THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF MATHEMATICS

Solving Differential Equations With Deep Neural Networks (DNNs)

JAYSA L. GRAFTON
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Mathematics
with honors in Mathematics

Reviewed and approved* by the following:

Leonid Berlyand
Professor of Mathematics
Thesis Supervisor

Victoria Sadovskaya
Professor of Mathematics
Honors Adviser

*Electronic approvals are on file.

Abstract

Overall, the goal of this project is to make use of the machine learning algorithm of deep neural networks (DNNs) to solve differential equations. Specifically, this project aims to solve two different second-order differential equations: Poisson and Ginzburg-Landau equations. Results for the Poisson equation show an accurate solution can be acquired using a single layer network with no activation function due to the linearity of the equation. These results demonstrate that finding solutions to differential equations is possible through the use of deep neural networks. For the Ginzburg-Landau equation, two different loss functions are utilized with adjustments being made to account for boundary conditions and derivatives. Results indicate an accurate approximation for various mesh sizes (i.e., coarse versus fine mesh) and allow for the comparison of network architectures for each mesh size in order to determine the parameters necessary for an accurate solution.

Table of Contents

List of Figures iii

List of Tables iv

Acknowledgements v

1 Defining the Project 1
- 1.1 Introduction . . . 2
- 1.2 Preliminaries . . . 2
- 1.3 General Differential Equation . . . 3
- 1.4 Results from Literature . . . 3

2 Deep Neural Network (DNN) 7
- 2.1 Project Design for Deep Neural Network . . . 8
- 2.2 General Structural Components . . . 10
  - 2.2.1 Hidden Layers . . . 10
  - 2.2.2 Activation Function . . . 10
  - 2.2.3 Momentum . . . 11
  - 2.2.4 Learning Rate . . . 12
- 2.3 Motivation for Utilizing DNNs . . . 12

3 Differential Equations and Results 13
- 3.1 Training Set Generation . . . 14
- 3.2 Linear ODE . . . 15
- 3.3 Non-Linear ODE . . . 17
  - 3.3.1 Case 1: Original Loss Function . . . 17
  - 3.3.2 Case 2: Updated Loss Function . . . 19

4 Conclusions and Future Work 25
- 4.1 General Discussion of Results . . . 26
- 4.2 Future Work . . . 26

Bibliography 27

List of Figures

2.1 Diagram of DNN w/ 2 layer functions . . . 8
2.2 ReLU activation function . . . 10
2.3 GELU activation function . . . 11

3.1 Simple network with no hidden layers or activation function solution . . . 16
3.2 Simple network with no hidden layers or activation function error . . . 16
3.3 Network with ReLU and three hidden layers . . . 16
3.4 Result for one DNN and one element of T . . . 18
3.5 Network with three hidden layers . . . 18
3.6 Solution and error with data size 16 . . . 20
3.7 Solution and error with data size 32 . . . 21
3.8 Solution and error with data size 64 . . . 21
3.9 Solution and error with data size 128 . . . 21
3.10 Solution and error with data size 256 . . . 22
3.11 Solution and error with data size 1024 . . . 22
3.12 Solution and error with data size 16 as input and 256 as output . . . 23
3.13 Solution and error with data size 32 as input and 256 as output . . . 23
3.14 Solution and error with data size 256 for 10000 epochs . . . 24
3.15 Solution and error with data size 1024 for 10000 epochs . . . 24

List of Tables

3.1 Nonlinear ODE original results 17
3.2 Nonlinear ODE results w/ different first hidden layer size 18
3.3 Nonlinear ODE results w/ different penalty coefficients 19
3.4 Nonlinear ODE results w/ different data sizes 20
3.5 Nonlinear ODE results w/ different data sizes for input and output 22
3.6 Nonlinear ODE results w/ 10000 epochs 24

Acknowledgements

First and foremost, I would like to thank my thesis supervisor Dr. Leonid Berlyand for taking me under his guidance and giving me the opportunity to complete this project over the course of the last several semesters. He has not only expanded my academic knowledge of machine learning but also taught me the important lesson of perseverance when results do not work out as intended on the first attempt. He has also taught me how to be concise and intentional with each word put to paper and for this I am extremely grateful.

In addition, I would also like to thank my honors advisor Victoria Sadovskaya for not only being a wonderful professor but also helping me every step of the way whether through graduate applications or providing feedback for this project. I would also like to thank Victoria for getting me involved in Women in Math (WIM) and supporting me as a female in this male-dominated field.

I would like to thank Hai Chi, Robert 'Robby' Creese II, and Alexander 'Sasha' Gavrikov for the many hours of assistance they have provided to this project. Hai has guided me through the construction of training sets as well as taught me much about the theory and motivations for using deep neural networks. Coding the networks and analyzing the results could not be completed without the help of Robby. And lastly, the compilation of presentations and construction of this thesis have been made possible with the assistance of Sasha. I cannot thank these three individuals enough for the tremendous guidance they have provided me.

My sincere thanks also go out to the members of Dr. Berlyand's research group. The feedback they have all provided me during group meetings and the examples they have set as presenters will guide me through my future mathematics studies.

In addition, I would like to thank the Women in Mathematics (WIM) group at Penn State for the support they have provided me to pursue a career in this field. In particular, I am incredibly thankful for the financial contribution I have received towards this project as part of the WIM Scholars Program Fellowship. This aid has helped tremendously in the completion of this project and demonstrates the strong support this group displays for one another.

Last, but certainly not least, I would like to thank my family: specifically, my parents Deron and Toni Grafton, my siblings Derek and Kylie Grafton, and my significant other Sean Ristey. The love and support you have provided me over my entire life and undergraduate career have been unmatched and there is truly no one else I would rather have by my side through this journey. Thanks for the words of encouragement, hugs, laughs, memories, and more you have given me over the years. Here's to the next chapter!

Chapter 1

Defining the Project

1.1 Introduction

Overall, the idea of this project is to find an approximate solution to a differential equation in a space of functions by minimizing a loss function with gradient descent method. This effectively converts the differential equation into an optimization problem that can be solved numerically. In other words, when the loss function provides a sufficiently small result, this implies that the approximation is sufficiently close to the true solution of the differential equation. In order to do these computations, deep neural networks (DNNs) are utilized with the network itself consisting of n input nodes based on the dimension of the equation's domain. The DNN then outputs to the last layer yielding the approximate solution. This solution is achieved by training the DNN with a specific loss function that is based in part on the (boundary and initial) conditions specified within the differential equation. Use of backpropagation then allows for efficient correction of the weight matrices within the deep neural network so that all derivatives necessary for the differential equation can be obtained.

In the past, methods for solving PDEs have included finite difference method (FDM) and finite element method (FEM). While FDM is a fairly intuitive and economical way to compute such solutions, it also tends to encounter issues when boundary conditions become increasingly complex. In addition, FEM tends to involve intensive coding techniques that may delay its efficient implementation. The method of using DNNs to solve differential equations, however, provides several advantages over its finite difference and finite element counterparts. One such advantage includes that DNNs are at least as good as finite element method when it comes to dealing with dimensionality. This can be seen by comparing the number of mesh points in FEM to the number of parameters involved with the DNN. This advantage can become especially apparent when considering higher dimensions, where DNNs perform much better than other methods. For more advantages of DNNs, see Section 2.3.

Since solving ordinary differential equations (ODEs) remains a central feature to problems in engineering, physics, biology, etc. acquiring robust algorithms for solving ODEs is crucial and deep neural networks provide such an algorithm. Overall, the idea is to create a framework to approximate a differential operator for a function depending on many parameters, determine the optimal DNN parameters for acquiring the best numerical approximation, and show that when the parameters are sufficiently optimized, the DNN framework provides good approximations to the solution of the ODE.

1.2 Preliminaries

We first introduce the concept of differential operators in order to build the appropriate background knowledge for how to solve differential equations with DNNs. A differential operator is a function of differentiation operators that act on a functional space. Perhaps, the simplest way to understand differential operators is through the following examples. Consider a differential operator of the form:

$$A = a(x)\frac{d^n}{dx^n} + b(x)\frac{d^{n-1}}{dx^{n-1}}. \tag{1.1}$$

This operator is applied to an (at least) nth differentiable function which then returns a new function after the operator is applied. Another example would be of the form:

$$A(F(x)) = \frac{dF}{dx} = f(x) \tag{1.2}$$

which simply takes the derivative of a function $F(x)$. If we take $F(0) = 0$, then we can define the inverse operator:

$$A^{-1}f = \int_0^x f(t)\,dt = F(x) - F(0) = F(x) \tag{1.3}$$

where A^{-1} is commonly known as taking the integral operator on the function f.

These examples help to motivate the idea of being able to find an inverse operator so as to solve for an unknown function. In this project, we will aim to find u in the differential equation $Au(x) = f(x)$ with boundary conditions, where A is a differential operator, by finding the inverse operator A^{-1}. That is, we will use DNNs to approximate $u = A^{-1}f$.

