

https://dspace.org/

Solving differential equations using learning-based techniques

Ilung, Tshibang Gideon

2026
University of the Witwatersrand, Johannesburg
https://hdl.handle.net/10539/50284
Ilung, Tshibang Gideon. (2026). Solving differential equations using learning-based techniques [Master's dissertation, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/50284

Downloaded from a DSpace repository. https://dspace.org/

RESEARCH REPORT

Solving differential equations using learning-based techniques

Author:
Gideon ILUNG (2241186)

Supervisors:
Prof. Byron JACOBS
Krupa PRAG



UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

submitted to
the Faculty of Science, in fulfilment of the requirements for the degree of
Masters of Science in Computational & Applied Mathematics

in the

School of Computer Science and Applied Mathematics

March 30, 2026

Declaration of Authorship

I, Gideon ILUNG (2241186), declare that this Research Report titled, "Solving differential equations using learning-based techniques" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this Research Report has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this Research Report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the Research Report is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

Abstract

Faculty of Science
School of Computer Science and Applied Mathematics

Masters of Science in Computational & Applied Mathematics

Solving differential equations using learning-based techniques

by Gideon ILUNG (2241186)

Physics-based machine learning models are an emerging approach in machine learning to solve differential equations. These models integrate the known physics of a system into the neural network training process, enhancing the efficiency of solutions to differential equations. This study will explore the capability of these models to tackle differential equations across various fields, including fluid dynamics, solid mechanics, and heat transfer. It will feature comparative analyses of different optimisation algorithms, determining under which conditions an algorithm is computationally more effective. In addition, the study will examine the impact of various hyperparameters and training strategies on the performance of these models. The potential of physics-based machine learning models lies in their capability to provide accurate and efficient solutions to differential equations, making them applicable to a wide range of crucial applications in science and engineering.

Acknowledgements

Would like to thank Prof Byron Jacobs and Krupa Prag for the assistance.

Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

1 Introduction 1
- 1.1 Aims, Objectives, and Research Questions . . . 2
  - 1.1.1 Research Questions . . . 2
  - 1.1.2 Aims . . . 2
  - 1.1.3 Objectives . . . 2
- 1.2 Model Overview . . . 3
  - 1.2.1 Logistic Equation . . . 3
  - 1.2.2 Lorenz System of Equations . . . 4
  - 1.2.3 Diffusion Equation . . . 5
  - 1.2.4 Burgers' Equation . . . 5
- 1.3 Method Overview . . . 6
- 1.4 Dissertation Layout . . . 7
- 1.5 Packages Used . . . 8

2 Background 9
- 2.1 Differentiation . . . 9
  - 2.1.1 Finite Difference . . . 10
  - 2.1.2 Spectral Differentiation . . . 10
  - 2.1.3 Automatic Differentiation . . . 11
- 2.2 Chebyshev Psuedo Spectral Method . . . 13
  - 2.2.1 Approximating Derivatives . . . 14
  - 2.2.2 Solution Method . . . 14
- 2.3 Optimisation Algorithms . . . 15
  - 2.3.1 Gradient based methods . . . 15
  - 2.3.2 Heuristic based methods . . . 15
- 2.4 Network Architecture and Algorithms . . . 16
  - 2.4.1 Artificial Neural Network . . . 16
  - 2.4.2 Nuero-Evolution . . . 17
  - 2.4.3 Compositional Pattern Producing Network . . . 18
  - 2.4.4 Physics Informed Nueral Network . . . 18

3 Gradient Based Methods in Comparision to Heuristic Based Methods 20
- 3.1 Gradient Optimisation Algorithms . . . 20
  - 3.1.1 Stochastic Gradient Descent . . . 20
  - 3.1.2 ADAM . . . 21
  - 3.1.3 L-BGFS . . . 21
- 3.2 Hueristic Optimisation Algorithms . . . 22

3.2.1 Genetic Algorithm . . . 22
3.2.2 Particle Swarm Optimisation . . . 23
PSO Algorithm . . . 24
3.3 Automatic Differentiation Approach . . . 25

4 Physics Informed Neural Networks 26
4.1 Introduction . . . 26
4.2 Methodology . . . 27
4.2.1 Point Sampling . . . 27
4.2.2 Hyperparameter tuning . . . 27
4.3 Results . . . 27
4.3.1 Logistic Equation . . . 27
4.3.2 Lorenz System of Equations . . . 31
4.3.3 Diffusion Equation . . . 32
4.3.4 Burgers' Equation . . . 37
4.4 Conclusion and Discussion . . . 40

