Optimizing finite-element mesh construction for singularly perturbed differential equations using neural networks

by

© **Gerry Harris-Pink**

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Masters of Scientific Computing.

Department of Mathematics and Statistics

Memorial University

September 2024

St. John's, Newfoundland and Labrador, Canada

Abstract

We use differential equations as a mathematical tool to enable us to model real world systems. This thesis focuses on developing new tools to help us understand models based on differential equations using numerical approximation. Instead of developing new approximation techniques, we focus on refining the inputs to these approximation tools by optimizing the mesh points given to them. To do this, we develop neural network algorithms to optimize mesh points for a given differential equation, utilizing both the exact solution to a problem and a higher order approximation as comparison tools for our loss function.

This work was done by combining the Python packages Firedrake and Pytorch. Firedrake was used to generate the numerical approximations for us, and allowed us to easily try different permutations of problems and use higher order approximations when needed. Firedrake also allowed us to use finite-element discretizations without having to manually rediscretize our problem if we wanted to change something. Pytorch was the software used to create our neural networks that allowed us to generate meshes that over time would adapt to better approximate the solution of the desired differential equation. With these tools, we show that neural networks are a good resource for optimizing mesh points for a given differential equation without knowing the solution to the problem already.

Lay summary

Differential equations are the mathematical tool that allows us to model the world around us. In this thesis, we will showcase how the worlds of differential equations and machine learning can be used together in order to accurately approximate solutions to differential equations via neural networks. In order to do this, we will introduce both topics individually and showcase some of the challenges each of them have, but also their strengths.

After they have been introduced separately, we will motivate why it might make sense to try combining them and what new problems arise when doing so. We will see that the kind of differential equation whose solution we try to approximate will determine how difficult it is for a machine learning algorithm to learn the behaviors of the solutions we want to approximate.

Our focus is not on using a neural network to approximately solve the differential equation directly, instead we want to suitably choose an ordered set of points on a domain on which to approximate the solution values. This means our end goal is to develop neural networks that can accurately optimize this ordered set of points to best represent the behavior of the solution to a particular class of differential equations.

Finally, we will showcase some of the experiments that we completed to help us derive conclusions about the validity of combining these two topics. We then conclude with an overview of our findings and discuss possible additional work that can be done on this topic.

Acknowledgements

I would like to acknowledge and thank those whom have supported me during my time as a masters student. Particularly, I would like to thank my supervisors Scott MacLachlan and Ronald Haynes for their unwavering support, optimism and constant feedback. This would work not have been possible without either of their insights.

I would also like to thank the School of Graduate Studies and Interdisciplinary Graduate Program in Scientific Computing for the opportunity to pursue my masters and for the financial support provided to me.

Lastly, I would like to thank my partner, Amanda Hannon. She has both supported me throughout my masters as a loving partner, and provided countless editorial suggestions that helped make this thesis what it is today.

Statement of contribution

All figures/plots included in this paper are produced or owned by the author, Gerry Harris-Pink except for Figure 1.1. There are references to the original source in captions for all figures not produced by the author. Initial ideas and techniques for optimizing mesh points for singularly perturbed differential equations using neural networks were provided by Dr. Scott MacLachlan and Dr. Ronald Haynes. The final algorithms and techniques presented here were co-developed by Gerry Harris-Pink, Ronald Haynes, and Scott MacLachlan.

Table of contents

List of figures

Chapter 1

Introduction

Physics and engineering are built on mathematical models that describe the behaviour of various systems, for example, relating forces applied to a fluid or solid body to the original state of a system. Differential equations (DEs) are often the mathematical tools that arise in these problems. For complex systems we typically require a system of multiple equations to accurately model these problems. In this thesis we will focus on developing tools to help us understand these models using accurate numerical approximations. These approximations will depend on the placement of grid points in the physical domain of the DE, and we will focus on the question of how to choose those grid points to lead to the best possible accuracy in the resulting approximation. More specifically, we aim to expand the current set of options available to choose these grid points by introducing neural networks (NNs) as a possible tool, by training a NN to optimize the locations of the grid points in order to maximize the accuracy of the resulting approximation.

1.1 Boundary layers

The specific kind of DEs we want to study are ones whose solutions have boundary layers. A boundary layer is a small region near the beginning or end of the interval the problem is defined on, in which the solution rapidly changes. These boundary layers bring many challenges with them since often the solutions that satisfy the model have to heavily compensate to match the layers. In Section 2.2 we will explore why these

layers cause many issues for numerical approximations, and how we must adjust the methods in order to handle problems with boundary layers.

First, we will look at a fluid flow problem to see how a boundary layer forms in a real world example. The modified Hartmann flow problem [2] is a two-dimensional steady-state problem with a square domain $\Omega = [-L, L]^2$. For this model, we consider the flow of a charged fluid, such as a plasma, that interacts with both classical physical forces on the fluid and electromagnetic forces as well. A critical parameter in the behaviour of this model is the Hartmann number, Ha, that measures a ratio of advective to diffusive terms in the system.

The Hartmann flow problem models a portion of a channel that a fluid is flowing through, and the fluid is subjected to a transverse magnetic field $\mathbf{B_0} = (0, B_0, 0)$, which is applied perpendicularly to the fluid flow. For this problem, the fluid flow is moving in the x-direction due to a force applied by a pressure gradient $\frac{\partial p}{\partial x} = -G_0$. We assume this channel has insulating walls, and we have Dirchlet boundary conditions for the fluid velocity and magnetic vector potential on all the walls.

In this situation, the velocity has only one non-zero component u_x, and the magnetic field has components B_x, B_y. It can be shown that $B_y = B_0$. The analytical solution to this problem is $\mathbf{u}= (u_x, 0, 0)$ and $\mathbf{B}= (B_x, B_0, 0)$ where these functions are described in [2].

By choosing the physical parameters of the system, we can produce solutions corresponding to different Hartmann numbers, yielding the solutions shown in 1.1. We note that the x component of the velocity, u_x has sharp changes near the boundary of the domain. These changes are due to a physical boundary layer that arises when the advection of the fluid in the x-direction is much stronger than diffusive terms in the system.

1.2 Neural Networks

A neural network is a machine learning algorithm inspired by the ideas behind the human brain. The network is made up of a large number of processing units or neurons that are connected to one another. These neurons work in parallel to solve a specific problem, and they learn by example. Neural networks are often used to

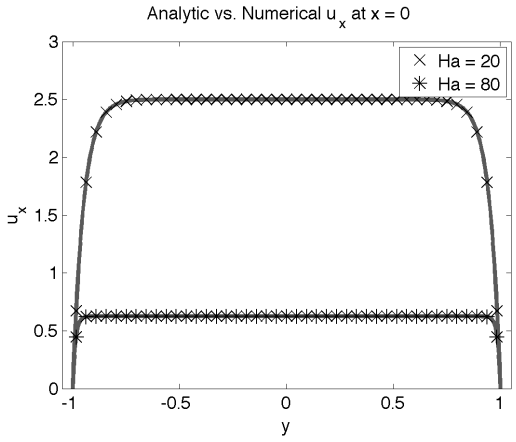

Figure 1.1: Image taken from [2] showing the numerical verification of the Block-Diagonal Braess-Sarazin solver for the Hartmann test problem. This graph compares the analytical velocity solution u_x along a cross-section at $x = 0$, shown by the solid lines, with a numerical solution, shown by the symbols, at two different values of the Hartmann number.

solve problems such as image recognition. They usually need much more data than a human to sufficiently learn how to solve the desired problem; however the speed in which they can parse each individual piece of data eclipses anything humans can do. Due to this, NNs are commonly chosen to solve optimization problems and very repetitive problems.

It is important to note that NNs are not programmed to solve a specific task like traditional computer algorithms, instead NNs rely on the data being given to them to be chosen wisely. For example, let us say we wanted to train a NN to look at pictures of animals and determine if the picture shown contains a dog. We would need to provide plenty of pictures and some flag which indicates if a picture is a dog or another animal. We also would need to provide data such as pictures of cats and birds; other kinds of animals that have their own distinct features from a dog that eventually we would want the network to learn to distinguish. Lastly, we should also include some complete outliers that we hope the network can easily realize are not dogs, such as a kitchen sink and a spaceship. It is difficult to figure out exactly how much data is required to effectively train a NN, but it is important to make sure it has a sufficiently wide range of data to study, containing difficult problems to solve, in order to learn the specific elements of the desired problem.

Suppose we choose to only train this network on pictures of dogs, and pictures of

things that are not animals. The concern with such training would be that the network would not learn the specifics of a dog such as the number of legs they have, and after training, when shown new images of animals may perform poorly. For example, there is a reasonable likelihood that if you showed this version of the network a black bear, it would claim it is a dog as it has similar shape to a dog but different dimensions, plenty of dogs are black and there are many other shared traits between dogs and black bears.

To look at this example more technically, we need to establish how exactly a NN learns. A NN takes in some number of inputs, and maps them into a numerical vectorized form that it then performs mathematical operations on. Specifically, a NN is made up of interconnected layers that work to adjust values known as weights and biases, to optimize the quality of output from the training of the NN. There are different ways to look at each layer but in our case we will assume linear layers which take the form $Ax + b$ where A is a matrix of weights, b is a vector of biases and x is the vectorized data.

It is important to also understand how a computer understands images to be able to justify the idea of a computer mapping inputs into a numerical form. An image can be translated into a collection of pixels where each individual pixel has three numbers assigned to it between 0 and 255 creating a vector, and each number represents which exact color a pixel is. A computer reads an image by representing the image as a grid of pixels and can apply mathematical operations to that data. For grey-level images there is only one value assigned per pixel instead of three meaning it is much more difficult to understand images with color for a computer. It is these values, describing the color (or greyscale value) for each pixel, that form the vector x the NN acts on. A much more in depth explanation of this is found in [14].

The layers of a NN are made up of one input layer, some number of hidden layers and one output layer. The input layer is where we feed data into the network, using the above example, the images we provide to train the network. We then take the output from each layer, and use it as the input to the next layer. The output layer produces the results, meaning that if the network thinks the picture is a dog it returns true, and the output layer is where that information comes from. The hidden layers are where the learning happens. Each hidden layer can have any number of nodes and typically they are all connected, meaning each node in each layer communicates with

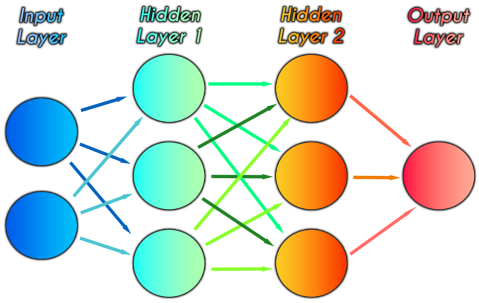

Figure 1.2: A sample fully connected network with two hidden layers.

all nodes in the next layer. However, there are some neural networks that do not have all of its layers connected that have been successful, but for the purposes of this thesis we will assume all nodes are fully connected and so are all layers. In theory, the more layers a neural network has the better it should be to solve problems. However, there are both concerns of overfitting and computational costs to keep in mind. Practically, we work with a limited size network and aim to find a reasonable network structure so that we learn sufficiently well and fast without an infinite number of layers or nodes. A sample fully connected neural network is shown in Figure 1.2 with 2 hidden layers.

It is important to note that the output layer of a NN can give multiple kinds of outputs. Given how a layer is constructed, often a true or false output is hard to exactly generate, but instead often NNs output the probability of something occurring. To continue the above example, it would be more common for a NN to return the probability that an image is a dog. One simplification that is often used if a true or false output is desired is to make an additional hidden layer that maps the probability to true or false. For example, if we wanted to claim if the NN returns an 80 percent or higher chance of a given picture being a dog, we round it to just say that is true, and anything less than 80 percent we claim to be false. This is a delicate balance. In a field like medical imaging, for example, there are good reasons to avoid such rounding if possible.

Now that we have described an example of a neural network, we need to also discuss some of the important keywords needed to understand a bit more about what is happening behind the scenes of a NN. First we have a loss function [24], which is some function $f(o)$ that we aim to minimize as we train the network where o is

the output of the neural network. This function should be chosen wisely to become large if o is farther from the expected output, and shrink as we get closer to optimal outputs. The choice of loss functions for this thesis is discussed in Section 3.1.

Backpropagation is the act of updating the state of a neural network via estimating the gradient of the loss function with respect to each of the parameters of the network. The goal is to update the parameters so that the size of the gradient of the loss function decreases in the next epoch. The details of this algorithm, including its derivation can be found in [21]. To be more specific, backpropagation takes the derivative of the loss function with respect to each of the parameters, and as each of these derivatives approach zero we consider a neural network optimal.

The next important concept is an epoch. An epoch is a complete pass of the dataset through the training algorithm, allowing each sample in a dataset the opportunity to update the parameters of the network [6]. The key idea is that as we do more epochs, in theory our network should better learn how to solve the problem it is desired to solve. Typically, per epoch we have a model determine outputs given training data and once it has determined outputs for all the data, we then update the network with backpropagation and repeat this process. It is important to note that sometimes we update the network more often than just every epoch. For example, although it would be inefficient we could backpropagate every time the network solves any problem on the dataset, but this would likely lead to overfitting [4].

The more realistic way to update the network more often than every epoch is with batching. Batching is the process of dividing your dataset into smaller datasets, solving the problem on all the pieces of data in an individual dataset and then updating the network after each dataset is complete. This can decrease the number of epochs required, since if we split a dataset into ten smaller datasets, we would update the network ten times as often. This does not always lead to better results, especially when working with loss functions that do not become large in magnitude, since it may be better to have more data analyzed before backpropagating in order to make the loss function larger. A more in depth discussion of batching is found in [12], and [6] contains discussion about the difference between batching and epochs.

Neural networks are a large scale optimization algorithm. Due to this, we need a parameter that controls how big of a step we take each epoch called the learning rate. The learning rate is used to make sure we both iterate towards the optimal network

structure but also help us stay near the optimal network structure once we reach it. A detailed explanation of the effects of a learning rate can be found in [15].

Lastly, throughout the layers of a neural network, input can pass through functions known as activation functions [22]. Simply, these activation functions take in an input, and map that input to an output based on what function is chosen. For example, the ReLU activation function takes in a number, and returns 0 if the number was negative, otherwise it returns the original input. Typically, activation functions are chosen in a way to address some concern we have with the neural networks inputs. For example, if we do not expect to have any negative outputs in our network, then we can use the ReLU activation function to ensure there are no negative outputs. It is important to note that multiple activation functions can be used in one neural network, but often we only use one kind, and either use it on every layer in the network, or only before the output layer.