1.3 General Differential Equation

We will consider ODE boundary value problems of the form:

$$A(u(x)) = f(x), \quad x \in (0,1) \tag{1.4}$$

$$u(0) = a \tag{1.5}$$

$$u(1) = b \tag{1.6}$$

where A is a differential operator (can be nonlinear) and u,f are from the appropriate functional spaces. In particular, for this project, we have A as a second order differential operator (i.e., Laplacian, $\frac{\partial^2}{\partial x^2} + \frac{\partial}{\partial x}$, etc.), $u \in C^2([0,1])$, and $f \in C^0([0,1])$. Note that a function in $C^k([0,1])$ is defined to be one with the property that all derivatives up to and including k exist and are continuous on the interval $[0,1]$. Thus, $C^0([0,1])$ is all continuous functions on $[0,1]$. The goal is then to use a DNN to numerically approximate $A^{-1}f = F(f)$, where F maps f into the solution u (if such F exists). Thus, given data f, a, b, we can quickly calculate an approximation to $u = A^{-1}f$.

1.4 Results from Literature

In the past, several research studies on solving differential equations have incorporated deep neural networks into their algorithms for approximating a solution. In one study by Aarts and van der Veer [1], a method using a single hidden layer feedforward network was introduced with the hopes of solving second-order linear differential equations. In order to solve the equations, three deep neural networks were run simultaneously with each of the three networks accounting for one of the function and its derivatives. It is important to note that the differential equations were time dependent with both Dirichlet and Neumann boundary conditions (BCs). One differential equation

which was investigated was the model for damped free vibration of the form:

$$\frac{d^2y}{dt^2} + 4\frac{dy}{dt} + 4y = 0, \quad t \in [0,4] \tag{1.7}$$

$$y(0) = 1 \tag{1.8}$$

$$\frac{dy}{dt}(0) = 1 \tag{1.9}$$

One network accounted for the Dirichlet BCs in (1.8) and was trained using a set consisting of input from a mesh on the points $t \in [0,4]$ and having 1 for all outputs. Another network then used the same inputs as the first network but handled (1.9) by having all inputs trained to produce 1 as the output. The third network then approximated the second derivative so that the combination of all three networks would produce the desired solution. That is, $\frac{d\phi}{dt}$ and $\frac{d^2\phi}{dt^2}$ approximate $\frac{dy}{dt}$ and $\frac{d^2y}{dt^2}$, where ϕ is the DNN. Further, the number of neurons per hidden layer for the networks approximating the boundary conditions were taken to be 6. Results for this approach indicated that all weights and biases needed to be restricted to the interval $[-5,5]$. With this restriction, the network approximation did tend to resemble that of the analytic solution, implying that deep neural networks can be used to solve differential equations. Perhaps, some of the biggest drawbacks of this study included the need for three networks to be run simultaneously. Thus, the computational time and complexity of implementing this network could be a problem for higher dimensional equations. Another drawback included the need for all outputs to have a value 1 in order to handle the boundary conditions. This poses another restriction on the generalizability of the model. Lastly, the weights of the network had to undergo a restriction to the interval $[-5,5]$ which leads to the question of what would occur without these constraints [1].

Other work on solving differential equations with deep neural networks has been conducted by Dockhorn [4] through an investigation of how to solve the Poisson and Navier-Stokes equations. The goal of this study was to find the DNN approximation to the solution of the differential equation by minimizing the loss function

$$L(\theta) = \frac{1}{N_{int}} \sum_{i=1}^{N_{int}} (A\hat{u}(x_i) - f(x_i))^2 + \frac{1}{N_{bou}} \sum_{j=1}^{N_{bou}} (\hat{u}(s_j) - g(s_j))^2 \tag{1.10}$$

Here A is a differential operator, u is the function being approximated, f and g are the known right-hand sides of the differential equation, x_i are points in the domain Ω, and s_j are points on the boundary $\partial\Omega$. This loss function takes a penalty approach to the optimization as it taxes the boundary conditions of the differential equation in addition to the equation itself. With solutions to both differential equations being known, Dockhorn was able to compare the DNN approximation to the true solution by making use of a finite difference inspired scheme. Upon investigation of the Poisson equation in two dimensions, results indicated that the most accurate approximation could be made via a network with one hidden layer with the maximum number of iterations being placed at 20000. This error was found to be of order around 10^{-5} to 10^{-6}. In particular, the DNN approximation encountered the most issues when attempting to capture the nature of the differential equation at the corners of the boundary. In addition, the correlation between the DNN approximation and the finite difference approximation was rather small which indicated that future work may need to take the direction of modifying the loss function. This became especially apparent as the

Navier-Stokes equation was able to be solved with a similar order of error but a higher correlation in the approximation and finite difference estimation [4].

Another study that has discussed the implications of solving differential equations with deep neural networks is one study by Lagaris, Likas, and Fotiadis [6]. In this study, the researchers considered how a differential equation can be broken into two parts with the first satisfying the boundary conditions and containing no changeable parameters and the second being a feedforward deep neural network that accounts of the other portions of the equation. The design of this study suggested to minimize the error in the function:

$$E(\bar{p}) = \sum_i \left(\frac{d^2u(x_i)}{dx^2} - f(x_i, u(x_i), \frac{du(x_i)}{dx})\right)^2 \tag{1.11}$$

where $\bar{p}$ are the weight parameters to be optimized. This loss function was constructed specifically for the second order ODE of the form:

$$\frac{d^2u(x)}{dx^2} = f(x, u, \frac{du}{dx}) \tag{1.12}$$

with Dirichlet BCs. By expressing the loss function in this way, the researchers were able to eliminate the need for a training set and take an unsupervised learning approach. Note that this study also discussed how to solve higher dimensional ODEs as well as PDEs. Overall, this study was able to determine several advantages of using deep neural networks to solve differential equations. The first of such advantages includes that DNNs are able to handle higher dimensional problems much easier than that of finite element methods. When dimensionality increases, the number of mesh points also increases leading methods that solve the equation locally to have large increases to computational time. However, with deep neural networks, the number of objects in the training set is the only aspect that changes with dimensionality and the ability to perform calculations in parallel eliminates the need for any extensive computational time. Another advantage found in this study is that finite element methods become less accurate on a coarse mesh when compared with the method of deep neural networks. In fact, this study found that in two dimensions, DNNs could handle a 10x10 point mesh whereas finite element method required a 18x18 point mesh to achieve an accurate solution. Lastly, this study suggested that based on the results, the computational time needed for using deep neural networks had a linear relationship as the number of parameters increased whereas finite element methods experienced an increase in computational time that appeared to be quadratic [6].

Additionally, the study by Pedro et al. [7] also utilized an unsupervised approach to solving differential equations with deep neural networks. In this study, the researchers aimed to solve the one-dimensional advection equation as well as the two-dimensional advection-diffusion equation. In order to implement unsupervised learning, the inputs to the network were taken to be the spacial independent variables as well as the time variable with the corresponding outputs being $\phi(\bar{x}, t)$ where ϕ is the dependent variable. After feeding the inputs through the network, the dependent variable and the appropriate derivatives of the dependent variable at a specific point could be evaluated. The loss function was then built by taking these derivatives to match those of the original differential equation and minimizing to get a value close to 0. In addition, a loss function based on the mean squared error was then constructed to approximate the boundary conditions. The researchers reported that the results from training could then be used to obtain an acceptable solution to other conditions not seen in training. However, there was no indication as to exactly what

constitutes the solution being acceptable. In addition, while taking this unsupervised approach eliminated the need for a training set, the researchers did encompass several drawbacks that motivate the need for a supervised method. The first of such concerns was that multiple loss functions had to be implemented in order to get an approximation. This increases the computational time necessary as well as the difficulty of the problem for more complex differential equations. Another drawback was the need to re-train for changing boundary conditions as one of the loss functions depended directly on the given conditions [7].

Overall, this previous research directs the design for this project in several ways. The work of Aarts and van der Veer [1] indicates the need for the integration into one deep neural network that can solve the differential equation rather than three separate networks. In addition, more flexibility is needed when dealing with different boundary conditions. In other words, if the boundary condition value is changed, the network needs to be better suited to handle these modifications without needing to be re-trained. The results of Pedro et al. [7] showed a similar concern with the lack of adaptability to changing boundary conditions. By taking a supervised approach, a training set consisting of a variety of boundary conditions can be used so that when inputs not from the training set are encountered, the network is able to approximate a solution without re-training. In addition, the study conducted by Dockhorn [4] suggests that a loss function of the form (1.10) may be used to acquire an accurate approximation to the differential equation. This work also motivates the idea that equations such as the Poisson equation may have some underlying property that requires less hidden layers in order to get an accurate approximation. There may also need to be some other hyperparameter that aims to control the approximation at the corners of the boundary. Additionally, the Lagaris, Likas, and Fotiadis [6] study delineates several advantages to utilizing DNNs, where DNNs are primarily able to eliminate issues pertaining to dimensionality and computational time.

