THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF MATHEMATICS

Creating a Solver for Nonlinear Differential Equations Using Unsupervised Neural Networks

Jacob Frishman
Spring 2024

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Mathematics and Physics
with honors in Mathematics

Reviewed and approved* by the following:

Leonid Berlyand
Professor of Mathematics
Thesis Supervisor

Sergei Tabachnikov
Professor of Mathematics
Honors Adviser

*Electronic approvals are on file.

Abstract

Differential equations are ubiquitous in science and engineering for describing the natural world and often appear as nonlinear differential equations. Unfortunately, there is no general method for solving all types of nonlinear differential equations. This work uses a machine learning process called deep neural networks (DNNs) to create a solver for the Ginzburg-Landau equation regardless of the boundary conditions or the right-hand side. This method overcomes challenges to previous methods that require recomputing the solution again for every change in the boundary conditions and right-hand side of the equation. The method develops a versatile solver capable of finding a solution using only the form of the differential equation without a predefined right-hand side or boundary conditions. Systematically varying the architecture of the network, the characteristics of the input data, the loss function optimized over, and the network's hyperparameters reveal that the method can find a general solution across a diverse range of boundary conditions and right-hand sides. The network can consistently find accurate approximations of slowly oscillating data and highly oscillating data built from many terms of the Fourier series. The model can generalize performance from training data to test data, indicating its success in creating a general inverse differential operator that solves the equation. For data with many oscillations and small magnitudes, the network suffers from the vanishing gradient problem. These challenges are addressed by implementing strategies such as batch normalization, varying initialization schemes, changing activation functions, modifying the network architecture, and altering the loss function. These changes help mitigate the problem, leading to more stable and robust solutions to the initial hyperparameters of the model. However, the vanishing gradient problem persists despite these changes. Developing a solver that works for nonlinear equations would be pivotal in developing a theory for solving differential equations, saving computational time and resources, and facilitating real-time applications of the network without retraining.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to extend my deepest gratitude to Alexander 'Sasha' Gavrikov and Professor Berlyand. I would never have finished this project much less have participated in research without their guidance and support. They encouraged me to start the research process, coached me through the process, wrote letters of recommendation, and consistently comprised to make time for me. I treasure the time I spent working with you during Penn State Summer 2022 and I appreciate you for making that possible and for future summers doing research. I would also like to thank Professor Berlyand's research group and all of its members for being so welcoming, willing to answer my questions and showing me the ropes. Especially Spencer Dang for tirelessly explaining complicated topics in simple ways. The strategies of drawing things out and thought process is something I still copy to this day. I would like to thank Nadia Khoury for her help in developing the appendix.

I also want to thank my friends and family for their unconditional support throughout my academic journey. Specifically, I want to acknowledge my parents, Suzanne Starbuck and Eric Frishman, for working so hard to provide for me my entire life. Your patience, love, and advice have helped me get where I am today.

Chapter 1

Project Overview

1.1 Introduction

Machine learning has undergone unprecedented growth with advances in computational resources and accessible data. It has been integrated into our daily lives, from filtering our social media connections to providing curated product recommendations. This same class of machine learning techniques called deep neural networks (DNNs) has been instrumental in solving challenging problems such as speech recognition [8], fraud detection [2], and image recognition [11]. These achievements occurred in "big data" environments where the abundance of information allows the use of readily available data for a flexible data-driven approach. Unfortunately, many science and engineering domains do not have the luxury of access to extensive datasets available, and acquiring these datasets is often prohibitively expensive.

Differential equations are the cornerstone of describing these physical problems in physics, biology, and engineering. Traditional numerical methods like Finite Element Method (FEM) and Finite Difference Method (FDM) conventionally were implemented to solve partial differential equations in these low data regimes because machine learning techniques are prone to failing with less available data [15]. FDM can achieve high accuracy and stability with a comparatively simple but flexible approach. FDM has the drawback that it needs help implementing the boundary conditions and mesh in difficult situations. FEM also represents a highly accurate method that is adaptable to unique features and domains at the cost of requiring significant memory and processing power and is often challenging to implement without manufactured software.

The appendix includes a more comprehensive examination of these traditional methods and a numerical comparison with neural network methods for solving a simple differential equation.

The field of scientific machine learning has emerged to bridge the gap in applying machine learning to scientific problems in sparse data scenarios. By framing the situation as an optimization problem, the network is able to alter weight and bias parameters to minimize a loss function based on the form of the differential equation and boundary conditions. The convergence of the solution amounts to finding an approximate solution to the differential equation. DNNs have already successfully solved intricate real-world differential equations, such as the incompressible Navier–Stokes equations used to model weather phenomena [10]. The advantages of DNNs for scientific machine learning are as follows.

1.2 Benefits of Neural Networks

Using DNN offers several key advantages for solving nonlinear differential equations:

1. **Flexibility in Training**: Neural networks offer users considerable control over the training process through many tunable hyperparameters, including network architecture, batch size, learning rate, and more. This flexibility allows the user to tailor their hyperparameters to accommodate the specific demands of the differential equation, available computational resources, and time constraints. Unlike traditional methods that cannot be stopped until the solving scheme is completed, neural networks iterative training process allows the network to end training once the technique has reached a desired accuracy level. This iterative training process also allows for further refinement if accuracy is still suboptimal, which is not available to some traditional numerical methods.

2. **Scalability to higher dimensions**: Introduced by American mathematician Richard E. Bellman in 1961, the term "Curse of Dimensionality" signifies the exponential increase in complexity as dimensions are added to Euclidean space. The implication is that traditional computational methods require more data to achieve the same level of granularity as the density of the data. Backpropagation allows for efficient computation that extends to higher dimensions. Neural networks can excel in high-dimensional spaces and robustly handle data sparsity issues to solve high-dimensional partial differential equations [13]. This dramatically expands the types and domains of problems we can solve and the choices we can make when defining the problem. No triangulation is strictly necessary so that the method can be extended to higher domains more easily, unlike FEM and FDM.

3. **High Approximation Power**: Neural networks can approximate differential equation solutions even in higher dimensions. The generalizability of the network comes from the physical laws directly incorporated into the loss function during training [9]. This allows for robust solutions even in the presence of noisy data. Furthermore, the loss function can compensate for noisy data by adding regularization terms.

4. **Quick Solution Implementation**: After training, applying the network approximation given initial conditions and data only requires performing matrix multiplication and often simple activation functions. Matrix multiplication has become incredibly fast with today's computing power and can be performed in seconds, even for large networks. On the contrary, computing a solution for traditional numerical methods such as FEM and FDM could take a long time, like days or longer.

5. **Ease of Implementation**: The wide adoption of neural networks has increased the amount of educational resources and user-friendly coding libraries such as PyTorch and TensorFlow. These resources allow for building neural networks relatively quickly with low coding knowledge requirements, especially in comparison to FEM.

Neural network methods have significant drawbacks as well. The field of scientific machine learning with neural networks is relatively new and needs further exploration. Empirically and theoretically, the exact relationship between hyperparameters and model performance has yet to be fully understood. Currently, there is no guarantee that the network will converge or converge optimally for changing parameters like increasing network size. The large number of network parameters can make it challenging to find the optimal solutions without this rigorous theory, especially when hyperparameters are interrelated. Additionally, the inherent randomness from initialization makes the network performance inconsistent between runs.

1.3 Preliminaries

We will focus primarily on a class of differential equations called boundary value problems (BVPs). These problems dictate a unique solution based on the conditions they specify at the domain's boundary. The boundary conditions are crucial for a well-posed problem and often ground the problem to physical conditions.

$$\begin{aligned} Au(x) &= f(x) \quad \text{for } x \in (0,1), \\ u(0) &= a, \\ u(1) &= b. \end{aligned} \tag{1.1}$$

Equation 1.1 is a canonical representation of a BVP in operator notation. The function $u(x)$ is the solution to the differential equation, and $f(x)$ is the right-hand side (RHS) of 1.1, where $u(x)$ and $f(x)$ belong to the appropriate functional space. The scalars $a, b \in \mathbb{R}$ define the boundary conditions (BCs), which we take here as Dirichlet BC. The inputs to this system are $f(x)$, a, and b where we are trying to find $u(x)$. In this context, A symbolizes a (potentially nonlinear) differential operator, mapping functions to functions and encapsulating one or several operations applied to a function. For example, A could represent the second-order derivative $A = -\frac{d^2}{dx^2}$.

Additionally, we can define an inverse operator:

$$u = A^{-1}f = \int_0^x f(t)dt = F(x) - F(0) = F(x) \tag{1.2}$$

where in Equation 1.2 we take $F(x) \in C^\infty$ under the condition that $F(x) = 0$. Therefore, in this case, the inverse operator A^{-1} now maps the function $f(x)$ to the solution $u(x)$. A^{-1} is commonly known as the integral operator or inverse differential operator on the function $f(x)$ in the context of differential equations.

1.4 Goal

Our goal going forward is to find this inverse operator A^{-1} so that we can solve the differential equation for a given input function. We are under the assumption that A^{-1} exists and can be found in the future. Truthfully, the existence of A^{-1} is not guaranteed and depends on the domain and functional class of f. To even define the operator and the inverse operator, we need to define the domain. An example in which an inverse operator would not exist is if we had taken $A(u) = \frac{d^2u}{dx^2}$ for $u \in C^\infty$. Since there are constant functions in C^∞ that will each get mapped to 0, then this implies that A is not invective and therefore does not have an inverse operator.

For past classical and neural network methods, finding a numerical approximation of the inverse operator depends on the differential equation's BCs and the RHS f. To find this approximation means performing the computationally expensive calculation for a specific BC and RHS. If another situation with a different RHS or BC arose, then the inverse differential operator for that problem would need to be recomputed, costing time and resources. Therefore, the goal is to create a more general inverse operator approximation that can accurately solve the differential equation for any RHS and BC. This is achieved by instead of training over a single differential equation $Au = f$ with BCs $u(0) = a$ and $u(1) = b$ where f, a, and b are fixed; instead we train over a large number of differential equations by varying f, a, and b in the training set of an unsupervised DNN. In practice, $10,000$ randomly generated right-hand sides and boundary conditions f, a, and b are trained over simultaneously.

The proposed solver has four advantages over traditional numerical methods like FDM, neural network methods like physics-informed neural networks (PINNs), and other conventional solution methods.

1. **Versatility**: The method would accommodate a wider range of differential equations that can be written in the form $Au = f$ in operator notation. The output of this method would be an approximate inverse differential operator that could solve for any RHS and BC of the same form without requiring retraining for each unique BC and RHS.

2. **Adaptability**: The ability to handle quickly changing conditions allows this method to potentially be implemented in real-time simulations for dynamic systems like weather forecasting. This could be a major breakthrough as the state-of-the-art neural network methods, PINNs, are not suited for real-time application [13].

3. **Cost-efficiency**: Since this method does not need to be retrained for varying conditions, it is less computationally expensive and saves more on computational resource consumption than methods that have to repeat the solving scheme for every condition change. Therefore, the method saves time from training and would lead to faster development times for specialized solvers.

4. **Research potential**: Developing a generalized solver that can create approximations for previously intractable or somewhat tricky problems through a general approach would allow for more research in understanding general problem-solving approaches and new physics gained from solving hard-to-calculate problems.

In this study, we test building a solver for general differential equations by building a solver specifically for the Ginzburg-Landau equation:

$$\begin{aligned} \frac{d^2u(x)}{dx^2} + u(x) + u^3(x) = f(x) \quad \text{for } x \in (0,1), \\ u(0) = a, \\ u(1) = b. \end{aligned} \tag{1.3}$$

subject to Dirichlet boundary conditions $u(0) = a \in \mathbb{R}$ and $u(1) = b \in \mathbb{R}$ on the interval $[0, 1]$. Since A is a second-order differential operator, then the functional class of the solution and the RHS are $u(x) \in C^2([0, 1])$ and $f(x) \in C^0([0, 1])$ The equation models the behavior of type-I superconductors, where $u(x)$ represents the magnetic vector potential of a superconductor [1]. This equation is useful for liquid crystal theory and understanding the nonlinear evolution of the superconductor amplitude near small perturbations. The original Ginzburg-Landau equation is a partial differential equation where the solution is a complex tensor. However, we consider the simpler 1D case where $u(x)$ is a real scalar function, and the equation turns into an ordinary differential equation.

Following this, chapter 2 will examine the mathematical foundation for neural networks and neural network methods to solve differential equations. Chapter 3 contains the solution procedure and numerical simulations for creating a solver for the Ginzburg-Landau equation to test the technique. Chapter 4 will synthesize the research findings and propose future directions of study. Lastly, the appendix will include a survey of a few different traditional numerical methods and compare them to the trial solution method in a computational experiment.

Chapter 2

Deep Neural Networks (DNNs)

2.1 Mathematical Framework for Neural Networks

DNNs are built from the ground up using neurons. Inspired by the biological neurons in the brain, the neurons are the building blocks of DNNs that are essential for each computation. The neuron function is defined as:

Definition 2.1.1: A neuron function $f : \mathbb{R}^n \to \mathbb{R}$ is a mapping of the form

$$f(x) = \lambda(\alpha \cdot x + \beta) \tag{2.1}$$

where $\lambda : \mathbb{R}^n \to \mathbb{R}$ is a continuous nonlinear function called the activation function, and $\alpha \in \mathbb{R}^n$ is a vector of weights, and scalar $\beta \in \mathbb{R}$ is called the bias. Here, $\alpha \cdot x$ is the inner product on $\mathbb{R}^n$.

The neuron function is an affine function that can approximate nonlinear functions because of the activation function. Combining multiple neurons together creates a layer function.

Definition 2.1.2: A layer function g: $\mathbb{R}^n \to \mathbb{R}^m$ is a mapping of the form

$$g(x) = (f_1(x), f_2(x), \cdots, f_m(x)) \tag{2.2}$$

where each $f_i : \mathbb{R}^n \to \mathbb{R}$ is a neuron function of the form (2.1) with its own vector of weights parameter $\alpha_i = (\alpha_{i,1}, \cdots, \alpha_{i,n})$ and biases $\beta_i = 1, \cdots, m$.

We define an artificial neural network along the lines of [3]. Principally, a neural network is simply a composition of these layer functions.

Definition 2.1.3: An artificial neural network (ANN) is a function h: $\mathbb{R}^n \to \mathbb{R}^m$ of the form

$$h(x) = h_M \circ h_{M-1} \circ \cdots \circ h_1(x), M \geq 1 \tag{2.3}$$

where each $h_i : \mathbb{R}^{n_i - 1} \to \mathbb{R}^{n_i}$ is a layer function with its own weight matrix A defined as:

$$A = \begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1n} \\ \vdots & \ddots & \vdots \\ \alpha_{m1} & \cdots & \alpha_{mn} \end{pmatrix} \tag{2.4}$$

and its own vector of biases β defined as:

$$\beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_m \end{pmatrix} \tag{2.5}$$

serving as the parameters of the network. Hence, (2.2) may be written as

$$g(x) = \bar{\lambda}(Ax + \beta), \tag{2.6}$$

where $\bar{\lambda} : \mathbb{R}^m \to \mathbb{R}^m$ is the vectorial activation function defined as

$$\bar{\lambda}(x_1, \ldots, x_m) = (\bar{\lambda}(x_1), \ldots, \bar{\lambda}(x_m)) \tag{2.7}$$

for a scalar activation function $\bar{\lambda}$ as in Definition 2.1.1.

The neural network structure can be represented as a computational graph shown in Figure 2.1. Each neuron function represents a connection from one node to another node, where along that edge, the weights $\alpha_{i,j}$ determine how much emphasis or weight is placed on each node. The figure omits biases, but each bias raises or lowers the importance of the node. The layers are represented graphically as the column of nodes. The connections between layers represent applying the affine function to each node in that layer to get to the next layer (2.6) and the use of the vectorial activation function (2.7).

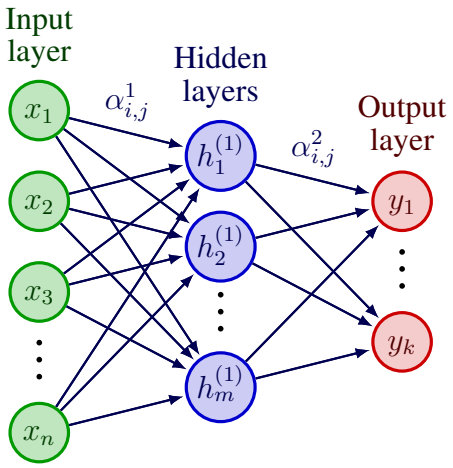

Figure 2.1: The figure shows the graphical representation of a fully connected DNN with three layers. The input layer is green, the hidden layer is blue, and the output layer is red. Each node represents a neuron with a value of x_i connected to another node by a weight $\alpha_{i,j}$ with the superscript denoting the which layer it connects and no biases. The output vector $y = (y_1, \cdots, y_k)$ are the solution.

The input layer, h_1, is determined by the dimension of the input data, where if we have n data points, then the input layer will have correspondingly n nodes in the green layer for Figure 2.1. Connected to the input layer is the series of layers called hidden layers, h_i, shown in blue for Figure 2.1. Many hidden layers are not required, but often to achieve a complex output many hidden layers are needed. There is no limit to the number of hidden layers, but it becomes computationally costly to add more hidden layers. DNNs often have more layers with fewer neurons per layer as opposed to wide neural networks with fewer layers but more neurons per layer. The network output is given by the last layer, h_M, for a network with M layers. The output layer shown in red for Figure 2.1 with the column of k nodes is what the network returns and is evaluated in subsequent steps.

Other network architectures, like recurrent networks, have graphical cycles between layers. In this study, we will consider only feedforward networks which means that the graph has no cycles or loops and so the information is only passed forward to the next neuron. There are other network architectures like recurrent networks that have graphical cycles between layers. Additionally, we also constrain the networks to be fully connected networks, which means that every node in one layer is connected to every node in the previous layer.

The goal of neural networks here is to find the solution to the differential equation, meaning to find an accurate approximation of the inverse differential operator $A^{-1}f$. Here the actual expres-

sion for the inverse operator is unknown, and the solution to the differential equation is unknown. However, we want to minimize the difference between the unknown actual solution u_{actual} and the approximate solution by the neural network $\hat{u}(x, \alpha)$ so that the difference $u_{actual} - \hat{u}(x, \alpha)$ goes to 0. A network achieves this formally by performing an optimization problem where the goal is to minimize a function representing this error known as the loss function $L(x, \alpha)$. The output of the neural network $\hat{u}(x, \alpha)$ depends on the tunable parameters α in each layer. By modifying these parameters, the network can minimize the error between the network approximation and the actual value. Training is the process of reducing this error by adjusting the parameters iteratively. For example, optimization techniques like gradient descent update the weights towards the direction that minimizes an objective function for the network known as the loss function, which is the negative of the gradient according to $\alpha_{\text{new}} = \alpha_{\text{old}} - \tau \nabla_\alpha L(\alpha)$. Here, τ is the learning rate, and $\nabla_\alpha L(\theta)$ is the gradient of the loss with respect to the parameters. The learning rate τ controls how large the step is in each direction. Frequently, the step size is adaptively changed to enhance convergence.

Since the loss function often represents a proxy for the error that we are trying to minimize then a common example for the loss function is to perform a similar optimization to a least squares minimization or find the L^2 error:

$$L(x, \alpha) = \min_{\alpha} \sum_{i=1}^{N} ||u_{actual} - \hat{u}(x, \alpha)||^2 \tag{2.8}$$

It would be ideal if the actual solution u_{actual} were known a priori. Neural networks are generally subdivided into three classes, supervised, semi-supervised, or unsupervised learning, based on whether the corresponding answer to that data point is known in the data set. The data set used by the neural network is known as the training set T.

2.1.1 Supervised Learning

In the supervised case, the objects in the training set T have known solutions and are labeled before training. In certain scenarios, one may already have access to an efficient solver for the differential equation in consideration. For the case of solving differential equations, this would correspond to T containing the RHS of the differential equation f and the solver computing solutions u for that RHS f. The network tries to create approximations for u and solve the differential equation where the exact error is computing the difference between the approximated and the label solutions. The network here tries to solve the differential equation for a large number of RHS, say 10000, at the same time giving the training set T the form:

$$T = \{(f_1, u_1), (f_2, u_2), \cdots (f_{10000}, u_{10000})\} \tag{2.9}$$

The idea is that giving the network training data with computed solutions that it can compare to the network will generalize the solution it learns from training to a broader number of problems and offer accurate outputs $\hat{u}(x, \alpha)$ for previously unseen inputs f. This method is more direct to set up and easily trainable. However, this method has the downside that a solver must already exist to have labeled training data and is often expensive to utilize.