To help highlight how a neural network learns, we will examine Figure 1.2 and each of its pieces. Typically, a neural network learns by completing epochs on some set of model problems, and we use the loss function to help compute the difference between the outputs from the output layer and the expected true results. These model problems are considered our training set. The goal is as we complete more epochs, the size of the loss function decreases and eventually we hope the derivative of the loss function approaches zero. The key piece to analyze is what do we change in order to make the loss function decrease over time, which is where our weights and biases come in.

In Figure 1.2 each of the lines connecting the circular nodes depicts a mapping from one node in a layer, to another node in the next layer. These lines are typically made up of the linear function $ax + b$ where a is the weight and b is the bias for that particular mapping. A learning algorithm tries to tweak the values of a and b for each layer each time it backpropagates, and as the derivative of the loss function with respect to a and b shrinks we should approach a optimal neural network. Many neural networks have a stopping criteria that makes the network stop learning once the derivative of the loss function reaches a pre-set small threshold.

1.3 Contributions of the thesis

This thesis focuses on merging two different fields of study; neural networks and numerical approximations to differential equations. In order to do so, we focus on introducing both of these areas of study individually and the challenges that arise from each of them. Then we start to showcase how these topics overlap and motivate the idea that utilizing neural networks to approximate solutions to these differential equations might be a reasonable idea.

After introducing these topics we then will demonstrate them working together and showcase the quality of the numerical results generated via a neural network. It is important to note we are not the first to try to combine these areas of study, but other papers have taken a different approach to ours. One common approach is to train the neural network to directly generate approximations to the solution of a given differential equation. Instead, we take the stance that there are strong enough numerical solvers that exist already, and focus on optimizing the data that gets sent into these numerical solvers; grid points. These grid points are what our neural network trains to optimize, setting it apart from the work others have done in this area.

Some similar work in this area comes from [18] which presents theoretical justification for why neural networks are a reasonable choice for approximating solutions to differential equations. It has become common in this area to try to approximate the solutions to differential equations using physics informed neural networks, and in [3] it highlights both the strengths and weaknesses of this structure. Specifically, it showcases that if we want to use prior knowledge about a problem to build neural networks, this structure can perform very well but for our purposes we want to use as little knowledge about a specific problem as possible.

The most similar work to ours comes in [8] and [9] which shares the same idea of optimizing the mesh points we use to perform our approximations, but instead uses a shallow neural network approach instead of a deep neural network approach. While the approach of [8] and [9] was not developed directly for differential equations with boundary layers, [9] includes preliminary results for problems similar to ours, showing promising results.

Chapter 2

Background information

In order to start using the ideas presented in Chapter 1, we need to establish the required background information to understand the methods and terminology used moving forward. This chapter focuses on establishing an interesting problem with boundary layers that we have used as our model problem for the rest of this thesis, and how we will then solve this problem numerically. To do that, we will discuss our methodology of choice, the finite-element method and then begin looking at layer adapted meshes and how these will be important to successfully utilize the finite-element method.

2.1 Our model problem

We want to start by looking at our model problem and what makes it an interesting enough choice of a problem to showcase the ideas of this thesis. Our model problem is given as

$$-\epsilon^2 u''(x) + u(x) = f(x), \tag{2.1}$$

for $0 < \epsilon < 1$ and boundary conditions $u(0) = u(1) = 0$. For the purposes of the model problem we first focus on the choice of $f(x) = e^x$, although other forcing functions will be considered in Chapter 4. We can solve this problem exactly by hand using

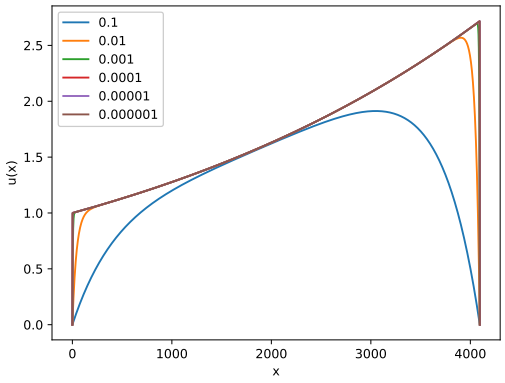

Figure 2.1: Solutions to the model problem in (2.1) for ϵ from 10^{-6} to 10^{-1}.

standard techniques for solving ODEs to find the solution

$$u(x) = \frac{(e - e^{\frac{-1}{\epsilon}})e^{(\frac{x-1}{\epsilon})} + (1 - e^{(1-\frac{1}{\epsilon})})e^{(-\frac{x}{\epsilon})}}{(\epsilon^2 - 1)(1 - e^{(-\frac{2}{\epsilon})})} - \frac{e^x}{(\epsilon^2 - 1)}. \tag{2.2}$$

We can plot this with varying ϵ to see the behavior of the true solution.

We can observe in Figure 2.1 that as $\epsilon \to 0$ the steepness of the solution at $x = 0$ and $x = 1$ increases. This is very important because it highlights the idea that as we decrease ϵ the solution of the problem has to compensate for the boundary conditions more. This will be crucial when we start looking at numerical approximations of the solutions defined in (2.2). We can also note that (2.2) contains terms with $\frac{1}{\epsilon}$ meaning that this solution is undefined at $\epsilon = 0$. This is intuitive since if $\epsilon = 0$ our equation reduces to

$$u(x) = e^x. \tag{2.3}$$

This function is defined but does not satisfy the boundary conditions, in fact $u(0) = 1$ but the boundary conditions require $u(0) = 0$ which is a contradiction. We can also see problems occur for $\epsilon = 1$ where $\epsilon^2 - 1 = 0$ meaning we divide by 0. However, there are other ways to write our model problem solution for $\epsilon = 1$ that gives the solution to be

$$u(x) = \frac{e^2}{2(1 - e^2)}e^{-x} - \frac{e^2}{2(1 - e^2)}e^x - \frac{xe^x}{2}. \tag{2.4}$$

Now we want to start thinking about approximating solutions and it is important to try to do this in some way that does not depend on the value of ϵ. For a problem

like this where we know the exact solution, it is trivial to approximate. However, we would like to be able to solve problems for which the true solution is unknown. With this in mind, we can restate our goal as how do we approximate the solution to this problem accounting for varying ϵ without utilizing the true solution.

2.2 Finite-element methods

In Section 2.1 it was mentioned that we would like to be able to solve our model problem using numerical approximations. One of the standard families of methods used to approximate solutions to boundary-value problems is that of finite-element methods, which are covered in more detail in [1]. These methods take our large problem, and break it into many smaller problems that we call elements. Then these methods try to solve each of these smaller problems and combine their solutions into one solution for the larger problem. Typically, the challenging part of finite-element methods is determining how to break the large problem into these smaller finite elements, as some ways to do this work better for some problems than others.

One way to approximate a function is to approximate its value at a finite set of points. This is because we are not able to sample all possible inputs to a function on the continuous level, so we sample a selection of inputs and these sets of inputs we will call meshes. A mesh is an ordered list of points on some domain with some amount of spacing between them, and the goal will be to approximate our solution based solely on approximate values of the solution at the mesh points. We denote the interval between two arbitrary consecutive points on a mesh to be $I_i = [x_{i-1}, x_i]$. Typically we start with a uniform mesh, which is a mesh where the distance between each consecutive pair of points is the same. This is shown in Figure 2.2. This is a reasonable starting point for many problems however, this approach can perform poorly if the solution to a DE begins to change rapidly in some region. As we will see below, uniform meshes are a particularly bad choice for problems with boundary-layer behaviour, so we continue this section considering non-uniform meshes, where the length of I_i can vary dramatically with i.

In order to break things into finite elements we must write the original problem we want to solve into a weak form. The original problem is considered a strong form, where all boundary conditions must hold and derivatives must be defined. To illustrate

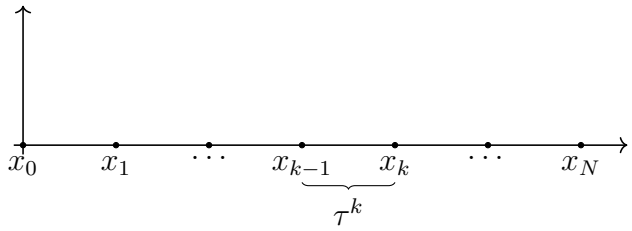

Figure 2.2: The one-dimensional uniform mesh.

what a weak form is, we will rewrite (2.1) by first choosing a smooth function $v(x)$ with $v(0) = 0$ and $v(1) = 0$. Then we can multiply both sides of (2.1) by $v(x)$ to get

$$(-\epsilon^2 u''(x) + u(x))v(x) = f(x)v(x). \tag{2.5}$$

We can then integrate both sides to get

$$\int_0^1 (-\epsilon^2 u''(x) + u(x))v(x)dx = \int_0^1 f(x)v(x)dx. \tag{2.6}$$

Using integration by parts on the left side we get

$$-\epsilon^2 u'(x)v(x)\Big|_0^1 - (-\epsilon^2)\int_0^1 u'(x)v'(x)dx + \int_0^1 u(x)v(x)dx, \tag{2.7}$$

which simplifies to

$$\int_0^1 \epsilon^2 u'(x)v'(x)dx + \int_0^1 u(x)v(x)dx = \int_0^1 f(x)v(x)dx. \tag{2.8}$$

The first term in (2.7) vanishes due to $v(0) = 0$ and $v(1) = 0$. From here we want to define the space of square-integrable functions on [0,1] as

$$L^2([0,1]) = \left\{ f : [0,1] \to \mathbb{R} \middle| \int_0^1 (f(x))^2 dx < \infty \right\}. \tag{2.9}$$

The space $L^2([0,1])$ is a Hilbert space, giving us an inner product defined by $\langle u(x), v(x)\rangle = \int_0^1 u(x)v(x)dx$. We can define the L^2 norm on $[0,1]$ as

$$\|u\|^2 = \langle u, u\rangle. \tag{2.10}$$

Moving forward any norm without a subscript implies the L^2 norm defined in (2.10).

We also use this inner product to define a bilinear form.

$$a(u,v) = \epsilon^2\langle u', v'\rangle + \langle u, v\rangle. \tag{2.11}$$

We note that $a(u,v)$ also defines an inner product, although that will not be proven in this thesis. We now will require $a(u,v)$ to be finite, in order for this to be true we need both $a(u,u)$ and $a(v,v)$ to be bounded above. This allows us to define the function space

$$\mathcal{V} = \left\{ v \in L^2([0,1]) \,\middle|\, v(0) = v(1) = 0 \text{ and } a(v,v) < \infty \right\}. \tag{2.12}$$

We also need to define the H^1 norm, which is a measure of energy or smoothness of a function. We define the H^1 function space as

$$H^1([0,1]) = \left\{ f : [0,1] \to \mathbb{R} \middle| \int_0^1 (f(x))^2 + (f'(x))^2 dx < \infty \right\}. \tag{2.13}$$

Following this definition, we define the H^1 norm on $[0,1]$ as

$$\|u\|_1^2 = \|u\|^2 + \|u'\|^2, \tag{2.14}$$

where the norms on the right side are L^2 norms. It is important to note that since $a(v,v)$ is finite, the H^1 norm is also finite. This implies that $a(u,v)$ defines both an inner product and an induced norm on $\mathcal{V}$, which is equivalent to the H^1 norm. This allows us to say that if $v \in \mathcal{V}$ then v is continuous, because (in 1D) $a(v,v)$ can only be finite if v is continuous. Thus, the weak form of this differential equation is to find $u \in \mathcal{V}$ such that

$$a(u,v) = \langle f, v\rangle \ \forall v \in \mathcal{V} \tag{2.15}$$

where $a(u,v)$ is defined in (2.10). Throughout the rest of this section, all mentions of $a(u,v)$ will refer to (2.10).

There are two other norms we will utilize throughout this thesis, first being the H^1 seminorm which we define as

$$|u|_1^2 = \langle u', u'\rangle. \tag{2.16}$$

The other is the energy norm defined as

$$\|u\|_e^2 = a(u,u) = \epsilon^2\langle u', u'\rangle + \langle u, u\rangle. \tag{2.17}$$

The main idea here is that when $u(x)$ is a solution to the strong form, it must also be a solution to the weak form. It turns out the converse is true if the solution to the weak form is sufficiently smooth then it is also a solution to the strong form. This is proven in Section 4.1 of [1]. Now we can focus on solving an integral based problem for the weak solution which is much easier to develop tools for compared to the general strong form.

Before we continue the discussion of finite elements, it is important to look at a general equation to note that not all weak forms are well posed. Instead, we need to look at additional properties of the bilinear form, $a(u,v)$.

Definition 2.1 ($\mathcal{V}$ - ellipticity)**.** *Given a Hilbert space, $\mathcal{V}$ with induced norm $||.||_{\mathcal{V}}$ we consider a bilinear form*

$$a(\cdot,\cdot) : \mathcal{V} \times \mathcal{V} \to \mathbb{R}, \tag{2.18}$$

$a(\cdot,\cdot)$ is coercive if there exists a constant $c_0 > 0$ such that

$$c_0||u||_{\mathcal{V}}^2 \leq a(u,v) \text{ for all } u \in \mathcal{V}. \tag{2.19}$$

The bilinear form $a(\cdot,\cdot)$ is continuous if there exists a constant $c_1 > 0$ such that

$$|a(u,v)| \leq c_1||u||_{\mathcal{V}}||v||_{\mathcal{V}} \text{ for all } u, v \in \mathcal{V}. \tag{2.20}$$

If both of these conditions are met, then $a(u,v)$ is said to be $\mathcal{V}$-elliptic.

With this in mind, we can now state the Lax-Milgram theorem.

Theorem 2.1 (Lax-Milgram)**.** *Let $\mathcal{V}$ be a Hilbert space and let $a(\cdot,\cdot)$ be a $\mathcal{V}$-elliptic bilinear form. We also assume that $g(\cdot)$ is a bounded linear functional on $\mathcal{V}$. Then there exists a unique $u \in \mathcal{V}$ such that*

$$a(u,v) = g(v) \text{ for all } v \in \mathcal{V}. \tag{2.21}$$

Another property that arises from $\mathcal{V}$-ellipticity is given by Céa's Lemma.

Theorem 2.2 (Céa's Lemma)**.** *Let $\mathcal{V} \subseteq H$ be a closed subspace of a Hilbert space H. Let $a(\cdot,\cdot)$ be a $\mathcal{V}$-elliptic form on $\mathcal{V}$. Lastly, for a bounded linear functional $g(\cdot)$, on $\mathcal{V}$, let $u \in \mathcal{V}$ satisfy*

$$a(u, v) = g(v) \text{ for all } v \in \mathcal{V}. \tag{2.22}$$

We consider a finite-dimensional subspace $\mathcal{V}^h \subseteq \mathcal{V}$ and $u^h \in \mathcal{V}^h$ that satisfies

$$a(u^h, v^h) = g(v^h) \text{ for all } v^h \in \mathcal{V}^h. \tag{2.23}$$

Then

$$||u - u^h||_{\mathcal{V}} \leq \frac{c_1}{c_0} \min_{v^h \in \mathcal{V}^h} ||u - v^h||_{\mathcal{V}} \tag{2.24}$$

where c_0 and c_1 are the coercivity and continuity constants for $a(\cdot,\cdot)$ respectively.