Chapter 2

Deep Neural Network (DNN)

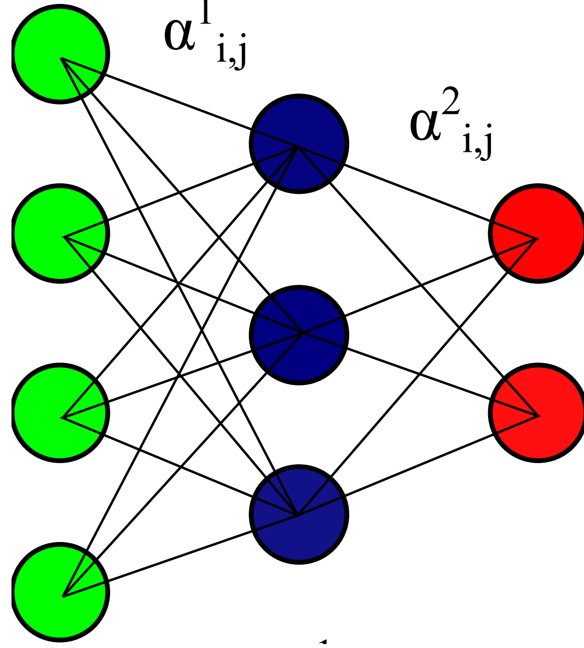

Figure 2.1: Diagram of DNN w/ 2 layer functions, X_1 mapping from the left column of neurons to the middle, and X_2 mapping from the middle column of neurons to the right.

2.1 Project Design for Deep Neural Network

We introduce a deep neural network (DNN) $\phi_\alpha(f)$ where α are the parameters and $f \in C^0([0,1])$ are the input. We have that ϕ_α is a composition with:

$$\phi_\alpha = X^M \circ X^{M-1} \circ \cdots \circ X^1 \tag{2.1}$$

where $X^i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ is a layer function. First, it is important to note that there are seemingly two definitions for a "layer" which must be distinguished. Mathematically, a "layer" is actually a layer function of the form $X^i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$. In Figure 2.1, a mathematical layer function can be thought of as the lines connecting the green and blue dots, where the dots are known as neurons (or sometimes called nodes). Geometrically, a "layer" is a column of neurons. In Figure 2.1, a geometric layer would be a column of neurons of the same color (e.g., all green dots). Note that when just the word "layer" is used in this project, it is referring to the geometric definition (as this is the more intuitive understanding).

Additionally, we also define the neurons in the input to X^1 as the *input layer*, neurons in each following layer as a *hidden layer*, and neurons in the last layer, or output of X^M, as the *output layer*. The *deep* in deep neural network refers to many layers (i.e., $M > 1$ in the DNN composition of mathematical layer functions). In this project, we considered:

- Parameters $\alpha = (\alpha_1, \cdots, \alpha_M)$ for ϕ_α, also called *weights*, where α_i is an $N_i \times N_{i-1}$ matrix,
- Mathematical layer functions given by $X^i(y) = \bar{\lambda} \circ Y^i(y)$ for $y \in \mathbb{R}^{N_{i-1}}$ where:
  - $Y^i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ is a linear map with matrix α_i,

– $\bar{\lambda} : \mathbb{R}^{N_i} \to \mathbb{R}^{N_i}$ is a non-linear *activation function*, e.g. ReLU (see Section 2.2.2).

Note that N_i is the number of neurons in the geometric layer i (where geometric layer 0 is the input layer and geometric layer M is the output layer).

The objective of this project was then to approximate the inverse differential operator $A^{-1}f : C^0([0,1]) \to C^2([0,1])$ where A^{-1} was unknown for most $f \in C^0([0,1])$. In other words, the goal was to find α so that $\phi_\alpha(f)$ approximated A^{-1} for each $f \in C^0([0,1])$. We then wanted the difference between $\phi_\alpha(f)$ and $A^{-1}f$ to be minimized via choosing an optimal α.

Since computers can only output finite-dimensional objects, we utilized a discretization so that $\phi_\alpha(f) \in \mathbb{R}^{N_M}$, N_M finite. This left the question of how we could compute the difference between $\phi_\alpha(f)$ and $A^{-1}f \in C^2([0,1])$? The solution was to discretize $u = A^{-1}f$, i.e.

$$u \to \tilde{u} = \left(u\left(\frac{1}{N_M+1}\right), u\left(\frac{2}{N_M+1}\right), \cdots, u\left(\frac{N_M}{N_M+1}\right) \right) \in \mathbb{R}^{N_M} \tag{2.2}$$

One example of this discretization would be if we discretize u and f on the interval $[0,1]$ so that we only see the values at every 0.1,

$$\tilde{u} = \{u(0), u(0.1), u(0.2), u(0.3), \ldots, u(0.9), u(1)\} \tag{2.3}$$

$$\tilde{f} = \{f(0), f(0.1), f(0.2), f(0.3), \ldots, f(0.9), f(1)\} \tag{2.4}$$

The “best” network (α) was then found by minimizing the following error called loss:

$$\bar{L}(\alpha) = \frac{1}{|T|} \sum_{f \in T} \|\tilde{u} - \phi_\alpha(f)\|_{\ell^2}^2 \tag{2.5}$$

on a training set $T \subset C^2([0,1])$ where the target $u = A^{-1}f$ was known for each $f \in T$. Note that the ℓ^2 norm of a vector $\bar{x} = (x_1, x_2, ..., x_n)$ is given by:

$$\|\bar{x}\|_{\ell^2} = \sqrt{\sum_{i=1}^{n} |x_i|^2}. \tag{2.6}$$

Minimizing the loss in (2.5) is called training because it is an iterative process during which the DNN continuously improves. It is typically done through some form of gradient descent, i.e. the starting parameters $\alpha^{(0)}$ are chosen at random and refined through the iteration:

$$\alpha^{(n+1)} = \alpha^{(n)} - \tau \nabla_{\alpha^{(n)}} \bar{L} \tag{2.7}$$

where $\tau \in \mathbb{R}$ is called the learning rate. In particular, we made use of stochastic gradient descent (SGD) where $\bar{L}$ was calculated as an average over $T^{(n)} \subset T$, where $T^{(n)}$ was randomly chosen with fixed size for each iteration (also called a batch). Note that SGD is often more efficient than gradient descent [2].

Why did this work? Because for optimal α, the DNN $\phi_\alpha(f)$ was similar to $A^{-1}f$ even when A^{-1} was unknown. That is, for every $\epsilon > 0$ there existed $\delta > 0$ such that if loss $\bar{L} < \delta$ then $|u - \phi_\alpha(f)| < \epsilon$ for all $f \notin T$ as suggested by [3].

2.2 General Structural Components

2.2.1 Hidden Layers

The most commonly altered component of the deep neural networks in this project was the number of hidden layers in the network as well as the number of neurons in each layer. First, note that the deep neural networks in this project were all feedforward networks meaning each neuron in the previous layer was connected to each neuron in the following layer [5]. The number of hidden layers was typically placed between 2 to 5 so as to keep the network running efficiently. As a whole, the general deep neural network scheme was to have a decreasing number of neurons in each layer throughout the hidden layers. The number of neurons per layer typically ranged between 16 and 2000 depending on the dimensions of the input and output data. Many of the networks also used the following scheme of varying the layer size depending on the input data dimension (N):

$$N + 2\ (input\ with\ BCs) \longrightarrow 1.5N \longrightarrow 1.3N \longrightarrow 1.1N \longrightarrow M\ (output) \tag{2.8}$$

It is important to note that the input data sizes were based on the partition of the mesh being used which was either considered fine (1024) or coarse (16).

Overall, these structures were chosen because we wanted to ensure that the network itself was not too large with the goal of keeping computational time small. For the non-linear ODE we also needed at least one hidden layer due to the non-linear nature of the equation. This also explains the lack of hidden layers necessary for the linear ODE (results explained further in Chapter 3). In addition, the second layer of the network also needed to store information on the first and second derivatives in order to find the solution.

2.2.2 Activation Function

For most of the deep neural networks in this project, the Rectified Linear Unit (ReLU) activation function $\lambda(x)$ was used. If we consider just one dimension, without loss of generality, ReLU is characterized as a piecewise linear function that will either output zero or a positive real number depending on if the input is negative/zero or positive respectively [2].

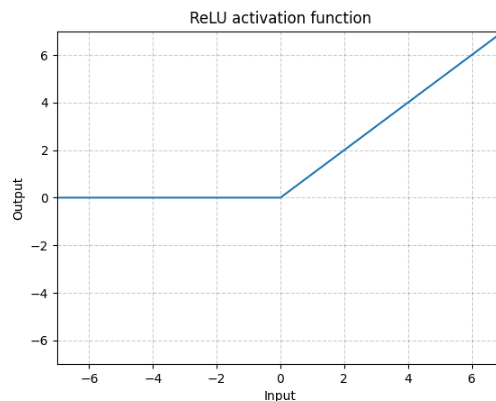

Figure 2.2: Graph courtesy [8]

In addition to ReLU, we also utilized the Gaussian Error Linear Units (GELU) function. This function is similar in nature to ReLU, excluding points around 0. In particular, this activation function is based on the cumulative distribution function for the Gaussian (normal) distribution. The goal of using GELU was to smooth out the DNN approximation since the function itself does not have the corner that is present in the ReLU function [2].

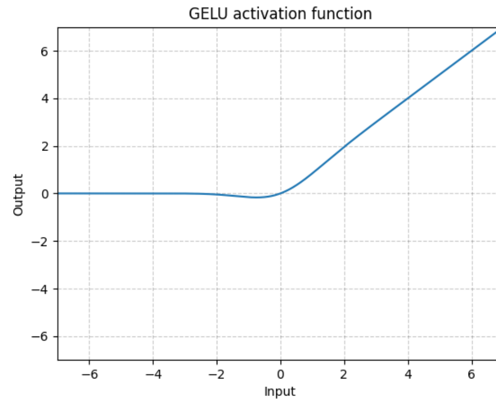

Figure 2.3: Graph courtesy [8]

Note that in higher dimensions, these activation functions are applied component-wise to the components of a vector. For example, if we have the vector $\bar{x} = (1, -2, 3)$, then for the ReLU activation function we have $\bar{\lambda}(\bar{x}) = (\lambda(1), \lambda(-2), \lambda(3)) = (1, 0, 3)$ [2].

