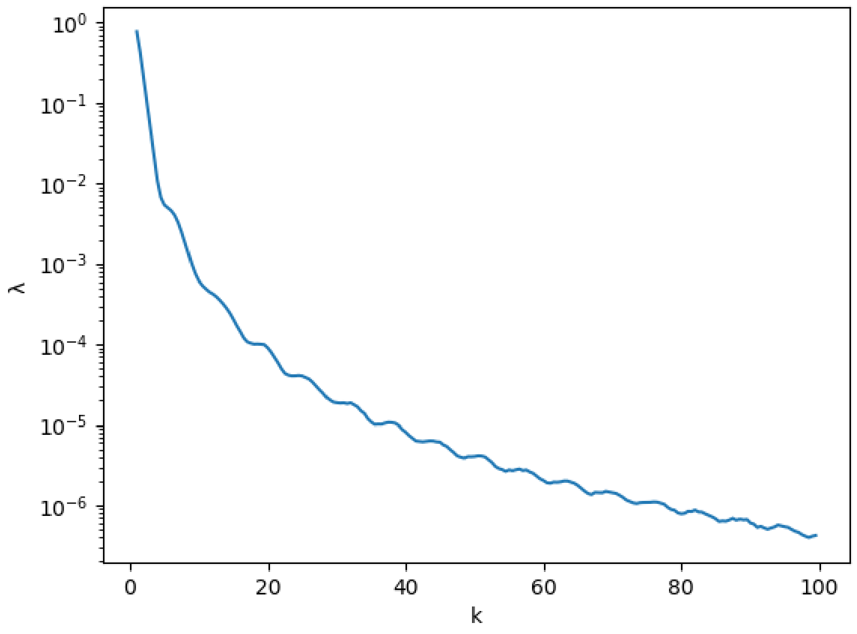


Figure 19: The optimal loss weight as defined in eq. 213 as a function of the wavenumber k.

5.3.3 Method Comparison

In this section we apply the four different methods to the family of problems defined in eq. 207, specifically using $k = 5$ and $k = 25$. We first consider the methods that use constant loss weights. The hyperparameter configurations of all methods used here are described at the start of this chapter.

For each wavenumber there are two loss weights that we consider: the default $\lambda = \frac{1}{2}$, corresponding to basic physics informed neural networks, and the optimal loss weight as given in fig. 19. The mean square errors that followed from the default loss weight are given in table 21, and the mean square errors after training weighted physics informed neural networks are given in table 22.

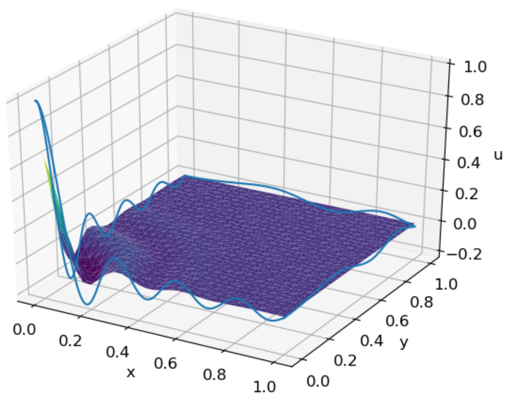
k \ Method	NSGD	Adam	L-BFGS
5	1.76e-3	1.82e-4	6.16e-5
25	4.41e-3	2.28e-3	1.99e-3

Table 21: Results after training a physics informed neural network using the default $\lambda = \frac{1}{2}$ with NSGD, Adam and BFGS. None of these methods were able to accurately solve the high frequency problem.