For our model problem 2.1, $c_0 = \epsilon^2$ and $c_1 = 1$. To show this we start with coercivity and derive the bound

$$a(u, u) = \epsilon^2 ||u'||^2 + ||u||^2 \geq \epsilon^2 ||u||_1^2 \tag{2.25}$$

for $\epsilon < 1$. This gives a lower bound of $c_0 = \epsilon^2$ where $||\cdot||_1$ represents the H_1 norm. For continuity, we use the triangle inequality and the Cauchy Schwarz inequality to show

$$\begin{aligned}
a(u, v) &= \epsilon^2 \langle u', v' \rangle + \langle u, v \rangle && (2.26)\\
&\leq \epsilon^2 ||u'||||v'|| + ||u||||v|| && (2.27)\\
&= \begin{bmatrix} ||u|| \\ \epsilon||u'|| \end{bmatrix} \begin{bmatrix} ||v|| \\ \epsilon||v'|| \end{bmatrix} && (2.28)\\
&\leq (||u||^2 + \epsilon^2 ||u||^2)^{1/2} (||v||^2 + \epsilon^2 ||v'||^2)^{1/2} && (2.29)\\
&\leq (||u||^2 + ||u'||^2)^{1/2} (||v||^2 + ||v'||^2)^{1/2} && (2.30)\\
&= ||u||_1 ||v||_1 && (2.31)
\end{aligned}$$

giving us a continuity constant of $c_1 = 1$.

Now that we have continuity and coercivity constants we know our model problem is considered $\mathcal{V}$ - elliptic. Importantly, we also know our weak form does have a unique solution in our space $\mathcal{V}$ and we can bound the quality of this solution. We also can

look at the bound from 2.2 for our model problem to see we get

$$||u - u^h||_{\mathcal{V}} \leq \frac{1}{\epsilon^2} \min_{v^h \in \mathcal{V}^h} ||u - v^h||_{\mathcal{V}}. \tag{2.32}$$

We can see that our error is bounded by $\frac{1}{\epsilon^2}$ so as $\epsilon \to 0$ this bound grows rapidly. Thus, unless we can ensure that $\min_{v^h \in \mathcal{V}^h} \|u - v^h\|_{\mathcal{V}}$ is bounded by a factor smaller than ϵ^2, we expect to have difficulties in achieving good approximation. We now develop the standard theory for finite-element approximation, making use of the Ritz-Galerkin framework.

Definition 2.2 (Ritz-Galerkin approximation)**.** *Let $a : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$ be a bilinear form, and let $\mathcal{V}^h$ be a finite-dimensional subspace of $\mathcal{V}$. Then we find $u^h \in \mathcal{V}^h$ such that*

$$a(u^h, v) = \langle f, v \rangle \ \forall \ v \in \mathcal{V}^h. \tag{2.33}$$

We say that u^h is the Ritz-Galerkin approximation of the weak solution $u \in \mathcal{V}$.

In order for this to be useful, we need to examine uniqueness and existence of solutions to (2.33).

Theorem 2.3 (Existence and Uniqueness)**.** *If $f \in L^2([0,1])$, then there must exist a unique solution to the Ritz-Galerkin approximation assuming $\mathcal{V}^h \subseteq \mathcal{V}$.*

This is proven in [1]. In order to study the relationship between the weak form of the solution and the Ritz-Galerkin approximation, we need to develop results about the quality of this approximation. First, we need to look at an orthogonality property in the energy inner product.

Theorem 2.4 (Orthogonality relationship)**.** *If $u \in \mathcal{V}$ is the solution of the weak form and $u^h \in \mathcal{V}^h$ is the solution to the Ritz-Galerkin approximation, then $a(u - u^h, v) = 0, \forall v \in \mathcal{V}^h$.*

Proof. As u^h is the Ritz-Galerkin approximation, it must satisfy the property $a(u^h, v) = \langle f, v \rangle, \forall v \in \mathcal{V}^h$. Also, since u is the solution to the weak form it requires $a(u, v) = \langle f, v \rangle, \forall v \in \mathcal{V}$. Given that $\mathcal{V}^h \subseteq \mathcal{V}$ which implies $a(u, v) = \langle f, v \rangle, \forall \mathcal{V} \in \mathcal{V}^h$. Thus, $a(u - u^h, v) = a(u, v) - a(u^h, v) = \langle f, v \rangle - \langle f, v \rangle = 0 \ \forall v \in \mathcal{V}^h$. □

Next we want to determine how to quantify the best approximation error in the subspace $\mathcal{V}^h$. We need to define a mesh in order to do this. A mesh, Ω^h, on $[0,1]$ is an ordered set of points, $0 = x_0 < x_1 < x_2 < \ldots < x_N = 1$, defining the intervals $I_i = [x_{i-1}, x_i]$ for $1 \leq i \leq N$. Then we define the piecewise polynomial space of degree at most k.

Definition 2.3. *For integer $k \geq 1$ we define*

$$P_k(I_i) = \{u : I_i \to \mathbb{R} \,|\, u \text{ is a polynomial of degree at most } k\}. \tag{2.34}$$

Then given a mesh Ω^h on $[0,1]$

$$\mathcal{V}_k^h = P_k(\Omega^h) = \left\{u \in C^0([0,1]) \middle| u(x) \in P_k(I_i), \forall i \text{ and } u(0) = 0, u(1) = 0\right\} \tag{2.35}$$

where

$$C^m([0,1]) = \{v | v(x) \text{ is } m \text{ times continuously differentiable on } [0,1]\} \tag{2.36}$$

We specifically want to look at when $k = 1$, and $\mathcal{V}_1^h$ is the space of continuous piecewise linear functions. We note that for any function $u \in P_1(\Omega^h)$ we can write

$$u = \sum_{i=1}^{N-1} u_i \phi_i(x), \tag{2.37}$$

defining basis functions for $\mathcal{V}_1^h$ using a basis function $\phi_i(x)$ that satisfies

$$\phi_i(x_j) = \delta_{i,j}, \tag{2.38}$$

for all grid points x_j where $\delta_{i,j}$ is the Kronecker delta function [23]. These are called nodal basis functions and for $1 \leq i < N$ we define them to be

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}}, \text{ for } x \in [x_{i-1}, x_i], \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, \text{ for } x \in [x_i, x_{i+1}], \\ 0 \text{ otherwise.} \end{cases} \tag{2.39}$$

We could extend (2.39) to define basis functions associated with x_0 and x_N but we do not need these, since any function $u \in P_1(\Omega^h)$ satisfies $u(0) = u(1) = 0$. Lastly

we can check that $a(\phi_i, \phi_i) < \infty$ for $1 \leq i \leq n$ and see that $\mathcal{V}_1^h = \text{ span}\{\phi_1 \cdots, \phi_N\}$. For the purposes of this thesis, we will focus on $\mathcal{V}_1^h$, and thus we will refer to it as $\mathcal{V}^h$ moving forward. A standard bound on the right-hand side of (2.32) arises from quantifying an approximation assumption for the space $\mathcal{V}^h$.

Definition 2.4 (Approximation assumption)**.** *Given $\mathcal{V}^h \subset \mathcal{V}$, the approximation assumption for $\mathcal{V}^h$ is that $\exists \sigma > 0$ such that $\forall w \in C^2([0,1]) \cap \mathcal{V}, \min_{v \in \mathcal{V}^h} |w - v|_1 \leq \sigma ||w''||$.*

To show that the approximation property holds for $\mathcal{V}_1^h$ we show that for any $w \in C^2([0,1]), \exists w^h \in \mathcal{V}_1^h$ such that $|w - w^h|_1 \leq \sigma ||w''||$. Typically for finite elements, we prove this relationship instead using the interpolant of w.

Definition 2.5 (Interpolant)**.** *Let $w \in \mathcal{V}$ and Ω^h be given. The piecewise linear interpolant of w is the function $w_I^h(x) \in \mathcal{V}^h$ given by $w_I^h(x) = \sum_{i=1}^{N-1} w(x_i)\phi_i(x)$, where x_i are the nodes of the mesh.*

We can see that from this definition, if $w \in \mathcal{V}_1^h$ then $w = w_I^h$. This means that any function in $\mathcal{V}_1^h$ is its own interpolant. We can now bound $|w - w_I^h|_1$, by proving the required approximation property in terms of our maximum value of h_i for $1 \leq i \leq N$, defined as

$$h_i = x_i - x_{i-1}. \tag{2.40}$$

Theorem 2.5 (Approximation property for $\mathcal{V}_1^h$)**.** *Given a mesh Ω^h on $[0,1]$, let $h = \max_{1 \leq i \leq N} h_i, w \in C^2([0,1]) \cap \mathcal{V}$. Then*

$$|w - w_I^h|_1 \leq \frac{h}{\sqrt{2}} ||w''||. \tag{2.41}$$

The proof of this theorem can be found in [1]. We can apply this to the Ritz-Galerkin approximation

Corollary 2.1. *Let $f \in C^0([0,1]), u \in C^2([0,1]) \cap \mathcal{V}$. Let u^h be the Ritz-Galerkin approximation of $u \in \mathcal{V}$ over $\mathcal{V}_1^h$, defined by $a(u^h, v) = \langle f, v \rangle \ \forall v \in \mathcal{V}_1^h$. Then*

$$||u - u^h|| \leq \frac{h}{\sqrt{2}} |u - u^h|_1 \leq \frac{h^2}{2} ||u''||. \tag{2.42}$$

With 2.2 and 2.1 we are able to now combine these to give a bound on the Ritz-Galerkin approximation for our problem which importantly, depends on both h and ϵ, this bound is shown in (2.43), where we notice that the bound scales like $\frac{h}{\epsilon^2}$.

$$||u - u^h||_1 \leq \frac{1}{\epsilon^2}||u - u_I^h||_1 \leq \frac{1}{\epsilon^2}\frac{h}{\sqrt{2}}||u''||. \tag{2.43}$$

Before even considering the size of $\|u''\|$, this shows that the standard bound gives us no guarantee that the Ritz-Galerkin approximation will be a good approximation. Even for mild values of ϵ, such as $\epsilon = 0.01$, this would require that the largest mesh interval be no larger than $h = 10^{-4}$ for this bound to guarantee a good approximation. Instead, in Section 2.3 we will explore other ways to construct a mesh that does not require such a strict bound.

2.3 Layer adapted meshes

In Section 2.2 we concluded that the standard bound on finite-element approximation does not offer a suitable guarantee of approximation quality for solutions to equations like (2.1). In this section we will introduce the idea of adapted meshes which are prescribed meshes that are adapted to the expected behaviour of the solution to the problem. There are a few kinds of layer adapted meshes that were used as reference meshes in this thesis, one of which is the Shishkin mesh.

The Shishkin mesh is a mesh obtained by breaking the domain into 3 uneven subdomains and using a uniform spacing within each subdomain. The key to this grid is figuring out where to divide those domains. Shishkin meshes are often used for problems where we believe the interesting and hard to approximate behavior occurs near the end points of the interval, so we cluster more points around those areas than a uniform grid would. For problems on intervals with 2 boundary layers, this grid typically assigns one fourth of its points to the beginning of the interval, another one fourth to the end of the interval and the last half of the points to the middle. Importantly, the Shishkin mesh makes the assumption that the parts of the solution that are far from the boundary layers are easy to approximate, meaning the points in this section can be further spread apart than those with boundary layers. The Shishkin mesh can be formed using the following structure; we first choose a τ to represent where we begin to divide each of our domains. The Shishkin mesh transition points are defined as

$$\tau = \min\left\{1/4, 2\beta^{-1}\epsilon\log(N)\right\}. \tag{2.44}$$

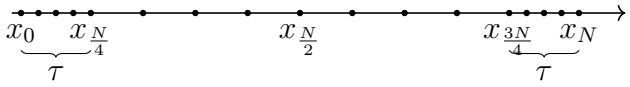

Figure 2.3: The one-dimensional Shishkin mesh

These transition points can be found in [16]. For our model problem we have $\beta = 1$. Then the intervals $[0, \tau]$ and $[1-\tau, 1]$ are made into $\frac{N}{4}$ subintervals, while $[\tau, 1-\tau]$ are divided into $\frac{N}{2}$ subintervals. So we have

$$\Omega^N = \{x_i | i = 0, \ldots, N\}, \tag{2.45}$$

where

$$x_i = \begin{cases} \frac{4i\tau}{N}, & i = 0, \ldots, \frac{N}{4} \\ \tau + \frac{4(0.5-\tau)(i-\frac{N}{4})}{N}, & \frac{N}{4}+1 \ldots, \frac{3N}{4}-1, \\ 1-\tau+\frac{4\tau(i-\frac{3N}{4})}{N}, & i = \frac{3N}{4} \ldots, N. \end{cases} \tag{2.46}$$

The Shishkin mesh is shown in Figure 2.3.

From [16, Equation 6.4], for the remainder of this section we consider a function $\psi \in C^2[0,1]$ that admits the derivative bounds

$$|\psi''(x)| \leq C(1 + \epsilon^{-2}(e^{\frac{-x}{\epsilon}} + e^{\frac{-(1-x)}{\epsilon}})) \tag{2.47}$$

noting that this is true of the solution to (2.1) and related problems. We can bound the energy norm and L^2 norm error of the difference between ψ and its interpolant with the following theorem.

Theorem 2.6 (See [16], Theorem 6.2)**.** *Suppose ψ satisfies* (2.47)*. Then*

$$||\psi - \psi_I^h|| \leq C(\vartheta_{rd}^2(\Omega^h))^2 \tag{2.48}$$

and

$$||\psi - \psi_I^h||_e \leq C(\epsilon^{\frac{1}{2}} + \vartheta_{rd}^2(\Omega^h))\vartheta_{rd}^2(\Omega^h), \tag{2.49}$$

for

$$\vartheta_{rd}^2(\Omega^h) = \max_{i=1,\ldots,N} \int_{I_i} (1 + \epsilon^{-1} e^{\frac{-s}{2\epsilon}} + \epsilon^{-1} e^{\frac{-(1-s)}{2\epsilon}}) ds. \tag{2.50}$$

We can also give similar bounds on our solution and approximation with the following theorem.