2.2.3 Momentum

Momentum is an adaption to the method of stochastic gradient descent that works to prevent local minima from being reached while the DNN is minimizing the error. This value typically ranges between 0 and 1 with larger values indicating that the network is using more information (magnitude and direction) from the previous step's gradient to determine the magnitude and direction of the current step size. In other words, momentum acts as a hyperparameter that uses information from previous gradient calculations to update the current iteration. Note that if the value for momentum is 0 then this is simply a DNN with no momentum. Perhaps, the most intuitive way to understand momentum in a deep neural network is to consider a bicycle going down a hill. As the bicycle travels down the hill, imagine the rider applying the brakes as it approaches a local minimum. The bicycle will slow down due to the braking as it approaches the local minimum but the momentum, in terms of physics, will force the bicycle to skid past this point as the tires become locked up. This causes the bicycle to continue on to the global minimum. Similarly, the DNN with momentum will try to stop (similar to the bicycle braking) at a local minimum. However, the momentum will force the error to move past any local minima (similar to the bicycle continuing to skid through the local minimum) and continue in the updated direction until, if the momentum factor is chosen correctly, it arrives at the global minimum. Caution must be taken with using momentum, however, as too much momentum may cause oscillations for unstable systems [2]. In this project, the momentum value was set to 0.9 for all deep neural networks.

2.2.4 Learning Rate

The learning rate is a hyper-parameter that controls how much the weights within the deep neural network are updated based on the error found. It is important to note that too large of a learning rate can result in unstable training whereas too small of a rate can cause the training process to stall out. In other words, the learning rate affects the convergence rate of the network to the solution [2]. Values for the learning rate typically range between 0.0-1.0 and in our case were typically placed at 0.001 and 0.00001 for the linear ODE. In addition, for the non-linear ODE, we used two different adjusted learning rates in order to help train the network. The first adjusted rate implemented involved the rate being cut in half every 50 epochs. This change prevented the learning rate from updating the weights too much resulting in a more stable deep neural network. This was tested on the non-linear ODE as the error appeared to oscillate after 80-100 epochs which may be a result of the learning rate. We then transitioned to a learning rate that was more adapted to the network results as it was calculated by picking the starting value (τ_0) to be 0.01 and updating τ by finding:

$$max[(\frac{CurrentTrainingSetLoss}{OriginalTrainingSetLoss})^{1/2} * \tau_0, 0.0005] \tag{2.9}$$

2.3 Motivation for Utilizing DNNs

When attempting to solve differential equations, there is often an issue, known as the "Curse of Dimensionality," which involves the problem becoming exponentially more difficult in higher dimensions for traditional algorithms (finite difference, finite element, etc.). This is likely due to the fact that differential equation solving algorithms approximate the solution on a grid or a collection of points in a region of $\Omega \subset \mathbb{R}^n$. Therefore, traditional methods require a small distance between points for an accurate approximation. So as n increases, these algorithms need exponentially more points to maintain small distance between points. With this in mind, DNNs are more robust to maintaining the distances between points in the grid as volume increases when compared with traditional algorithms.

An additional advantage of DNNs arises when solving an ODE such as (1.4) where f is variable. For example, if one uses a traditional algorithm to solve for u, one must repeat the process from the beginning for every f, taking time for each use. On the other hand, after some initial computation time, a DNN can quickly output a solution u for any input f.

In addition, once a DNN is trained, it can compute the solution to (1.4)-(1.6) very quickly for a given f. That is, the DNN only needs to apply matrix multiplication and activation functions in order to find a solution. This typically takes less than a second for even very large networks. In contrast, when considering complex nonlinear differential equations, solving via finite elements or finite difference methods can take a much longer time, even days or weeks.

Lastly, by utilizing DNNs, we gain the advantage of easy customization associated with DNNs. In particular, we may easily change the number of layers, layer size, activation function, learning rate, etc. to best suit the given differential equation.

While using DNNs has its advantages, there are several disadvantages that must be addressed. Perhaps, the most prominent one involves the long duration of time DNN may need to train. In addition, we do not know a priori if the DNN will generalize well to new functions, f, sufficiently different from those of the training set.

Chapter 3

Differential Equations and Results

3.1 Training Set Generation

For each differential equation considered, a training set T was created by randomly generating a collection of C^2 functions $u_1, u_2, \cdots, u_N$ for the given equation and calculating $f_i = Au_i$ for each i (satisfying boundary conditions). Thus, the training set T consisted of pairs of the form:

$$T = \{(u_1, Au_1), (u_2, Au_2), \cdots, (u_N, Au_N)\}. \tag{3.1}$$

Each function u_i was constructed as a fourth-degree polynomial whose coefficients were generated randomly from a Gaussian distribution (mean=0, sd=1). Upon this generation, the training and test sets then consisted of input data f and exact solutions u that could be used to determine the solution of the differential equation.

For example, if we select 10 solutions from 4th degree polynomials with integer coefficients between 0 and 5

$$\{u_1, u_2, \ldots, u_{10}\} = \{2x^4 + x^2 + 5x + 1, \ldots, 4x^4 + 2x^3 + 2x^2 + 4x\}. \tag{3.2}$$

Then for $f_i := Au_i$ with $A = -\dfrac{d^2}{dx^2}$, we would obtain

$$T = \{(u_1, Au_1), \cdots, (u_{10}, Au_{10})\} \tag{3.3}$$

$$= \{(2x^4 + x^2 + 5x + 1, -24x^2 - 2), \ldots, \tag{3.4}$$

$$(4x^4 + 2x^3 + 2x^2 + 4x, -48x^2 - 12x - 4)\}. \tag{3.5}$$

Another such example would be if we again select 10 solutions from 4th degree polynomials with integer coefficients between 0 and 5

$$\{u_1, u_2, \ldots, u_{10}\} = \{2x^4, \ldots, x^4 + 1\}. \tag{3.6}$$

Then with Ginzburg-Landau differential equation $\dfrac{d^2u}{dx^2} + u - u^3 = f$ and $f_i = Au_i$ we would obtain

$$T = \{(u_1, Au_1), \cdots, (u_{10}, Au_{10})\} \tag{3.7}$$

$$= \{(2x^4, -8x^{12} + 2x^4 + 24x^2), \ldots, \tag{3.8}$$

$$(x^4 + 1, -x^{12} - 3x^8 - 2x^4 + 12x^2)\}. \tag{3.9}$$

Overall, 80% of T in (3.1) constituted the training set and was used to find optimal parameters $\alpha^{(end)}$ for the DNN. The remaining 20% of T comprised the test set and was used to evaluate the accuracy of $\phi(\alpha^{(end)}, \cdot)$. This 80/20 split is a typical choice for machine learning algorithms in DNN training. It is also important to note that all randomly generated numbers were acquired from different generation seeds. Lastly, the boundary conditions were chosen to match the u_i at the endpoints 0 and 1. After training the DNN, the mean squared error was calculated for:

- training set w/ initial parameters $\alpha^{(0)}$
- training set w/ parameters $\alpha^{(end)}$ at the end of training
- test set w/ parameters $\alpha^{(end)}$

We then calculated the relative ℓ^2 error for training and test sets:

$$\text{Relative } \ell^2 \text{ error} = \frac{\bar{L}}{\frac{1}{|T|}\sum_{\tilde{u}_i \in T} \|\tilde{u}_i\|_{\ell^2}^2}$$

3.2 Linear ODE

The first type of differential equation solved by the deep neural network was the linear second-order ODE:

$$u'' = f(x) \text{ in } (0,1) \tag{3.10}$$
$$u(0) = a \tag{3.11}$$
$$u(1) = b \tag{3.12}$$

In this ODE, we have $u \in C^2([0,1])$ and $f \in C^0([0,1])$ and the equation itself is known as the Poisson equation which has connections to electrostatics, holomorphic functions, and more.

First, it is important to note that we originally tried to solve the differential equations with the deep neural network software made available by the MatLAB Deep Learning Toolbox. However, after many attempts on a computer with 4 i7 cpu, the network took almost an hour to run one iteration. As we were unable to resolve this issue of slow computing time, a transition to the Python-based Jupyter Notebook was necessary. This transition was successful in that we were able to significantly reduce the computational time necessary to run the DNN.

After training over 30 different DNNs for the linear ODE, we determined it could be solved by a 1-layer (rather than deep) neural network with no activation function. Overall, the DNN training time was measured in epochs where 1 epoch consists of enough iterations so that each element of the training set can be chosen once for stochastic gradient descent. For the linear ODE, a network with a single linear layer provided an error of about 10^{-4} after a couple hundred epochs. This single linear layer neural network connects the input directly to the output with no hidden layers or activation function and is sometimes called support vector machine. This network structure outperformed any of the networks tested containing multiple layers with the ReLU activation function.