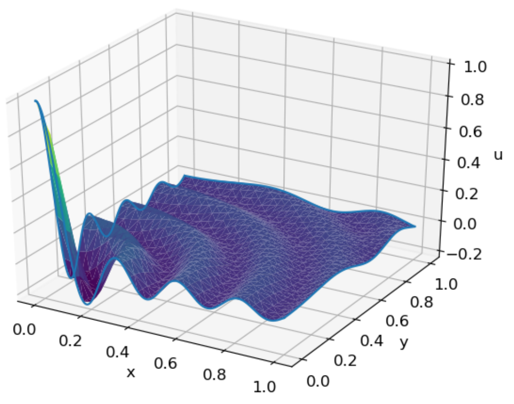
k \ Method	NSGD	Adam	BFGS
5	1.45e-7	2.94e-6	6.82e-7
25	3.03e-3	6.01e-4	6.82e-5

Table 22: Mean square errors after training a weighted physics informed neural network with NSGD, Adam and BFGS using λ = 5.4e-3 for $k = 5$ and λ = 4.0e-5 for $k = 25$. While all three methods were able to solve the low frequency problem, only BFGS managed to solve the higher frequency problem, while Adam came reasonably close.

There are a couple of observations to make from these results. Firstly, note that the impact of the loss weight is about as significant as one would expect from fig. 19; the high frequency problem has an optimal loss weight that is about two orders of magnitude smaller, and similarly, the best results of table 21 are about two orders of magnitude worse. Secondly, we observe that although the best performance was obtained by solving the problem with a fixed optimal loss weight, the results of those methods for the high frequency problem only improved the results by about one order of magnitude. The solutions obtained by BFGS using both loss weights are depicted in fig. 20.



(a)



(b)

Figure 20: The solutions of the problem with $k = 25$ found by BFGS using constant loss weights. Fig. 20a shows the solution found using $\lambda = \frac{1}{2}$ and fig. 20b shows the solution found using λ =4.0e-5. The difference between the mean square errors is about two orders of magnitude.

It seems that the small differences in mean square errors are mainly a product of the shape of the analytical solution; large portions of the solution lie close to 0, and thus solutions that completely flatten out seem to achieve relatively accurate results.

Next, we consider variance normalization. Applying variance normalization to the Helmoltz equation results in the loss functional

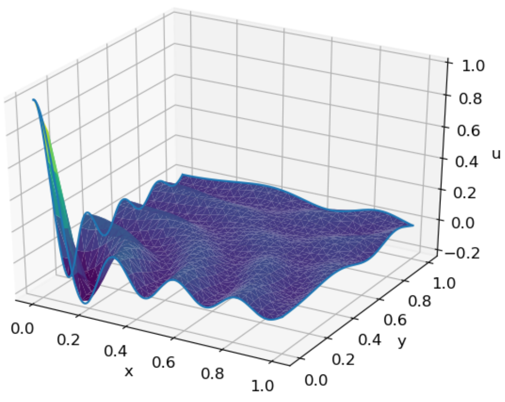
$$L(u) = \frac{\int_\Omega \left[u_{xx}(\boldsymbol{x}) + u_{yy}(\boldsymbol{x}) + k^2 u(\boldsymbol{x}) - F(\boldsymbol{x})\right]^2 d\boldsymbol{x}}{\int_\Omega [|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})| + k^2\,|u(\boldsymbol{x}) - \overline{u}|]^2 d\boldsymbol{x}} + \frac{\int_{\partial\Omega}(u(\boldsymbol{x}) - G(\boldsymbol{x}))^2 d\boldsymbol{x}}{\int_{\partial\Omega}(u(\boldsymbol{x}) - \overline{u})^2 d\boldsymbol{x}}, \tag{225}$$

where $\overline{u}$ denotes the mean of u over the respective domain. Table 23 gives the results of variance normalization using NSGD, Adam and BFGS.