2.1.2 Unsupervised Learning

Unsupervised learning provides a mechanism for reducing the computational overhead commonly associated with traditional methods for constructing training sets. By forgoing the need for another solver, computational costs are directly reduced. The training set is now just composed of only the RHS that the network trains over.

$$T = \{f_1, f_2, \cdots f_{10000}\} \tag{2.10}$$

Unsupervised learning presents the benefit that the methodology's design is agnostic to the actual solution of the differential equation, which removes any dependencies on prior knowledge or direct supervision.

2.2 Defining the Loss Function

The defining question of neural networks is how to choose the loss function. The loss functions form and exact implementation are dependent on the resources available and whether training data can be labeled. Another question arises: how do we find an appropriate loss function that is able to solve differential equations? The field of scientific machine learning and DNN methods for solving differential equations apply the same basic principles for choosing a loss function as laid out in Lagaris et al. [12].

2.2.1 Mathematical Framework Solving Differential Equations

The answer comes from I. E. Lagaris, A. Likas, and D. I. Fotiadis in their paper "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations" [12]. This paper laid out a general framework and a specific technique for solving differential equations with neural networks. Beforehand, the attempts to solve differential equations with DNNs were limited to trying linear systems of algebraic equations, and neural networks were mainly used for regression and classification problems only. However, in the paper [12], they suggest a framework for solving differential equations that is generally applicable, and most modern DNN differential equation solvers are built off this framework, at least implicitly.

If we consider the bounded domain $\Omega \subset \mathbb{R}^n$ with $x \in \Omega$, then the differential equation with boundary conditions can be moved to one side, so G is an operator that contains both the right-hand side and left-hand side of the differential equation and $g(x)$ is still the term for the boundary conditions.

$$\begin{aligned} G(x, u(x), \Delta u(x), \Delta^2 u(x)) = 0 \quad \text{for} \quad x \in \Omega \\ u(x) = g(x) \quad \text{for} \quad x \in \partial\Omega \end{aligned} \tag{2.11}$$

Here:

- $u : \Omega \to \mathbb{R}$ represents the solution.
- $G : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n \times n} \to \mathbb{R}$ characterizes the differential operator and its associated boundary conditions.

- $g : \mathcal{B} \to \mathbb{R}$ sets the BC which are taken to be Dirichlet or Neumann in the paper.

Evaluating this functional G gives the solution's deviation from the differential equation. If the function found is the solution, then the operator G will return 0, and any deviation leads to an error.

For example, given a differential equation of the form (1.1) $Au = f$, we can define an operator G as $G(u) = Au - f$. Therefore, plugging in the exact solution u^* to the differential equation into G gives $G(u^*) = 0$; however, any other that is not the solution would lead to an error.

The idea is now to parameterize the solution u using a neural network, denoted as $u = u_\alpha$, with α representing the neural network parameters. Now after creating a functional representation of an operator then we are now able to find the inverse mapping to solve the differential equation. Virtually every future version of solving a differential equation with a neural network tries to find G in the loss function with an additional term for g to account for the boundary loss.

To specifically solve the problem for DNNs, we break the domain Ω into a series of points $\bar{x} = (x_1, x_2, \cdots, x_\mu)$ from some form of discretization. We separate the points of the discretization into the interior points (formally collocation) and boundary points. We now solve a series of equations for the operator G for each point x_i for $i = 2, \cdot\mu - 1$ on the interior:

$$\begin{aligned} G(x_i, u(x_i), \Delta u(x_i), \Delta^2 u(x_i)) = 0 \quad \text{for} \quad x \in \Omega \\ u(x_i) = g(x_i) \quad \text{for} \quad x \in \partial\Omega \end{aligned} \tag{2.12}$$

Transitioning from a continuous domain into a discrete one is necessary for computational implementation but introduces an error. The error in the computation is not as clearly defined as for more traditional methods. Solving the differential equation amounts to finding the parameters of the network that find the minimum of the loss function $G = 0$. Relaxing the conditions of the loss function to minimize the difference between every point along the continuum to select discrete points relaxes the conditions necessary to find a perfect solution to the differential equation. Depending on the neural network method, the discretization of the domain can be uniform or nonuniform, allowing for various choices for the number of points and their location. For example, the method allows for taking the number of points μ in $\bar{x} = (x_1, x_2, \cdots, x_\mu)$ to be as high as we want within the limits of computational processing power. There is also a reduced risk of overfitting and higher accuracy with more input data, which makes "big data" regimes especially sought after. To recover the continuous loss function from Equation 2.11 to the discrete loss function actually used in Equation 2.12, would be the equivalent of taking infinitely taking points on the domain $\lim_{\mu\to\infty} \bar{x}$. This is realized by taking many points, with some papers taking as many as 500 million data points.

More advanced methods of solving neural networks do not require the points x_i to be taken from a uniform mesh. For methods like the deep Galerkin Method [17], the points are chosen stochastically on the interior. Without requiring the choice of points in time and space directly to be the same as the grid, then we do not ever have to form a mesh. Therefore, unlike many traditional solvers, this method can be meshless. Making the system meshless greatly allows us to expand the problems we can solve and allows for both unsupervised and supervised methods. This also makes the gradient descent steps we calculate now over the stochastically chosen points x_j. We have the problem that using gradient descent for G requires calculating the derivatives of G and then one derivative higher in the gradient descent algorithm to take a step. For example, if the differential equation had a second derivative term, then the computational complexity to calculate the operation for the arithmetic and memory costs would be $O(\mu^2 N)$ where N is the batch size.

Then, the stochastic gradient descent algorithm

$$\alpha^{(n)} = \alpha^{(n-1)} - \tau \nabla G(\alpha^{(n-1)})$$

requires taking now the third-order derivative, which is a massive computational increase as the size of μ increases. This is why, in practice, the higher-order derivatives are approximated using Monte-Carlo methods. Introducing Monte-Carlo integration has its own challenges, such as introducing bias and a higher amount of variance to the stochastic gradient descent, which already has inherent variance but no bias. This noise introduced by Monte-Carlo methods should average out over many iterations, but it is still an intrinsic problem when approximating the derivatives.

In general, for solving differential equations, we define the loss function as to reduce the error between the solution and the differential equation as a residual error:

$$L(\alpha) = \int_{\Omega} G(x, \hat{u}(x,\alpha), \Delta\hat{u}(x,\alpha), \Delta^2\hat{u}(x,\alpha))^2 \, dx \tag{2.13}$$

Therefore, if we find parameters α such that $\hat{u}(x,\alpha)$ in some space H, the neural network solution so that $L(\alpha) = 0$, then we have a solution to the differential equation. This can also be seen as equivalent to defining the loss function using a variational approach. In this approach, we turn the problem into an optimization problem where we try to minimize the network parameters on the interior to form the closest approximate solution:

$$\min_{\tilde{u} \in H} \int_{\Omega} G(x, u(x), \Delta u(x), \Delta^2 u(x))^2 \, dx \tag{2.14}$$

This framework works incredibly well in solving the differential equation on the interior. However, the BC for the solution $\hat{u}(x,\alpha)$ is important as it uniquely defines the problem and has physical significance. This can be seen as finding a solution to the integral in equation 2.13, but without specifying initial or boundary conditions, there is an extra constant $+C$ term when solving denoting existence but not uniqueness. Therefore, we need to be able to introduce into the loss a method to ensure that $\hat{u}(x,\alpha)$ satisfies the boundary conditions to create a unique solution.

Borrowing the naming system from [4], two general methods exist for modifying the loss function to satisfy the boundary conditions.

1. Employing a method inherently constructed to satisfy the boundary conditions is called a hard assignment. An example of a hard assignment method is the *trial solution method*. This method is further expanded upon in the appendix and compared to traditional numerical methods.

2. Adding a penalty term to the loss function to penalize deviation from the initial or boundary conditions is called soft assignment.

2.2.2 Soft Assignment: Physics-informed Neural Network (PINNs)

Physics-Informed Neural Networks (PINNs), introduced by Karniadakis et al. in 2018 [16], represent the state-of-the-art method for solving differential equations with neural networks. Interest in the field grows significantly each year and now the method has been adapted thousands of times to generate different kinds of PINNs for different circumstances. PINNs represent an an

extension of the loss function in the same form as the approach from Lagaris et al. [12] while incorporating boundary conditions through soft assignment. The loss function consists of two main terms $L = L_{physics} + L_{boundary}$, where:

- $L_{physics}$ corresponds to the discrepancy between the network's approximations and the differential equation, embedding the system's physical laws.
- $L_{boundary}$ corresponds to the discrepancy from the precise boundary and initial conditions, typically employing the mean square error (MSE) for its flexibility.

This approach is incredibly flexible, and any boundary or initial condition can be accounted for relatively easily. Additionally, this formulation allows for the straightforward integration of further constraints as just an additional term in the loss function to optimize over. The straightforward integration of additional constraints into the network's loss function allows it to be applied to difficult problems like high-dimensional PDEs where traditional methods would suffer, and the formulation facilitates both supervised and unsupervised learning approaches.

Further mathematically expanding on the PINN idea in [18], the general framework for solving PINNs relies on the minimizing the two main terms of the loss function the differential equation and boundary term. If we consider the bounded domain $\Omega \subset \mathbb{R}^n$ with $x \in \Omega$, then we can represent any differential equation similar to (1.1) but now on the entire domain. We can define operators representing these terms:

$$\begin{aligned} N(x,u) &= f(x) \quad \text{for} \quad x \in \Omega \\ B(x,u) &= g(x) \quad \text{for} \quad x \in \partial\Omega \end{aligned} \tag{2.15}$$

then the differential equation with boundary conditions can be moved to one side so that we have new operators representing the error on the interior due to the differential equation $\mathcal{N}$ and the error due to the boundary conditions $\mathcal{B}$. These operators are defined such that:

$$\begin{aligned} L_{physics} := \mathcal{N}(x,u) &= N(x,u) - f(x) = 0 \quad \text{for} \quad x \in \Omega \\ L_{boundary} := \mathcal{B}(x,u) &= B(x,u) - g(x) = 0 \quad \text{for} \quad x \in \partial\Omega \end{aligned} \tag{2.16}$$

Therefore, the loss function $L = L_{physics} + L_{boundary} = 0$ when the network finds parameters that are an exact solution to the differential equation and adhere to the boundary conditions. More formally, for higher dimensions and general L^p spaces, the loss functional has the form:

$$\hat{L}(u) = c_1 \int_\Omega \mathcal{N}(x,u)^2 dx + c_2 \int_{\partial\Omega} \mathcal{B}(x,u)^2 dx, \tag{2.17}$$

for some fixed p and $c_1, c_2 > 0$. c_1, c_2 represent weights that balance the model's focus on solving points on the interior, satisfying the differential equation between the boundary accuracy. The approximate solution $\hat{u}$ by the network of which the first k (potentially partial) derivatives exist and have finite L^p norm. However, in most practical cases $p = 2$ and can be viewed as a mean square error (MSE) for the interior and boundary terms.

The loss functional ideally is an integral with infinite points, but the data we use is finite since this cannot be done by a computer unless we discretize the domain. We can do so by using Riemann sums as an approximation of the continuous integral and evaluate the loss function on the series of points in the domain, where we create a uniform mesh of the domain $[0, 1]$ by taking a

uniform partition (made out of $n+2$ points for example) $0 = x_1 < x_2 < ... < x_{n+2} = 1$. Then, by using the Riemann sum of the function in the integrand we obtain:

$$L(\alpha) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{N}(x_i, u(x,\alpha))^2 + \lambda \left(\frac{1}{n_B} \sum_{i=1}^{n_B} \mathcal{B}(x_i^B, u(x,\alpha))^2 \right) \tag{2.18}$$

where:

- x_i and x_i^B denote interior and boundary points, respectively.
- λ is a tunable hyperparameter that arbitrates the balance between the model and boundary conditions.
- After training, the neural network $\hat{u}(\alpha, x)$ should approximate the solution of the differential equation, denoted as $u(x)$.

The points on the interior are chosen similarly to collocation methods for solving differential equations. The robust framework of PINNs allows for exceptional flexibility in dealing with differential equations problems and constraints. Since the loss function minimizes the deviance from the differential equation, this leads to high generalizability and generalization power [13]. However, this method has the drawback that the BCs might not be adhered to rigorously. This could be problematic as the BCs represent physical information to the problem.

From [18], there is a guarantee that if the neural network starts $\delta > 0$ close, then the neural network's solution is ϵ close to the actual solution. Therefore for any $\epsilon > 0$, there exists a $\delta > 0$ such that the approximate solution $\hat{u}$ converges as:

$$\hat{L}(u) < \delta \rightarrow ||\hat{u} - u|| < \epsilon \tag{2.19}$$

This theorem suggests a consistent closeness between the computed and true solutions when the loss is sufficiently small. Despite their universal nature, achieving a low loss does not always guarantee the accuracy of differential equation approximations. Specifically, if $L < \delta$, it implies $\hat{u}(x) \approx u(x, \alpha)$. To be more specific, a sufficiently small loss $L < \delta$ suggests that the approximation $\hat{u}(x)$ is close to the true solution $u(\alpha, x)$. However, these two solutions might still diverge substantially. Furthermore, reaching this particular loss threshold may not always be feasible due to various factors, including the complexity of the differential equation and the limitations of computational resources. Therefore the initialization of PINNs is important, but has not built enough theory yet to find the optimal initialization methods.

The general problem of minimizing the loss function here is a multi-objective optimization problem, where the network's finite computational capacity results in trading accuracy from boundary to interior. The boundary conditions often relate to physical constraints on the system that have to be satisfied for the system to make sense, therefore it is essential that the boundary conditions are satisfied. Since this method is a soft assignment method, the BCs are often not exactly satisfied. Moreover, training the network is essentially trying to minimize over the interior and boundary values of the problem simultaneously. The network might not be capable of doing both at the same time. To solve this, we introduce certain weights λ to tune the importance of the boundary and interior. However, the network only has a certain amount of nodes and capacity to tune this and all

that is strictly known is that $\lambda > 0$. There is not a known optimal method for choosing λ, but λ is most likely a large value and is rarely equal to the same weight as the interior. One method of finding the ideal λ value is brute force where we can iterate over the possible values; however, the number of hyperparameters make testing a sufficient number of combinations very computational expensive and slow.

Chapter 3

Computational Method, Setup, and Results

3.1 Unsupervised Machine Learning for Solving Ordinary Differential Equations

Traditional and neural network techniques to solve differential equations can find solutions for differential equations with fixed RHSs and BCs. Typically, changing either the RHS or BC requires the network to be completely retrained using the methods above. Our goal is to create a solver that is able to approximate solutions to any differential equation given any RHS or BCs without needing to be retrained. Having a solver for any RHS or BC would streamline the solution process for solving differential equations and potentially broaden the space of applications.

Grafton [7] was able to create a generalized solver for varying RHS and BC using supervised learning. This work extends the same idea for unsupervised learning, which is inherently more complex to perform. Unsupervised learning without explicit labeling of the data set, attempts to uncover the solution from the inherent structure of the data without human intervention. The benefit of this is we are able to circumvent the need for a separate solver and the computation with the explicit labeling of training data. Overall implementing the method should reduce the computational time and resources necessary, especially for difficult to solve differential equations.

Without any knowledge of the actual solution the network will use a loss function borrowing the form from [12]. The objective is to create a solver that minimizes the error between the network approximated solution and the actual solution for a differential operator A for the equation $Au = f$. By finding the parameters that minimize the loss the network will approximate the inverse mapping of the differential operator $u = A^{-1}(f)$, where A^{-1} maps f to its corresponding solution u for the Ginzburg-Landau equation $Au := u'' + u - u^3 = f$.

3.1.1 Training Set Generation

To study the network, a training set T is generated as a series of hypothetical solutions in C^2 composed of the first ten modes of the real discrete Fourier series to synthesize the data. The collection of functions are constructed with uniformly randomly assigned coefficients in a specified range. Specifically, each u_i has the form:

$$u_i = \sum_{k=0}^{10} \left(\frac{a_k}{k} \sin(kx) + \frac{b_k}{k} \cos(kx) \right), \tag{3.1}$$

where $a_k, b_k \in U(-10, 10)$. Note that the boundary conditions $u(0) = a$ and $u(1) = b$ are not prescribed beforehand. Following this, we calculate the corresponding RHS $f_i = Au_i$ for each solution u_i, for $i = 1, \ldots, 10,000$. Consequently, the training set T is comprised of these calculated values:

$$T = \{f_1, f_2, \ldots, f_{10,000}\}.$$

Each object in the training set f_i corresponds to a known solution u_i that the network does not know but is useful as a benchmark of comparison. The number of terms, the number of each u_i, and the uniformly random distribution for the coefficient are all modifiable parameters to the data. The decay term and the $1/k$ term on the coefficients were introduced to ensure that the solution converges and to dampen higher-order oscillation terms.

In J. Grafton's [7] approach, the collection of solutions were generated from fourth-degree polynomials with integer coefficients between 0 and 5. This work extends the solution space from fourth-degree polynomials to a broader space of functions described by the Fourier series solutions. The set of functions that are representable by the Fourier series is a broader class than functions that can be represented by polynomials. Therefore this expansion to a wider class of functions allows for solving a more general class of solutions to be represented.

The data generation process starts by designating the number of functions to build the training set T, in this case $10,000$. Then the $10,000$ discrete Fourier series functions u_i are generated to build a matrix U containing each u_i. The input data f is prepared by solving Ginzburg-Landau equation $f = \frac{d^2u}{dx^2} + u - u^3$, where each RHS f_i is derived from $f_i = Au_i$. Following this to work with finite-dimensional data the computer can handle, we dissect the interval $[0, 1]$ into uniform segments with N points equal to $64, 128, 256, 512,$ or 1024 and size of the interval $h = \frac{1}{N-1}$ to evaluate u and f at these points. This procedure generates an input matrix F, containing the evaluations of each f_i, which serves as the neural network's input. The model does not use the data from the initial solution u_i except the endpoints $u(0)$ and $u(1)$ which serve as the boundary conditions in the loss function.

The dataset is partitioned for training by splitting 80% of the functions in the training set T for training and 20% of the functions to evaluate the model in the test set. This 80/20 split between training and testing set data is a standard practice in machine learning.

3.1.2 Training Procedure

Training begins by generating and setting the inputs: f representing the right-hand sides of the differential equation, along with a and b which are the boundary conditions. The network initializes weights and biases for each layer according to a predetermined distribution and initialization strategy. After initialization, the network evaluates the initial loss function using the provided input data F and modifies the weight and bias parameters α to step in the direction of minimizing the loss function or the negative gradient. The method is called gradient descent, which modifies the parameters after each time step called an epoch as:

$$\alpha_n = \alpha_{n-1} - \tau \nabla L(\alpha_n) \tag{3.2}$$

The learning rate τ controls how large the step is in each direction. Instead of directly applying gradient descent to the entire data set, the parameters are update from the average of continuously choosing a smaller collection of randomly chosen objects from the training set T, called a batch T_n. The network passes through enough batches to equal the size of the training set and updates α afterwards. The length of time to perform this computation is known as an epoch in a process called stochastic gradient descent (SGD). SGD is used in computation because it has a lower computational complexity and is able to better avoid local minima then gradient descent. The learning rate is decreased after a specified number of epochs to more finely resolve the ideal parameters as it gets closer to the global minima.