Overall, we concluded the results for the single layer network performed better than the deep neural networks due to the linear nature of the ODE. In other words, it is best to approximate the linear operator $\left(\frac{d^2}{dx^2}\right)^{-1}$ with a linear function with no nonlinear activation function, e.g., ReLU. These results can also be seen in Figures 3.1 and 3.2. In particular, Figure 3.1 illustrates the true solution of the ODE (blue) versus the solution approximated by the single layer network (red).

These results are similar to those acquired in the study by Dockhorn [4] in that Dockhorn also found the Poisson equation to require a rather small network in order to obtain a solution. In the study, one hidden layer was used to find a solution with error 10^{-5}. The results of this project then indicated that, perhaps, no hidden layers are required for the network to find an accurate solution. This idea is backed on the premise that the equation itself is linear and that a non-linear scheme could be leading to less accuracy.

In addition, when DNNs with multiple layers and the ReLU activation function were implemented, results indicated that adding more neurons to the first hidden layer effectively reduced the error. Further indication of these results can be see in Figure 3.3. While the error for these networks were significantly higher than that of the simple network, the results demonstrated that more neurons increased the accuracy of the DNN.

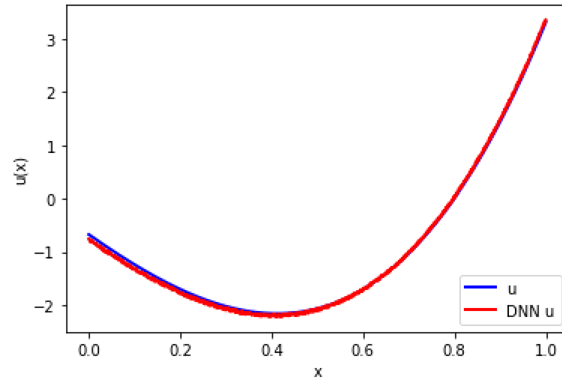

Figure 3.1: Simple network with no hidden layers or activation function: true solution (blue) vs. DNN approximation (red)

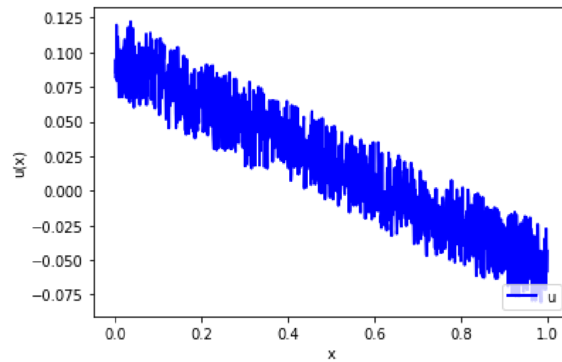

Figure 3.2: Simple network with no hidden layers or activation function: difference between exact solution and DNN approximation

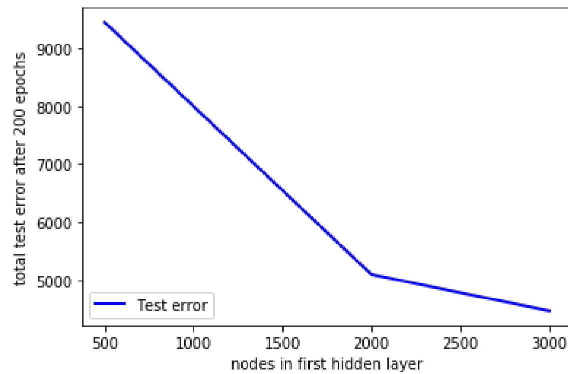

Figure 3.3: Network with ReLU and three hidden layers: differing # first hidden layer neurons

3.3 Non-Linear ODE

In addition to the linear case, we also considered the non-linear second-order ODE:

$$u'' + u - u^3 = f \text{ in } (0, 1) \tag{3.13}$$
$$u(0) = a \tag{3.14}$$
$$u(1) = b \tag{3.15}$$

Here, $u \in C^2([0, 1])$ and $f \in C^0([0, 1])$. This ODE is known as the real Ginzburg-Landau equation which is related to super-conductivity when u is a vector/complex number, liquid crystal theory for the cases when u is a tensor, and, for our purposes, we are considering the case when u is real scalar function [2].

3.3.1 Case 1: Original Loss Function

Using the same loss function as the linear ODE in (3.10), five DNNs with three hidden layers of sizes 2000, 1800, and 1500, respectively, where utilized to find a solution to the ODE. These networks feature an adjusted learning rate that started at 0.001 and was cut in half every 50 epochs. The total number of epochs permitted for training each DNN was 200 and the momentum was set to 0.9. This choice for the number of epochs was determined based on the motivation to find an accurate solution in a short computational time frame. Table 3.1 shows the mean squared error (MSE) and relative ℓ^2 error for five trials of the same DNN.

DNN	Init Train Set MSE	Final Train Set Rel ℓ^2 Err	Final Test Set Rel ℓ^2 Err
1	1.1280	0.0715	0.0733
2	1.1457	0.0825	0.0832
3	1.1086	0.0827	0.0899
4	1.1261	0.0757	0.0813
5	1.1237	0.0694	0.0753

Table 3.1: Nonlinear ODE original results

These results show that after 200 epochs the network was able to achieve a relative ℓ^2 error typically around 0.08 or 8% for both the training and test sets (see Figure 3.4). The approximation itself appeared to obtain some noise/oscillations that prevented an accurate solution from being obtained.

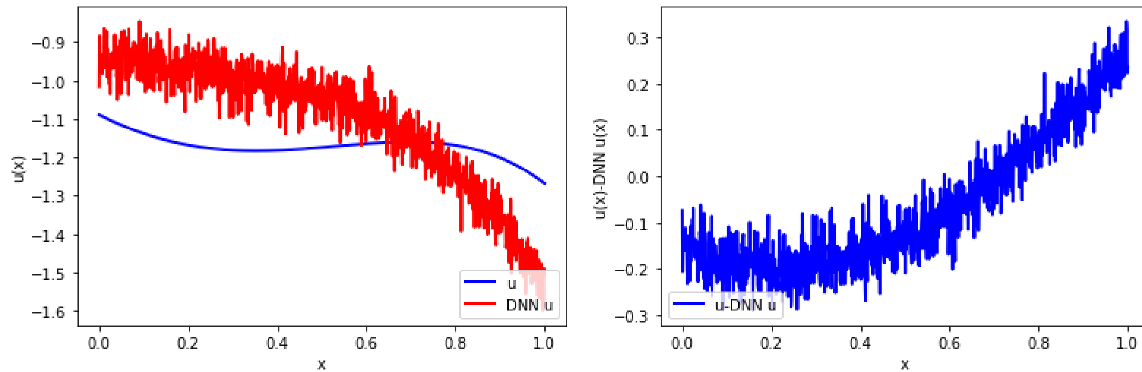

Figure 3.4: Result for one DNN and one element of T

These results indicated that one possible way to eliminate the oscillations was to utilize DNNs with a different number of hidden layer neurons. With this in mind, three different DNNs with first hidden layer sizes as 2000, 3000, 4000, and 5000 respectively were evaluated. Upon running these DNNs, the average mean squared error was found over five different networks of each structure type. Each DNN featured an adjusted learning rate that started at 0.001 and was cut in half every 50 epochs. Aside from the first hidden layer, all the networks also had the same number of neurons for other layers, 1800 and 1500 respectively. The total number of epochs each network was permitted to run was again set at 200.

# First Layer Neurons	Average Initial Train Set MSE	Average Final Train Set MSE	Average Final Test Set MSE
2000	1652.5379	140.2140	150.2999
3000	1661.4181	146.6518	163.2427
4000	1650.4153	144.1105	162.0282
5000	1641.0877	170.5839	188.6214

Table 3.2: Nonlinear ODE results w/ different first hidden layer size

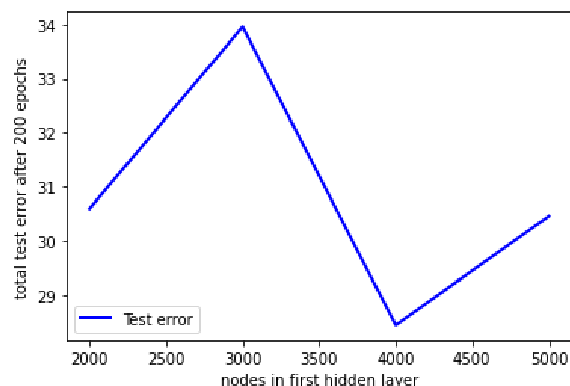

Figure 3.5: Network with three hidden layers: Differing # first hidden layer neurons

From these results, we determined that deeper networks with more neurons per hidden layer did not produce significantly better results than the corresponding shallower networks. In particular, the average mean squared error is nearly the same regardless of the increased number of neurons. It is important to note, however, that this is an AVERAGE over five approximations using the same DNN and there was typically an outlier with a large MSE for one of the approximations (see Figure 3.5).