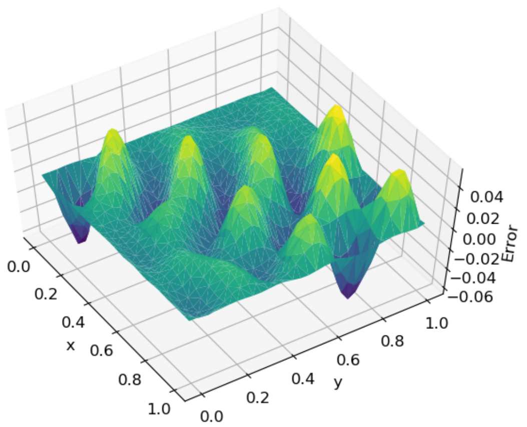
Method / k	NSGD	Adam	BFGS
5	1.22e-7	2.40e-6	1.88e-8
25	1.07e-2	1.22e-3	4.01e-4

Table 23: Mean square errors after training with variance normalization for $k = 5$ and $k = 25$. Compared to training with an optimal loss weight, this method loses accuracy for large wavenumbers.

Variance normalization yields excellent results for low wavenumbers, as even higher accuracy is achieved than when training with optimal loss weights. However, large wavenumbers result in poor approximations. To gain some insight in why this might be the case, the solution and the absolute error of the approximation obtained using BFGS for $k = 25$ are depicted in fig. 21.



(a)



(b)

Figure 21: The results of training with variance normalization using BFGS for $k = 25$. Fig. 21a depicts the output of the neural network, while fig. 21b depicts the absolute error of this approximation. Note that the error oscillates significantly, and is not shaped similarly to the analytical solution.

It seems that the error introduced near the source "propagates outward", resulting in inaccuracy where the analytical solution is small. Although the mean square error remains small, the relative error is much larger.

Comparing the error to the error that resulted from training with optimal loss weights, given in fig. 22, reveals an interesting pattern. Although the errors of both methods oscillate, the oscillations are very different in nature. In particular, the difference between these errors appears to be shaped like the eigenfunctions of the Helmholtz equation. Although this may seem coincidental, training larger networks with BFGS using variance normalization, or using different random seeds, results in approximately the same error distribution, suggesting that this error is systematic in nature.

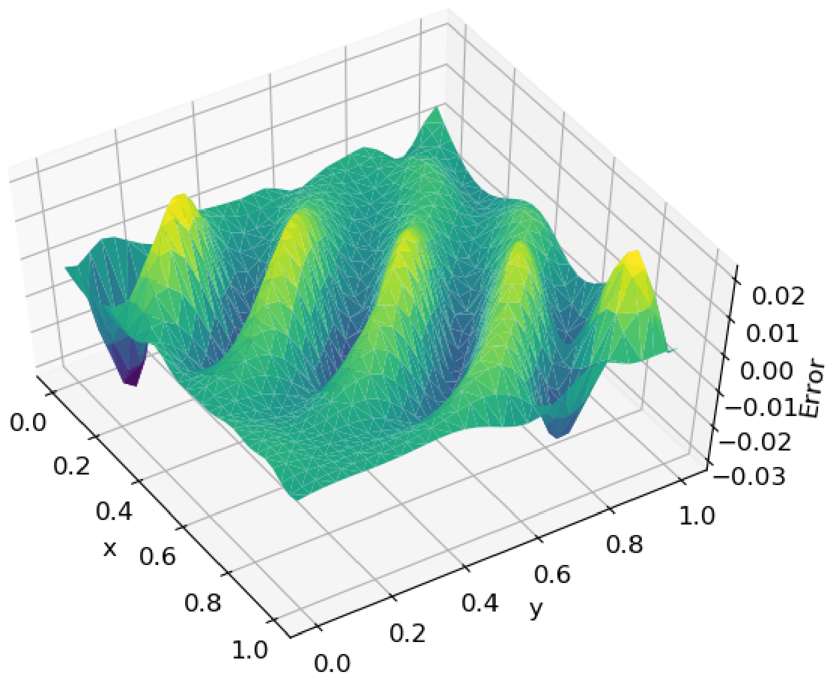


Figure 22: The absolute error resulting from training with optimal loss weight using BFGS for $k = 25$. The distribution of the error is similarly shaped as the analytical solution.