The loss function is designed to minimize the discrepancy between the computed solution and the actual solution of the differential equation. It is designed to reach a minimal value when the parameters find an exact solution to the differential equation satisfy the condition $Au = f$ where exactly at a solution where $L = 0$. The form of the loss function aligns with the design of the

simplified PINNs loss function from [18]:

$$\hat{L}(u) = \int_{\Omega} \mathcal{N}(u,x)^2 dx + \lambda \int_{\partial\Omega} \mathcal{B}(u,x)^2 dx \tag{3.3}$$

for general differential equations. Specifically, for the Ginzburg-Landau equation $\frac{d^2u}{dx^2}+u-u^3 = f$ with Dirichlet boundary conditions the loss function here is:

$$\hat{L}(u) = \frac{1}{N} \sum_{i=\frac{1}{N}}^{\frac{N-1}{N}} (A\hat{u}(x,\alpha) - f(x,\alpha))^2 + \lambda((\hat{u}-a)^2 + (\hat{u}-b)^2), \tag{3.4}$$

where is $\hat{u}$ is the network approximated solution, $\hat{f} = A\hat{u}$ is the network approximated RHS, α are the network parameters, and λ is a weighting factor emphasizing satisfying the boundary conditions. This loss function structure aims to minimize both the interior L^2 error of the solution's deviation from the differential equation, and the L^2 error of the boundary terms. The boundary conditions represent physical conditions and meaning to the differential equation. Therefore greater emphasis is placed on the network finding a physical solution that satisfies the boundary conditions emphasized by modifying the loss by a relative constant λ where $\lambda > 1$. Note that the loss function is agnostic to the actual solution, eliminating the need for prior knowledge or supervision.

The loss function's differential equation operator contains derivatives, and with only discrete data, an approximation of the derivative needs to be performed. There are multiple ways to perform this approximation. One is using automatic differentiation which leverages computational graphs and implicit differentiation. The approach here uses finite difference method derivatives to produce second derivative approximation of a point built using the neighboring values of the points around it (5.2). The overall RHS of the loss function after each training step becomes:

$$A\hat{u} = \hat{f}(x) = \frac{\hat{u}(x+h) - 2\hat{u}(x) + \hat{u}(x-h)}{h^2} + \hat{u} - \hat{u}^3$$

where h is a constant representing the uniform distance between points determined by the uniform discretization. The finite difference derivative does not requiring additional memory costs for finding derivatives like the computational graph approach. In some sense, the accuracy of the derivative in the network is capped to a resolution of h, which is predetermined by the data. The value of h in a uniform discretization here does scale with the number of points, providing further resolution for large discretization sizes of the interval $[0, 1]$. However, future methods could use adaptive stepping or stochastically taking points with other forms of the derivative to overcome this limitation.

3.1.3 Architecture

The neural network architecture is determined by the number of layers known as the networks depth and the number of neurons in each layer which is the layer's width. The first layer is the input layer its size is determined by the dimensions of the input data and boundary conditions. Similarly, the output layer is tailored to match the dimensions of the desired output, which here is the same size as the input layer's size. The number of neurons in the input was determined by the number of points N in the interval $[0, 1]$ from 64 to 1024. Then the input would have $N + 2$

nodes accounting for boundary conditions as well. The flexibility in the architecture lies within the hidden layers, where both depth and widths varied throughout the project. Optimal results were often achieved using a pyramid-like configuration, where the neuron count initially increases and then halfway through decreases across the layers.

An example of the pyramid-like network architecture would be a network with nodes in each layer as:

$$N + 2(input) \to 1.5N \to 2.25N \to 3.375N \to 2.25N \to 1.5N \to N + 2(output) \tag{3.5}$$

Another architectural variant employed a uniform layer width throughout the network. Although this configuration yielded marginally worse results compared to the pyramid structure, it offered the advantage of faster computation and function as a baseline of comparison.

The overarching goal of our neural network design was to balance computational efficiency with the ability to capture the complexities inherent in the input data. The structure of the layers determines the computational capacity of the network. By adjusting the neuron count to first increase then decrease, the network aims to refine and concentrate information critical for accurate output modeling within 4 to 8 layers which can be performed relatively quickly.

In addition to structural considerations, we enhanced the network's design with the integration of batch normalization and nonlinear activation functions at each layer. Initially, the Rectified Linear Unit (ReLU) served as our primary activation function due to its computational simplicity and effectiveness in providing a nonlinear transformation at minimal computational cost [14]. The ReLU function acting in 1D is a piecewise linear function that outputs zero for numbers less than 0 and x for positive numbers. The ReLU function is defined to be a vectorial activation function that applies to the entire vector component-wise. However, to address some of the limitations associated with ReLU, particularly in the context of negative input values, we transitioned to using the Leaky ReLU activation function. Leaky ReLU maintains the computational benefits of its predecessor while introducing a small, positive gradient for negative input values, thereby mitigating the dying ReLU issue without compromising the network's efficiency or its capacity for nonlinear function approximation.

The choice of activation function depends on a variety of factors like the type of problem, the gradient stability, the computational speed, and more, with the activation function effecting the number of computations the stability of the convergence, and the stability of the gradient. The initial choice of the activation function was the ReLU activation function. This was later updated to use leaky ReLU instead, which is the same as ReLU except for values below 0 result in $-cx$ where c is a positive constant resulting in a piecewise linear function with a negative slope on the left. Shown in the figure below is the leaky ReLU function. The choice of Leaky ReLU is that it is similarly computationally inexpensive nonlinear activation function, that also helped correct for some of the issues of the ReLU activation function at the cost of some of the nice properties.

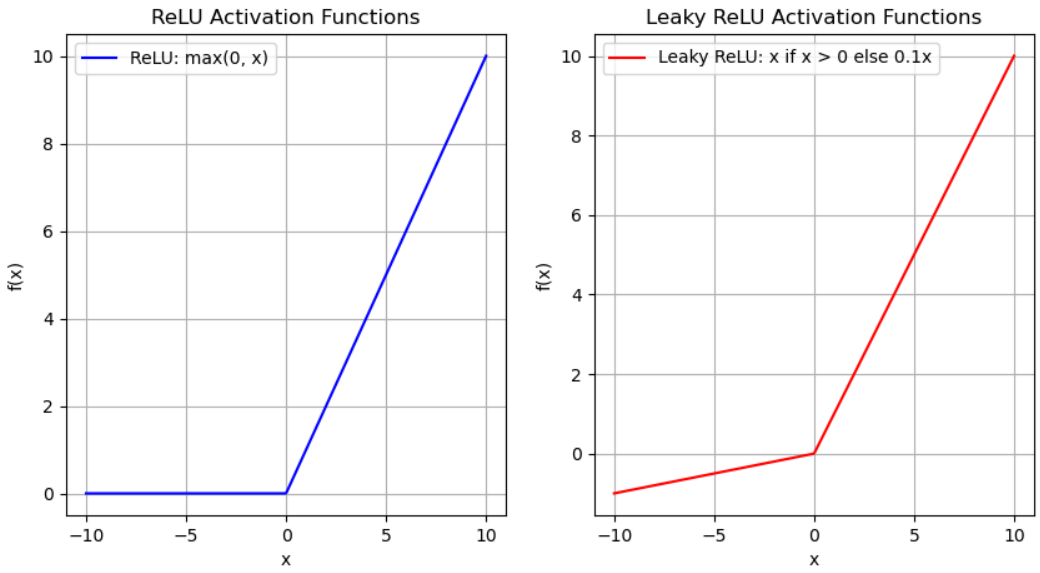

Figure 3.1: Left: The ReLU activation function. One of the simplest and most widely used activation functions [14], which outputs zero for negative inputs and the outputs the input for positive values. Right: The Leaky ReLU activation function. A variant of ReLU that has a small negative value that grows linearly for negative input values. The change to the negative values address the dying ReLU problem by preventing neurons from becoming inactive.

3.2 Results

3.2.1 Initial Results: Varying the Data

The experiments focus on training the DNN to solve the Ginzburg-Landau equation modeling the behavior of type-I superconductors in the simpler 1D case where we consider u as a real scalar instead of a complex tensor as in the original equations [1]. Using the loss function initially defined above in form (3.3) and implemented directly as (3.4) then the goal is to create a solver that given any RHS f and Dirichlet boundary condition is able to map those conditions to an approximate solution for the differential equation.

The efficacy of the model depends on several factors: the magnitude of the input data, the amount of data input, the structure of the network (width and depth of layers), the initialization procedure of the weights and biases α, and the magnitude of the boundary condition scaling parameter λ. We have found that the accuracy of the results is highly dependent on the magnitude of the data input, the number of neurons used, the initialization used, and the magnitude of the scaling parameter λ used to control the scaling of the boundary conditions to the interior. The experiment follows the training procedure above to generate matrices U and F composed of $10,000$ rows where the row entries represent the different points across the uniform discretization from 0 to 1 for the solutions and RHS respectively.

To define the ability for the network to approximate the solution the primary metric of performance was the coefficient of determination (R^2). This metric describes how well the neural

network model captures and predicts the variance in the target data [6]. Given a set of predictions $\hat{u}$ by the model and the actual solution u then R^2 is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(u_i - \hat{u}_i)^2}{\sum_{i=1}^{N}(u_i - \bar{u})^2}, \tag{3.6}$$

where $\bar{u}$ is the mean of the actual solutions and N is the number of data points. The numerator, $\sum_{i=1}^{N}(u_i - \hat{u}_i)^2$, represents the sum of squares of the residuals and measures the difference between the predicted and actual solutions. The denominator, $\sum_{i=1}^{N}(u_i - \bar{u})^2$, is the total sum of squares which measures the variance in the actual solutions.

Multiplied by 100 R^2 represented a percentage of capturing the variance and has three main regions. $R^2 = 1$ implies a perfect match between the model predictions and the actual data. An ($R^2 = 0$) implies that the prediction does not account for any of the variance in the solution. In this case, the model prediction is not any better than simply guessing the exact mean of the solution which is a straight line and does not represent any patterns in the data. A negative R^2 value occurs when the model's predictions are worse than the guessing just the mean. This often implies the model is overly complex or improperly trained.

The Relative Root Mean Square Error (RRMSE) was employed to assess solution accuracy, defined as:

$$\text{Relative Root Mean Square Error (RRMSE)} = \frac{\sqrt{\frac{1}{T}\sum_{\in T}(\hat{u} - u)^2}}{||u||^2}, \tag{3.7}$$

serving as a normalized measure of mean square error between the approximated and actual solutions. This metric allowed us to track improvements in model performance post-training. Notably, increasing the number of discretization points consistently enhanced network performance, indicating the benefit of providing more detailed data and a finer computational mesh. R^2 is the primary metric used here but it is complex and outliers in the data can disproportionately influenced its value. A high R^2 values are generally desirable, but a slightly lower R^2 might still meaningfully approximate the underlying dynamics of the solution. To get a more complete picture of the model's accuracy the Mean Square Error (MSE) and Relative Root Mean Square Error (RRMSE) are also used, where the RRMSE is defined as:

$$\text{Relative Root Mean Square Error (RRMSE)} = \frac{\sqrt{\frac{1}{T}\sum_{\in T}(\hat{u} - u)^2}}{\sqrt{\frac{1}{T}\sum_{\in T} u^2}},$$

This serves as a normalized measure for the solution error where the numerator gives the residual in L^2 and the denominator normalizes with the norm of the solution in L^2.

Characterizing the solution create by the real Discrete Fourier series, we find that the solution is smooth meaning that is continuously differentiable and has higher order oscillations created from the later terms of the Discrete Fourier series. The higher order oscillations induced by the later terms add complexity to the solution and add oscillations to the data. An initial test built the data form functions $u = c * sin(x)$ where c varied uniformly between functions from $[-100, 100]$ and had 128 discretization points between $[0, 1]$. For consistency, network outputs were generated using an architecture of fully connected layers and a small size of neurons per layer and hidden

layers were selected for its balance of computational efficiency and performance. Furthermore, the initialization procedure here is a Xavier initialization. Computations were conducted on a Python 3 Google Compute Engine backend via Google Collaboratory, utilizing an A4 GPU. All the results were performed using the Adam optimizer multiple times and averaged over five trials of rebuilding the network. Furthermore, all graphs in the future were plotted so that they return the prediction and network approximation closest to the average RRMSE of the $10,000$ functions in the network. With the singular sine term, the network is quickly and consistently able to find a network approximation in around 50 epochs. The network approximation found is quite accurate to the actual solution and was found relatively quickly in only 6 layers with 1024 neurons in each hidden layer. The average RRMSE error was 9.21×10^{-6} for both the training and test set with an initial MSE 82380 decrease to 7.31×10^{-9} on the test set after training. This implies that the network is easily able to find solutions to simple and smooth functions with this method, showing a increase MSE by 13 orders of magnitude. From now on, the solutions will be built with the first 10 modes of the Fourier series. Introducing more terms of the Fourier series add oscillations to the data and mirrors testing the network's performance against data having errors or noise. Overall this change makes the training on the network more difficult.

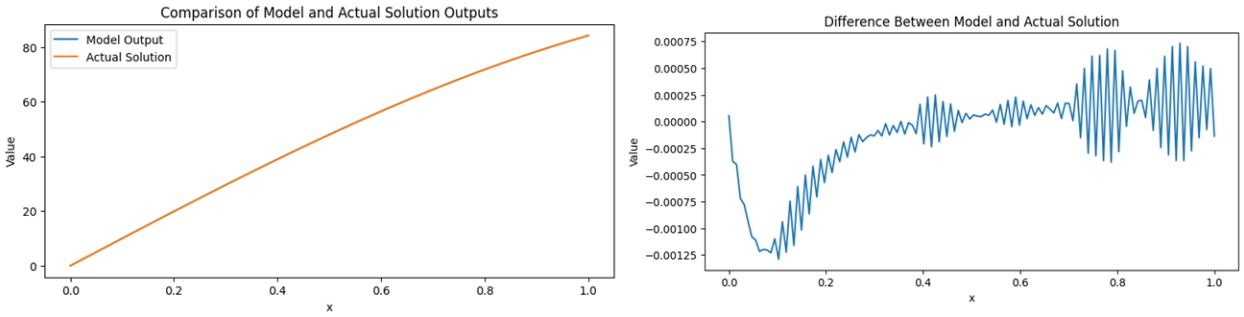

Figure 3.2: Left: The left figure shows the network approximation to finding the sine function in blue and the actual solution in orange. The network approximation is not visible the model virtually exactly matches the actual solution. The figure on the right shows the difference between the solutions plotted against their position on the interval. The network composed of only 6 layers and 1024 neurons per layer was able to converge to a solution accuracy of 9.21×10^{-4} % measured by RRMSE quickly within 50 epochs. This shows that the network is easily able to approximate functions with low oscillations and high magnitude. The network approximation appears to smoothly map to the sine function and is able to generalize well to the test set.

3.2.2 Changing Amount of Functions in the Data

So far, $10,000$ functions simulated for U and F has been the standard for computation. For future practical applications the RHS to a differential equation often corresponds to measurable phenomenon like forces and energy. To simulate the robustness of the method when there is not as much data, the network ran on different matrices of U and F with $5,000$ to $10,000$ rows representing functions. The tests were performed on coefficient ranges of $[-100, 100]$ with 256 discretization points between $[0, 1]$ for the pyramid-like network with 6 hidden layers with ReLU activation functions. The amount of data present does effect the accuracy of the solution as it was

found the larger the amount of data and functions available to the network, the higher the accuracy was. Based on the graphs for the R^2 error for this test in Figure 3.3, there is large improvement in the accuracy initially as the R^2 value on the training and test increased by roughly 9.0% between $5,000$ and $6,000$ input functions. However, there were diminishing returns as the relative increase in R^2 from $9,000$ to $10,000$ functions was achieved roughly a 0.5% increase. Therefore, this test indicates that increasing the data size past a certain point is not worthwhile. The higher accuracy comes at the cost of the greater the number of functions, the longer it takes for the model to run, whereas doubling the number of functions from $5,000$ to $10,000$ doubles the run time.

Data Size	**Initial MSE**	**Final MSE**	**Final R^2 Error**		**Time (s)**
			Training	**Test**	
5000	33 254.535	87.666	0.7358	0.7377	124.0548
6000	33 280.366	56.206	0.8254	0.8281	147.1863
7000	33 307.387	42.345	0.8736	0.8753	170.7534
8000	33 876.656	36.192	0.8815	0.8894	195.1211
9000	34 393.151	32.353	0.9029	0.9009	219.2694
10000	32 809.636	30.795	0.9083	0.9100	248.6532

Table 3.1: The figure shows the impact of changing the data size by varying the number of functions in the training and test set on neural network performance. The major highlights are the improvement in the R^2 error and MSE comparatively as there were more functions in the data and test set. The tests were performed on each data size on data that had 256 discretization points within a $[-100, 100]$ coefficient range for terms of the Fourier series. The network consisted of a six-layer pyramid-like architecture with ReLU activation between each layer. The computational time roughly doubled after doubling the size of the data from $5,000$ to $10,000$ points. This demonstrates the trade-off between data volume and accuracy, with diminishing returns on R^2 improvements beyond certain data sizes.

3.2.3 Varying Discretization Size

In general, the trend is that the more discretized points, the better the network performs until the network hits its computational capacity, which is determined by the width and depth. This observation could be a symptom of the "curse of dimensionality". Here, it is realized that the larger the dimensionality of the input data, the greater the number of effective neurons the network needs to process the information. The specific relationship between these architectural changes and performance is not entirely clear, but this empirically is found to be generally true while testing. Experimentally this effect can be seen by evaluating different numbers of discretization points between $[0, 1]$ for the pyramid-like network with 6 hidden layers, 1026 neurons in each layer, with Leaky ReLU activation functions ran for 250 epochs, coefficient sizes $[-10, 10]$, a learning rate of 0.0005, batch normalization, $\lambda = 1$ and a batch size of 128. There is a consistent improvement with each number of discretization points as the network can see a finer resolution of the interval $[0, 1]$. In the last case, once the dimension of the input data 1026 points reached larger than the number of neurons in each layer, the network struggled to improve. It did not have the same relative decrease in MSE on the test set of around two orders of magnitude that the smaller discretization sizes had.

Figure 3.3: The graph shows the comparison between the R^2 error between the test and training set while varying the data size. The R^2 error consistently improves with the data size showing that the network performs better given more data. The R^2 error between the test and training set is almost exactly the same, implying that the network is able to generalize learned patterns on the training set.

Furthermore, the largest source of error consistently in each of the tests was approximating the boundary points of the solution. This can be seen in the Figures 3.5-3.9 below for each test at different discretization sizes. The model mostly performs smoothly when approximating the RHS and interior, but unexpectedly diverges to a sharp point on the boundary. The RRMSE and R^2 variance metrics are particularly sensitive to large outliers and the metrics were heavily penalized by the boundaries. It is clear that we are using soft assignment as the term in the loss function penalizes the distance away from the boundary, but the network does not need to exactly satisfy the boundary. This poses a challenge since the large distance form the boundary can make the solution less physical. Additionally, the boundary values often pick out an exact solution for the problem, so finding a network for the wrong boundary value problem could effectively be finding a different solution entirely. This could be due to the effect of the finite difference method derivatives effect of ”compressing” data at the endpoints, where the size of the second derivative shrinks to contain two less points then the overall solution. Another possible reason why the results in the Table 3.2 suggest that the error decreases with the larger lattice sizes is that the differential equation could feel the effects of the boundary less when it is further away. Closest to the boundary is the largest source of error, but as the number of data points from the boundary increases the more the bulk and differential equation play a further role and of the minimization of the approximated solution with the actual solution. The more points on the interior and further away from the boundary then

the lower the overall error. Note that this is just speculation. Similarly, an increase in discretization points introduced greater oscillatory behavior in the solutions, likely due to the network's fixed capacity struggling with larger datasets. This could also be due to the dominance of large discrete derivatives in the loss function. The goal of smoothing out the second derivative and the RHS constructed by the network would most likely lead to a smoother solution; however, the networks goal of decreasing the loss function does not always directly align with decreasing the error. This is especially true in unsupervised learning where the exact solution is unknown. This is exemplified with the right image of Figure 3.10 plotting in blue the value of f_i for that specific function in the test set and in orange the model recreates RHS $f = Au = \frac{d^2u}{dx^2} + u - u^3$ from the predicted model solution. The matching of the RHS was particularly close and smooth; however, this did not always translate to smoothly producing a solution like in the case of 128, 1024 point discretizations, and the example below.

One positive note was that consistently throughout all the tests the error between the test and training set were very low and in some cases higher on the test set show in Figure 3.4. This suggests that the network is successfully generalizing finding a solution to the inverse differential operator without overfitting.