Theorem 2.7 (See [16], Theorem 6.6)**.** *Let u be the solution of* (2.1) *and u^h be its finite-element approximation in the space $\mathcal{V}_1^h$ over a given mesh Ω^h. Also let u_I^h be the nodal interpolant of u. Then*

$$||u^h - u_I^h||_e \leq C(\vartheta_{rd}^2(\Omega^h))^2 \tag{2.51}$$

and

$$||u - u^h||_e \leq C(\epsilon^{\frac{1}{2}} + \vartheta_{rd}^2(\Omega^h))\vartheta_{rd}^2(\Omega^h). \tag{2.52}$$

With some work, we can bound (2.50) on a Shishken mesh to get

$$\vartheta_{rd}^2(\Omega^h) \leq CN^{-1}\ln N. \tag{2.53}$$

We can then apply this bound to (2.52) to get

$$||u - u^h||_e \leq C(\epsilon^{\frac{1}{2}} + CN^{-1}\ln N)CN^{-1}\ln N. \tag{2.54}$$

for a suitable C. This gives first-order convergence in the energy norm. An in depth proof of this result can be found in Section 2.2 of [16].

In higher dimensions, we construct tensor-product grids to generalize this. On the two-dimensional domain $\Omega = [0,1]^2$, we can define

$$\tau = \min\{\frac{1}{4}, 2\epsilon\beta^{-1}\ln(N)\}. \tag{2.55}$$

Then we partition Ω as $\overline{\Omega} = \Omega_{11} \cup \Omega_{21} \cup \Omega_{12} \cup \Omega_{22}$ where

$$\begin{aligned}
&\Omega_{11} = [\tau, 1-\tau] \times [\tau, 1-\tau],\\
&\Omega_{21} = ([0,\tau] \cup [1-\tau, 1]) \times [1-\tau, 1],\\
&\Omega_{12} = [\tau, 1-\tau] \times ([0\tau] \cup [1-\tau, 1]),\\
&\Omega_{22} = ([0,\tau] \times ([0,\tau] \cup [1-\tau, 1])) \cup ([1-\tau, 1] \times ([0,\tau] \cup [1-\tau], 1)).
\end{aligned} \tag{2.56}$$

Then we denote a mesh on Ω as $0 = x_0 < x_1 < \ldots < x_N = 1$ and $0 = y_0 < y_1 < \ldots < y_N = 1$ with mesh sizes $H_i = x_i - x_{i-1}$, and $K_j = y_j - y_{j-1}$ defined by

$$H_i = \left\{ \begin{array}{ll} H_1 = 4\tau N^{-1}, & \text{for } i = 1, \ldots, \frac{N}{4}; \frac{3N}{4} + 1, \ldots, N, \\ H_2 = 2(1-2\tau)N^{-1}, & \text{for } i = \frac{N}{4} + 1, \ldots, \frac{3N}{4}, \end{array} \right\} \tag{2.57}$$

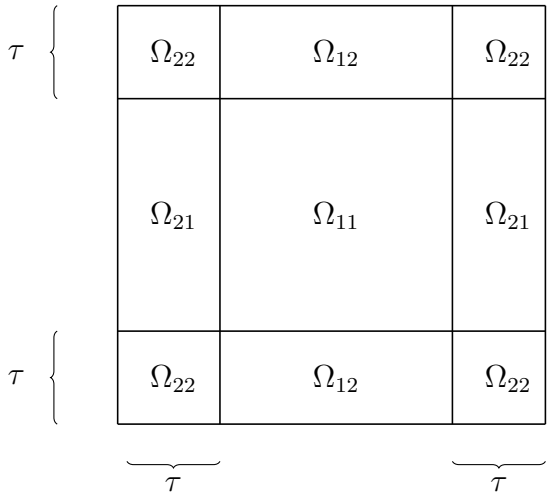

Figure 2.4: The Shishkin mesh in two dimensions

and

$$K_j = \left\{ \begin{array}{ll} K_1 = 4\tau N^{-1}, & \text{for } j = 1, \ldots, \frac{N}{4}; \frac{3N}{4} + 1, \ldots, N, \\ K_2 = 2(1-2\tau)N^{-1} & \text{for } j = \frac{N}{4} + 1, \ldots, \frac{3N}{4}, \end{array} \right\} \tag{2.58}$$

for positive even integers $N \geq 4$. The domain can be partioned to take the form in Figure 2.4.

There are other suitable choices of meshes. The Shishkin mesh requires us to know a lot about the solution to the problem in advance, for example knowing there are complicated boundary layers at both ends of the domain. For a more general approach that relies less on knowing the solution in advance, we can look into mesh equidistribution.

Definition 2.6 (Equidistribution principle)**.** *Let $M : [0,1] \to \mathbb{R}$ be a positive function. A mesh equidistributes the monitor function M if*

$$\int_{x_{i-1}}^{x_i} M(t)dt = \frac{1}{N} \int_0^1 M(t)dt \text{ for } i = 1, \ldots, N. \tag{2.59}$$

Given a monitor function M the associated mesh generating function $x(\xi)$ can be implicitly written as

$$\int_0^{x(\xi)} M(t)dt = \xi \int_0^1 M(t)dt \text{ for } \xi \in [0,1]. \tag{2.60}$$

Before we look into utilizing the equidistribution principle, the monitor function should be discussed. There is no one size fits all monitor function. Typically, we pick a behavior we want to study, maybe the size of the derivative or second derivative of the solution to our DE and then try to pick a monitor function that rapidly increases where the behavior we are interested in becomes more prevalent. This means some monitor functions are better than others for specific problems. For example, for our model problem we want to pick a monitor function that becomes large near the end points of our interval. This can be done via the first derivative of the solution function since the function undergoes a rapid rate of change, or if we had a problem with layers in the middle of the function we may want to study the second derivative of the solution function.

The Bakhvalov mesh is a mesh found via equidistribution of the monitor function

$$M_{Ba}(s) = \max\{1, K_0\rho\epsilon^{-1}e^{\frac{-\rho s}{\sigma_0\epsilon}}, K_1\rho\epsilon^{-1}e^{\frac{-\rho(1-s)}{\sigma_1\epsilon}}\} \tag{2.61}$$

for suitable choices of $K_0, K_1 > 0$, $\rho > 0$ and $\sigma_0, \sigma_1 > 0$.

We are also able to bound (2.50) on a Bakhvalov mesh. Since our solution to the reaction-diffusion problem on the mesh satisfies (2.47) we get the bound

$$\vartheta_{rd}^2(\Omega^h) \leq CN^{-1}. \tag{2.62}$$

We are also able to bound (2.50) on a Bakhvalov mesh, getting

$$||u - u^h||_e \leq C(\epsilon^{\frac{1}{2}} + CN^{-1})CN^{-1}. \tag{2.63}$$

for a suitable C. We can also compare this to (2.54) to see the Bakhvalov mesh does give a slightly better convergence result as it does not have the $\ln N$ factor. This gives first-order convergence in the discrete maximum norm while being independent of ϵ. An in-depth proof of this result can be found in Section 2.2 of [16].

For the sake of this thesis, we have utilized the Bakhvolov mesh as a reference mesh, and our goal is to train a NN to generate a mesh that is capable of outperforming both the Shishkin and Bakhvalov mesh in an appropriate choice of norm.

One key point for equidistributing meshes, is that a mesh generating function represents a mapping from $[0, 1] \to [0, 1]$ where $x(\frac{i}{N})$ defines the mesh points. In

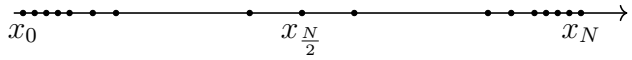

Figure 2.5: The one-dimensional Bakhvalov mesh with $\sigma = 2$, $\epsilon = 0.01$, $\rho = 0.5$.

order to do this, we do a change of variables allowing us to write $\hat{u}(\xi) = u(x(\xi))$ and $\hat{v}(\xi) = v(x(\xi))$ for $x = x(\xi)$. Now we are going to look at our weak form and see how it changes with the change of variables. We can restate our weak form of our model problem and apply the change of variables to it, starting with (2.64).

$$a(u, v) = \int \epsilon^2 u' v' dx + \int uv dx \tag{2.64}$$

Rewriting our dx gives us

$$dx = \frac{dx}{d\xi} d\xi. \tag{2.65}$$

We can also apply the chain rule to see how the derivative of our solution changes

$$\frac{\partial}{\partial \xi}(u(x(\xi))) = \frac{\partial \hat{u}}{\partial x}\frac{\partial x}{\partial \xi} \tag{2.66}$$

and

$$\frac{\partial}{\partial \xi}(v(x(\xi))) = \frac{\partial \hat{v}}{\partial x}\frac{\partial x}{\partial \xi}. \tag{2.67}$$

Simplifying this gives us

$$\frac{\partial u}{\partial x} = \left(\frac{\partial x}{\partial \xi}\right)^{-1} \frac{\partial \hat{u}}{\partial \xi} \tag{2.68}$$

and

$$\frac{\partial v}{\partial x} = \left(\frac{\partial x}{\partial \xi}\right)^{-1} \frac{\partial \hat{v}}{\partial \xi}. \tag{2.69}$$

Applying this to (2.64) we get

$$a(u, v) = \epsilon^2 \int_0^1 \left(\left(\frac{dx}{d\xi}\right)^{-2} \left(\frac{\partial \hat{u}}{\partial \xi}\frac{\partial \hat{v}}{\partial \xi}\right)\right) \frac{dx}{d\xi} d\xi + \int_0^1 \left(\hat{u}\hat{v}\right) \frac{dx}{d\xi} d\xi. \tag{2.70}$$

Although this is not discussed heavily throughout the thesis, this is crucial for allowing us to create numerical implementations of adaptive mesh methods as it preserves differentiability which is required for our NNs to be able to backpropagate. While using automatic differentiation tools to backpropagate through direct changes

in the mesh is possible (and was done in initial work in this thesis), Firedrake [13] provides much better tools for backpropagation that rely on defining the mesh transformation $x(\xi)$ instead of directly manipulating grid points.

2.4 Neural networks and approximating differential equations

Approximating differential equations utilizing neural networks is not a new concept, with conventional approaches often skipping the idea of generating a mesh and instead focusing on learning the actual solution to the differential equation. These approaches have had varying amounts of success depending on how the neural network is structured and what kind of problem is being solved.

One of the most common approaches to approximating solutions to differential equations with neural networks are physics informed neural networks (PINNs) [10], [20], [7], [11]. PINNs are a branch of NNs that can embed the knowledge of any physical laws that govern a dataset during the training process. For example, if our dataset studies the construction of a box, it should never have a negative volume because negative volume does not exist in the real world. Although we did not use a PINN for the purposes of this thesis, many papers that try to approximate solutions to differential equations are focused on utilizing PINNs.

A paper that focuses on PINNs and improving upon them is [3]. This paper shows that PINNs are a powerful tool for approximating solutions to differential equations, but can struggle in cases such as when a boundary layer is thin. This paper shows some steps that can be taken for PINNs to help perform better during these cases. The key difference in this and other PINNs papers from our work is that they are prescribing information about the problem they want to solve into the neural network before training it. This does help set the network up for success, but our goal has always been to minimize the amount of information our neural network requires about our specific problem in order to make it more robust at solving other problems.

Another similar study was done in [17] which focused on utilizing a feed forward neural network to approximate solutions to partial differential equations. The major difference in this paper from our work is they again focus on actually optimizing the

solution to the differential equation using the neural network, skipping mesh generation entirely. They also crucially transform the piecewise linear activation functions used to splines using constraints on the weights of the network. The specific spline utilized is a polynomial spline using the Chapeau function but requires the weights and biases to be updated based upon the Chapeau function as well. This paper also relied on some post-processing of the network which is not something we experimented with. This performed well and gave results within an expected theoretical order of accuracy.

A similar study to ours was done in [9] which focused on minimizing the H^1 norm of the error for similar problems as we have been solving. This paper used the damped block Newton (dBN) method as a fast iterative solver for 1D diffusion problems as well as a damped block Gauss-Newton (dBGN) method. The differential equations presented in this paper were solved with a piecewise linear finite element scheme. [9] and [8] focus on applying shallow neural network structure to generate meshes for many different problems, but do not focus on the case of problems with boundary layers. Our work does focus on this case and compares our work to classical approaches to solve problems with boundary layers. Our work also utilized the tanh activation function while [9] and [8] utilize the rectified linear unit (ReLU) activation function.

These specific activation functions are discussed more in Section 3.3 however we found that the ReLU activation function was unable to detect smaller changes in the quality of results from our network as we get closer to our optimal result. This means that the ReLU activation function sometimes would be great for our initial descent towards the optimum, but refused to put mesh points close enough together to get within quasi-optimal of the optimal mesh. [8] is a part 2 of [9] which expands on this work.

One paper that showcases the theory behind applying neural networks to approximate solutions to differential equations is [18]. This paper focuses entirely on what results are possible in theory using neural networks, helping showcase that work like the experiments in this thesis are worth trying. It does not contain any numerical experiments, and does not account for any real world limitations such as processing power. Due to this, there may be some problems showcased in this paper that in order to use a neural network to approximate, may require an extremely deep network, or

other unrealistic approaches in order to make work.

Chapter 3

Methodology

Now that we have laid out the numerical methods required to approximate the solution to the model problem, and what a neural network is, we need to discuss combining these ideas. In order to do so, we need to break down the specific pieces of our neural network and see how we apply those to fit the ideas of a numerical approximation.

It is important to note that all computations will be done via Pytorch [19] and Firedrake [13]. Pytorch is a python package designed to create NNs, allowing us to structure our network however we desire without having to handle any of the back-end ourselves. Firedrake is a software package designed to numerically approximate solutions to differential equations by writing the problem in its weak form, and then approximating the solution on an appropriate mesh and function space. Combining these two packages was challenging as integrating them together became problematic, but once they were able to work together it became trival to conduct a variety of experiments in a timely manner. In order to combine these two softwares, we primarily had to deal with challenges that came from the data types they work with. Pytorch works on torch tensors, while Firedrake works on Numpy arrays via PYOP2. The issue with mixing and matching these data types is preserving differentiability since all variables need to be differentiable for Pytorch to be able to perform backpropagation. There has been considerable work done on the integration between Firedrake and Pytorch, and an in depth discussion of this can be found in [5].

Throughout this chapter, we will establish our neural network structure and tune the parameters to help build a robust neural network. The goal is then to use the results of this chapter to answer questions posed in Chapter 4. For each experiment,

we will also be retraining the neural network instead of reusing the same network throughout all experiments. When a network is initialized, we set the weights and biases of each layer to 0 by default.

3.1 Loss function

When designing a NN, one of the key steps in the process is determining what we are interested in minimizing. Once it is decided what we want to minimize, there are many tools we can use to measure how well we are performing this task. These tools are known as loss functions. There is not one perfectly correct loss function for all problems, instead a loss function needs to be some unit of measure that accurately picks up changes in results made by our neural network. For example, if we know the optimal solution for some problem, we would want a loss function that gets smaller as we approach the optimal solution. This would mean by minimizing the loss function, we approach the desired solution.