To investigate whether the eigenfunctions may be related to these errors, we consider these eigenfunctions in more detail in the next section.

5.3.4 Eigenmodes of the Vibrating Membrane

The previous section has shown that variance normalization leads to worse results than expected. Since the error showed similarities to eigenfunctions with different wavenumbers, these eigenfunctions may be the underlying cause of these poor results. In this section we investigate these eigenfunctions and their impact on variance normalized training.

Again consider the family of problems defined in eq. 207, with a fixed wavenumber k that does not satisfy eq. 210 for any $k_x, k_y \in \mathbb{N}$. As already mentioned, such problems are well-posed. However, there may exist $k_x, k_y \in \mathbb{N}$ for which the equation is approximately satisfied, i.e.

$$\left(k_x^2 + k_y^2\right)\pi^2 = k^2(1 \pm \epsilon) \equiv \hat{k}^2 \tag{226}$$

for some small ϵ. It turns out that cases like these are exceedingly common. For the wavenumber $k = 25$, we can satisfy eq. 226 with an error of approximately $\epsilon \approx$2.6e-2 with the wavenumbers $k_x = 7, k_y = 4$ and vice versa, as well as $k_x = 8, k_y = 1$ and vice versa. This existence of approximate eigenfunctions leads to interesting scenarios. For example, for eigenfunctions with at least one even wavenumber, the loss function corresponding to the homogeneous Helmholtz equation is given by

$$\begin{aligned} L_I(u) &= \int_\Omega \left[u_{xx}(\boldsymbol{x}) + u_{yy}(\boldsymbol{x}) + k^2 u(\boldsymbol{x})\right]^2 d\boldsymbol{x} \\ &= \int_\Omega \left[-\hat{k}^2 u(\boldsymbol{x}) + k^2 u(\boldsymbol{x})\right]^2 d\boldsymbol{x} \\ &= (k^2 - \hat{k}^2)^2 \int_\Omega (u(\boldsymbol{x}))^2 \, d\boldsymbol{x}. \end{aligned} \tag{227}$$

The scale factor on the other hand, is given by

$$\begin{aligned} S_I(v) &= \int_\Omega \left(|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})| + k^2 \left|u(\boldsymbol{x}) - \overline{u}\right|\right)^2 d\boldsymbol{x} \\ &= \int_\Omega \left(k_x^2\pi^2 |u(\boldsymbol{x})| + k_y^2\pi^2 |u(\boldsymbol{x})| + k^2 |u(\boldsymbol{x})|\right)^2 d\boldsymbol{x} \\ &= \int_\Omega \left(\hat{k}^2 |u(\boldsymbol{x})| + k^2 |u(\boldsymbol{x})|\right)^2 d\boldsymbol{x} \\ &= (\hat{k}^2 + k^2)^2 \int_\Omega (u(\boldsymbol{x}))^2 \, d\boldsymbol{x}. \end{aligned} \tag{228}$$

Thus, the interior part of the variance normalized loss function is given by

$$\frac{L_I(u)}{S_I(u)} = \frac{(\hat{k}^2 - k^2)^2}{(\hat{k}^2 + k^2)^2} = \frac{\left(\pm\epsilon k^2\right)^2}{((1 \pm \epsilon + 1)k^2)^2} \approx \frac{\epsilon^2}{4}. \tag{229}$$

For the eigenmodes close to $k = 25$ we mentioned earlier, this would result in an interior loss of 1.69e-4, a value much smaller than typically encountered for solutions with high frequencies. When such eigenmodes are added to approximations of inhomogeneous problems, the normalization factor S_I tends to be much larger, resulting in an even lower loss. Since such functions satisfy homogeneous boundary conditions, they affect neither the boundary loss nor the boundary normalization factor. This makes eigenmodes with wavenumbers close to the wavenumber of the problem very difficult to eliminate. In fact, in some scenarios they may even be exploited by the network to increase the interior normalization factor without significantly increasing the interior loss. Adding such eigenmodes with large amplitudes to the solution may therefore be a way for the neural network to asymptotically reduce the interior part of the loss to $\frac{\epsilon^2}{4}$. This ties in well with the theoretical results derived in section 3.3.3; not all problems may be suited for these scaled loss functionals.