Discretization Points	**Final R^2 Error**		**Final RRMSE**	
	Training	**Test**	**Training**	**Test**
64	0.871 25	0.867 89	0.358 00	0.361 85
128	0.925 55	0.926 70	0.273 45	0.271 02
256	0.945 65	0.945 11	0.233 15	0.234 46
512	0.953 71	0.954 44	0.219 00	0.215 35
1024	0.267 87	0.268 16	0.857 04	0.855 05

Table 3.2: This table evaluates the impact of data density on neural network accuracy. Testing was conducted with a pyramid-like network architecture comprising six hidden layers and 1026 neurons per layer, using Leaky ReLU activation functions over 250 epochs. The number of uniform discretization points in the interval $[0, 1]$ varied from 64 to 1024, with coefficients in the Fourier series building the data ranging within $[-10, 10]$. The network had a learning rate of 0.0005, included batch normalization between layers, had equal waiting between loss components $\lambda = 1$, and a batch size of 128. Results indicate an overall improvement in model accuracy with increased discretization until the network's capacity is overwhelmed when input dimensions exceed the networks computational capacity. This point occurs around when there are a similar number of neurons in the hidden layers as discretization points. The network demonstrated strong generalization capabilities across both training and test sets suggesting the method effectively learned the inverse differential operator without overfitting.

3.2.4 Coefficient Ranges

The greatest change to the characteristics of the network solution were the coefficient ranges of the inputs in the Real Discrete Fourier series when building the data. Linear functions have the nice property of homogeneity where for any coefficient $c \in \mathcal{R}$ then $f(cx) = cf(x)$. However, nonlinear functions do not necessarily have this property. Therefore changing the magnitude of the

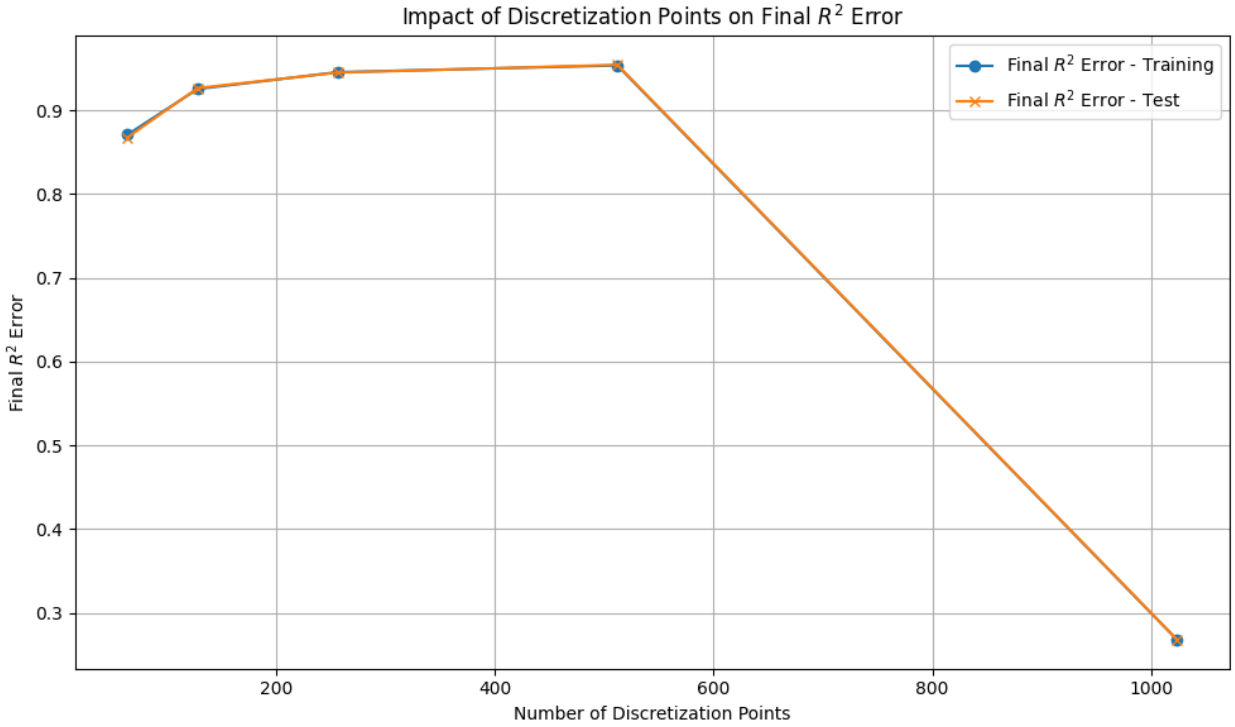

Figure 3.4: The graph shows the comparison between the R^2 error between the test and training set while varying the number of discretization points. The R^2 error consistently improves with the number of discretization points until it hits a computational limit and the performance significantly drops. Once again, the R^2 error between the test and training set is almost exactly the same, implying that the network is able to generalize learned patterns on the training set.

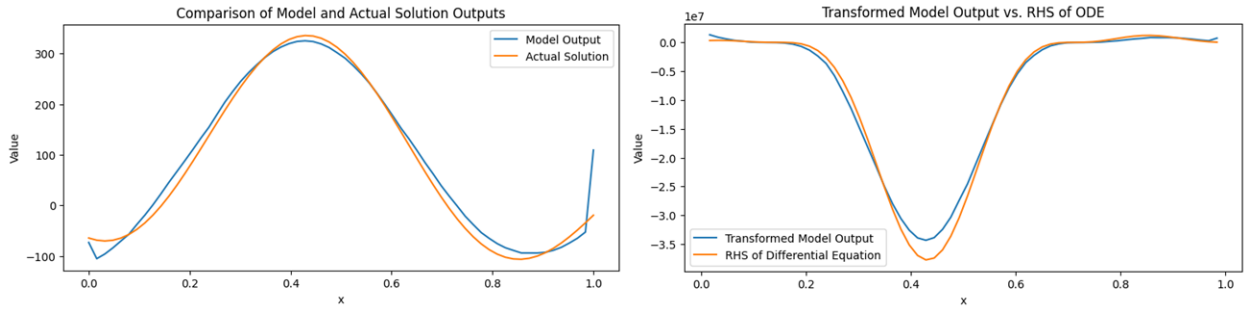

Figure 3.5: The comparative approximation of the solution u and RHS f taking 64 data points varying the number of discretization points.

initial input data changes the solution and solution characteristics for nonlinear equations like the Ginzburg-Landau equation.

It was observed that there are two distinct cases impacting network performance. The first case is for large sizes of the coefficient ranges the data, the network is able to handle the data well using at least four to eight hidden layer sizes. The results increase with the number of layers and other factors but the network is able to handle the data and produces a good approximation of the

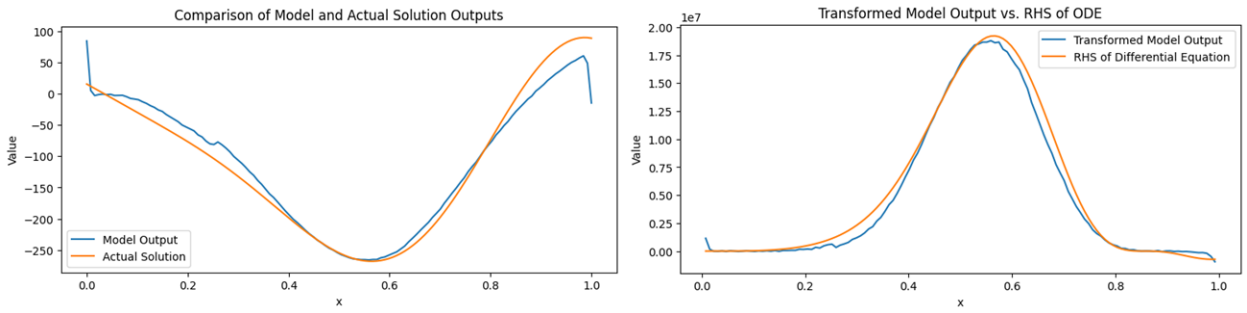

Figure 3.6: The comparative approximation of the solution u and RHS f taking 128 data points varying the number of discretization points.

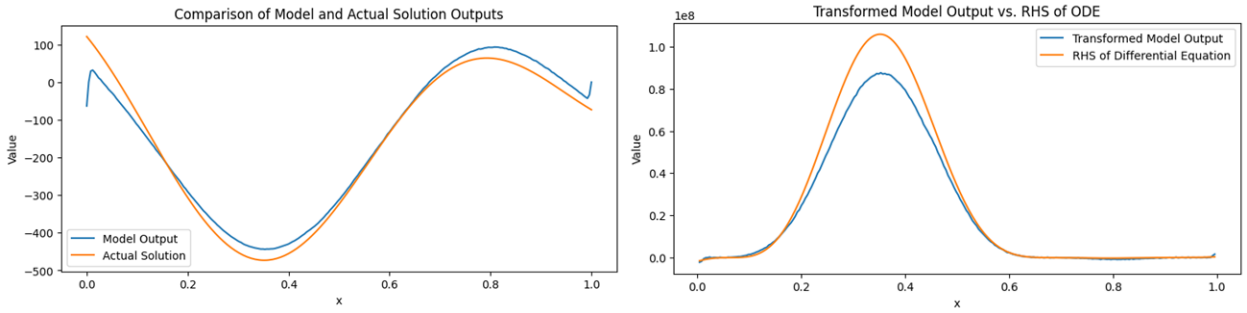

Figure 3.7: The comparative approximation of the solution u and RHS f taking 256 data points varying the number of discretization points.

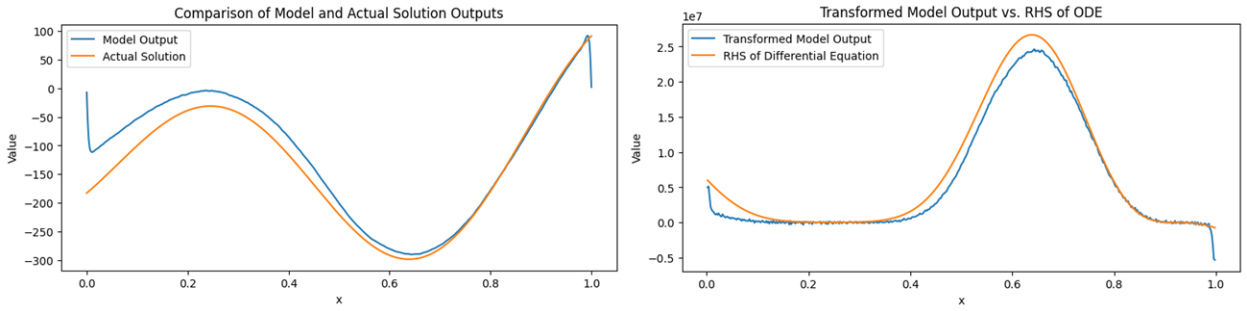

Figure 3.8: The comparative approximation of the solution u and RHS f taking 512 data points varying the number of discretization points.

solution. This can be seen in the table below showing the network was able to perform the best on the largest size coefficient range of $[-100, 100]$. The table was computed for 256 discretization points between $[0, 1]$ for the pyramid-like network with 8 hidden layers with ReLU activation functions ran for 250 epochs. Generally the tests performed on coefficient ranges of $[-100, 100]$ throughout the project, which holds true here as well. The network had the highest R^2 value in the training and test set at the $[-100, 100]$ coefficient range with a sharp decrease in the MSE. The functions fit well; however, there was large error on the boundaries and therefore the RRMSE appears quite large.

The second case is for lower magnitude data. Examining the other coefficient ranges than

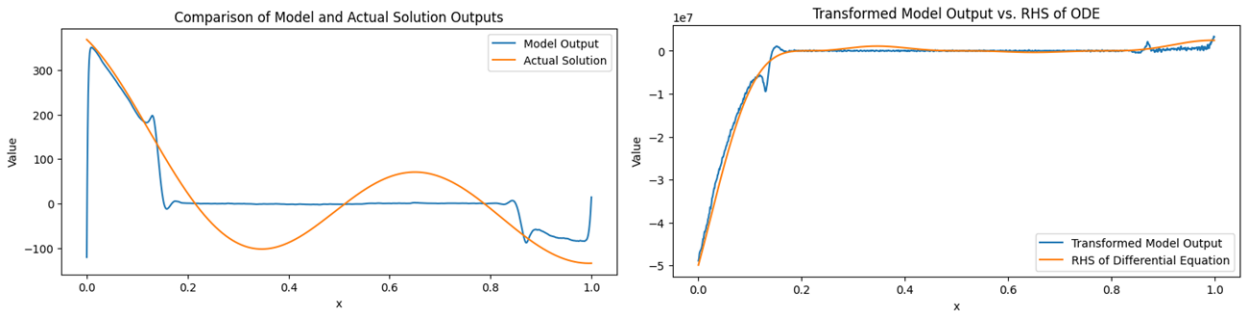

Figure 3.9: The comparative approximation of the solution u and RHS f taking 1024 data points varying the number of discretization points.

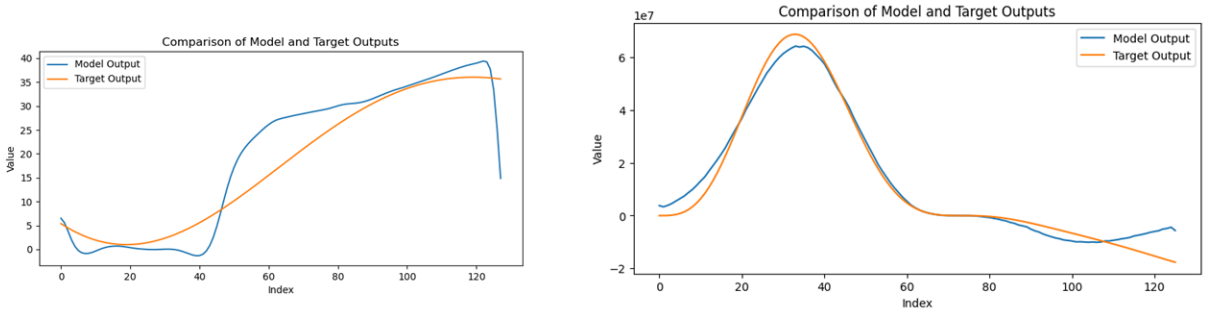

Figure 3.10: An example of initial results. We notice that the network is able to approximate the function well but has a large error on the boundary and nonsmooth structure. General improvements in the loss of the function do not directly correspond to improvements in the error even though the loss function represents the error in the differential equation.

$[-100, 100]$ for the coefficients in the Fourier series, it seems that none of the other coefficient ranges produced an adequate approximation to the actual solution. The MSE on the test set did not reduce much from the initial value after initialization. Additionally, the value of R^2 decreases with each decrease in magnitude of the coefficient range and shows that in the smallest size of the coefficient range from $[-0.01, 0.01]$ the network was blatantly wrong. However, note that the measures of the error scale also with the size data as well, where smaller error normalized on smaller coefficient ranges appear worse. Overall, it appears that in all metrics the network fails to approximate small coefficient sizes.

Specifically examining the instance in the coefficient range between $[-0.1, 0.1]$ shown below in Figure 3.11, it was observed that the network quickly converges to a suboptimal solution shown by the final training RRMSE of 141.2%. This premature convergence happens typically within 50 epochs and exemplifies a problem known as the vanishing gradient problem. The plateau in the loss function comes from the gradient no longer updating because it has suddenly become small hence ”vanishing”. Specifically, it seems the issues is the dying ReLU problem which is a kind of vanishing gradient problem where neurons effectively 'turn off' due to non-positive inputs, reducing the network's computational capacity and flexibility.

Coefficient Range	MSE		Final R^2 Error		Final RRMSE		Time (s)
	Initial	Final Test	Train	Test	Train	Test	
0.01	8.67×10^{-6}	5.09×10^{-4}	-150.600	-150.400	12.325	12.243	141.52
0.1	3.39×10^{-4}	6.60×10^{-4}	-0.984	-0.985	1.410	1.412	143.5
1	0.032 44	0.0331	-0.021	-0.022	1.010	1.010	142.64
10	3.3088	3.1483	0.0493	0.0483	0.9751	0.9754	142.36
100	3.36×10^2	18.509	0.9456	0.9451	0.2332	0.2345	141.01

Table 3.3: Analysis of neural network performance for different coefficient ranges shows that the network can accurately approximate solutions for large coefficient ranges but fails for smaller coefficient sizes. The $[-100, 100]$ coefficient range exhibited the network's best performance with an R^2 score for training and testing of 0.945, reflecting high precision and successful learning. However, the R^2 value and other performance metrics consistently worsen with the smaller the coefficient range. This occurs as the network approximates small magnitudes, and the gradient vanishes, leading to a lack of learning and plateaus in training earlier and earlier.

3.2.5 Dying ReLU Problem

The dying ReLU problem is a phenomenon where the values for the weights become highly negative for certain weights, this means that because of the structure of the ReLU activation function those neurons did not get used. Therefore there is less computational capacity of the network and large changes in the weights for those neurons still lead to the same result of having no weight on certain neurons. Effectively the number of neurons in the network are reduced and the network gets stuck. As seen in [14], there are three general approaches to fixing the dying ReLU problem.

1. **Adjust Network Architecture:** Modifying the network's structure like changing the width and depth of layers, changing the activation function type, changing the learning rate and modifying the batch size.

2. **Add Training Steps:** Incorporating steps in training like dropout and batch normalization during training significantly impacted the network's ability to maintain active neurons and prevent gradient vanishing. Introduced in 2015, batch normalization adjusts each layer's inputs to have fixed means and variances, thereby reducing the dependency on batch size and initialization while enhancing generalization. This approach allowed for higher learning rates and mitigated the risk of exploding or vanishing gradients. [5] showed that initialization is important for determining the convergence to solving differential equations.

3. **Modify the Initialization Method:** Changing the initialization strategy is used to modify the weights and biases without changing the network architecture.

All three methods were implemented into this network to help fix this problem, but the effect each one had were not equal. Implementing these changes throughout the tests, the most significant difference was in changing the number of layers. At the cost of a large increase in computational time with each increase in layer as the effective number of neurons increases, so does the accuracy of the solution and the range of parameters that the network can effectively handle. Additionally, the effects of batch normalization are quite dramatic. During batch normalization the for each layer fixes the means and variances of each layers inputs. This implies that the normalization causes the

layers to no longer be independent and identically distributed. This has its problems, but overall batch normalization empirically allowed for higher learning rate without vanishing or exploding gradients. There was a regularization effect for the batch normalization that improves the ability for the network to generalize. After adding the batch normalization the network was more easily able to handle coefficient ranges with smaller magnitudes such as $[-10, 10]$, less dependent on the learning rate, less dependent on the batch size, and less dependent on the activation function.

In these cases, in addition to changing the initialization and adding batch normalization, the activation functions and network structure were changed to produce a much better result. This implies that the ideal network parameters and architecture depend on the data itself, which is a problem that needs to be taken in consideration by the user.

Modifying the activation function to deal with the dying ReLU problem did not seem to help greatly for very small coefficient sizes. The results for Leaky ReLU which is a variant on the ReLU function seemed to help the most out of Leaky ReLU, Tanh, and GeLU; however, the vanishing gradient problem still persisted. For larger coefficients ranges like $[-100, 100]$ that do not suffer from the vanishing gradient then ReLU activation's work well and outperform other activation functions like Tanh while being less computationally expensive and only slightly slower. The results for the test with coefficient ranges of $[-100, 100]$, with learning rate 0.005, 128 batch size, and six layers with 1026 neurons per layer with a flat architecture are composed in Table 3.4. The activation function did not make a huge difference for larger coefficient ranges but did lead to noticeable improvement and similar computational times. However, for small coefficient ranges like $[-0.1, 0.1]$ the model has a higher probability of failing even with adding dropout, adding batch normalization, varying the architecture and changing the initialization strategy. The results were not included for smaller coefficient sizes because the results were not consistent or robust. These observations highlight that the optimal network parameters and architecture are contingent on the input data, especially the size. Therefore selecting and tuning the parameters of the network to match the data's specific attributes requires careful consideration by the user. This shows that designing effective unsupervised neural networks for nonlinear BVPs is tricky.

Activation Function	**Time Taken (s)**	**Training Set RRMSE**	**Test Set RRMSE**
ReLU	142.79	0.0769	0.0762
GeLU	132.53	0.0913	0.092
Leaky ReLU	134.42	0.1188	0.1193
Tanh	135.60	0.1186	0.1196

Table 3.4: The best performing activation function was ReLU in the large coefficient sizes $[-100, 100]$ regime, consistent with the theory [14]. Surprisingly, ReLU is the simplest function, but it took the longest to train. Leaky ReLU and Tanh activation functions performed the worst but were slightly faster. GeLU emerged as a middle ground in accuracy between ReLU and Leaky ReLU while performing the fastest but marginally.