In our case, we want to minimize some mathematical error function that shows how close we are to the desired solution to the DE. One challenge with this is making sure this error function is robust enough at finding slight discrepancies in our error using only limited data. We have chosen to use a L^2 norm as this function which is computed by Firedrake. This performed well, so other options were not explored in this thesis. This is spoken about more in Chapter 4.

We implemented two versions of this loss function, given the model problem in (2.1) we first compared the results of our neural network to the exact solution given in (2.2). Due to this being the exact solution, we know that we are comparing against the true optimum values for any given point on our mesh. To show how this applies to (2.9) we let

$$L_1(\Omega^h, u^h, u) = \|u^h - u\|^2 \tag{3.1}$$

where u^h is our Ritz Galerkin approximation, and u is our exact known solution.

The flaw with this approach is that there are many problems for which we do not have analytical solutions, where we cannot write out the solution by hand. This makes using u impossible if we want to be able to approximate any given differential equation without prior knowledge. Our goal is to be able to solve any DE using this

method, including ones without known analytical solutions, due to this we needed another approach.

The other version of this approach is to compute the solution on different function spaces using Firedrake, specifically different order solutions. The idea behind this is if we pick a first-order linear solution and a higher-order solution on the same mesh, say second order, if we compare those two solutions treating the higher order solution as our exact solution, we should be able to improve the first-order solution and optimize that. To show how this applies to (2.9), we let

$$L_2(\Omega^h, u^h, u_*^h) = \|u^h - u_*^h\|^2 \tag{3.2}$$

where u_*^h is our higher order approximation. This approach is more costly since we are computing two solutions every time we solve the problem instead of one, however by doing this, we no longer need the exact solution. Since we are interested in building robust algorithms that do not require knowledge of the solution of the differential equation, for the remainder of this thesis we will be using (3.2) as our loss function with a piecewise quadratic approximation as u_*^h.

3.2 Neural network structure

During the process of optimizing our network, we experimented with multiple neural network structures. The first structure we tried had 3 hidden layers, all linearly connected using a rectified linear unit (ReLU) activation function. The ReLU activation function was chosen simply as it is the most common first activation function to try for most neural networks. The ReLU activation function is

$$f(x) = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x, \text{ if } x > 0, \\ 0, \text{ otherwise,} \end{cases} \tag{3.3}$$

where x is the input to the neural network. We structured the network to have three hidden layers. The first layer maps a given value of ϵ to $\frac{N}{4} - 1$ points, the -1 is because we require that 0 and 1 be the first and last points in our meshes, so this gives a mesh with $\frac{N}{4}$ intervals. The second layer then maps these $\frac{N}{4} - 1$ points to $\frac{3N}{4} - 1$ points, and the final layer maps these $\frac{3N}{4} - 1$ values to $N - 1$ points.

This structure worked, however was limited in its capabilities since the ReLU activation function made it difficult to put multiple points on our mesh extremely close together if needed. We then tried changing the activation function to a tanh activation function. The tanh activation function is as follows

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.4}$$

where x is the input to the neural network. This function transforms the problem we solve to be on the domain $[-1, 1]$ and then when we want to test the differential equation we are learning, we transform it back to $[0, 1]$ as that is the domain we are interested in. We use tanh activation, so that the output of our network is a valid mesh on $[-1, 1]$ with points $-1 = y_0 < y_1 < \cdots < y_N = 1$ which we map back to a valid mesh on $[0, 1]$ by taking

$$x_i = \frac{y_i + 1}{2}. \tag{3.5}$$

This performed better, allowing us to more easily generate points that were close to each other and ended up being the final activation function we landed on using.

3.3 Tuning Parameters of the Neural Network

Now that we have established the problem we want to solve, and the structure of our neural network, we need to focus on how do we actually get the network to perform. In order to do this, we needed to create proper training criteria such as reasonable data, loss functions, and to figure out sensible cases to use as our test problem.

It is important to understand what a successful run is, throughout this thesis many of the results will show the error in our approximate solution being very close to the Bakhvalov mesh error. As they are within the same order of accuracy, and the Bakhvalov mesh is quasi-optimal meaning it is optimal within a constant, this would be considered a success. As mentioned in Section 2.3 both the Shishkin and Bakhvalov meshes require knowing specific information about the problem before constructing the grids, giving them an advantage for specific kinds of problems. However, we are not giving our neural network any of this information and so being quasi-optimal is a very good result since we do not need to know anything about the solution to the problem we are solving.

First we will establish a baseline of results, and then show how choice of parameters may or may not help improve our approximations. Before doing so, it is important to note that in machine learning theory we expect that with an infinitely small learning rate, and an infinite amount of time we should always be able to converge to an optimal result assuming the network is reasonably constructed. This is not a viable path in the real world, so we will be trying to vary parameters and structure tests in a way that minimizes the amount of real world time it takes to converge to an optimal result, or at least an optimal enough result where we can see that the rate of change of the loss function approaches 0.

Convergence of the finite-element approximation has been talked about in Section 2.3 but, in this section, we will discuss convergence of the neural network instead. If our neural network has converged, it means the difference between the gradients of each of our components of the neural network has approached zero meaning the difference between the current epoch and the previous one is almost zero. If this occurs, the neural network can continue to perform more epochs, however the network will no longer change a significant amount anymore, concluding the learning process.

First let us look at the test problems. By default, we choose to solve our model problem (2.1) with $\epsilon = 0.01$. For all experiments we use a small enough number of mesh points so that any ϵ that we consider is still a challenging problem, since for a large number of points a uniform mesh would be sufficiently accurate. Specifically, we will only solve problems where $\epsilon N \ll 1$ since for $\epsilon N \geq 1$ a uniform mesh will be sufficient to resolve any layers. The choice of $\epsilon = 0.01$ is a small enough value of ϵ in order for the boundary layers to become prevalent, while also being big enough to not worry about numerical instability. In order to figure out how the network is doing after each epoch of training, we take this value of ϵ and ask the network to produce a mesh using this value. Then we solve (2.1) and record the value the loss function returns from this specific problem. The hope is that as we train the network further, we become better at solving our test problem while also improving our ability to solve any other problem within an interval of ϵ values.

We did try other values of epsilon as well, for example $\epsilon = 10^{-5}$. The solution for this value of ϵ is shown in Figure 3.1. The smaller the value of ϵ the steeper the solution, meaning the boundary layer occurs in a more narrow region. Due to this, we predicted it should be much harder to solve this kind of problem to a high degree

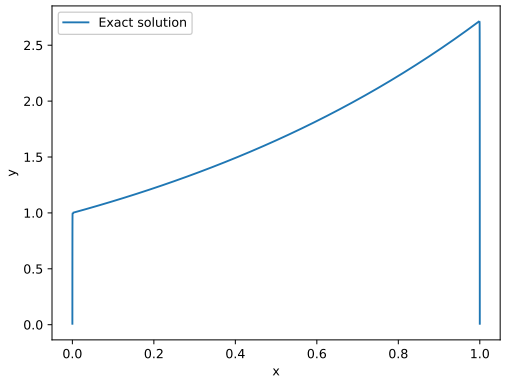

Figure 3.1: Our model problem with $\epsilon = 10^{-5}$

of accuracy with a small number of points. Assuming a wisely chosen loss function, we expected to find the smaller the value of ϵ, the longer it would take to converge to a desired level of accuracy.

For training, we pick some range of ϵ and then randomly select batches of ϵ from this range. For the results in this Chapter, we only consider test problems for values of ϵ within this range. Typically, we choose the range $[10^{-7}, 10^{-2}]$ on a logspacing, as the weak form of our model problem has one term weighted by ϵ^2 and when this is around 10^{-14}, we start to worry about numerical instability. We often choose 10 values of ϵ from this range, this served as a middle ground between enough values of ϵ and computational cost. After each epoch we perform backpropagation through the network to update the weights and biases based upon the sum of the outputs of the loss function of each batch in that epoch. It is important to note that for all experiments in this thesis, we retrain the neural network instead of reusing it.

We also did parameter studies to see what would improve our results. One question that was important to look at was how different batch sizes and numbers of epochs compare to each other. Doing more epochs with smaller batches is more computationally expensive, and so the only reason to do this would be if it improved our rate of convergence to our final answer during the training phase.

By default, we used a batch size of 10, and we tried experiments using 10000 epochs, giving us 100000 problem solves. To give us a starting point, we trained a network for $N = 16$, with a learning rate of 10^{-2}, a test ϵ of 10^{-2}, batch size of 10 and 10000 epochs. The results of this are shown in Figure 3.2 and we aim to vary the

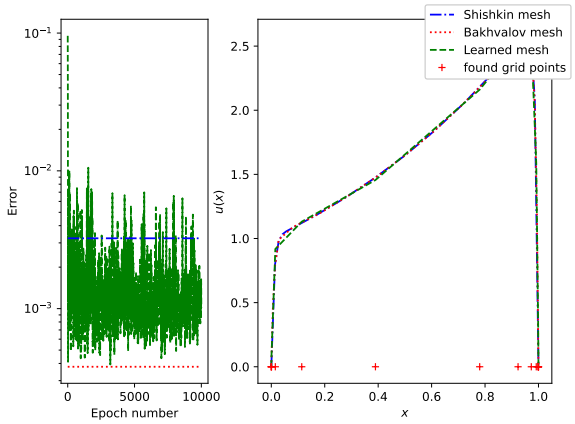

Figure 3.2: Our model problem with $f(x) = e^x$ for $N = 16$ with a learning rate of 10^{-2}, 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$.

parameters in some way that we improve the results of this starting point.

For all figures moving forward, on the left hand side we plot errors of the Shishkin mesh, the Bakhvalov mesh and a loss history plot of our network. Each epoch we ask the network to solve our model problem with a specific value of ϵ and we record the error of that test, plotted as the dashed green line in the left-hand figure. On the right side, we plot the solution approximations generated on the Shishkin and Bakhvalov meshes, along with that generated on the mesh points returned from the NN at the final epoch. The red plus signs are the mesh points our network generates when we ask it to solve the model problem with our testing value of ϵ. We can see that in Figure 3.2 our error value oscillates heavily, and it is important to try to diagnose what causes this. We know that the more points we put on a mesh, the better the initial mesh is going to be at resolving the boundary layers. Due to this the next thing to try is to check what happens if we increase the number of points to $N = 32$ and $N = 64$ but keep the rest of the parameters the same.

We can see that in Figure 3.3 that for both $N = 32$ and $N = 64$ we still have the same oscillations we found in Figure 3.2. It is also important to note that we want to optimize our neural network so that it is robust for solving problems on $N = 16$, $N = 32$ and $N = 64$. From here, we aimed to study the learning rate to see if that was able to resolve the oscillations.

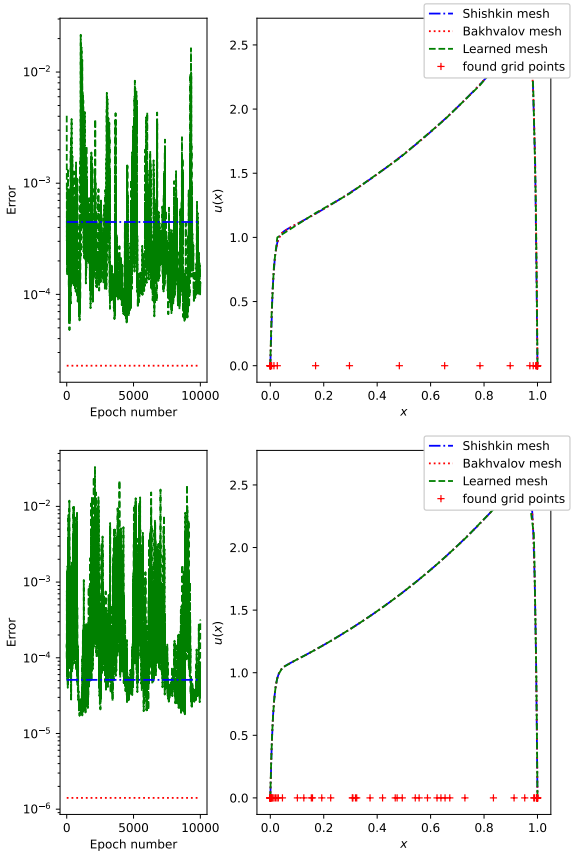

Figure 3.3: Our model problem with $f(x) = e^x$ for $N = 32$ (top) and $N = 64$ (bottom) with a learning rate of 10^{-2}, 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$.

Learning Rate

One of the challenges for any practical NN is going to be determining an appropriate learning rate. Learning rates serve as a step size for a neural network, a big learning rate encourages the network to try to make big steps towards optimal solutions, risking over shooting and missing the true optimum. A smaller learning rate takes longer to find a optimum, but also risks getting stuck in a valley like pattern, where moving a point in either direction only makes the results worse but if it were to move further it may realize it can improve its results.

Due to this, we studied different possible learning rates specifically focused on when we use a large number of points such as 64. The idea is that although a large number of points means the uniform mesh for this number of points starts off better, there is simply more variables to optimize so it should take much longer to reach the desired optimal solution.

We analyze different learning rates in Figure 3.4 to see if there is a clear best learning rate option. We can see that although the approximated solution to each of these figures looks reasonable, the learning curves indicate we are not improving our meshes for some of these learning rates. We can see that for a large learning rate, the error oscillates heavily as it is likely taking too large of steps and missing small changes that might impact our error significantly. For small learning rates, we can see that the rate in which we learn is very slow, and in order for us to generate meaningful results we would need to allocate a large amount of computational time to solve the problem. From this, we want to find a middle ground for $\epsilon = 10^{-2}$ where we get relatively fast convergence to an optimal solution, and a high level of accuracy. From Figure 3.4 we can see that our learning rate of 10^{-4} seems to perform best, but we need to make sure this is still true for $N = 16$ and $N = 32$.

In Figure 3.5 we can see that by lowering our learning rate from 10^{-2} to 10^{-4} we were able to fix the oscillations from Figure 3.2 and begin converging to errors similar to the Bakhvalov mesh. We did not heavily experiment with smaller learning rates, this is because the smaller the learning rate is the longer it takes for a network to converge and as the learning rate approaches 0, the harder it is for a network to make progression in its training.