Although addressing these problems is beyond the scope of this thesis, we can use this property to our advantage, since it gives us a way to find eigenfunctions. By solving the homogeneous Helmholtz equation with variance normalization, the neural network should find nearby eigenfunctions, as variance normalization prevents the zero function from being found.

To this end, consider the boundary value problem given by

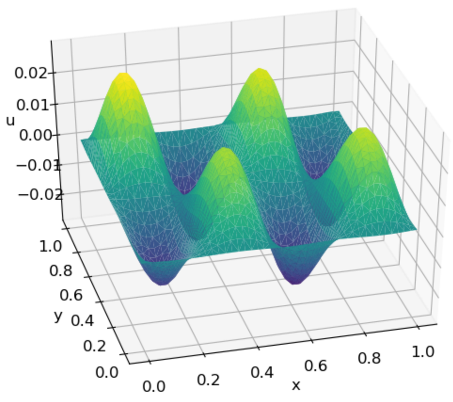
$$\begin{aligned} \left(\nabla^2 + k^2\right) u(x,y) &= 0 \qquad \text{for } (x,y) \in [0,1] \times [0,1], \\ u(x,y) &= 0 \qquad \text{on } \partial\left([0,1] \times [0,1]\right). \end{aligned} \tag{230}$$

We consider $k = 14$ and $k = 25$. With $k = 14$, eigenmodes that would result in very low loss are given by $k_x = 2, k_y = 4$ and vice versa. The eigenmodes of $k = 25$ were mentioned earlier. To find these eigenmodes, we use the loss function

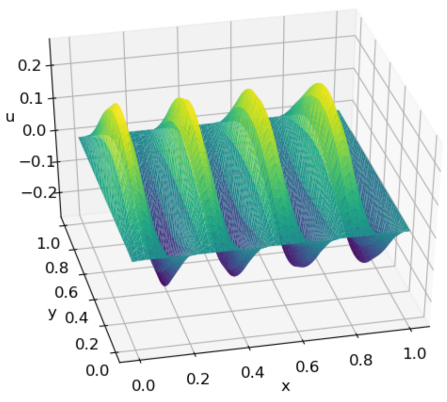
$$L(u) = \frac{\int_{\partial\Omega} \left(u(\boldsymbol{x})\right)^2 d\boldsymbol{x}}{\int_{\Omega} \left(u(\boldsymbol{x}) - \overline{u}\right)^2 d\boldsymbol{x}} + \frac{\int_{\Omega} \left[u_{xx}(\boldsymbol{x}) + u_{yy}(\boldsymbol{x}) + k^2 u(\boldsymbol{x})\right]^2 d\boldsymbol{x}}{\int_{\Omega} \left(|u_{xx}(\boldsymbol{x})| + |u_{yy}(\boldsymbol{x})| + k^2 \left|u(\boldsymbol{x}) - \overline{u}\right|\right)^2 d\boldsymbol{x}}. \tag{231}$$

Note that instead of normalizing the boundary loss with the variance of the solution along the boundary, we use the variance of solution over the entire domain. The reason for this is that with homogeneous boundary conditions the boundary variance is identical to the boundary loss function itself.

Minimizing this loss function with BFGS leads to the solutions depicted in fig. 23. The eigenmodes $(4, 2)$ and $(8, 1)$ are clearly visible. Interestingly, the eigenmode $(4, 7)$ was not picked up by the network, suggesting that this method deals poorly with eigenvalues with higher multiplicity.