3.2.6 Adding weights

Throughout the test there has been an issue that the largest source of error was on the boundary leading to large penalties to the metrics for the MSE, R^2, and RRMSE. To remedy this problem,

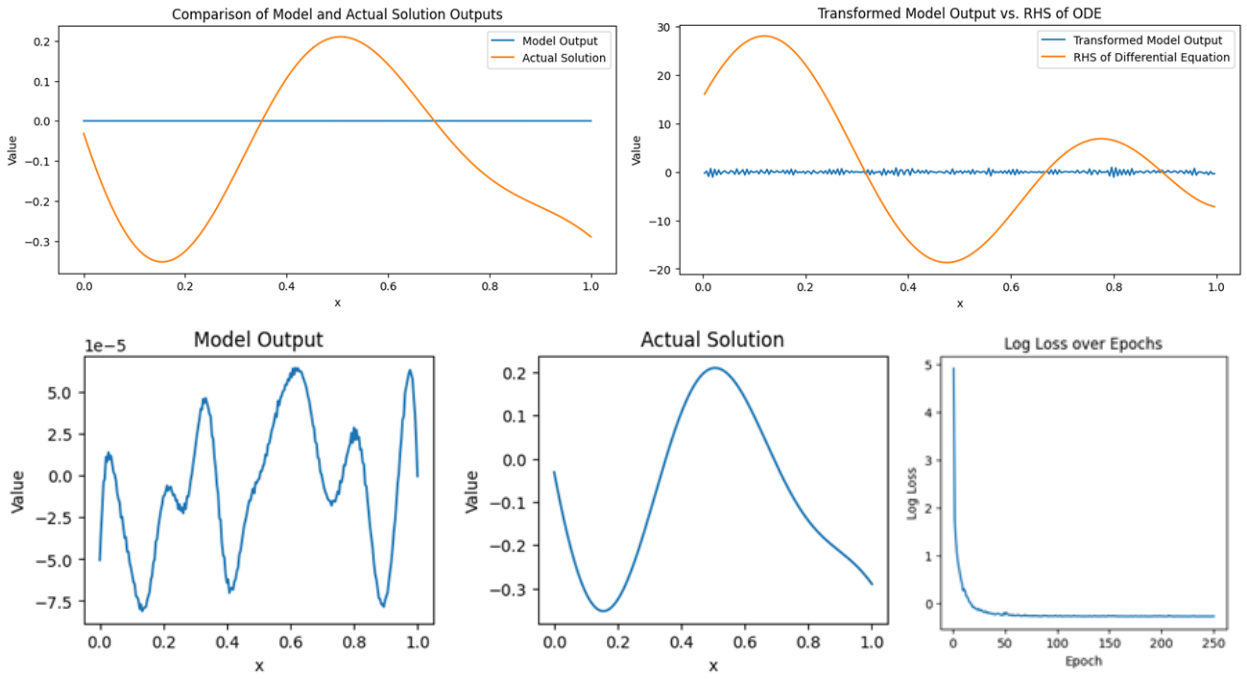

Figure 3.11: The ReLU activation function does poorly at small coefficient ranges and exhibits the dying ReLU problem where the network plateaus early to a poor result. The top left figure shows the output of the neural network and the actual solution comparison; below are the separately plotted outputs to show that the network output is at a different magnitude. The top right shows the network recreated $f = Au$ from the solution approximation vs. the actual RHS for error comparison. The bottom left shows that the loss plateaus quickly and can no longer adequately update after around 50 epochs.

extra weighting λ was placed on the boundary term in the loss function. Following the ideas of [18], by placing weighting on the boundaries we can have the network prioritize the boundary more heavily. Since the neural network has a fixed computational capacity determined by architecture width and depth, then placing more emphasis on the boundaries takes away emphasis on solving the differential equation interior. However, the interior does not need an equal weighting and there is diminishing returns placing all the weighting on the interior so it is more efficient to place more weight on the boundaries. Finding the hyperparameters for the weights is a difficult procedure and the paper suggests that the it is known that the weights placed on the boundary should be higher, but there is no defined way to find the ideal hyperparameter weighting. The methods to find the weighting so far have been through systematic testing, intuition, and trial and error.

To effectively evaluate the effects of adding the weights to a network, a test was performed with a network with 256 discretization points between $[0, 1]$. The network had a pyramid-like architecture with 8 hidden layers, Leaky ReLU activation functions for each layer, data with coefficients generated from $[-100, 100]$ uniformly, and Xavier initialization. The model was run for 250 epochs for each test, at a learning rate of 0.0005, and a batch size of 128. The results for placing more weight had the effects of overall reducing the boundary error significantly, especially for small discretization sizes where most of the error comes from the boundaries. The weighting is different for every run and seems to scale with the size of the data but not in a clear manner.

Additionally, the cost of placing more weight on the boundaries is shown by the transformed solution and the network output having more oscillations. These oscillations are quite large and seemingly non-periodic. Adding more weight on the boundary terms did marginally decrease the error and increase the R^2 value, as seen in Figure 3.12. The increasing R^2 values indicate a better network approximation, especially with the high values of the boundary error used. Additionally, at least for larger magnitude data, the network is robust to the boundary weight. Employing a large boundary weight or too small of a boundary weight has a relatively small effect on the overall accuracy of the network approximation from the baseline of an equal weighting between the interior differential operator and the boundary operator. This is true until the terms of the interior and boundary reached relatively equal magnitude in the loss function. After increasing λ so that the boundary had greater weighting in the loss at this point results in the emphasis no longer placed on solving the differential equation. The model performance decreased drastically, now prioritizing the boundaries over everything. There is dramatic improvement in the error on the boundary by roughly 33 times decrease from the initial value of an equal weighting $\lambda = 1$ and closer to the optimal value $\lambda = 1 \times 10^3$. This same improvement corresponded with an increase in the R^2 value in both the test and training set showing that model was performing more accurately and able to generalize to solving the differential equation well for large values of λ. The improvement in the R^2 value was less drastic in the larger coefficient regime in Table 3.5 but the fit was already pretty strong so further improvements are more difficult, costly, and impressive. Notably, there were also improvements in the interior error as well, as the boundary error improved the network was better able to solve the differential equation leading to improvements on both until a point.

$\log_{10}\boldsymbol{\lambda}$	**Final R^2 Error**		**Test Set Error**	
	Training	**Test**	**Boundary**	**Interior**
0	0.942 195	0.942 084	28 380.0	1747.7
1	0.941 525	0.941 418	29 656.9	1760.2
2	0.944 257	0.943 729	29 302.8	1683.4
3	0.944 950	0.943 877	29 226.6	1681.6
4	0.943 291	0.942 851	28 884.8	1716.3
5	0.940 187	0.939 758	29 937.0	1813.3
6	0.945 543	0.945 629	17 920.4	1709.6
7	0.949 525	0.950 001	6069.5	1651.8
8	0.954 796	0.954 039	4668.7	1525.8
9	0.951 416	0.950 680	1799.7	1661.6
10	0.958 740	0.958 114	853.1	1416.8
11	0.446 947	0.452 427	932.0	18 586.5

Table 3.5: Varying the weighting λ in the loss function on the boundary term showed that the higher the λ, the more significant the increase in performance measured by R^2. This occurs until the network hits a critical limit in computational capacity resulting in a significantly worse performing network. The tests were done on a network with 256-point discretized points across a $[-100, 100]$ coefficient range. The network architecture had a pyramid-like architecture with eight hidden layers, Leaky ReLU functions, and Xavier initialization. Measuring the interior and boundary error on the test sets revealed that once the boundary error reached greater weighting in the loss function than the interior, the network failed to solve the differential equation accurately.

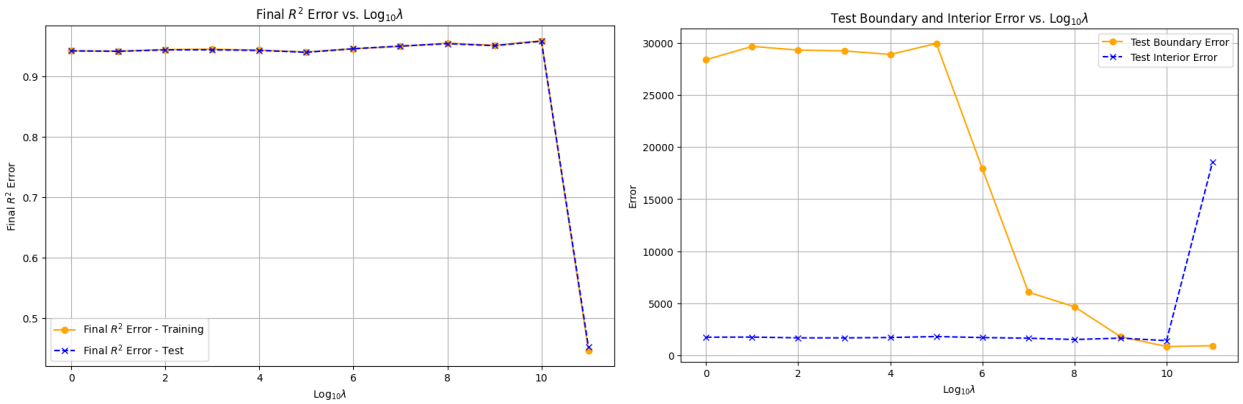

Figure 3.12: The graph on the left shows the R^2 error from the tests varying the boundary weighting in the large magnitude data regime. The graph on the right shows the boundary and the interior errors compared against each other. When the boundary weighting increases past the point where interior and boundary weighting are comparable, the R^2 error decreases heavily and the interior error increases heavily. The network performs much worse past the comparatively small gains made from increasing the weighting before the critical point.

In contrast, testing the solution on the same hyperparameters above but now using data with Fourier term coefficients of size [-1,1] the solution accuracy is much lower overall. The network clearly suffers from the vanishing gradient as the $R^2 = 0$ indicates that the model fit is similar to guessing the mean and is stuck around 0. Now there are dramatic improvements when varying the weighting on the boundary. There is an increase in the R^2 value with each successful iteration reaching a peak at $\lambda = 1 \times 10^6$ with a roughly 15.7% improvement in the R^2 value on the test set. The larger improvement on the test set then the training set implies that the model is generalizing well to finding the solution to the inverse differential operator more exactly while not overfitting. However, there is still a large difference between the test and training R^2 value at some points meaning this generalization is not always guaranteed in this regime, as seen in Figure 3.13. Furthermore, with each increase in λ the error in the boundary and interior on the test set decreases. This shows that it is both beneficial for optimizing the boundary and the interior of the differential equation to place more relative weighting on the boundary. By placing more weight on the boundary the differential equation is better able to solve that specific boundary value problem determined by the a and b boundary values rather than a different problem. There is a sharp increase in the improvement on the boundary error going from $\lambda = 1 \times 10^3$ to $\lambda = 1 \times 10^4$ showing that the improvement is not linear. Additionally, the network stops improving on the test set after reaching $\lambda = 1 \times 10^7$, where the model still improves in the R^2 value on the training set but decreases the R^2 on the test set implying that the model is now beginning to overfit and $\lambda = 1 \times 10^6$ was closer to the ideal value for the network to optimize over.

The model overall is robust to changes in λ where large changes in the loss function still lead to convergence and proper fitting of the model. The significant improvements due to varying the weights suggest that one possible avenue for dealing with the problem of the network being unable to improve for smaller lattice sizes is to modify the loss function L. This changes the optimization problem and therefore the gradient leading to improvement from the stagnant base state. Additionally, the largest source of error typically on the models so far is fitting the boundary points. With equal or relative weighting of the interior and boundary then there is significantly

more terms in the interior then the boundary. The interior is entirely separate to the boundary. The network will not see the boundary when modifying the loss as modifying the boundary is a small relative change in the loss compared to the large values of the interior. Coupled with the fact that the network using the discrete derivative over oscillating points often leads to very large magnitudes for the second derivative then the boundary and stays large even small variations in the network then the boundary is further not seen when optimizing. This can be seen by very large values of λ are necessary for the network to see the boundary. The boundary term seems to have a regulating effect on being able to see the magnitude of the equation and allows the network to center itself around solving the specific boundary value problem at hand.

$\log_{10} \lambda$	Final R^2 Error		Test Set Error	
	Training	**Test**	**Boundary**	**Interior**
0	0.046 53	0.044 02	4340.1	3137.1
1	0.045 01	−0.1125	4235.8	3144.2
2	0.047 74	−0.0892	3939.2	3140.6
3	0.101 41	0.103 59	3097.8	2983.4
4	0.171 11	0.179 66	1009.1	2771.2
5	0.189 08	0.189 50	413.1	2732.7
6	0.189 50	0.201 36	315.5	2705.0
7	0.201 36	0.189 52	323.6	2751.3

Table 3.6: The tests were performed on the same architecture, but different coefficient ranges of the Fourier series at $[-1, 1]$ than Table 3.5. The results show that increasing the weighting on the boundary term improved the performance of the network with each magnitude increase. The ability for the network to increase its performance despite being in the low magnitude regime and suffering from the vanishing gradient problem presents a means of potentially fixing the vanishing gradient issue with further analysis.

3.2.7 Varying Initialization Strategy

Initialization is especially important when evaluating nonlinear neural networks. As supported by the results from [5], nonlinear differential equations are highly sensitive to the initialization method and hyperparameters. This implies that the loss landscape of the nonlinear differential equation is complex and has many local minima and maxima that the solution can fall into. The complex loss landscape can often be seen by the network failing into and converging far away from the actual solution. Additionally, the Landau-Ginzburg equation describes a process with two large potential wells representing the global minima to the solution. Therefore which global min the network ends up in would be dependent by the distance of each well to the initial starting location. One the best ways to deal with this is just matching the initialization to be closer to the global minima, which can be hard to predict.

The initialization of the distribution of weights and biases is crucial as it is one of the two sources of randomness in neural networks. The two sources of randomness are weight initialization and randomness inherent to the data. This is a blessing and a curse because the randomness allows the network to have a higher amount of generalizability by introducing randomness means that

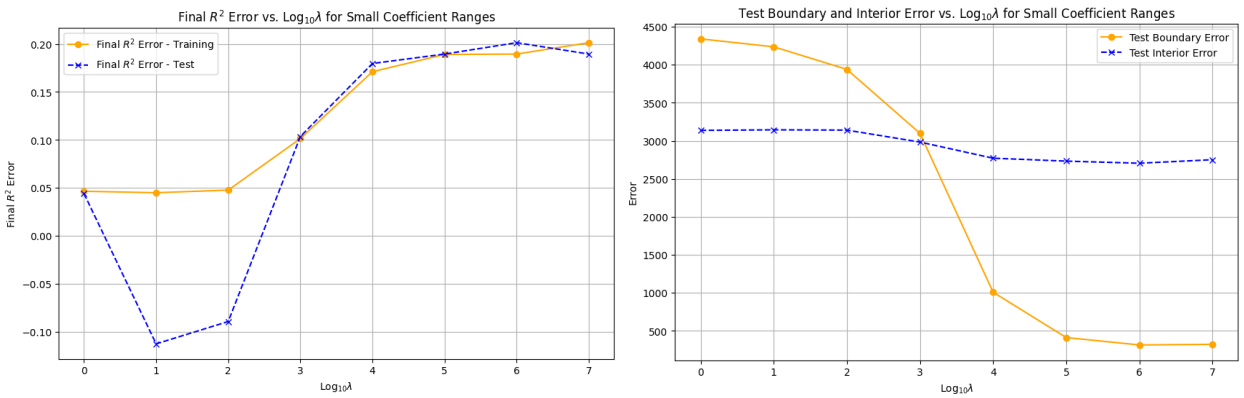

Figure 3.13: The graph on the left shows the R^2 error from the tests varying the boundary weighting in the small magnitude data regime. The graph on the right shows the boundary and the interior errors compared against each other. Unlike before in the large magnitude data regime, the R^2 error consistently improves and the interior error is relatively constant even past the point where the errors become comparable. The lack of change in the interior error shows the network approximation for the differential equation is not improved upon, but the R^2 error increases as the boundary gets closer and the network approximation becomes closer the solution in magnitude. There is a significant gap in the R^2 value between the test and training sets showing that under the dying ReLU problem training does not necessarily generalize well to the test set.

the network is not symmetrical and can leverage its nonlinearity to approximate any function and learn new features. This randomness also allows for better exploration of the space to find the optimal parameterize and converge faster. However, this also requires careful selection of the initialization procedure, as the weight initialization controls the variability of how well the space will be explored. Classical computation methods are more deterministic in their results, but it is often not entirely clear how to choose the initialization the best way to leverage it. Here, weight initialization was varied to find the best-performing initialization for a network with coefficient ranges of $[-100, 100]$, 256 discretization points between $[0, 1]$ for the pyramid-like network with eight hidden layers with Leaky ReLU activation functions for each layer. The model was run for 250 epochs for each initialization at a learning rate 0.0005 and a batch size of 128. Table 3.7 shows the outcome of the test, and overall, the initialization strategy did not have a significant effect on the outcome. The five initialization procedures, Xavier, Kaiming, Normal, Orthogonal, and Sparse initialization, all had relatively similar final values of R^2 in the training and test set, along with the final value of the MSE. The network performed well, and the best-performing model was the Normal initialization by a percent better than its peers.

Comparing the results for the well-behaved case with the range of coefficient sizes in the Fourier series is $[-100, 100]$ to the case to the same setup, but a coefficient range from [-1,1] shows a drastic decrease in the overall ability of the network to find a solution like expected. In this case, the effect of initialization is less pronounced when there is equal weighting between the boundary and interior terms, as the network generally suffers from a vanishing gradient. The network plateaus early regardless of the initialization strategy, indicating that varying the initialization, although important, is not the primary driver to fixing the vanishing gradient problem.

However, the last case considers the same hyperparameters as the model above except now

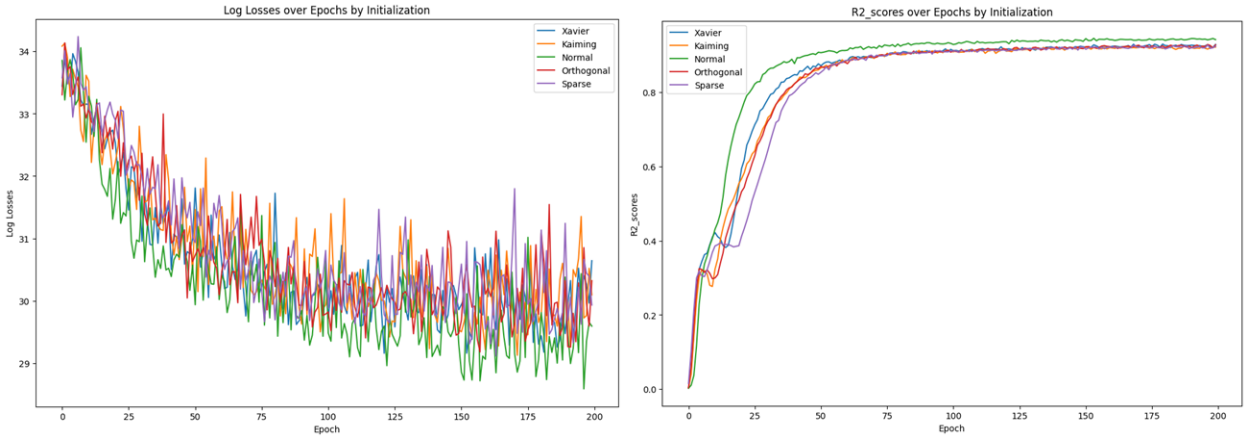

Figure 3.14: The two graphs measure the log loss vs. epochs and R^2 vs. epochs for coefficient ranges of $[-100, 100]$ showing the initialization is less important when the network can approximate well.