3.3.2 Case 2: Updated Loss Function

After evaluation of these initial results, we determined that some aspect other than network structure must need altered in order to produce more accurate results for the nonlinear ODE. With this in mind, adjustments were made to the loss function so that the new loss function took the form:

$$\bar{L}(\alpha) = \sum_{f \in T} \|\tilde{u} - \phi_\alpha(f)\|^2_{\ell^2} + \lambda_2 \|\tilde{u}' - \phi'_\alpha(f)\|^2_{\ell^2} \tag{3.16}$$

$$+\lambda_1(\tilde{u}(0) - \phi_\alpha(f))^2 + \lambda_1(\tilde{u}(1) - \phi_\alpha(f))^2 \tag{3.17}$$

Here $\|\tilde{u} - \phi_\alpha(f)\|^2_{\ell^2}$ penalized the network if the values of the approximation were far away from u and $\lambda_1(\tilde{u}(0) - \phi_\alpha(f))^2 + \lambda_1(\tilde{u}(1) - \phi_\alpha(f))^2$ imposed a penalty with weight coefficient λ_1 when the boundary conditions did not hold for the approximation. We also had a $\lambda_2\|\tilde{u}' - \phi'_\alpha(f)\|^2_{\ell^2}$ term that aimed to restrict the derivative. Note that the $\tilde{u}'$ here is the derivative of u taken over its discretization and that we considered $\lambda_1 = \lambda_2$ in all cases for this project. Therefore, this loss enforced both the differential equation and boundary conditions of (1.4)-(1.6).

We then tried a variety of values for the penalty coefficients λ_1 and λ_2 in the loss function. The goal of trying various values was to penalize the boundary conditions and derivative, thus allowing for the approximation to converge to the true solution more accurately. The following results used data size 1024 and had hidden layer sizes 2000, 1800, and 1500, respectively. We also trained the network for 200 epochs for comparison purposes with the original loss function results. Note again that we considered $\lambda_1 = \lambda_2$.

λ_1 and λ_2 Values	Initial Train Set MSE	Final Train Set Relative ℓ^2 Error	Final Test Set ℓ^2 Error
0.00001	1489.6370	0.6713	0.6899
0.001	1477.2289	0.6447	0.6935
0.005	1479.8466	0.6492	0.6728
0.01	1468.6582	0.6555	0.6751

Table 3.3: Nonlinear ODE results w/ different penalty coefficients

Against intuition, the results became much worse upon changing the loss function initially. Thus, we decided to change the dimension of the data for u and f to determine if the mesh size being used was leading to the oscillations seen in the approximation. Table 3.4 and Figures 3.6-3.11 display the results for each data size. The deep neural network structure was also edited with the

hidden layer sizes being changed to adapt to the data size. In particular, each layer was the integer part of 1.5 times the data size, 1.3 times the data size, 1.1 times the data size, and the data size itself, respectively. For example, in the 256 case, the schematic network structure would be:

$$258\ (input\ with\ BCs) \longrightarrow 384 \longrightarrow 332 \longrightarrow 281 \longrightarrow 256\ (output) \tag{3.18}$$

This adjusted network structure allowed for more neurons and thus more parameters to be optimized when the data size increased. In addition, training time was increased to 500 epochs as the stopping criteria and the lambda values were placed at 0.0001 based on the previous results.

Data Size	Initial Train Set MSE	Final Train Set Relative ℓ^2 Error	Final Test Set ℓ^2 Error
16	23.2916	0.0004419	0.0008099
32	46.2514	0.0004209	0.0005536
64	93.0453	0.0003634	0.0004714
128	179.1681	0.002039	0.003736
256	364.0416	0.002812	0.01068
1024	1479.8466	0.6492	0.6728