(a)



(b)

Figure 23: Minimizing eq. 231 leads to the discovery of nearby eigenfunctions. For $k = 25$, the found eigenfunctions eigenfunctions correspond to the modes $k_x = 1, k_y = 8$.

The loss of the resulting solutions can be used as a measure of how well these eigenmodes line up with the wavenumber of the problem. Small losses indicate that the eigenmodes have very similar wavenumbers as the original problem. For $k = 25$, the resulting variance normalized loss was 2.11e-4, which is quite close to the predicted 1.69e-4. For $k = 14$, the loss was similarly small.

6 Conclusions

Recent studies have developed methods to solve partial differential equations using neural networks. These methods approach solving partial differential equations in a completely novel way. In order to exploit the properties of these methods, which may be vastly different compared to more traditional numerical methods, a better understanding of these methods on a fundamental level is needed. The main goal of this thesis was to gain a deeper understanding of these methods and their interactions with several different classes of partial differential equations.

To gain more insight in when and why these methods work, we reconstructed them from scratch in chapter 3, starting from a function space point of view. This process identified four properties that loss functionals should satisfy to guarantee these methods to work. We then proceeded to show that the norms of the operators of the partial differential equation may be used to define a loss functional which satisfies all four of these properties for well-posed linear problems. We have shown that solving a partial differential equation can be formulated as the simultaneous minimization of two loss functionals, one of which describes the boundary conditions while the other describes the partial differential equation itself. In this light, the loss function developed in [8] can be viewed as a scalarization of a multi-objective optimization problem. However, this scalarization can be performed in a more general way by introducing a parameter to represent the relative importance of both loss functionals. The original method can then be obtained by giving the loss functionals equal weight.

We analyzed the potential effects of this newly introduced loss weight on the accuracy. Under certain assumptions on the distribution of the error of trained neural networks, the optimal value of the loss weight can be shown to be dependent on both the partial differential equation and its analytical solution. To avoid relying on the analytical solution to derive parameters that are required to find it, we introduced a heuristic method that uses the output of the neural network to approximate the optimal loss weight, a method which we called variance normalization. We then defined an alternative way of solving the multi-objective optimization problem. This method, which we labeled gradients scaled SGD, computes an update vector that optimizes all loss functions equally when the learning rate is infinitely small.

The behaviour of the original and the proposed methods has been investigated empirically in chapters 4 and 5 by solving several different model partial differential equations. We showed that the original methods are unable to solve some relatively simple problems. Using the optimal loss weight derived in chapter 3 significantly improved the accuracy for all problems we investigated, enabling these methods to solve the four-dimensional Laplace equation. Variance normalization yielded similarly good results for the Laplace equation, but resulted in poor approximations for high frequency problems of the Helmholtz equation, likely due to the presence of nearby eigenmodes. By applying variance normalization to solve the homogeneous Helmholtz equation, we obtained a method that is able to find these nearby eigenfunctions.

Our theoretical findings provide a good starting point of obtaining a complete understanding of these methods. These findings have made clear that under idealized circumstances, i.e. when using perfect optimization algorithms, these methods are able to solve any well-posed linear problem. This implies that potential convergence issues can only originate from the properties of neural networks and the training algorithms. In practice, convergence issues prevent the loss functions from being optimized to arbitrary levels of accuracy. Our proposed methods mitigate some of these issues by allocating the network's resources better over the loss functions that need to be optimized, which leads to significantly improved accuracy. This makes the methods more suitable for a wider range of problems, which is important for methods that are so general in nature.

There are, however, several limitations to these methods. Most importantly, the accuracy of these methods as they currently stand remains insufficient for many applications. Even though traditional methods only provide information on a discrete set of points or over a discretized mesh, the accuracy of such methods tends to be far superior to the accuracy of neural network based methods. On top of that, neural networks are computationally expensive to train, and convergence guarantees remain evasive. Without significant breakthroughs, these methods are unlikely to completely replace traditional methods for certain classes of problems.