Initialization	**MSE on Test Set**		**Final R^2 Error**	
	Initial	**Final**	**Training**	**Test**
Xavier	362 020.3	14.141 42	0.958 81	0.958 07
Kaiming	411 905.3	16.090 05	0.952 01	0.952 19
Normal	261 462.5	10.213 38	0.969 99	0.969 68
Orthogonal	369 743.4	14.443 10	0.956 91	0.957 18
Sparse	382 924.7	14.957 99	0.955 41	0.955 59

Table 3.7: The impact of five different initialization schemes, Xavier, Kaiming, Normal, Orthogonal, and Sparse, was relatively equal in the extensive magnitude data range built from coefficient sizes $[-100, 100]$ in the Fourier series. In this regime the network appears to be robust to the initialization scheme with small improvements for the optimal choice. The R^2 value was greatest for the Normal initialization scheme, with the other initialization schemes performing relatively similarly.

changing $\lambda = 1 \times 10^5$ instead of having equal weighting. With a greater emphasis on the boundary, the network can better find the correct magnitude of the solution. The R^2 value on the test set improved on average by 16.0%, excluding the results for the normal initialization as an outlier.

The improvement further provides evidence that the introducing weights helps mitigate the vanishing gradient problem. Clearly, it does not resolve the problem as the model is still unable to find a good approximation to the solution, but there is clear improvement. In this case, the best strategy was sparse initialization, which performed a few percent better than the Xavier, Orthogonal, and Kaiming initialization strategies, which performed relatively equally. The normal initialization performed much worse compared to the other initialization schemes and was relatively unable to function well when the vanishing gradient was present. This is surprising given how well it performed in the case where the coefficient range was $[-100, 100]$; seemingly, the Normal initialization does not scale as well with the magnitude of the data.

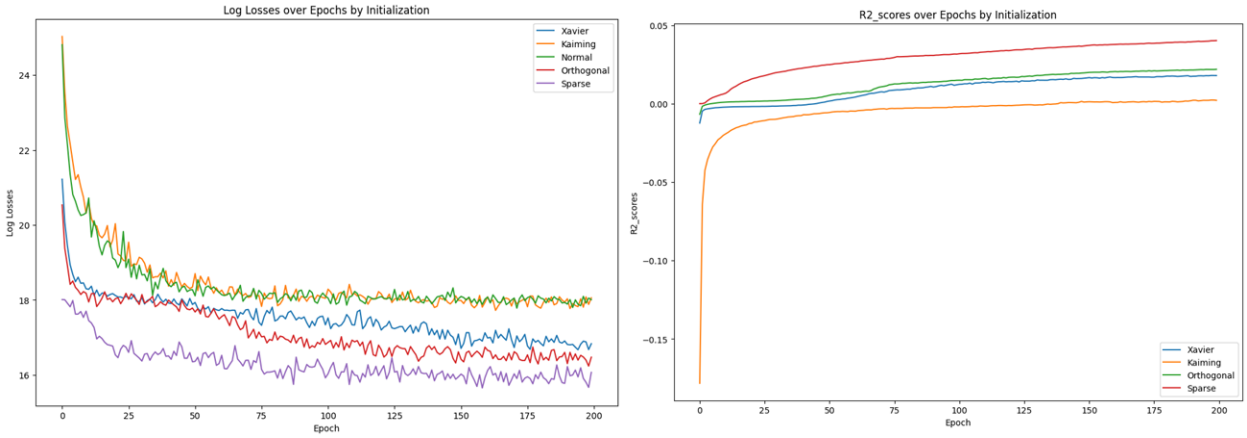

Figure 3.15: The two graphs measure the log loss vs. epochs and R^2 vs. epochs for coefficient ranges of $[-1, 1]$ with an equal weighting between the boundary and interior $\lambda = 1$. This shows that that the initialization is less influential when the problem exhibits a vanishing gradient.

Initialization	**MSE on Test Set**		**Final R^2 Error**	
	Initial	**Final**	**Training**	**Test**
Xavier	3.235 28	3.140 72	0.0290	0.028 41
Kaiming	3.300 14	3.203 68	0.009 98	0.008 95
Normal	6.387 89	6.201 19	−0.938 55	−0.917 80
Orthogonal	3.281 62	3.185 71	0.0164	0.014 63
Sparse	3.311 17	3.214 39	0.006 24	0.005 69

Table 3.8: Unlike the results previously displayed in Table 3.7, the initialization method had a large impact on the performance of the network even under the same hyperparameter conditions as Table 3.7. The vanishing gradient problem is evident in the near zero R^2 value for each of the initializations as the networks plateau early. However, the Normal initialization that performed the best in previously performs significantly worse here, demonstrating the importance of choosing the proper initialization.

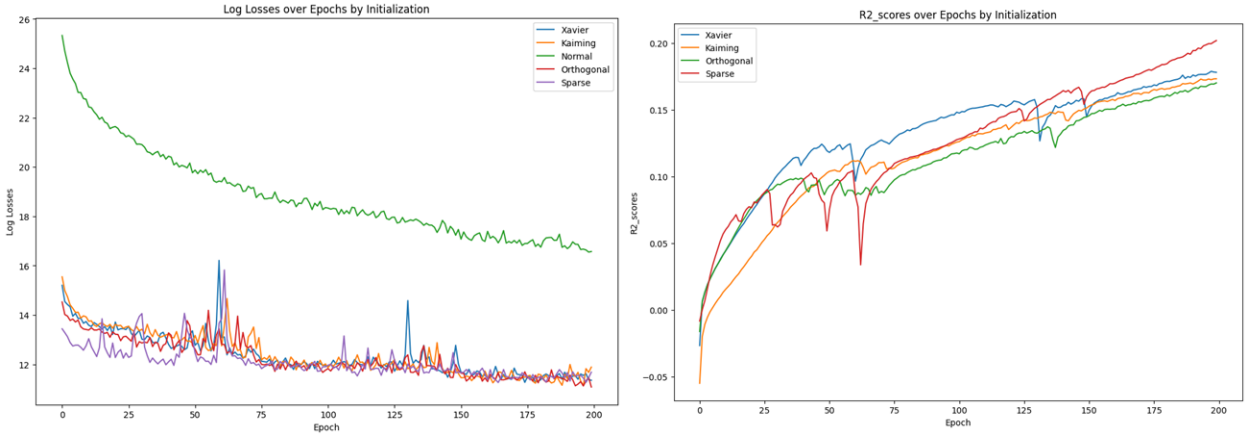

Figure 3.16: The two graphs measure the log loss vs. epochs and R^2 vs. epochs for coefficient ranges of $[-1, 1]$ with an weighting on the boundary term in the loss of $\lambda = 1 \times 10^5$. This shows that that the initialization is is influential and leads to large accuracy increases for less behaved solutions with weighting.

Initialization	**MSE on Test Set**		**Final R^2 Error**	
	Initial	**Final**	**Training**	**Test**
Xavier	2.750 06	2.669 68	0.179	0.172 98
Kaiming	2.778 54	2.697 33	0.173	0.164 46
Normal	5.165 42	5.014 44	−0.574 41	−0.554 26
Orthogonal	2.783 30	2.701 95	0.170	0.162 97
Sparse	2.667 57	2.589 60	0.205	0.197 61

Table 3.9: Proper initialization combined with increasing the boundary weighting in the loss function to $\lambda = 1 \times 10^5$ demonstrated a remarkable improvement in the overall performance of the network as illustrated by the increase R^2 values. The emphasis on putting more weight on the boundary helped slightly overcome the problem of vanishing gradients. Notably, the initializations Xavier, Kaiming, Orthogonal, and Sparse performed similarly once again, except the Sparse initialization developed a slight edge over the rest. However, the Normal Initialization scheme continues to perform relatively poorly and sees improvement from the previous test in Table 3.8 with the same hyperparameters in training, but is still an overall ineffective approximation to the solution.

3.2.8 Regularization Terms

Another way to regulate the loss function for improvement is to add regularization terms. Often neural networks will add additional terms to the network to get desired outputs. It was empirically observed that the network tended to exhibit frequent oscillations when building the RHS. These oscillations in matching the approximate RHS of the network $\hat{f} = A\hat{u}$ occurred when the network was unable to reconcile the size of the data. The examples shown in Figure 3.17 demonstrate a case where the network solution had high-frequency oscillations in matching the built network RHS to the actual RHS function. This network was performed for a coefficient range of $[-100, 100]$ with equal weighting at the boundary and six hidden layers with 1024 neurons per layer and Xavier initialization. The solution performs relatively well with a R^2 of 92.3% on the test set, but the oscillations lead to a solution that also has high-frequency oscillations that disrupt the performance.

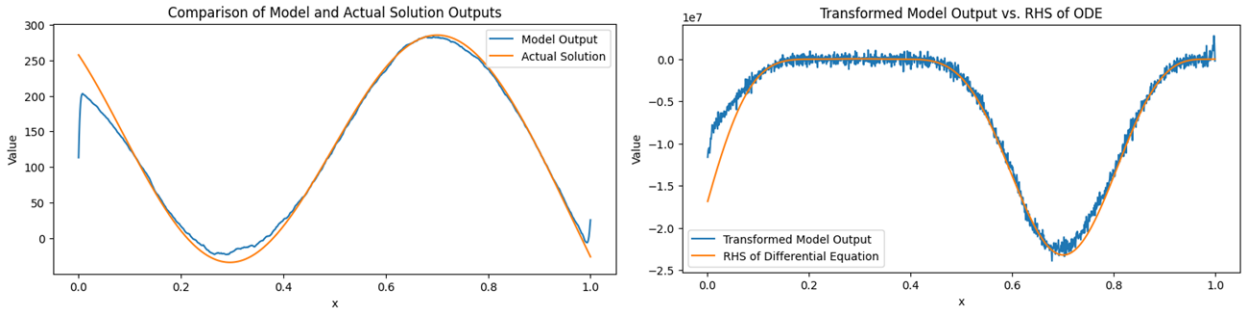

Figure 3.17: The graph compares the network approximation of the solution on the left and the RHS of the equation on the right, each built from 1024 discretization points. The conditions for the network were 1024 neurons per layer, and a flat architecture with ReLU activation functions running for 250 epochs. The network shows that even in the regime of large magnitude data, the network experienced high frequency oscillations that made the network perform worse than it ideally could have.

These oscillations plague solutions, especially in the regime of small-magnitude data. Shown in Figure 3.18, the network was run with 8 hidden layers, 2056 neurons per layer, a combination of Leaky ReLU activation functions and Xavier initialization. For this network, the approximate solution matches the relative boundary of the solution successfully; however, the oscillations in building the approximated RHS are a couple of orders of magnitude larger than the value for the actual RHS. Despite the network trying to minimize the difference in the RHS, it failed to minimize the discrepancy significantly and still generated a relatively accurate solution. Minimizing the difference between the approximated network RHS and actual RHS does not directly correspond to finding an accurate approximate solution, as shown before, except this time it worked out in the network's favor. The network was able to have a final MSE on the test set of 1.15×10^{-5}. However, without these oscillations the network would most likely have a smoother quality solution and have a lower MSE. This case exemplifies that the network can find accurate solutions in the low-magnitude regime; it is just not consistently able to do so.

The regularization term: $L_{\text{oscillation}} = \frac{1}{T}\sum_{\in T} ||\hat{f}||^2$ was introduced to penalize large oscillations from the mean of the center of the network. The term works to regulate the network to attempt to minimize the magnitude of $\hat{f}$. $\hat{f} = Au$ contains derivatives from the form of the differential equation, so decreasing these derivatives hopefully would smooth out the function, leading to

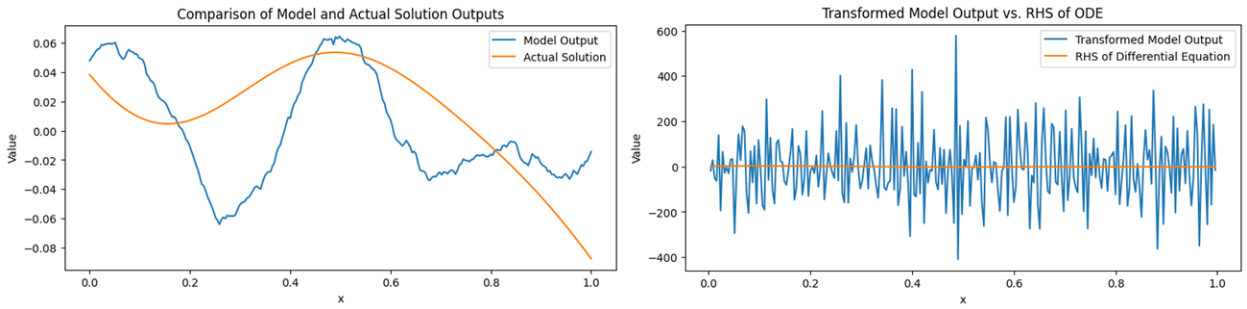

Figure 3.18: The network found an accurate approximate solution in the low-magnitude regime. However, as seen in 3.11, the solutions found in this regime, even when accurate, often have large oscillations that make the approximated solution nonsmooth

better results. Since a majority of the empirical observations of largely oscillating approximated RHS come from when the network fails to approximate in the low magnitude data regime or fails to approximate in the high magnitude data regime but past the computational capacity of the network, then the goal was to add this term and improve upon these two situations.

Adding a regularization term helped improve the base state with initially equal weighting. Shown in the Figures 3.19 and 3.20 below, in some cases, adding the regularization term led to smooth approximations of the solution even past the previous computational capacity of the network. However, empirical testing shows that there is a much larger cost to modifying the weighting on the regularization. The error often increases and the network is not as robust to changes in the weighting. There is a diminishing benefit in increasing the weighting, but the relationship is complicated and depends on many hyperparameters such as the weighting on the boundary term λ, the magnitude of the data, the discretization size, the amount of data, and the architecture. Adding the regularization term complicated an already difficult challenge of choosing the ideal hyperparameters. The results obtained were inconsistent and had large variations between trials of the same network.

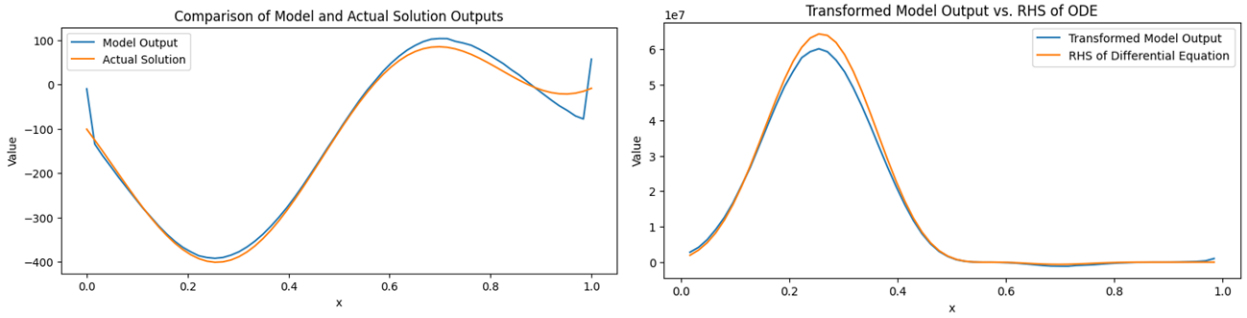

Figure 3.19: The network with the additional regularization term was able to better handle coefficient ranges $[-10, 10]$. The ideal weighting on the regularization term is roughly equal to the interior size. This network was run with eight hidden layers, 1026 neurons per layer, a combination of Leaky ReLU activation functions, and Xavier initialization.

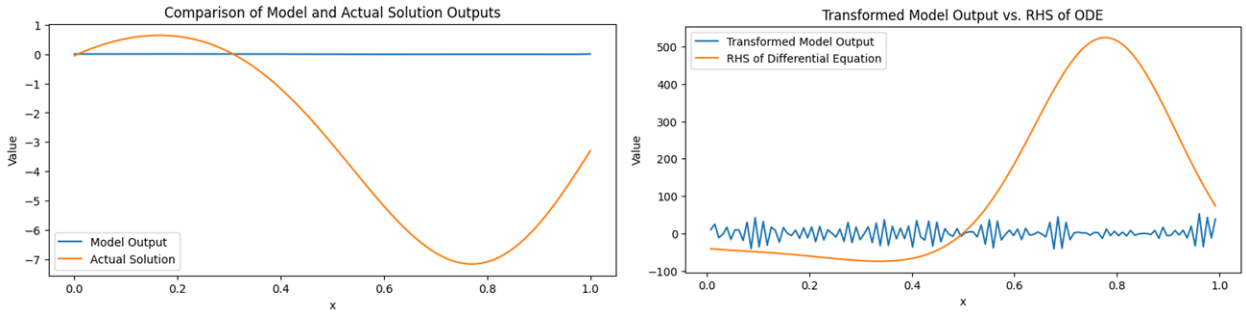

Figure 3.20: The network with the regularization term still fails to produce consistent, accurate approximations, as lower magnitude coefficient ranges like $[-1, 1]$ here. With equal weighting between all three terms in the loss function, the network solution still had high-frequency persistent oscillations. The network was still unable to adequately update its weights, running into the vanishing gradient problem

Chapter 4

Conclusion

4.1 Discussion of Results

The study was successful in creating a nonlinear differential equation solver regardless of RHS f and the Dirichlet boundary conditions from a feedforward fully connected unsupervised network applied to solving the specific case of the second-order nonlinear Ginzburg-Landau equations without specific RHS or boundary condition constraints. As shown for at least the case when the data is large in magnitude, the neural network was able to accurately approximate the solution for a variety of a large variety of varying RHS and boundary conditions. Furthermore, the versatility of the proposed approach to be applied to any differential equation of the form $Au = f$ allows for a large variety of problem applications and a more general solution outcome than previous methods for specific RHS and boundary conditions. This success lends hope that the network results could be further improved to have a more accurate solver that could be widely adopted for industrial applications.

To understand the impact on the network's ability to solve differential equations, the data was systematically varied along four main dimensions:

1. **Complexity of the Solution**: The network was able to approximate simple functions like $u = sin(x)$ incredibly quickly within a few epochs and accurately to mean square error of 7.31×10^9. Therefore greater complexity was added to the solution through by building the RHS f from data containing the first 10 modes of the discrete Fourier Series. This greatly enhanced the complexity of the data and the higher-order terms mimic oscillations in the real world from noise. Adding complexity to the data showed the network depending on the parameters of the data. All the tests going forward had this complexity and the ability to represent solutions as a Fourier Series represents a wide class of solutions the network can find.

2. **Amount of Data**: By randomly generating different amounts of amounts of functions to feed to the network, it was found that the network performed best given a few thousand functions as input data. The accuracy increased the more data was fed to the network, but the network was able to perform reasonably well on with less data.

3. **Mesh Size and Dimensionality of Data**: Adjusting the amount of points the interval [0,1] was uniformly cut simulated having different amounts of precision of the input data and solution. The network was able to perform well at smaller mesh sizes, but larger mesh sizes led to a more accurate approximation. However, the network does have a limit on the dimensionality that can be input for a given architecture. In accordance with the curse of dimensionality, the number of computations necessary seems to scale with the data. The exact relationship between how the dimensions of the data scale with the necessary number of layers and neurons per layer to find an accurate solution is unknown.

4. **Magnitude of Input Data**: The magnitude of the input data was altered by modifying the range of the distribution the coefficients in the Fourier Series randomly chose from. The network demonstrated an ability to accurately predict solutions in the large data scale that was robust to alterations in the architecture and other hyperparameters. However, challenges arose with lower magnitude data. The network encountered the vanishing gradient problem where the parameters in the network would stop updating and the progress would plateau.

To attempt to resolve the issues, several adjustments to the architecture, training procedure, hyperparameters, and loss function were made:

1. **Batch Normalization**: Batch normalization applied between hidden layers worked to stabilize the network. After introducing batch normalization the network was less sensitive to hyperparameters like the activation function applied to the layers, learning rate, and batch size. The network became more robust to changes in magnitude and the selection of hyperparameters.

2. **Boundary Weight in Loss Function**: Modifying the loss function by a large positive constant allowed for greater emphasis on matching the boundary conditions of the network. The emphasis on the boundary in the loss should take away importance on finding the solution to the differential equation on the interior; however, increasing the weight on the boundary term consistently improved the accuracy of the network on the interior and the boundary until reaching a critical point. After the critical point the network performs worse with increasing the boundary weighting. The quality of the approximations were fairly smooth generally but prone to oscillations. With the boundary weighting, the network solution was able to reduce the error and some of the oscillations, especially coupled with an appropriate initialization scheme.