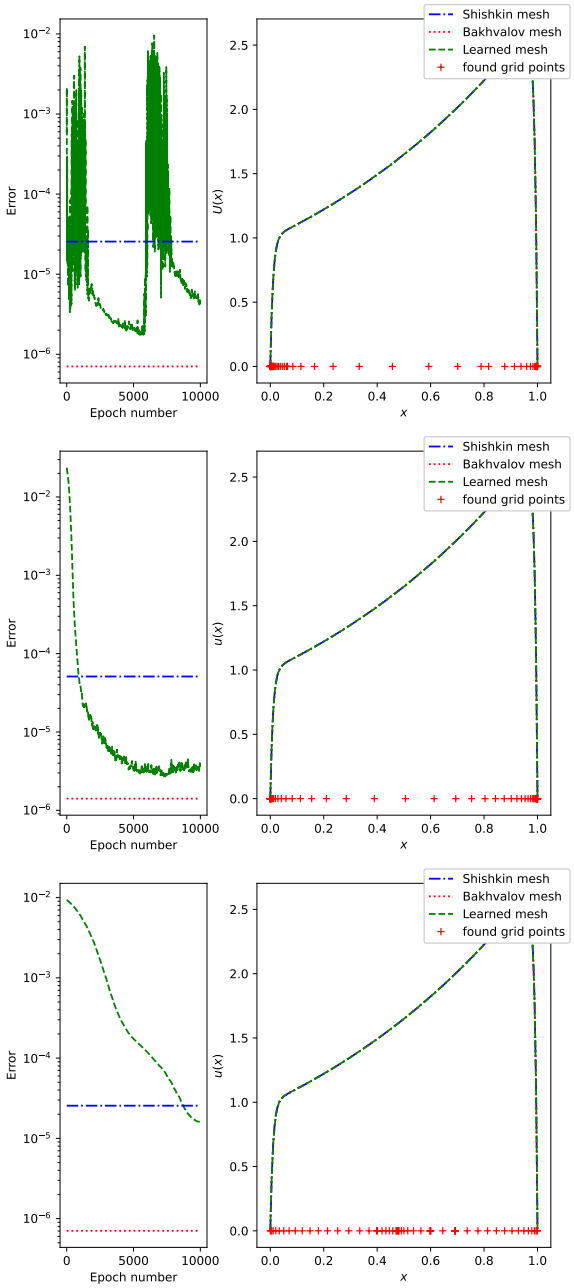

Figure 3.4: Our model problem with $f(x) = e^x$ for $N = 64$ with learning rates of 10^{-3} (top), 10^{-4} (middle) and 10^{-5} (bottom), 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$.

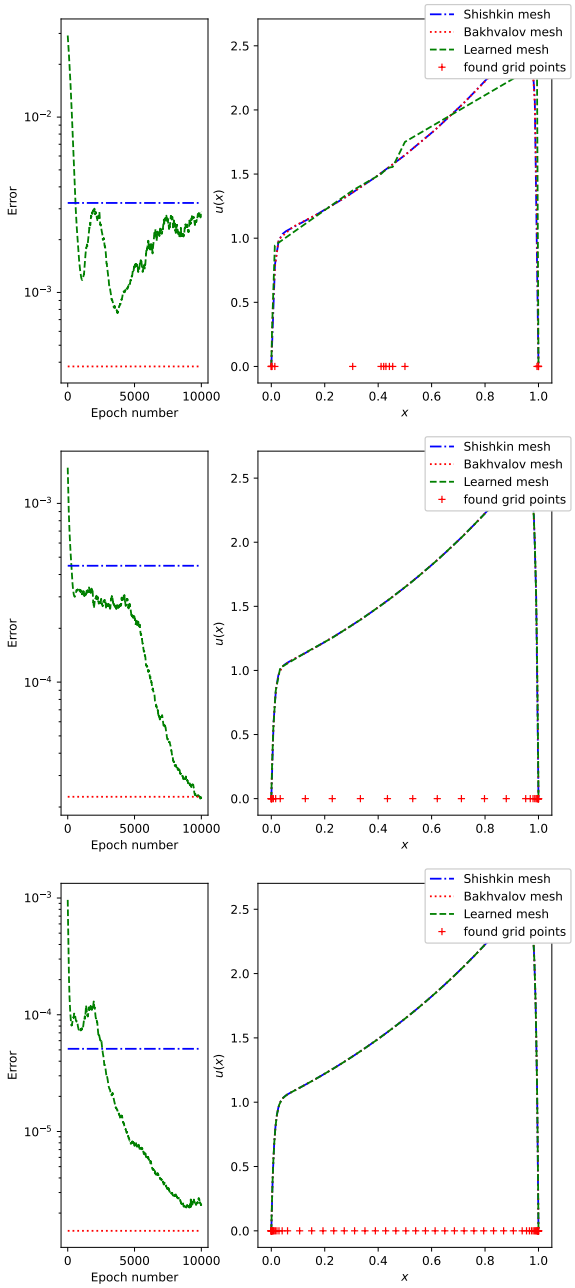

Figure 3.5: Our model problem with $f(x) = e^x$ for $N = 16$ (top), $N = 32$ (middle) and $N = 64$ (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$.

Batch sizes

Next, we tried to solve the same problem but with a batch size of 1, and 10000 epochs meaning we do the same number of backpropagation steps, but fewer problem solves. First we tried $N = 16$, $N = 32$ and $N = 64$ and a small learning rate of 10^{-4}. In Figure 3.6 we can see the learning converged very rapidly. Importantly, for $N = 16$ although our network converged, we converged to a poor quality mesh. From this, it seems that a batch size of 1 may have benefits for some cases, but may lack the robustness we are looking for in our network structure.

Next we tried the same setup, except we shrunk the learning rate even further to the point where typically the learning rate is too small with batching to solve the problem successfully. We can see in Figure 3.7 that for a learning rate of 10^{-6} we slowly were able to make progress towards convergence but the time it would have took for the algorithm to finish was large. For a learning rate of 10^{-7} we can conclude that this learning rate is simply too small to be able to solve the problem, as the rate of change of the loss function is essentially zero no matter how we change the problem.

The number of layers in the neural network

Another question to be asked is what if we increase the number of layers in the NN, does that help compensate for the larger number of variables and will it allow us to use a larger learning rate for more problems. The new neural network structure will take in the inputs from the input layer into a layer of size $\frac{N}{4} - 1$ where N is the number of mesh points. Each additional layer will then add another $\frac{N}{4}$ points to the network, meaning the final layer will contain $N - 1$ full connected nodes.

We can see in Figure 3.8 that by adding a fourth layer to our network, we have greatly increased the rate of convergence and for some of our smaller learning rate tests, we are now actually converging. This poses questions such as if we added an arbitrary number of layers should we have a very rapid rate of convergence which should be true in theory, but in practice the more layers we add the more variables there are and the algorithm becomes more expensive.

To investigate further, our best results seem to come from a learning rate between 10^{-4} and 10^{-5} so we can look at some plots for learning rates in between those values

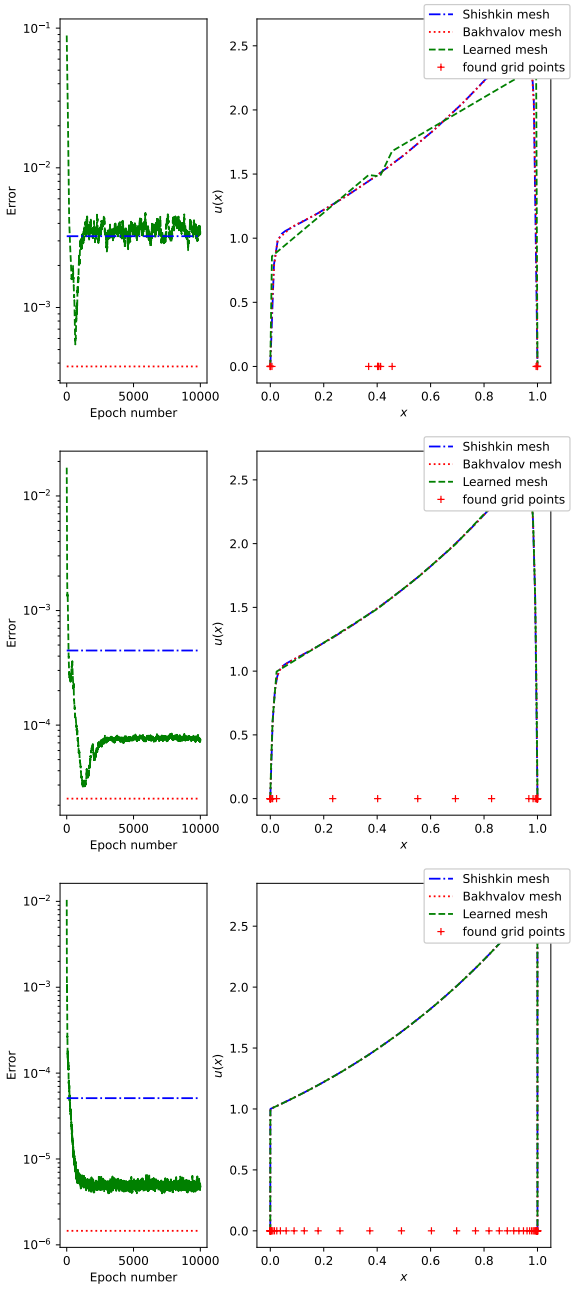

Figure 3.6: Our model problem with $f(x) = e^x$ for $N = 16$ (top), $N = 32$ (middle) and $N = 64$ (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 1, tested with $\epsilon = 10^{-2}$.

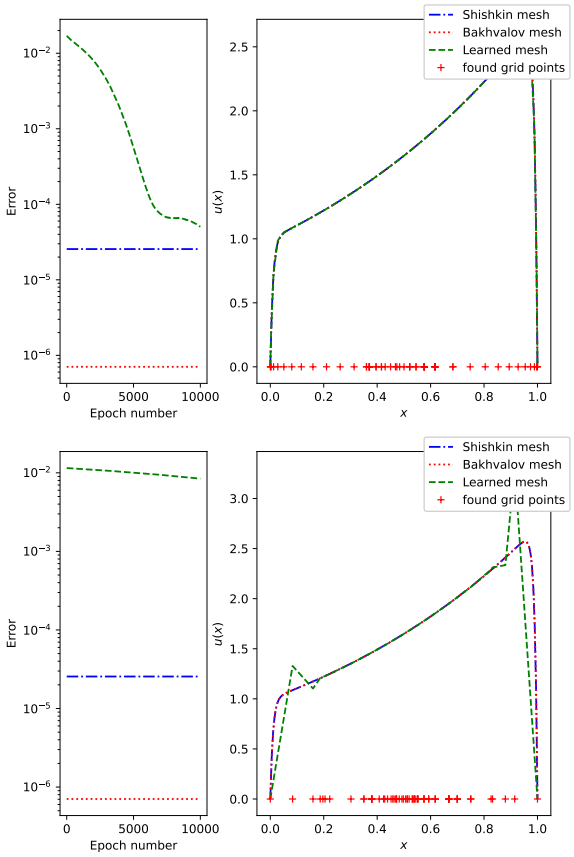

Figure 3.7: Our model problem with $f(x) = e^x$ for $N = 64$ with learning rates of 10^{-6} (top), 10^{-7} (bottom), 10000 epochs, a batch size of 1, tested with $\epsilon = 0.01$.

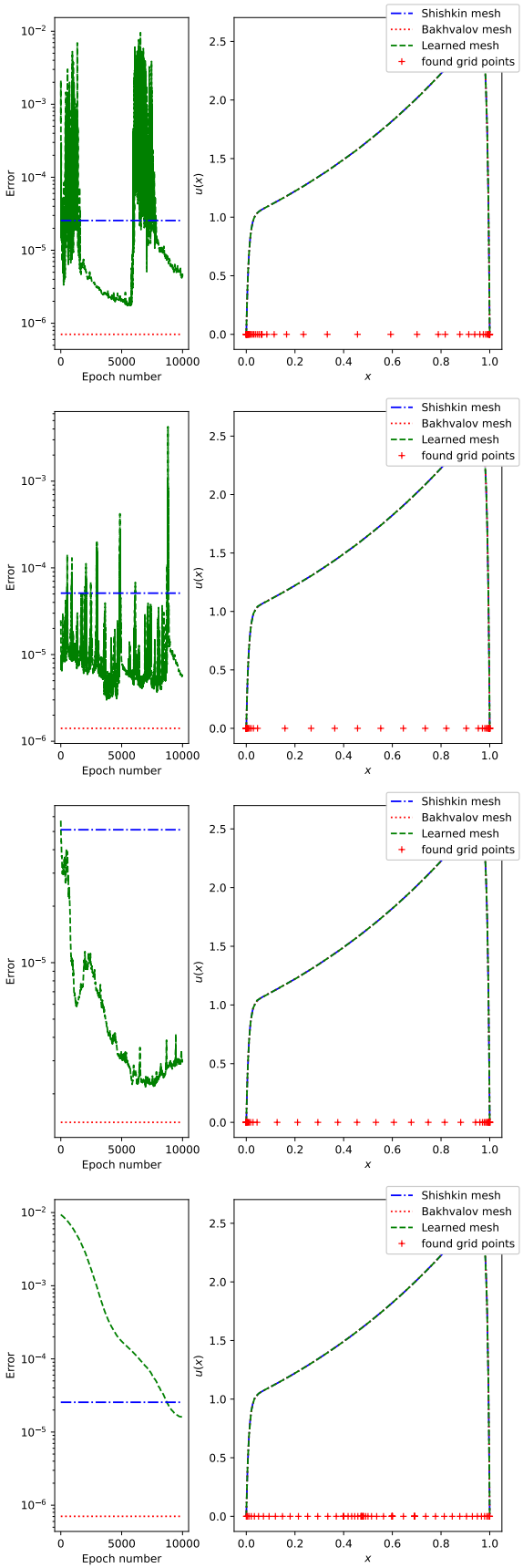

Figure 3.8: Our model problem with $f(x) = e^x$ for $N = 64$ with learning rates of 10^{-2} (first), 10^{-3} (second), 10^{-4}, (third) and 10^{-5} (final) with 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$ with 4 layers.