Table 3.4: Nonlinear ODE results w/ different data sizes

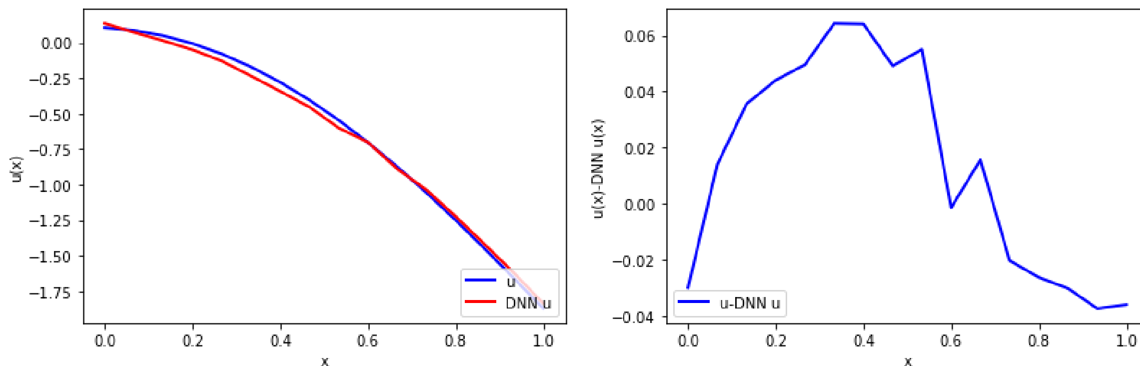

Figure 3.6: Approximate solution and error with data size 16

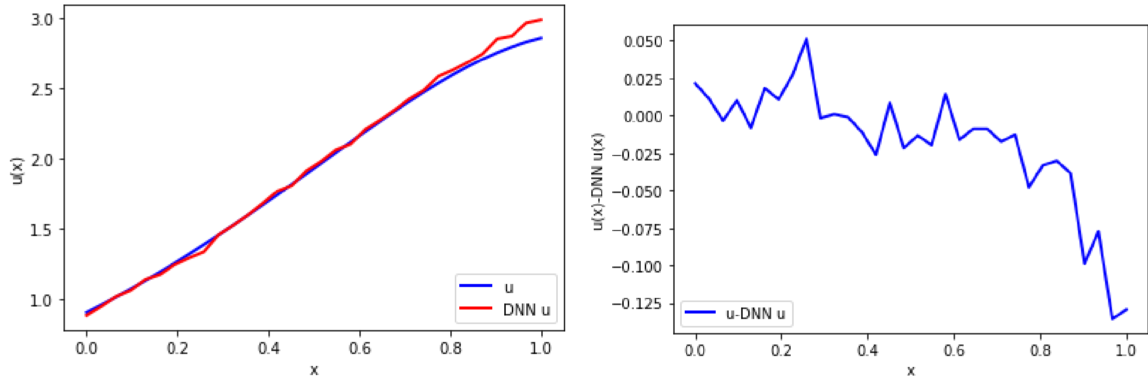

Figure 3.7: Approximate solution and error with data size 32

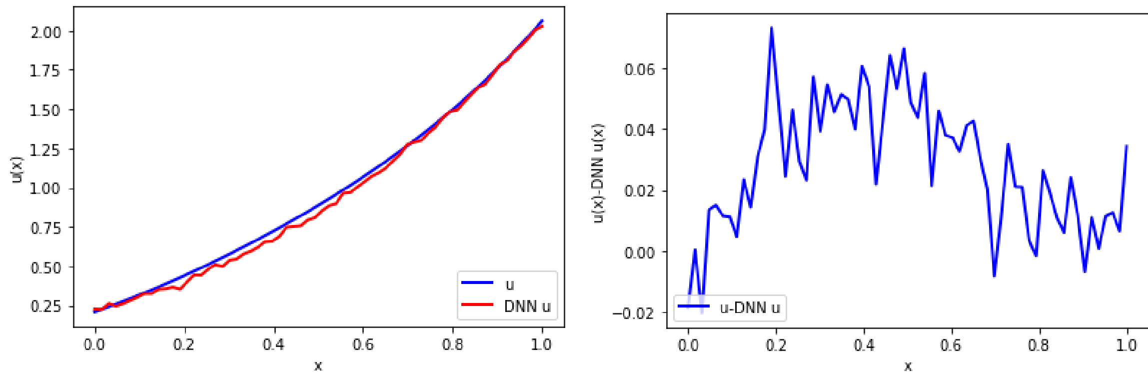

Figure 3.8: Approximate solution and error with data size 64

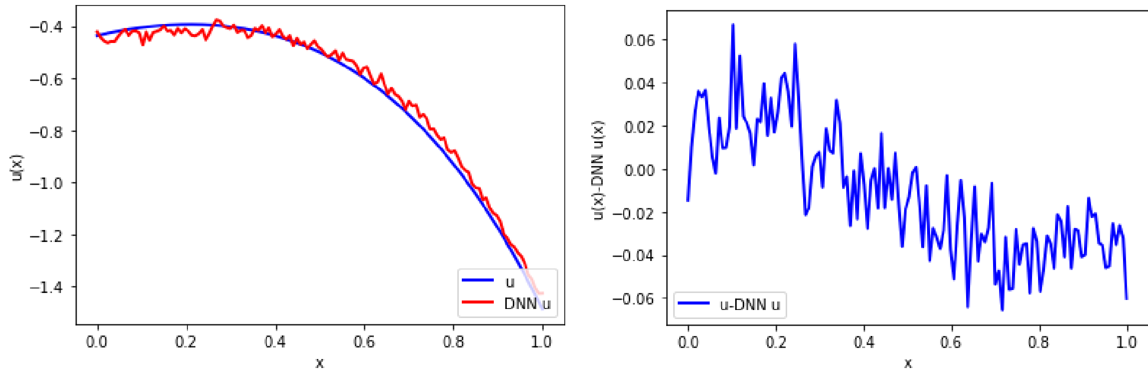

Figure 3.9: Approximate solution and error with data size 128

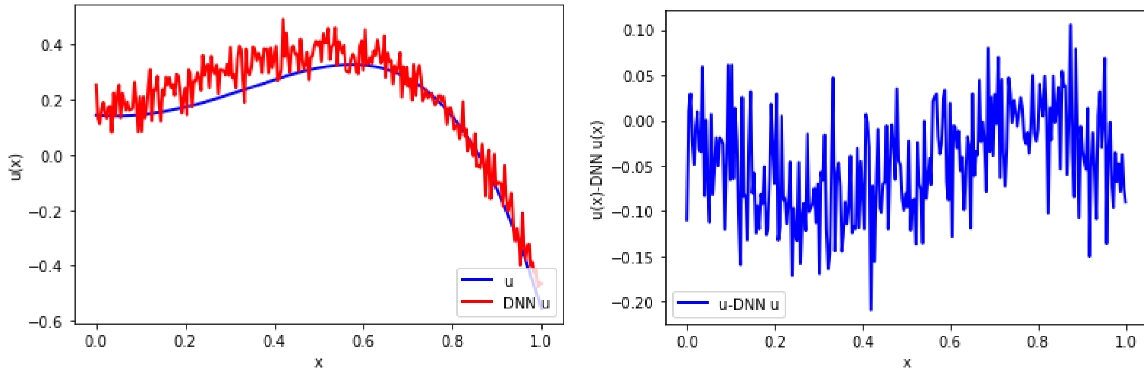

Figure 3.10: Approximate solution and error with data size 256

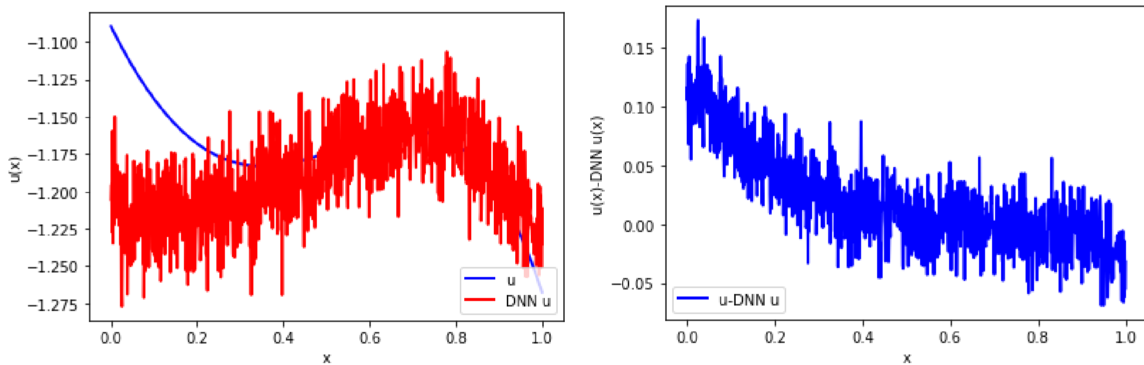

Figure 3.11: Approximate solution and error with data size 1024

As the goal of changing the loss function was to remove the oscillations found in the approximate solution, we thought such a result could be obtained via restricting the derivative as in (3.17). However, this did not solve the issue because, as demonstrated by the data size 1024 case, the oscillations have not been eliminated. One possible explanation for these results is due to the way the data set is generated using fourth degree polynomials. The resulting f will be a twelfth degree polynomial which is very oscillatory when discretizing using very high N. In order to move forward, we proposed to use different dimensions for f and u. In particular, we started with a smaller dimension for the input f and have a larger output dimension for u (i.e., 16 to 256). Making such changes, lead to the results:

Data Size (Input to Output)	Initial Train Set MSE	Final Train Set Relative ℓ^2 Error	Final Test Set ℓ^2 Error
16 to 256	363.6518	0.007617	0.007996
32 to 256	368.3765	0.005122	0.005537

Table 3.5: Nonlinear ODE results w/ different data sizes for input and output

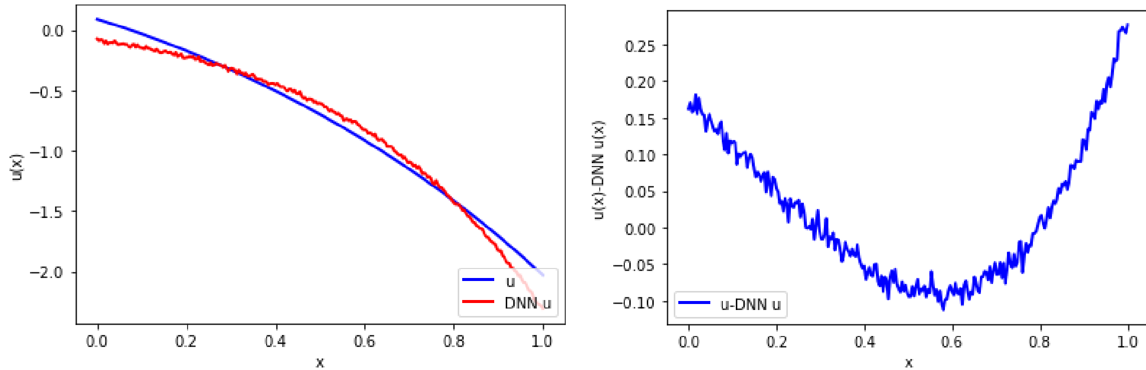

Figure 3.12: Approximate solution and error with data size 16 as input and 256 as output

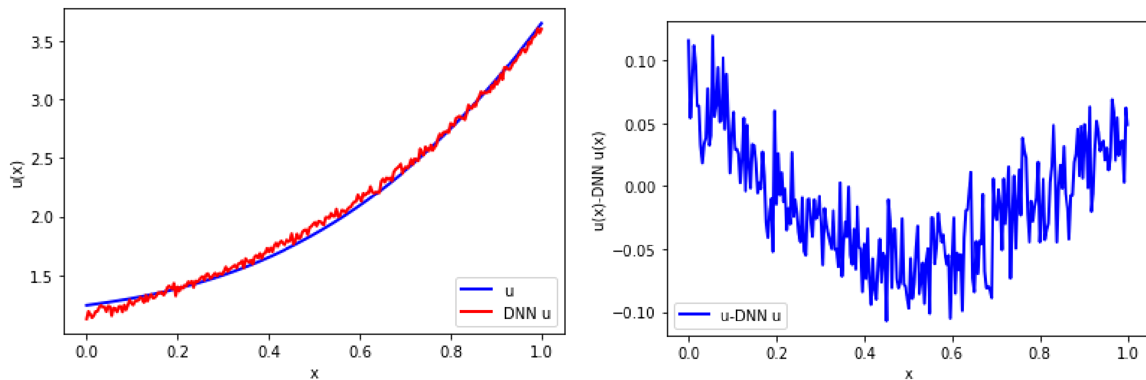

Figure 3.13: Approximate solution and error with data size 32 as input and 256 as output

Overall, these approximations were able to capture the nature of the true solution with about 0.6% error. This shows that using data from a coarse mesh can be generalized to give a solution in a finer mesh. Visually, this generalization removed the oscillations found previously in the data size 256 case, but also accumulated a slightly larger error. It appeared that when going from a smaller data size to a larger, the boundary conditions tended not to converge to the true value as in the previous cases. With this in mind, we again returned to the case with the same input and output data size in order to investigate how the oscillations could be removed, focusing specifically on 256 and 1024 data sizes.

As data size increased, the number of parameters needing to be optimized also increased. This motivated the idea that longer amounts of training were required for larger data sizes and, therefore, number of epochs must be adjusted. In the following results, the networks were run for 10000 epochs. In addition, the activation function was switched to the GELU function with the goal of smoothing out the oscillations in the data.

Data Size	Initial Train Set MSE	Final Train Set Relative ℓ^2 Error	Final Test Set ℓ^2 Error
256	365.0387	0.0001242	0.0003966
1024	1463.6688	0.0001609	0.0004484