5 Spectral Based Physics Informed Extreme Learning Machine 42
5.1 Mathematical Derivation of Proposed Method . . . 42
5.2 Solution Method . . . 43
5.2.1 Logistic Equation . . . 44
5.2.2 Diffusion Equation . . . 44
5.2.3 Burgers Equation . . . 46
5.3 Results . . . 48
5.3.1 Logistic Equation Results . . . 48
5.3.2 Diffusion Equation . . . 50
5.3.3 Burgers' Equation . . . 53
5.3.4 Discussion of results . . . 55
5.4 Conclusion and Future Work . . . 55

6 Deep Neural Operators 57
6.1 Introduction . . . 57
6.2 Methodology . . . 59
6.3 Results . . . 60
6.3.1 Logistic Equation . . . 60
6.3.2 Burgers' Equation . . . 66
6.4 Conclusion and Discussion . . . 72

7 Compositional Pattern Producing Networks 74
7.1 Introduction . . . 74
7.2 Methodology . . . 75
7.2.1 Representation of Genomes . . . 75
7.2.2 Mutations . . . 76
7.2.3 Crossover . . . 76
7.2.4 Speciation . . . 77
7.2.5 Algorithm . . . 77
7.3 Results . . . 78
7.3.1 Logistic Equation . . . 78
7.3.2 Lorenz Equation . . . 80
7.3.3 Diffusion Equation . . . 81
7.3.4 Burgers' Equation . . . 83

7.4 Conclusion and Discussion 86

8 Conclusion **87**

Bibliography **89**

List of Figures

1.1 Engineers, scientists, and data scientists: how different professions balance accuracy and time in their work. . . . 1
1.2 Example solution of the Logistic equation. . . . 3
1.3 Example solution of the Lorenz system of equations. . . . 4
1.4 Example solution of the diffusion equation. . . . 6
1.5 Example solution of Burgers' Equation. . . . 7

2.1 Diagram showing how forward mode automatic differentiation works for $f(x) = cos(x) + sin(x)$. $y_1 = cos(x)$ and $y_2 = sin(x)$ . . . 12
2.2 Diagram showing how reverse mode automatic differentiation works for $f(x) = cos(x) + sin(x)$. $y_1 = cos(x)$ and $y_2 = sin(x)$ . . . 13
2.3 Example of an ANN . . . 17
2.4 Example of an ANN without a fixed topology . . . 18

3.1 Genetic Algorithm Flowchart . . . 22
3.2 Example of crossover in GA . . . 23
3.3 Visualisation of the bit-flip. . . . 23
3.4 PSO Update . . . 24

4.1 Convergence Analysis of Different optimisation techniques on the logistic equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.1 and measuring the mean loss . . . 28
4.2 Comparision Analysis of Different optimisation techniques on the logistic equation . . . 29
4.3 Error Analysis of Different optimisation techniques on the logistic equation . . . 29
4.4 Gradient Analysis of Different gradient based optimisation techniques on the logistic equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.1 and measuring the norm of the gradient vector . . . 30
4.5 Convergence Analysis of Different optimisation techniques on the Lorenz system of equations . . . 31
4.6 Comparision Analysis of Different optimisation techniques on the Lorenz system of equations . . . 31
4.7 Error Analysis of Different optimisation techniques on the Lorenz system of equations . . . 32
4.8 Convergence Analysis of Different optimisation techniques on the diffusion equation . . . 33
4.9 Ground Truth Solution to the Diffusion equation . . . 34
4.10 Comparision Analysis of Different optimisation techniques on the diffusion equation . . . 34

4.11 Error Analysis of Different optimisation techniques on the diffusion equation . . . 35
4.12 Gradient Analysis of Different gradient based optimisation techniques on burgers equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.4 and measuring the norm of the gradient vector. L-BFGS results returned NaN consistently across the trials; hence, the gradient results were ommitted 36
4.13 Convergence Analysis of Different optimisation techniques on Burgers' equation . . . 37
4.14 Ground Truth Solution to Burgers' equation . . . 38
4.15 Comparision Analysis of Different optimisation techniques on Burgers' equation . . . 38
4.16 Error Analysis of Different optimisation techniques on Burgers' equation . . . 39
4.17 Gradient Analysis of Different gradient based optimisation techniques on burgers equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.6 and measuring the norm of the gradient vector . . . 39