3. **Initialization Scheme**: The initialization scheme for choosing the initial distribution of the weights in biases in the layers was shown to not significantly impact performance in the high magnitude case or the low magnitude case, with equal weighting on the boundary. However, when the network is at a low magnitude and active, which can be spurred on by placing weight on the boundary term in the loss function, then the initialization scheme did influence the convergence behavior and accuracy of the network. The start point of the network is important for the intermediate cases where the network is active but does not perform as well.

4. **Regularization Terms**: Incorporating regularization terms into the loss function offered improvements to the network when applied carefully. However, introducing weighting on the regularization term significantly complicated the finding optimal hyperparameters for the network empirically. The sensitivity and optimization of different types of regularization terms warrant further investigation into their dynamics.

Across all experiments, the network exhibited a remarkable ability to generalize from the training set to the test set. This suggests that the model successfully is able to approximate a mapping of the inverse differential operator to find the solution for a variety of RHS and boundary conditions. The key challenge of the network is that the accuracy of the results greatly depends on the nature of the input data. This intricate relationship makes finding the optimal network hyperparameters and architecture difficult. The experiments highlight the characteristics of the network in different regimes. In the case of low magnitude data, the experiments elucidate potential strategies to mitigate the effects of the vanishing gradient and find the best network approximation to the solution. It is an open question to resolve these issues.

4.2 Future work

Using neural networks to solve differential equations is area of research that has exploded in interest in recent years. There are plenty of intriguing potential modifications that could be made to the networks to examine characteristics of the network and improve it.

One future area of investigation would be verifying the methods ability to solve differential equations of form $Au = f$. Theoretically, it should be able to solve these problems for an unsupervised network, but verifying and classifying the range of equations that can be solved would further develop the theory behind the solver. The project here investigated solving primarily boundary value problems with Dirichlet boundary conditions, but other types of boundary conditions like Neumann or initial value problems could just as easily be implemented with an additional term in the loss function.

Expanding on this, one potential area of focus would be expanding on the applications of the method to problems where terms in the differential equation are changing. This would increase the variety of applications with the method. One type of problem like this would be Strum-Louiville Eigenvalue problem. Solving eigenvalues problems appears frequently in industrial and theoretical applications like quantum mechanics and heat conduction. The Strum-Louiville boundary value problem describes a general class of second-order differential equations of the form:

$$\begin{aligned} \frac{d}{dx}\left(p(x)\frac{dy}{dx}\right) + q(x)y &= -\lambda w(x)y \text{ for } x \in (a,b) \\ \alpha_1 y(a) + \alpha_2 y'(a) &= 0 \\ \beta_1 y(b) + \beta_2 y'(b) &= 0 \end{aligned} \tag{4.1}$$

where $p(x)$, $q(x)$, and $w(x)$ are given functions, y is the solution, λ is a parameter, and α_1 and α_2 are constants that are both not 0 simultaneously, and β_1 and β_2 are constants that are both not 0 simultaneously. λ is often an eigenvalue which has physical meaning for representing the harmonics of the solution. The solution for the Strum-Louiville is not known in all cases or guaranteed to be unique. There was a long attempt to solve the Strum-Louiville eigenvalue problem by the author; where uniqueness of the solution was introduced by the network finding the number of oscillations in the solution on an interval and assigning that as the eigenvalue. However, the network failed to find convincing solution that matched the expected magnitude of the data. In part, this could be because the landscape is complicated, but this could also be due to more analysis on the vanishing gradient to be performed.

A potential future application would be to apply the method to partial differential equations. By describing a partial differential equation that varies in time and space with two separate update mechanisms this method of solving ordinary differential equations could be applied to solve an ordinary differential equation that varies in space and time separately. The implications for whether this is realizable need further study and analysis.

Another potential avenue of research could be studying further the properties of the network like the implications of adding more regression terms, varying the means of taking a derivative, or varying the different ways to build the solution like polynomial, Fourier series, or others. Moreover empirical comparisons for analyzing the impact of different means of measuring the error in the loss in H^1, L^1 to see if it provides any benefits. However, the results from this testing were inconsistent and did not match the existing theory, therefore this exists as a promising method of

varying the method. Along this same idea, different network structures could be use like using convolutions networks instead of feedforward fully connected neural networks. Convolutional networks are especially good at finding local patterns and potentially convolutional neural networks would be good at matching the local characteristics of the inverse differential operator landscape. This could possibly be a solution to the vanishing gradients.

Lastly, modifying the architecture and hyperparameters that go into making the network to deal with low magnitude data is an open question and being able to find a solution or correctly optimizing the parameters to solve this problem would be a future avenue of research.

Chapter 5

Appendix A

5.1 Traditional Computation of Differential Equations

Understanding traditional methods for computing differential equations is instrumental when employing neural networks for similar purposes. These methods, although in computer science are thought of as a blackbox, are also methods to approximate the map $A^{-1}f = F(f)$ where F maps f into the solution u. There are methods to solve differential equations that both use a mesh and are mesh-free, where a mesh is a specifically defined discretization of the domain. Typical ordinary differential equation solvers use a mesh and rely on discretizing the problem from continuous functions to a discrete representation that is solvable numerically through algorithms; however, this transformation also introduces a error associated with solving the equations. This transformation can be chosen a number of different ways, but the general ordinary differential equation solver can be broken down into the steps:

1. **Mesh Creation**: Split the domain into parts. This subdivision is often uniform but it does not have to be and can be defined depending on the method.
2. **Discretization**: Now we convert the governing equations from a continuous form to a discrete form so that we can evaluate the problem numerically using the mesh.
3. **Solution**: Use a method of stepping or traversing from different points to solve the system of discrete equations.
4. **Post-processing**: After finding the solution, we use the solution to extract desired quantities and create visualizations.

This general format is widely applicable and has incredible variation in the choice of mesh, descretization, derivative approximation, and more.

5.1.1 Finite Difference Method (FDM)

In the finite difference method, the idea is to approximate continuous functions by a series of discrete grid points where at a specific grid point we use information from neighboring grid points to compute necessary derivatives. Derivatives are necessary to compute the differential equation and the stepping mechanism. The grid chosen for FDM is most of the time a rectangular grid with points arranged subdivided uniformly but can be defined on non-uniform or non-rectangular domains. These derivative approximations can be seen as discrete versions of the definition of the derivative which become equal to the continuous definition of the derivative in the case where $\lim_{h\to 0}$. We define finite difference approximation of the derivative for the first-order and second-order derivative, but higher order derivatives with differing points taken can be taken.

$$u'(x) \approx \frac{u(x+h) - u(x)}{h} \quad \textit{(First-order Forward Difference)} \tag{5.1}$$

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \quad \textit{(Second-order Central Difference)} \tag{5.2}$$

where h is the distance between neighboring grid points. For a uniform domain h remains fixed. In FDM, we are approximating the derivatives in the differential equation with a system of finite

difference approximations which are solved through a direct or iterative method. The FDM was derived using Taylor series approximations and we can define the accuracy of the approximation through the Taylor series expansion. Examining the first-order forward difference for a smooth function $u(x)$ the Taylor series expansion is given by:

$$u(x+h) \approx u(x) + u'(x)h + \frac{u''(x)}{2}h^2 \tag{5.3}$$

Then the error for Equation 5.1 is found through plugging in Equation 5.3

$$\begin{aligned} \left|\frac{u(x+h)-u(x)}{h} - u'(x)\right| &\approx \left|\frac{u(x)+u'(x)h+\frac{u''(x)}{2}h^2-u(x)}{h} - u'(x)\right| \\ &\approx \left|\frac{u''(x)}{2}h^2\right| = O(h^2) \end{aligned} \tag{5.4}$$

The computational accuracy of equation 5.1 goes as $O(h^2)$ and higher order derivatives increase there order by one for every further derivative and the coefficient of the approximation depending on the scheme and the number of points taken for the derivative approximation.

5.1.2 The Euler Method

The Euler method works in iterative steps, by approximating the solution $u(x)$ by its Taylor polynomial of around the data points.

Step 1: Starting from the initial condition we have: $u(0) = a$

Step 2: We start the iterative process:

- Pick N (the number of iterations we want to perform) and set: $x_0 := a$.
- For every $1 \leq i \leq N$ define: Δx_i to be a small distance we pick at each iteration.
- For every $1 \leq i \leq N$ Define: $x_i = x_{i-1} + \Delta x_i$
- Approximate the solution u at the point: x_i by using the Taylor expansion of u around $x_{i-1} + \Delta x_i$:

  $u(x_i) \approx u(x_{i-1}) + u'(x_{i-1})\Delta x_i + ... + \frac{u^{(n)}(x_{i-1})}{n!}\Delta x_i^n$.

  The degree of Taylor polynomial matches the order of the ODE. The derivatives $u'(x_{i-1})$ can be computed by using the given ODE.

Step 3: Once we complete the iterative process, we now have estimated the solution in N points. The larger N, the more precise the approximation is.

The computational accuracy of the Euler method is $O(N)$.

5.1.3 Finite Element Method (FEM)

The finite difference method follows a similar idea to FEM, but its often times more complicated to define and use but is often more powerful. FEM is a specific method in a class of Galerkin Projection Methods. The idea is that we can expand functions into a linear combination of their basis elements with coefficients. Then after expanding functions we solve differential equations in their weak form (integral form) with the product between the equation and the basis functions. The domain is specially discretized so that for a set of elements Ψ_i then they represent the space $\cup\Psi_i = \Psi$ and $\Psi_i \cap \Psi_j = \phi$ for $i \neq j$ where ϕ are piecewise continuously differentiable functions called basis functions. Then the basis functions can be approximated by low degree piecewise polynomials. This method therefore breaks complicated functions into simple polynomials which we solve in the weak form.

5.1.4 Shooting Method

The idea behind the Shooting method is that we are given BVPs and we can reformulate the problem into an initial value problems (IVP) by adding sufficient conditions to one of end of the IVP until the conditions are satisfied. This IVP can then be solved more easily using methods like finite difference and the solution to the IVP that creates a solution that matches the boundary value on the other side then obtains a solution to the BVP.

For example: If we consider the BVP given by:

$$Au(x) = f(x) \quad \text{with } u(0) = a \text{ and } u(1) = b \tag{5.5}$$

Then we can solve this problem by guessing choosing to keep $u(0) = a$ and guessing an appropriate value of c for the condition $u'(0) = c$ to reformulate the problem as an IVP problem.

$$Au(x) = f(x) \quad \text{with } u(0) = a \text{ and } u'(0) = c \tag{5.6}$$

Then numerically integrating the equation or solving using numerical methods we continue varying values of c systematically until we get a value that matches $u(1) = b$. At this point then solution to Equation 5.6 solves the BVP 5.5.

5.2 The Trial Solution Method for Differential Equations

5.2.1 Introduction

In 1997, I. E. Lagaris, A. Likas, and D. I. Fotiadis in their paper "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations" [12] laid out both a general framework

and specific technique for solving differential equations with neural networks. Before the paper, the attempts to solve differential equations with DNN were limited to trying to linear systems of algebraic equations. With the framework Lagaris et. al. formualted allowed for the implementation of DNNs to solve any type of diff eq and is still generally applicable and most modern DNN diff eq solvers are built off of this framework at least implicitly. However, the trial solution method proposed by Lagaris et al. in 1997 offers a unique approach to solving diff equation that has comparable efficiency while resulting in a smooth approximation of the analytical solution.

5.2.2 Hard Assignment: Trial Solution Method

This method is characterized by two key components: **the loss function** and the **form of the trial solution**, referred to as u_{approx}, which takes the form:

$$u_{\text{approx}}(x) = A(x) + F(x)N(w, x), \tag{5.7}$$

where:

- $A(x)$ is a prechosen function that inherently satisfies the boundary conditions of the differential equation.
- $F(x)$ is a function chosen such that it does not alter the boundary conditions, i.e., $F(x) = 0$ for x on the boundary of the domain.
- $N(w, x)$ represents the output of a feedforward neural network parameterized by the set w of weights and biases.

The neural network $N(w, x)$ now represents a solution to the differential equation with homogeneous boundary conditions that is scaled by the function $F(x)$. $N(w, x)$ still represents a solution to the differential equation with a closed analytical form given by $u_{\text{approx}}(x) = A(x)+F(x)N(w, x)$ which we could substitute $N(w, x)$ back into to find the actual solution. This is a smooth approximation the actual solution with infinite differentiability. We ensure the solution is a smooth approximation by taking only smooth activation functions like the sigmoid activation function, so we can smoothly take derivatives. However, the function $N(w, x)$ is easier to find and approximate than the actual solution to the differential equation, because the term $A(x)$ satisfies the boundary conditions and $F(x) = 0$ for x on the boundary of the domain so $N(w, x)$ does not effect the boundary. Therefore since the network adjusts the weights and biases of $N(w, x)$ to find the form which minimizes the error. Then the problem is turned from a constrained optimization problem to an unconstrained optimization problem where the network does not have to worry about the boundaries.

5.2.3 Motivation

This same idea and form of the trial solution is present in classical diff equation techniques for example in solving the one-dimensional heat equation with non-homogeneous boundary conditions:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

$$u(0,t) = u_0,$$
$$u(L,t) = u_L.$$

$$u(x,0) = \phi(x).$$

where $u(x,t)$ represents the temperature at position x and time t, and α is the thermal diffusivity of the material. Now using the superposition principle we can decompose the solution $u(x,t)$ is decomposed into two parts:

$$u(x,t) = u_{ss}(x) + u_{tr}(x,t).$$

$u_{ss}(x)$: Steady-state solution, capturing the long-term behavior of the solution with homogeneous boundary conditions. $u_{tr}(x,t)$: Transient solution, capturing time-dependent behavior of the solution with non-homogeneous boundary conditions.

For the steady state solution ($\frac{\partial u}{\partial t} = 0$), the problem reduces to:

$$\frac{\partial^2 u_{ss}}{\partial x^2} = 0$$

which after solving this differential equation leads to a solution of the form

$$u_{ss}(x) = (u_L - u_0)\frac{x}{L} + u_0.$$

Now we can find the equation for the transient solution by substituting the superposition principle decomposition into the original PDE:

$$\frac{\partial u_{tr}}{\partial t} = \frac{\partial^2 u_{tr}}{\partial x^2}$$

$$u_{tr}(0,t) = 0,$$
$$u_{tr}(L,t) = 0.$$

$$u_{tr}(x,0) = \phi(x) - u_{ss}(x).$$

where using the form of the steady state solution you could find the transient solution after solving the differential equation.

Both the trial solution method and the superposition principle method both have similar forms of the decomposition where the term $A(x)$ similar to $u_{tr}(x,t)$ is the term that satisfies the boundary conditions. Additionally the term in the trial solution $N(w,x)$ is the neural network output, and $F(x)$ is a function ensuring that the boundary conditions are not affected by N, which is similar to the term $u_{ss}(x)$ which is similarly not effected by the boundary conditions and an easier problem to solve than then the exact form of the solution but given $u_{tr}(x,t)$ or $A(x)$ could be used to recreate the full solution. However, in the case of the trial solution method it is not as important that we find the exact form of the differential equation that decomposition into $A(x) + F(x)N(w,x)$ creates but rather we care that we solve the differential equation once we find $N(w,x)$.

5.2.4 The Loss Function

The loss function is central to training the neural network. It is defined to minimize the residual of the differential equation over the domain.

Definition: Consider the general ODE given in (1), and let w be the set of weights of the DNN $N(x)$ mentioned in definition (3). We define the following loss function:

$L(w) = \int_0^1 (Au_{approx}(x, w) - f(x))^2 dx$

This definition of the loss function follows rather naturally, since minimizing $L(w)$ also minimizes the integrand in this case (follows from the non-negativity of the integrand, and hence u_{approx} will be, in fact, an approximation of the analytical solution u.

Note: Derivatives are taken using built-in autodifferentiation functions like the Autograd package of PyTorch which is able to implement backpropogation using the chain rule in the form of a computational graph to calculate the partial derivatives of each term and build the necessary derivatives.

The following example best illustrates how the Trial Solution method can be used to solve ODEs.

5.3 An Example of using The Trial Solution Method

Let's consider the following ODE:

$u'(x) + \frac{1}{5}u(x) = e^{-\frac{x}{5}} cos(x) \qquad\qquad x \in [0, 2]$

$u(0) = 0$

By using the method of integrating factors, we can obtain the analytical solution: $u = e^{-\frac{x}{5}} sinx$.

Now we will use the trial solution method to find an approximation u_{approx} to the analytical solution u.

Since the Trial function needs to satisfy the same initial conditions as the analytic function, then the trial function $u_{approx}(x)$ must satisfy: $u_{approx}(0) = u(0) = 0$.

One choice of u_{approx} can be:

(1) $u_{approx}(x) = xN(w, x)$.

Here:

- $F(x) = x$ and $A(x) = 0\ \forall x \in [0, 2]$.
- $N(w, x)$ is a DNN with one hidden layer and 10 nodes, with weights $w = (w_1, ..., w_{10}, v_1, ..., v_{10})$, biases 0, and input x, described in the following way:

$N(w, x) = \sum_{i=1}^{10} v_i \sigma(x w_i)$. $\qquad\qquad$ $w = (w_1, ..., w_{10}, v_1, ..., v_{10})$

Where σ is the sigmoid function: $\sigma(y) = \frac{1}{1+e^{-y}}$.

We wish to minimize the Loss function:

$L(w) = \int_0^2 (u'_{approx}(x) + \frac{1}{5} u_{approx}(x) - e^{-\frac{x}{5}} cos(x))^2$

Keep in mind that integration is a continuous process, and therefore cannot be done by a computer unless we discretize the domain. We can do so by using Riemann sums.

We create a mesh of the domain $[0, 2]$ by taking a uniform partition (made out of 10 points for example):

$0 = x_1 < x_2 < ... < x_{10} = 2$. Then, by using the Riemann sum of the function in the integrand, we obtain:

(2) $L(w) \approx \gamma \sum_{j=1}^{10} (u'_{approx}(x_j) + \frac{1}{5} u_{approx}(x_j) - e^{-\frac{x_j}{5}} cos(x_j))^2$

Where $\gamma = \frac{2}{9}$ is the size of each interval in the partition.

Now minimizing (2) is the same as minimizing:

$\sum_{j=1}^{10} (u'_{approx}(x_j) + \frac{1}{5} u_{approx}(x_j) - e^{-\frac{x_j}{5}} cos(x_j))^2$.

Hence, we can minimize $L(w)$ by minimizing:

($\star$) $\qquad \sum_{j=1}^{10} (u'_{approx}(x_j) + \frac{1}{5} u_{approx}(x_j) - e^{-\frac{x_j}{5}} cos(x_j))^2$.

Now we proceed to computing ($\star$) explicitly:

By (1), we have: $u'_{approx}(x_j) = N(w, x_j) + x_j \frac{\partial N}{\partial x}(w, x_j)$

Hence:

$(\star) = \sum_{j=1}^{10} (x_j \frac{\partial N}{\partial x}(w, x_j) + (1 + \frac{x_j}{5}) N(w, x_j) - e^{-\frac{x_j}{5}} cos(x_j))^2$

Which after computing $\frac{\partial N}{\partial x}(w, x_j)$ and substituting the DNN:

$N(w,x) = \sum_{i=1}^{10} v_i \sigma(xw_i) = \sum_{i=1}^{10} \frac{v_i}{1+e^{-xw_i}}$

one obtains:

$(\star) = \sum_{j=1}^{10}(x_j \sum_{i=1}^{10} \frac{v_i w_i e^{-x_j w_i}}{(1+e^{-x_j w_i})^2} + (1+\frac{x_j}{5})(\sum_{i=1}^{10} \frac{v_i}{1+e^{-xw_i}}) - e^{-\frac{x_j}{5}} cos(x_j))^2$

This is a continuously differentiable function w.r.t $w_1, ..., w_{10}, v_1, ..., v_{10}$. Therefore we can minimize $(\star)$ by computing the derivative and finding the critical points.
By doing so, we find the parameters $w_1, .., w_1, v_1, .., v_{10}$ that minimize the loss function $L(w)$. The trial solution method does not specify a specific optimization algorithm to minimize the loss function and it is left as a free choice to the user.