Table 3.6: Nonlinear ODE results w/ 10000 epochs

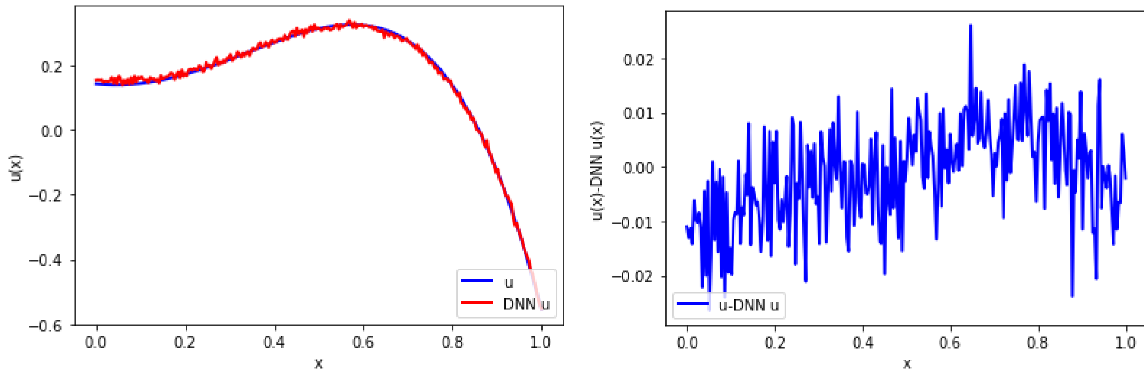

Figure 3.14: Approximate solution and error with data size 256 for 10000 epochs

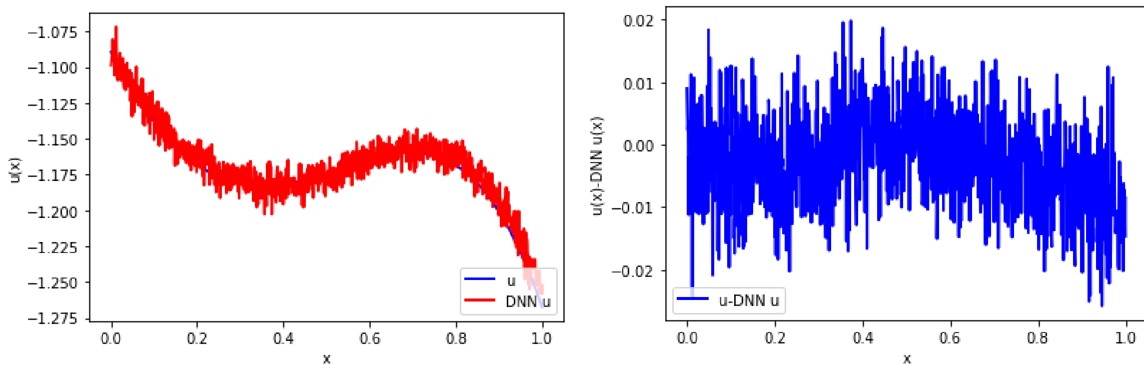

Figure 3.15: Approximate solution and error with data size 1024 for 10000 epochs

With the error for data size 256 around 0.01%, we concluded that issues with oscillations had been resolved for this case. For data size 1024, the oscillations were not completely eliminated, however, the approximation was able to converge to the solution as demonstrated by Figure 3.15. Ultimately, these results indicated that solutions to the Ginzburg-Landau ODE were able to be acquired for a variety of mesh sizes and that when the number of parameters increased, the training time also needed to be increased.

Chapter 4

Conclusions and Future Work

4.1 General Discussion of Results

Based on the acquired results, we were able to determine that the linear ODE showed that networks with no hidden layers or activation function produced the smallest error. This is likely due to the linear nature of the ODE and needing to avoid a non-linear approximation for the equation. In addition, we also demonstrated that with both the linear and non-linear ODE, increasing the number of neurons per layer decreased the error (at least slightly). This result correlates with the idea that adding infinitely more neurons per layer should lead to an accurate approximation.

For the non-linear ODE, we determined that decreasing the learning rate by cutting it in half every 50 epochs effectively reduced the error for the non-linear ODE. Similar results were also found for the second adjusted learning rate scheme implemented. Overall, we found that changing the loss function to penalize derivatives and boundary conditions did not effectively remove oscillations for higher dimensions upon initial approximations. Such oscillations were removed when the data was input from a coarser mesh and output to a finer mesh. Oscillations were also removed by running the networks for more epochs and, therefore, allowing the increased number of parameters to receive more time to be optimized.

4.2 Future Work

As solving ODEs with DNNs is still an area of mathematics that is being actively investigated, we have recommended that future work in this area take one of several directions. The first such direction involves changing the deep neural network structure to make use of convolutional layers. Roughly speaking, instead of each neuron in each layer being connected to every neuron in the following layer (fully connected), convolutional layers have a parameter $k \in \mathbb{Z}$ (i.e., $k = 3$) that says each neuron in a given layer is to be connected to k neurons in the following layer. The idea of adding these layers revolves around the fact that a convolutional layer can facilitate identifying local features of the input. This is especially useful in that ODEs are typically solved locally. Focusing on local features may also allow solutions to be approximated more efficiently.

Another direction the results of this project could promote is the transition to an unsupervised learning approach. In general, since a DNN takes input in the form of a vector $\bar{f}$ and provides as output a vector $\bar{u}$, we could discretize the Ginzburg-Landau ODE in the following sense:

$$u'' + u - u^3 = f \to B(\bar{u}) = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + u_i - u_i^3. \tag{4.1}$$

We are then able to minimize the loss function of the form:

$$\bar{L} = \|B(\bar{u}) - f\|. \tag{4.2}$$

Making this transition, would allow for the removal of a training set that relies on obtaining values by evaluating the function in the ODE. In addition, there is no longer the restriction to the type of functions in the training set (i.e., fourth degree polynomials).

Lastly, future work could also consider higher dimensional problems for ODEs to exploit the strength of DNNs in this area. Since DNNs are able to be easily adapted to higher dimensional data, the usual restrictions of traditional methods are not encountered.

Bibliography

[1] Aarts, L. P., & van der Veer, P. (2001). Neural network method for solving partial differential equations. *Neural Processing Letters*, 14:261–271.

[2] Berlyand, L. (2022). *Personal Collection of Math 452 Notes*. Pennsylvania State University, State College, PA.

[3] Borovykh, A., Oosterlee, C. W., & van der Meer, R. (2020). Neural networks for solving odes: Using optimally weighted loss functions. Imperial College London.

[4] Dockhorn, T. (2019). A discussion on solving partial differential equations using neural networks. *arXiv preprint*, pages 1–9.

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[6] Lagaris, I. E., Likas, A., & Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9:987–1000.

[7] Pedro, J. B., Maronas, J., & Paredes, R. (2019). Solving partial differential equations with neural networks. *arXiv preprint*, pages 1–7.

[8] PyTorch Team. (2019). *Torch.nn - PyTorch 1.10 documentation*. PyTorch.

JAYSA LEIGH GRAFTON
Academic Vita

EDUCATION

Bachelor of Science in Mathematics May 2022
The Pennsylvania State University, University Park, PA
Emphasis: Graduate Studies
Minor: Psychology

Schreyer Honors College, Women in Math (WIM) Scholars Fellowship, Euler Memorial Scholarship, Forest Honors Scholarship - Math, Joseph and Joline Harrington Scholarship, President's Freshman Award, Dean's List

MATHEMATICS COURSEWORK

- Math 140/141: Calculus with Analytic Geometry I/II
- Math 220H: Honors Matrices
- Math 230: Calculus and Vector Analysis
- Math 250: Ordinary Differential Equations
- Math 311W: Concepts of Discrete Mathematics
- Math 312: Concepts of Real Analysis
- Math 403: Classical Analysis I
- Math 404: Classical Analysis II
- Math 412: Fourier Series and Partial Differential Equations
- Math 414: Introduction to Probability Theory
- Math 415: Introduction to Mathematical Statistics
- Math 421: Complex Analysis
- Math 429: Introduction to Topology
- Math 435: Basic Abstract Algebra
- Math 436: Linear Algebra
- Math 452: Deep Learning Methods
- Math 455H: Introduction to Numerical Analysis I

COMPUTER SKILLS

- Programming languages:
    - Proficient: Python/Jupyter Notebook, Latex/Overleaf
    - Basic: MatLab, R
- Additional software:
    - Microsoft Office (Word, PowerPoint, Excel)
    - IBM SPSS Statistics

RESEARCH EXPERIENCE

Pennsylvania State University **University Park, Pennsylvania**
Undergraduate Mathematics Researcher September 2020 - Present

- Applied machine learning algorithm of deep neural networks (DNNs) to solve differential equations in a robust manner
- Created DNN framework through Python-based Jupyter Notebook to approximate the differential equation solution
- Implemented numerical optimization techniques such as stochastic gradient descent
- Supervised by Dr. Leonid Berlyand, Professor
- Presented results to supervisor and fellow students during weekly research meetings
- Compiled acquired knowledge and results into Schreyer Honors College thesis "Solving Differential Equations with Deep Neural Networks"

JAYSA LEIGH GRAFTON

TEACHING EXPERIENCE

Pennsylvania State University **University Park, Pennsylvania**
Selected Learning Assistant, Mathematics 405 Course August 2021 – December 2021

- Participated as part of instructional team with Dr. Diane Henderson for Math 405: Advanced Calculus for Engineers and Scientists I
- Provided assistance throughout lectures to enhance student-centered approach to learning
- Facilitated and organized weekly review sessions regarding homework content and student questions
- Corresponded with students via Zoom and email

Pennsylvania State University **University Park, Pennsylvania**
Selected Learning Assistant, Mathematics 140 Course August 2019 - December 2020

- Participated as part of instructional teams with Professor Neena Chopra and Dr. Matthew Willyard for Math 140: Calculus with Analytic Geometry I
- Provided guided examples throughout lectures to assist with student questions
- Administered weekly quizzes to approximately 45 undergraduate students
- Facilitated and organized weekly review sessions regarding current course content as well as practice examinations

Hazleton One Community Center **Hazleton, Pennsylvania**
Virtual Tutor, 4th Grade January 2020 - April 2020

- Designed weekly academic and linguistic goals to educate English Language Learners
- Facilitated weekly 45-minute virtual tutoring sessions with two students from Hazleton Elementary/Middle School regarding homework aid and language acquisition activities
- Encouraged use of students’ home language to expand academic language repertoire