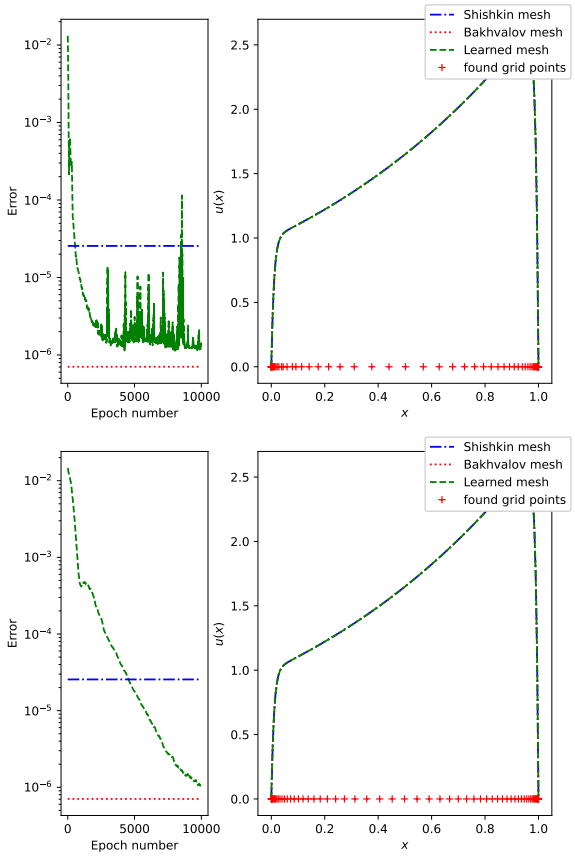

Figure 3.9: Our model problem with $f(x) = e^x$ for $N = 64$ with learning rates of 5×10^{-5} (top), 5×10^{-6} (bottom), 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$ with 4 layers.

with 4 layers.

We can see in Figure 3.9 that with a smaller learning rate, we are able to converge to a quasi-optimal solution with a four layer neural network. This indicates that we may be able to shrink our learning rate further if we increase the number of layers. For the purposes of this thesis, we did not experiment with any network structures with more than four layers, nor did we experiment on the size of these layers in a significant way.

For all the results presented in Chapter 4 we utilize the results of this study to construct our network. All experiments were completed using a learning rate of 10^{-4}, a batch size of 10 and 4 layers. We do not claim that these are the perfect values for each of these parameters, as there are too many permutations to test, however we believe this set of parameters provides a robust enough network structure for

answering the questions posed in Chapter 4.

Chapter 4

Numerical results

Now that our foundation has been laid for this project, we can start to examine if it was actually successful and if we were able to improve our ability to generate meshes that have a similar error of those of the Shishkin and Bakhvalov meshes while using less information. In Chapter 3 parameter studies were performed to help figure out an optimal neural network structure based upon our model problem. For this chapter, we will use the structure found in Chapter 3 and then ask new questions such as how our network performs when we consider different values for ϵ or different forcing functions, $f(x)$.

Testing over values of ϵ

Starting with a small learning rate of 10^{-4} we can produce our first results for $N = 16, 32, 64$. This should be a reasonable baseline for us to work with since $\epsilon = 10^{-2}$ is a relatively large value meaning we will not have to worry about round off error, and this learning rate should allow us to quickly see results without needing a ton of epochs.

We can see that in Figure 4.1 for $N = 16, 32, 64$ with 10000 epochs we are able to consistently outperform the Shishkin mesh and be within a small factor of the results of the Bakhvalov mesh. During the training process for $N = 16$ and $N = 32$ we are able to temporarily outperform the Bakhvalov mesh, however the network did eventually converge a bit away from this point which is not abnormal. This is because the network needs to be able to solve any ϵ within its training range, so by moving

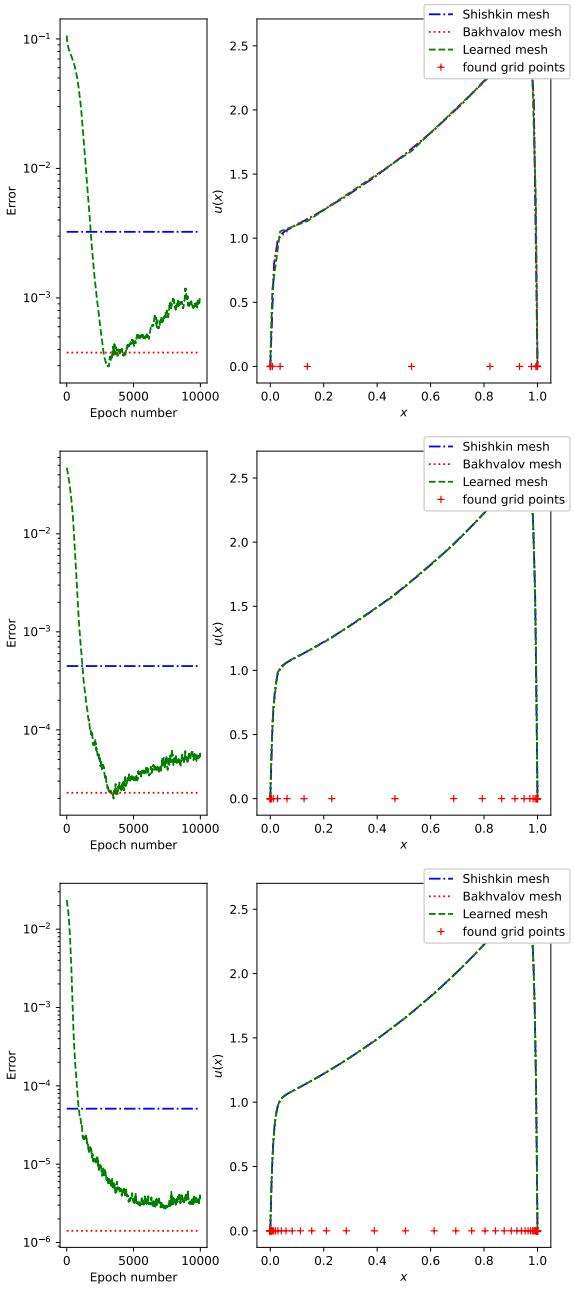

Figure 4.1: Our model problem with $f(x) = e^x$ for $N = 16$ (top), $N = 32$ (middle) and $N = 64$ (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$.

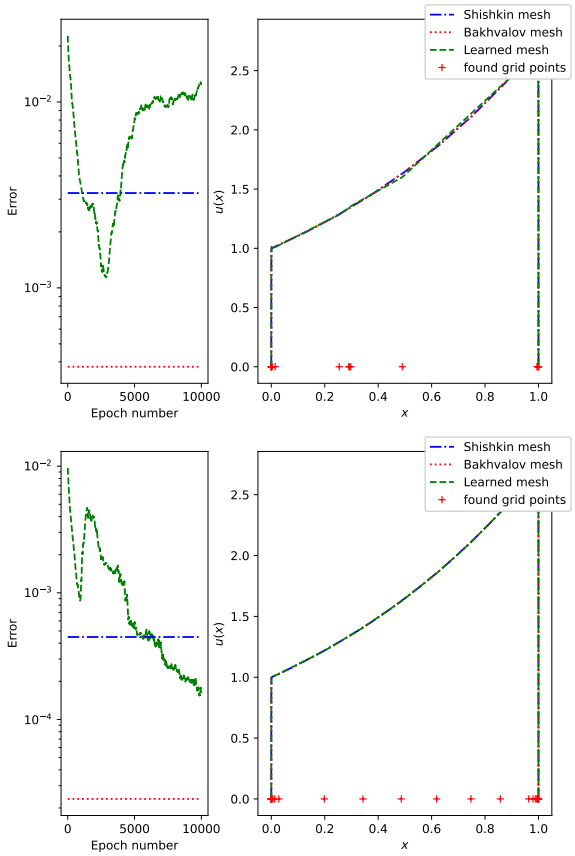

Figure 4.2: Our model problem with $f(x) = e^x$ for $N = 16$ (top), $N = 32$ (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-4}$.

away from outperforming Bakhvalov it likely was able to perform better on a wider range of ϵ instead of overtraining on our test value of ϵ.

We can also see what happens if we try a smaller ϵ. Figure 4.2 shows results for $N = 16$ and $N = 32$ with $\epsilon = 10^{-4}$, showing that although the loss values do decrease, we do not reach a true convergence after 10000 epochs. We still can outperform the Shishkin mesh for $N = 32$, but we can see that for a smaller value of ϵ the solution becomes very steep and with a small number of points, we do not perform as well as we do in tests such as Figure 4.1. We note that most experiments comparing the number of mesh points use $N = 16, 32$ and 64 however we observed inconsistent results with $N = 64$ and, so, omit them from the Figure 4.2.

To help show that the smaller ϵ is, the more difficult it is for us to learn the problem we performed an experiment looking at many values of ϵ. In Figure 4.3 we

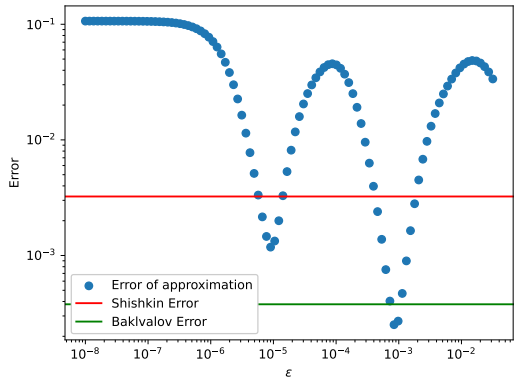

Figure 4.3: A scatter plot of the error between the approximated solution obtained on a mesh provided by our neural network and a higher order approximation for our model problem with $f(x) = e^x$. The neural network is trained on values of ϵ in the interval $[10^{-6}, 10^{-2}]$ with a batch size of 10, trained with $N = 16$ and learning rate of 10^{-4}.

trained our neural network with 1000 epochs and a learning rate of 10^{-4} but once the testing process concluded, we asked it to solve our model problem with 100 different values of ϵ on a logspacing. We use a training range of $[10^{-6}, 10^{-2}]$ and sample 100 values of ϵ from this range with logspacing. We can see in the scatter plot that for values of ϵ within our training range the smaller the value of ϵ the harder it is to train our network to solve the problem. For values of ϵ outside our training range, our network does perform poorly which is not unexpected.

In Figure 4.3 we observed that for values of ϵ outside our training range, we performed poorly. To further explore this, we train a network on test problems with ϵ in the range $[10^{-6}, 10^{-3}]$, and test the network with $\epsilon = 10^{-2}$. We trained the network with $N = 16, 32$ and 64 and the results can be found in Figure 4.4. We can see that for all values of N, the network performs extremely poorly and does not learn the behavior of the model problem. This is not unexpected as $\epsilon = 10^{-2}$ has less steep boundary layers than those the network was trained upon. Since $\epsilon = 10^{-2}$ is a relatively easy problem to solve due to the easier to resolve boundary layers, we can conclude that our network structure does not generalize for values outside our training range. This means if we want to be able to test as many possible values of ϵ as possible we need to train our network on as wide of a range of ϵ as possible.

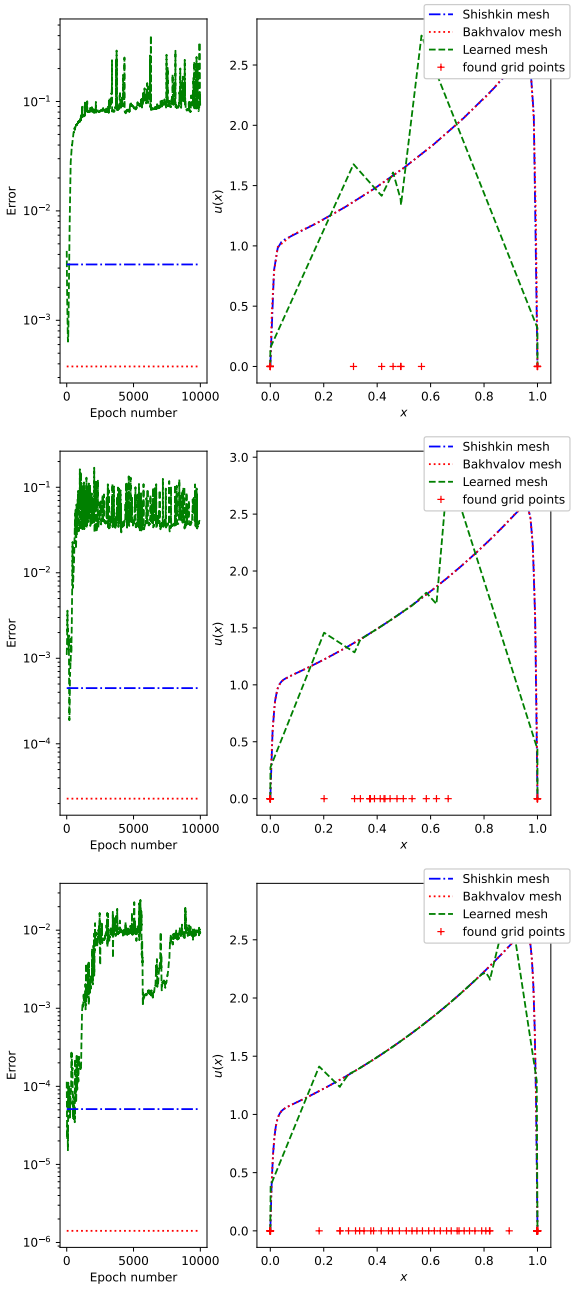

Figure 4.4: Our model problem with $f(x) = e^x$ for $N = 16$ (top), 32 (middle), 64 (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 10, tested with $\epsilon = 10^{-2}$ and trained on a range of $[10^{-6}, 10^{-3}]$.

The impact of the right hand side of our model problem

Lastly, an important question to answer is whether we can solve other problems besides our model problem (2.1). To attempt this, we will solve a series of similar problems, starting with $f(x) = \cos(x)$. We choose this problem as it contains two boundary layers like our usual $f(x) = e^x$, and has a similar final solution structure. Due to this, we should be able to solve this problem effectively. The true solution to this problem is

$$u(x) = \frac{-1 + e^{\frac{-1}{\epsilon}}\cos(1)}{(\epsilon^2+1)(1-e^{\frac{-2}{\epsilon}})}e^{\frac{-x}{\epsilon}} + \frac{-\cos(1) + e^{\frac{-1}{\epsilon}}}{(\epsilon^2+1)(1-e^{\frac{-2}{\epsilon}})}e^{\frac{x-1}{\epsilon}} + \frac{\cos(x)}{\epsilon^2+1}. \tag{4.1}$$

In Figure 4.5 we can see that for all three values of N, we are able to create a strong approximation to the solution of this problem with $f(x) = \cos(x)$. For $N = 16$ it does not look like we perform particularly well, but this is because the Shishkin mesh does a good job resolving the layers and so the difference between it and the Bakhvalov mesh is smaller than in previous problems. However, for $N = 32$ and especially $N = 64$ our network is able to perform very well for this problem. This indicates we should be able to solve other problems with two layers, but what happens for problems with only one?

A problem to solve with one boundary layer is our model problem with $f(x) = \sin(x)$ which has a true solution of

$$u(x) = \frac{-\sin(1)}{(\epsilon^2+1)(1-e^{\frac{-2}{\epsilon}})}e^{\frac{x-1}{\epsilon}} + \frac{e^{\frac{-1}{\epsilon}}\sin(1)}{(\epsilon^2+1)(1-e^{\frac{-2}{\epsilon}})}e^{\frac{-x}{\epsilon}} + \frac{\sin(x)}{\epsilon^2+1}. \tag{4.2}$$

This problem has a similar structure to our model problem except it only has a boundary layer near $x = 1$. We tried this problem for $N = 16, 32$ and 64.