Note: Derivatives are taken using built-in autodifferentiation functions like the Autograd package of PyTorch which is able to implement backpropagation using the chain rule in the form of a computational graph to calculate the partial derivatives of each term and build the necessary derivatives.

5.3.1 Results and Error Analysis for Euler vs. FDM vs. Trial Solution Method

Evaluating the network returns a closed analytic form of the trial function u_{approx} is obtained.

This problem was realized and solved using a neural network using the trial solution method. The implementation used the architecture of a neural network with one input node which we give x, one hidden layer with 50 neurons, and one output node which outputs $N(w, x)$. The network was evaluated on the domain of $[0, 2]$ where we take 100 points uniformly. Note the trial solution is a meshless method so we did not have to take a specific mesh or uniform partition of the domain it was done in this way to compare to classical methods. The network was evaluated for 10 epochs, which is relatively small amount of epochs, but the solution quickly converges using the gradient descent optimization algorithm.

To measure the accuracy of the network we use evaluate the error as the difference between the approximated solution through the network and the actual solution $\Delta u = u_{approx} - u_a$, where u_a is the analytic solution which after the method of using the method of integrating factors to the original problems gives $u_a(x) = e^{-\frac{x}{5}} \sin(x)$. From this error, the maximum deviation of the error and average of the error were taken over the uniform mesh from [0,2] to evaluate how well the network performs at worst and on average.

The network results in the trial function closely approximated the analytical solution, with an average difference of average diff of 6×10^{-5} and maximum deviation of 1×10^{-5} over the 10 epochs. The trial function approximation of the analytic solution practically overlaps the actual

solution showing a very strong matching, and only after finding the error do we see

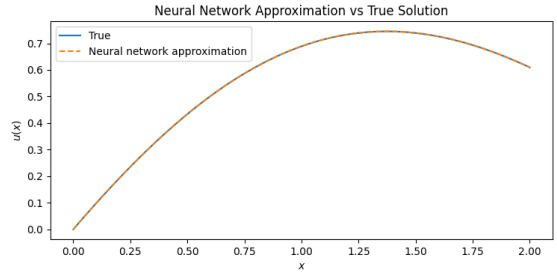

Figure 5.1: Trial Solution Approximation vs. Analytical Solution

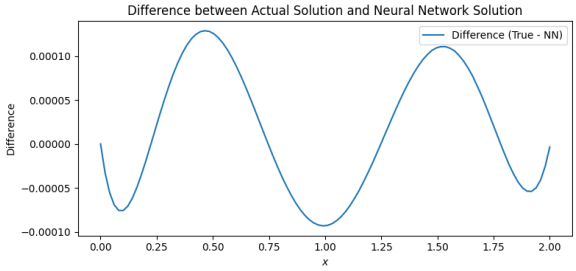

Figure 5.2: Trial Solution Error $\Delta u = u_{approx} - u_a$

Furthermore, by comparing the evaluation of this example problem using two different types of classical methods such as the Euler method and FDM an iterative and matrix method, then the trial solution can be directly compared to classical methods. Using $N = 100$ iterations of the Euler's method showed an average difference of 6.1×10^{-3} and a maximum deviation of 1.2×10^{-2} from the analytical solution. Euler's method has a linear computational complexity $O(N)$ making it computationally efficient for a small number of iterations. However, the precision of the Euler method is considerably lower than that of the trial solution method.

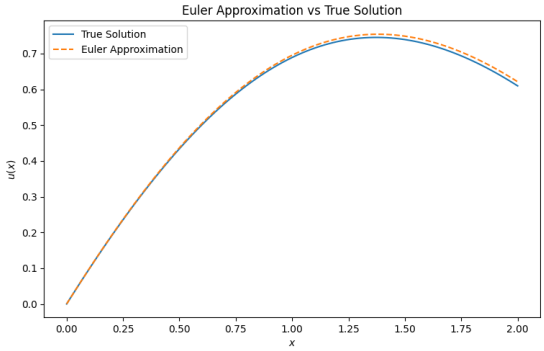

Figure 5.3: Euler Method Approximation vs. Analytical Solution

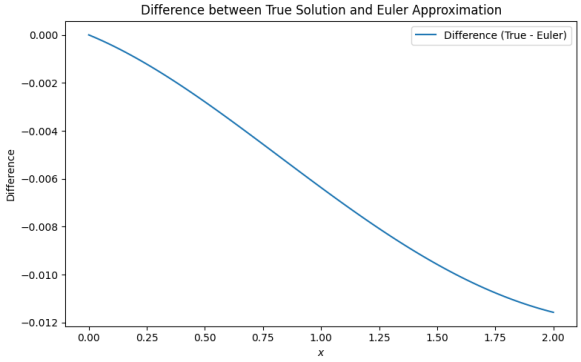

Figure 5.4: Euler Method Error $\Delta u = u_{Euler} - u_a$

Again using $N = 100$ points on the domain the FDM is evaluating by finding the inverse matrix A^{-1} that finds the solution $u = A^{-1}f$. The results were FDM exhibited an average difference of 1.2×10^{-2} and a maximum deviation of 2.1×10^{-2}. Similar to the Euler method, FDM has a computational complexity of $O(N)$. The FDM method performs similarly to Euler's method and has a considerably higher error than the trial solution method by three orders of magnitude. The three methods were performed over the same domain and the same mesh of uniform partition of 100 points of the domain [0,2]. The trial solution method is conjectured to have a slightly higher computational complexity then the classical methods; however, both Euler and FDM methods show greater errors compared to the neural network-based trial solution method where we have chosen a similar number of computations in all three cases. This difference becomes more pronounced as the complexity of the differential equation increases. The adaptability of neural networks to the characteristics of the differential equation at hand contributes to their superior performance.

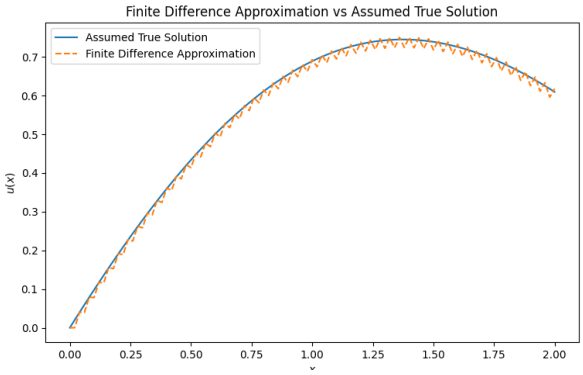

Figure 5.5: FDM Approximation vs. Analytical Solution

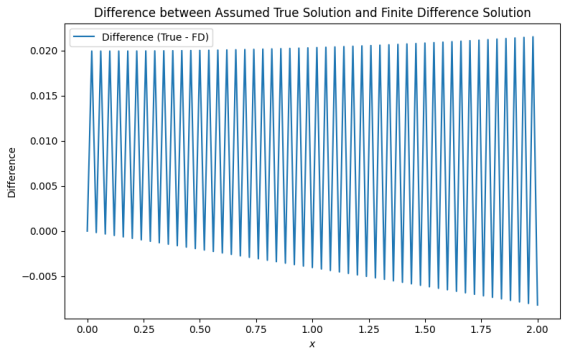

Figure 5.6: FDM Error $\Delta u = u_{Euler} - u_a$

5.3.2 Comparative Analysis with Finite Element Method (FEM)

In addition to simpler classical methods like Euler and FDM, the trial solution method was compared with the Finite Element Method (FEM) in the original Lagaris et. al. paper [12] to show the difference between the neural network methods and the state of the art classical methods. The paper compares the maximum deviation between FEM and the trial solution method on the training set and test set, for four problems much more complicated time-dependent nonlinear PDEs with homogeneous and nonhomogeneous boundary conditions to compare this method to more difficult higher dimensional problems. The results can be summarized in the table below from [12]:

	Trial Solution Method		FEM	
Problem No.	Training set	Test set	Training set	Test set
1	5×10^{-7}	5×10^{-7}	2×10^{-8}	1.5×10^{-5}
2	0.0015	0.0015	0.0002	0.0025
3	6×10^{-6}	6×10^{-6}	7×10^{-7}	4×10^{-5}
4	1.5×10^{-5}	1.5×10^{-5}	6×10^{-7}	4×10^{-5}

Table 5.1: Comparison of trial solution method and FEM maximum deviation for various problems between the approximate solution and the actual solution $\Delta u = u_{approx} - u_a$ and u_a is the analytic solution. The data and table are results from [12].

Observations and Implications

From the table, several key observations and implications can be drawn:

- **Training Set Performance:** FEM shows better performance on the training set in most cases, indicating high accuracy during the learning phase.
- **Test Set Performance:** On the test set, the trial solution method outperforms FEM. This suggests the DNN has a greater capacity to generalize its learning on the training set to the test set, which is great as the goal of training is the greatest accuracy on the test set and the training set is only a means to get there.

- **Consistency:** The trial solution method exhibits consistent performance between training and test sets, highlighting its stability and reliability across different data sets to the test set.
- **Implications:** These results imply that while FEM is highly accurate in well-defined, controlled settings (training set), its performance can degrade in more varied or uncontrolled environments (test set). In contrast, the trial solution method, though potentially less accurate on the training set, provides a more robust and generalizable solutions.

5.4 Conclusion

5.4.1 Advantages of the Trial Solution Method

- **Flexibility and Generality to Any Diff Eq :** This method is applicable to a wide range of differential equations, including single ODEs, PDEs, and to coupled systems of ODEs and PDEs. The general framework for the loss function can be applied to any differential equation paving the way for future methods for solving diff eqs with neural networks.
- **Inherent Satisfaction of Boundary Conditions:** The term $A(x)$ ensures that the trial solution inherently satisfies the boundary conditions by construction. This greatly simplifies the problem and transform the minimization problem from a constrained optimization problem to an unconstrained optimization problem which is much easier.
- **Differentiable Solutions:** Unlike classical methods like FEM and FDM which provide solutions that often are only discrete solutions and have limited differentiability, which greatly limits their applications and the problems they can be applied to. The trial solution method provides solutions in a differentiable, closed analytic form, which is advantageous for further analytical and numerical applications. Derivatives of the solution can be taken easily, are known to be smooth, and an exact analytical form allows for further analytic form analysis of different problems, like when you then have to compute gradients of the approximated solution to get desired quantities.
- **Efficiency of Computation and Implementation and Parallelizability:** The trial solution method extends to higher high-dimensional problems as well. Additionally, the nature of the method allows for implementation on parallel architectures, suitable for large-scale computational problems in higher dimensions. This is due to the architecture for this process often only requiring a single hidden layer and the exclusion of biases in the layers.

  DNN are both efficient in the adaptive time cost and implementation. DNNs work in an iterative process, where we take the output after one epoch and use plug it back into the network at the next epoch. During this process, unless the loss converges early, it will continue to decrease for longer epochs. Then if we only require a certain accuracy we can stop the DNN once that accuracy is hit in the training set during validation. However, for traditional solvers we cannot stop the solving scheme early, we have to wait for all values to compute in the full scheme until we get a result. We cannot further tune this result to run for longer if we need a higher accuracy like DNN.

DNNs are also incredibly fast in implementing found solutions. Once we train a DNN we have a series of weights and given a set of inputs we can compute an answer to a future problem quickly often in seconds just using matrix multiplication, which has gotten incredibly fast with today's computing power.

One example illustrating this benefit is say that you wanted to predict how major disturbances under the ocean or sea caused by shifting fault lines or earthquakes would generate tsunamis or waves. You could use the shallow water equations which are complicated hyperbolic PDEs that give the propagation of a disturbance in a fluid. From an earthquake in some location under the water you detect you want predict how high the waves would be from the water once it reached the coastline of a major city. This is a very complicated model especially in three dimensions. If a tsunami is coming towards a major city especially if the city is close to the source you will not have a lot of time to solve the problem and which the current numerical methods calculation would be way too long. However, if you had a DNN that solved the differential equation then all you would have to do is give the DNN the inputs and it could then perform matrix computation to solve the problem in seconds. Saving lives and giving people time to evacuate before its too late.

- **Strong Generalization from Training Set to Test Set:** Since the trial solution method is trying to match the error in the differential equation specifically rather than just minimizing the discrepancy between the training set and the solution. Then when the domain of the problem is extended and the approximate solution is evaluated on the test set, the trial solution method experiences a much smaller loss the classical methods.

- **Reduced Parameter Demand:** Requires a significantly lower number of model parameters compared to other techniques, leading to compact solution models with minimal memory requirements.

- **Meshless Nature:** The method does not rely on mesh generation, enhancing flexibility and reducing complexity of problem setup to problems with irregular domains like a star or handle data arriving at irregular intervals that are traditionally difficult or impossible for classical methods. This greatly expands the types and domains of problems that we can solve and the choices we can make when defining the problem.

5.4.2 Disadvantages of the Trial Solution Method

- **Convergence Issues:** Due to the recency of the method and DNNs in general, there is a lack of mathematical underpinning to DNNs and the theory. So far there is no proof of the convergence rate of the trial solution method. It is conjectured with numerical support that the method should have the same or better computational complexity as classical methods, but this is not proven. This means there is no guaranteed error reduction with increased network size or complexity, implying uncertainty in achieving higher accuracy through network expansion. This is a feature of all neural network methods.

- **Randomness in Initialization:** Randomness during network initialization can lead to variations in results. This may necessitate multiple runs for desired results and additional prob-

lems the randomness and initialization leads to like the exploding gradient problem when training. This is another feature of all neural network methods.

- **Dependence on Choice of $A(x)$ and $F(x)$:** The appropriate design of $A(x)$ and $F(x)$ is determined by the implementer and incredibly important to achieving desired results. Incorrect specifications can lead to inconsistent or inaccurate solutions. While there exist multiple methodologies to derive $A(x)$ and $F(x)$, often a more intuitive or "obvious" choice presents itself given the nature of the differential equation. The crux of the neural network approach remains the optimization of the parameters α according to the same general loss function above. However, if there were some sort of error or poor results generated it is unclear if the problem is due to suboptimal choices of $A(x)$ and $F(x)$, which would need to be changed by hand if ran again, a problem in the neural network hyperparameters, or some other additional problem that could effect results. Overall, making these neural networks hard to adjust and decipher when they have suboptimal results.

Bibliography

[1] The Ginzburg-Landau Equation 5.1 the Real Ginzburg-Landau Equation.

[2] Khalid I. Alkhatib, Ahmad I. Al-Aiad, Mothanna H. Almahmoud, and Omar N. Elayan. Credit Card Fraud Detection Based on Deep Neural Network Approach. In *2021 12th International Conference on Information and Communication Systems (ICICS)*, pages 153–156, 2021.

[3] Leonid Berlyand and Pierre Emmanuel Jabin. *Mathematics of Deep Learning: An Introduction*. de Gruyter, Germany, April 2023. Publisher Copyright: © 2023 Walter de Gruyter GmbH, Berlin/Boston. All rights reserved.

[4] Salvatore Cuomo, Vincenzo Schiano di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific Machine Learning through Physics-Informed Neural Networks: Where We Are and What's Next, 2022.

[5] Tim Dockhorn. A Discussion on Solving Partial Differential Equations using Neural Networks, 2019.

[6] Jian Gao. R-Squared (R2) – How much variation is explained? *Research Methods in Medicine & Health Sciences*.

[7] Jaysa Grafton. Solving Differential Equations With Deep Neural Networks (DNNs), 2022.

[8] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[9] Zheyuan Hu, Ameya D. Jagtap, George Em Karniadakis, and Kenji Kawaguchi. When Do Extended Physics-Informed Neural Networks (XPINNs) Improve Generalization? *SIAM Journal on Scientific Computing*, 44(5):A3158–A3182, September 2022.

[10] Xiaowei Jin, Shengze Cai, Hui Li, and George Em Karniadakis. NSFnets (Navier-Stokes Flow Nets): Physics-informed neural networks for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 426:109951, 2021.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[12] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial Neural Networks for Solving Ordinary and Partial Differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[13] Zaharaddeen Karami Lawal, Hayati Yassin, Daphne Teck Ching Lai, and Azam Che Idris. Physics-Informed Neural Network (PINN) Evolution and Beyond: A Systematic Literature Review and Bibliometric Analysis. *Big Data and Cognitive Computing*, 6(4), 2022.

[14] Lu Lu. Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics*, 28(5):1671–1706, June 2020.

[15] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal Differential Equations for Scientific Machine Learning, 2021.

[16] 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. *Journal of Computational Physics*, 378:686–707, 2019.

[17] Justin Sirignano and Konstantinos Spiliopoulos. DGM: A Deep Learning Algorithm for Solving Partial Differential Equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[18] Remco van der Meer, Cornelis W. Oosterlee, and Anastasia Borovykh. Optimally Weighted Loss Functions for Solving PDEs with Neural Networks. *Journal of Computational and Applied Mathematics*, 405:113887, 2022.

Jacob Frishman

237 Willow Avenue
Wayne, PA 19087
(215)-359-5001
jsf5388@psu.edu

Education

Physics and Honors Mathematics B.S.
Penn State University, University Park*, PA* — *Expected graduation May 2024*

Awards & Honors

- Schreyer's Honors College
- Evan Pugh Scholar Senior Award
- Sigma Pi Sigma Honors Society
- President's Sparks Award
- Dean's List Fall 2020- Fall 2023
- President's Freshman Award

Research Experience

Schreyer's Honors Thesis
Penn State University, University Park, PA — *May 2022 - Present*

- Implemented a new unsupervised machine learning technique with the capacity to approximate solutions to ordinary differential equations like the Ginzburg-Landau equation for any boundary condition and right-hand side
- Iteratively tested the network hyperparameters to show the network was able to consistently have 95% accuracy depending even on data with large oscillations
- Compiled acquired knowledge and results into Schreyer Honors College thesis "Solving Differential Equations with Deep Neural Networks"

Statistics Part-Time Research Assistant
Penn State University, University Park, PA — *Sept 2023 - Present*

- Wrote code for and simulated the Ising model with Markov Chain Monte Carlo (MCMC) methods to determine how the estimates of the Maximum Likelihood Estimate (MLE) and the Maximum Pseudo-likelihood Estimate (MPLE) converged.
- Implemented a stepping method to efficiently improve the ability for the MLE to approximate an answer where it was previously not possible.

Physics Research REU/Intern
Texas A&M University, College Station, TX — *May 2023 - Aug 2023*

- Synthesized and simulated multiple physical models to predict, analyze, and optimize precision control of ferromagnetic maghemite (γ-Fe_2O_3) 15 nm nanoparticles using magnetic fields generated from copper coils
- Conducted experimental validations with an impedance analyzer, to test the model's predictions and demonstrate the capability of controlled nanoparticle trapping.
- Synthesized and presented work at Texas A&M Summer Undergraduate Research Poster Session under the title "Modeling and Manipulating Ferromagnetic Microparticles Using Copper Coils"

Mathematics Part-Time Research Assistant
Penn State University, University Park, PA — *May 2022 - Aug 2022*

- Collaborated with published authors in Nature's Communications in Physics Journal to build intricate fluid simulations that model the collective behavior of bacterial colonies in viscoelastic liquid crystals (VLC).
- Validated mathematical models and experimental analysis utilizing FreeFem++, MATLAB, and Mathematica to simulate the collective motion of over 50 microswimmers in VLC and conducting data analysis to discern statistical patterns in their movement.

Programming Languages

- Python
- Mathematica
- R
- HTML/CSS
- MATLAB
- LaTeX

Extracurricular Activities and Leadership

- Physics Test Committee Chair for Science Olympiad Alumni at Penn State
- University Physics Teaching Assistant and Math Grader
- Professor Berlyand research group: PDE, homogenization theory, and machine learning
- Pulsar Search Collaboratory Mentor and Data Analyzer
- Operations Chair Board Member and D1 Esports Athlete, Penn State Esports Club
- Nittany AI Alliance and Nittany Data Labs club member

Courses

- Discrete Mathematics
- Deep Learning Algorithms
- Linear Algebra
- Quantum Mechanics
- Computational Physics
- Math Modeling
- Quantum Information and Qubit Implementation
- Real Analysis
- Introduction to Probability Theory
- Electricity and Magnetism
- Fluids and Thermal Physics
- Technical Communication
- Subatomic and Particle Physics
- Mathematical Statistics
- ODEs/PDEs
- Calculus I-III
- Theoretical Mechanics
- Experimental Physics
- Programming and Computation
- Electronics
- Game Theory