However, there are many areas that could benefit from these methods despite their drawbacks. Of particular interest are high dimensional partial differential equations, which may be solved effectively using such methods. Many traditional methods suffer from the curse of dimensionality, making it infeasible to solve such problems using those methods.

Other areas that might benefit from these methods are areas where similar problems are repeatedly solved. Although training neural networks is expensive, evaluating the output of a trained network is cheap, and this property can be exploited by learning the solution to many partial differential equations at once. In this thesis we already learned solutions to a parametrized problem. Extending this by adding initial conditions as input variables for neural networks may result in methods that could replace traditional solvers when they are repeatedly used to solve the same problem for different initial conditions.

Another promising research topic would be to combine the methods with importance sampling; the study of [18] has shown that the training process can be accelerated by using the loss function as the importance metric, and we suspect that applying this to the methods studied in this work may also lead to a further improved accuracy. This could be particularly interesting for problems of which the solution has highly localized features, such as boundary layers. Furthermore, importance sampling may enable these methods to deal with source functions with high variance, such as Dirac delta functions, making them more generally applicable.

Besides these practical concerns, there are also theoretical follow-up questions that might improve our understanding of these methods further. Particularly, investigating the loss surfaces as done in the study of [19] might explain our results in an entirely different way; the multi-objective optimization problem defines two separate loss surfaces, which we simultaneously traverse during training. Understanding how the proposed methods affect the resulting loss surface may lead to further improvements.

It would also be interesting to investigate the connection between connection between the neural network methods and traditional numerical solvers. Such a connection would provide acces to a very mature field of research, which contains many techniques that might be useful in the context we considered in this work, such as preconditioning.

Ultimately, the methods we discussed in this work are still in their infancy, and there are many more topics that would be interesting to investigate. The new insights we obtained should pave the way for further research.

References

[1] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec 2014.

[2] Juergen Schmidhuber. Deep Learning in Neural Networks: An Overview. *arXiv e-prints*, page arXiv:1404.7828, Apr 2014.

[3] D. Soudry and Y. Carmon. No bad local minima: Data independent training error guarantees for multilayer neural networks. *ArXiv e-prints*, May 2016.

[4] Kenji Kawaguchi. Deep Learning without Poor Local Minima. *arXiv e-prints*, page arXiv:1605.07110, May 2016.

[5] J. He, L. Li, J. Xu, and C. Zheng. ReLU Deep Neural Networks and Linear Finite Elements. *ArXiv e-prints*, July 2018.

[6] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *arXiv e-prints*, page physics/9705023, May 1997.

[7] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, Dec 2018.

[8] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. *ArXiv e-prints*, November 2017.

[9] Tim Dockhorn. A Discussion on Solving Partial Differential Equations using Neural Networks. *arXiv e-prints*, page arXiv:1904.07200, Apr 2019.

[10] Yogi Ahmad Erlangga. *A robust and efficient iterative method for the numerical solution of the Helmholtz equation*. PhD thesis, Delft University of Technology, dec 2005.

[11] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399, 2018.

[12] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[14] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[15] Pablo Márquez-Neila, Mathieu Salzmann, and Pascal Fua. Imposing Hard Constraints on Deep Networks: Promises and Limitations. *arXiv e-prints*, page arXiv:1706.02025, Jun 2017.

[16] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 9:249–256, 13–15 May 2010.

[17] V. I. Avrutskiy. Enhancing approximation abilities of neural networks by training derivatives. *arXiv e-prints*, page arXiv:1712.04473, December 2017.

[18] Angelos Katharopoulos and François Fleuret. Biased Importance Sampling for Deep Neural Network Training. *arXiv e-prints*, page arXiv:1706.00043, May 2017.

[19] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets. *arXiv e-prints*, page arXiv:1712.09913, Dec 2017.