We can see that in Figure 4.6, we learn the behavior of the solution very well including detecting the sharpness of the boundary layer. It seems that for all three of our possible values of N we are able to converge to an error better than the Shishkin mesh, and perform similarly to the Bakhvalov mesh.

The final similar problem we want to look at is one with no layers, for which we consider our model problem with $f(x) = \sin(\pi x)$. This problem should be easy to approximate but the usual comparsion tools of the Shishkin and Bakhvalov mesh are

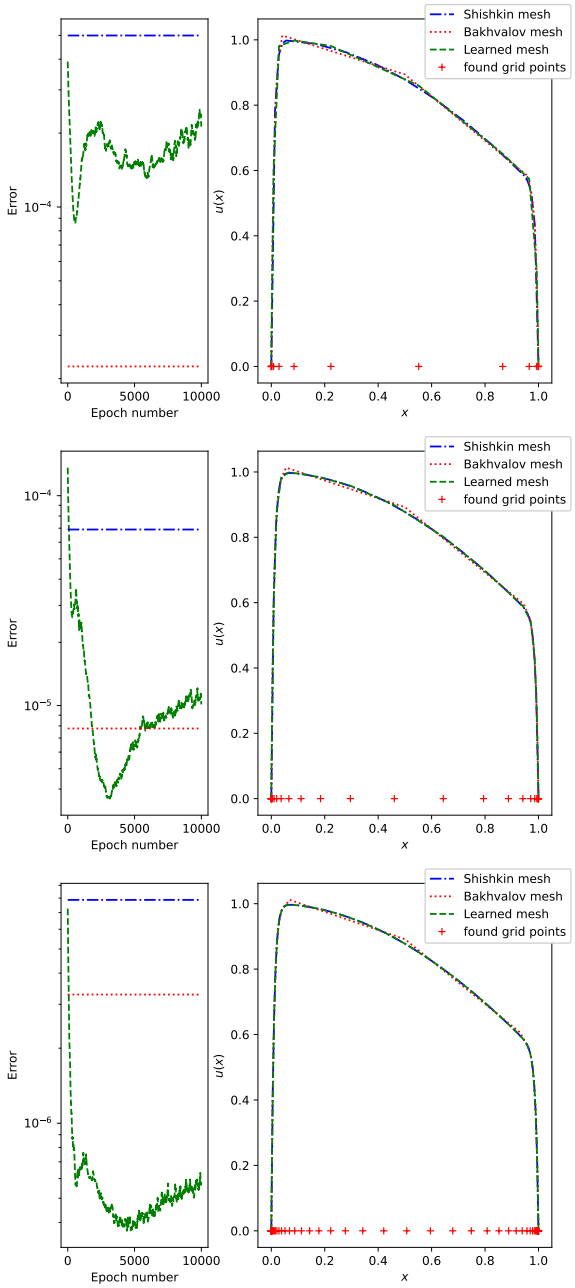

Figure 4.5: Our model problem with $f(x) = \cos(x)$ for $N = 16$ (top), $N = 32$ (middle) and $N = 64$ (bottom). Calculated with a learning rate of 10^{-4}, batch size of 10 and tested with $\epsilon = 10^{-2}$.

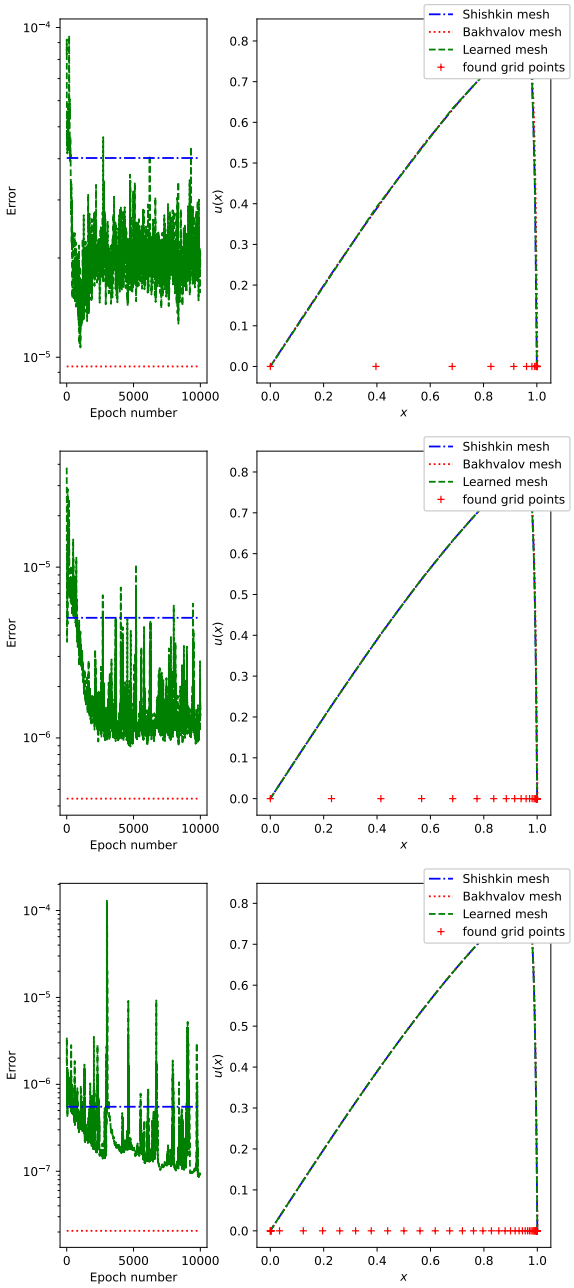

Figure 4.6: Our model problem with $f(x) = \sin(x)$ for $N = 16$ (top), 32 (middle), 64 (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 10 and tested with $\epsilon = 10^{-2}$.

not a good reference point. This is because those meshes are designed for problems with layers so instead, we would expect the uniform mesh to be a reasonable choice for this problem and so we will compare our approximation to that. The exact solution to this problem is

$$u(x) = \frac{\sin(\pi x)}{\pi^2 \epsilon^2 + 1}. \tag{4.3}$$

We can see in Figure 4.7 that for a problem with no boundary layers, we are able to learn how to approximate the solution to the problem very rapidly. This is because the initial mesh the network works with is the uniform mesh, and for this problem that is much closer to optimal than problems such as $f(x) = \cos(x)$. There are still better meshes than the uniform mesh for this problem; as we can see our network does learn how to create a slightly more optimal approach.

Using the past three experiments, it indicates that our network is fairly robust at optimizing mesh points to approximate solutions to differential equations regardless of the number of boundary layers. It is important to note that all experiments were still done using (2.1) meaning we do not know yet if our network could handle a different differential equation.

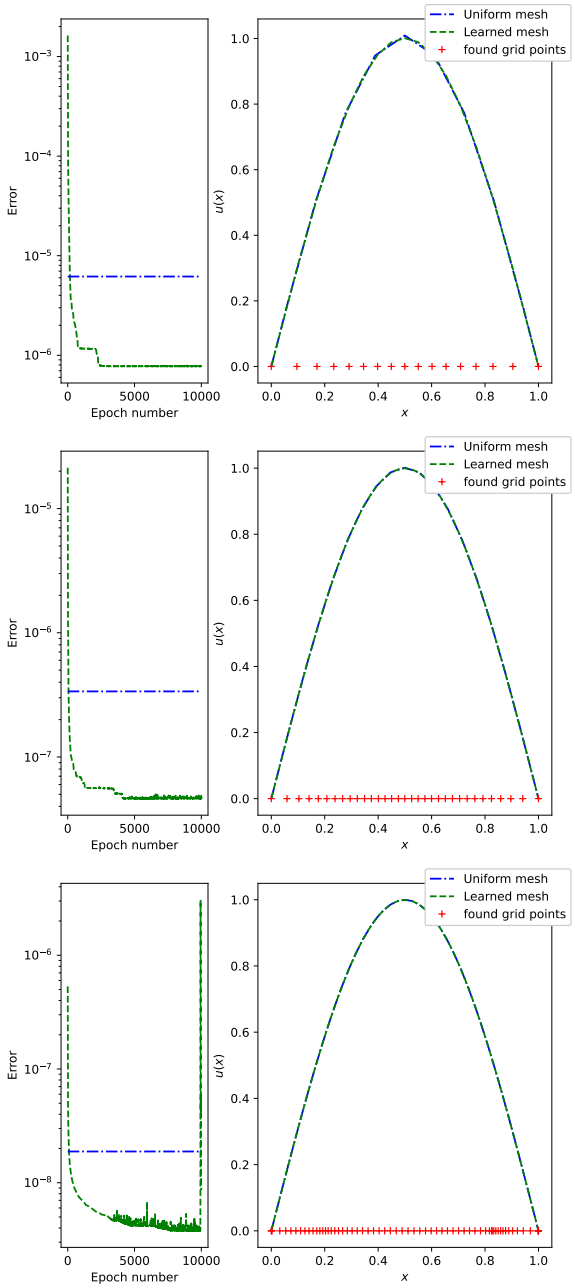

Figure 4.7: Our model problem with $f(x) = \sin(\pi x)$ for $N = 16$ (top), 32 (middle), 64 (bottom) with a learning rate of 10^{-4}, 10000 epochs, a batch size of 10 and tested with $\epsilon = 10^{-2}$.

Chapter 5

Conclusion and future work

In this thesis, we focused on utilizing neural networks to optimize the mesh points we use to approximate differential equations with a finite element discretization. We specifically focused on problems with at least one boundary layer as those posed a more difficult problem to generate meshes for.

We were able to successfully approximate multiple solutions to differential equations with either one or two boundary layers using the grid points found using our neural networks. We were able to consistently reach a quasi-optimal error for our final testing problem without having to provide our network information about what the ideal mesh should look like, unlike the Shishkin and Bakhvalov meshes we used as our reference tool.

This opens up the possibility of potentially moving away from building meshes with knowledge of the solution and instead, focusing on optimizing how we train a general neural network to help optimize the mesh points for a specific differential equation. There are many questions not answered in this thesis that are worth exploring, such as what happens for problems with an interior layer in the middle of the interval or how does this scale when we move beyond one dimension. Importantly, we would need to be able to solve these other more difficult problems with a similar level of accuracy as we did in this thesis for this approach to be viable in general but nothing so far has indicated we could not reach quasi-optimal meshes for other problems assuming we know how to best build our neural networks.

Bibliography

[1] J. Adler, H. De Sterck, S. MacLachlan, and L. Olson. *Numerical Partial Differential Equations*. SIAM, Philadelphia, 2024. To appear.

[2] J. H. Adler, T. R. Benson, E. C. Cyr, S. P. MacLachlan, and R. S. Tuminaro. Monolithic multigrid methods for two-dimensional resistive magnetohydrodynamics. *SIAM Journal on Scientific Computing*, 38(1):B1–B24, 2016.

[3] A. Arzani, K. W. Cassel, and R. M. D'Souza. Theory-guided physics-informed neural networks for boundary layer problems with singular perturbation. *Journal of Computational Physics*, 473:111768, 2023.

[4] M. M. Bejani and M. Ghatee. A systematic review on overfitting control in shallow and deep neural networks. *Artificial Intelligence Review*, 54(8):6391–6438, 2021.

[5] N. Bouziani and D. A. Ham. Physics-driven machine learning models coupling PyTorch and Firedrake. *arXiv preprint arXiv:2303.06871*, 2023.

[6] J. Brownlee. What is the difference between a batch and an epoch in a neural network. *Machine Learning Mastery*, 20:1–5, 2018.

[7] S. Cai, Z. Mao, Z. Wang, M. Yin, and G. E. Karniadakis. Physics-informed neural networks (PINNs) for fluid mechanics: a review. *Acta Mech. Sin.*, 37(12):1727–1738, 2021.

[8] Z. Cai, A. Doktorova, R. D. Falgout, and C. Herrera. Fast iterative solver for neural network method: I. 1d diffusion problems. *arXiv preprint arXiv:2404.17750*, 2024.

[9] Z. Cai, A. Doktorova, R. D. Falgout, and C. Herrera. Fast iterative solver for neural network method: II. 1d diffusion-reaction problems and data fitting. *arXiv preprint arXiv:2407.01496*, 2024.

[10] S. Cuomo, V. S. Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

[11] S. Cuomo, V. Schiano Di Cola, F. Giampaolo, G. Rozza, M. Raissi, and F. Piccialli. Scientific machine learning through physics-informed neural networks: where we are and what's next. *J. Sci. Comput.*, 92(3):Paper No. 88, 62, 2022.

[12] A. Devarakonda, M. Naumov, and M. Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[13] D. A. Ham, P. H. J. Kelly, L. Mitchell, C. J. Cotter, R. C. Kirby, K. Sagiyama, N. Bouziani, S. Vorderwuelbecke, T. J. Gregory, J. Betteridge, D. R. Shapero, R. W. Nixon-Hill, C. J. Ward, P. E. Farrell, P. D. Brubeck, I. Marsden, T. H. Gibson, M. Homolya, T. Sun, A. T. T. McRae, F. Luporini, A. Gregory, M. Lange, S. W. Funke, F. Rathgeber, G.-T. Bercea, and G. R. Markall. *Firedrake User Manual*. Imperial College London and University of Oxford and Baylor University and University of Washington, First edition, 5 2023.

[14] A. Koschan and M. Abidi. *Digital Color Image Processing*. John Wiley & Sons, New York, 2008.

[15] Y. Li, C. Wei, and T. Ma. Towards explaining the regularization effect of initial large learning rate in training neural networks. *Advances in neural information processing systems*, 32, 2019.

[16] T. Linß. *Layer-Adapted Meshes for Reaction-Convection-Diffusion Problems*. Springer, Heidelberg Dordrecht London New York, 2009.

[17] A. J. Meade Jr and A. A. Fernandez. Solution of nonlinear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling*, 20(9):19–44, 1994.

[18] J. A. A. Opschoor, C. Schwab, and C. Xenophontos. Neural networks for singular perturbations. *arXiv preprint arXiv:2401.06656*, 2024.

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[20] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.*, 378:686–707, 2019.

[21] R. Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[22] S. Sharma, S. Sharma, and A. Athaiya. Activation functions in neural networks. *International Journal of Engineering and Applied Sciences and Technology*, 6(12):310–316, 2017.

[23] F. Szabo. *The Linear Algebra Survival Guide: Illustrated with Mathematica*. Academic Press, 2015.

[24] H. Zhao, O. Gallo, I. Frosio, and J. Kautz. Loss functions for neural networks for image processing. *arXiv preprint arXiv:1511.08861*, 2015.