5.1 Performance Analysis of PINNs vs. S-PIELM on the Logistic Equation: Averaged Results and Standard Deviation over 30 Trials. . . . 49
5.2 Performance Evalutation of S-PIELM on the logistic equation. The initial condition $u_0 = 0.6$ and $r = 1$.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$ . . . 49
5.3 Performance evaluation of PINN on the logistic equation. The initial condition $u_0 = 0.6$ and $r = 1$. The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$ . . . 50
5.4 Performance Analysis of PINNs vs. S-PIELM on the Diffusion Equation: Averaged Results and Standard Deviation over 30 Trials. S-PIELM was omitted from the convergence plot as the optimal weights were determined in a single iteration . . . 51
5.5 Performance of the S-PIELM on the diffusion equation in comparision to the exact solution. The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. $u(x, t)$ is the exact and $u^*(x, t)$ is the approximation . . . 52
5.6 Performance of the PINN on the diffusion equation in comparision to the exact solution.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. $u(x, t)$ is the exact and $u^*(x, t)$ is the approximation . . . 52
5.7 Performance Analysis of PINNs vs. S-PIELM on Burgers' Equation: Averaged Results and Standard Deviation over 30 Trials. . . . 53
5.8 Performance of the S-PIELM on Burgers' equation in comparision to the exact solution.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. The error computed is by comparing the numerical solution using spectral methods $\tilde{u}(x, t)$, with the S-PIELM approximation $u^*(x, t)$. . . . 54
5.9 Performance of the PINN on Burgers' equation in comparision to the exact solution.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. The error computed is by comparing the numerical solution using spectral methods $\tilde{u}(x, t)$, with the PINN approximation $u^*(x, t)$. . . . 54

6.1 Diagram of the Deep Neural Operator Architecture 59
6.2 Convergence Analysis of Different optimisation techniques on the logistic equation 61
6.3 Comparision of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.6$ and $R = 1$ 62
6.4 Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.6$ and $R = 1$ 62
6.5 Comparision of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.7$ and $R = 0.5$ 63
6.6 Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets. Where $u_0 = 0.7$ and $R = 0.5$ 64
6.7 Comparision of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.2$ and $R = 1$ 65
6.8 Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets. Where $u_0 = 0.2$ and $R = 1$ 65
6.9 Convergence Analysis of Different optimisation techniques on Burgers' Equation 67
6.10 Comparision of Different optimisation techniques on the Burgers' equation using DeepONets.Where $a = 1.14$,$b = 2.88$,$c = 1.80$ 68
6.11 Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $a = 1.14$,$b = 2.88$,$c = 1.80$. 68
6.12 Comparision of Different optimisation techniques on the Burgers' equation using DeepONets.Where $a = 0.93$,$b = 3.58$,$c = 2.89$ 69
6.13 Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $a = 0.93$,$b = 3.58$,$c = 2.89$. 70
6.14 Comparision of Different optimisation techniques on the Burgers' equation using DeepONets.Where $a = 0.93$,$b = 3.49$,$c = 2.93$ 71
6.15 Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $a = 0.93$,$b = 3.49$,$c = 2.93$. 71

7.1 Genome representation of the network in 7.2 76
7.2 Network representation of genome in 7.1 76
7.3 Crossover between 2 networks 77
7.4 Convergence Analysis of the CPPN algorithm on the logistic equation 78
7.5 Comparision of the CPPN algorithm on the logistic equation 79
7.6 Error Analysis of the CPPN algorithm on the logistic equation 79
7.7 Convergence Analysis of the CPPN algorithm on Lorenz System of Equations 80
7.8 Comparision of the CPPN algorithm on Lorenz System of Equations . 80
7.9 Error Analysis of the CPPN algorithm on Lorenz System of Equations 81
7.10 Convergence Analysis of the CPPN algorithm on the Diffusion equation 82
7.11 Ground Truth Solution to the Diffusion equation 82
7.12 Comparision of the CPPN algorithm on the Diffusion equation 83
7.13 Error Analysis of the CPPN algorithm on the Diffusion equation 83
7.14 Convergence Analysis of the CPPN algorithm on Burgers Equation . . 84
7.15 Ground Truth Solution to Burgers' equation 84
7.16 Comparision of the CPPN algorithm on Burgers Equation 85
7.17 Error Analysis of the CPPN algorithm on Burgers Equation 85

List of Tables

4.1 Results of hyperparameter Tuning on the logistic equation (rounded to four significant digits). Abbreviations: Alg. – algorithm; LR – learning rate; Epo. – epochs; Loss – final loss; Hist. – history size (L-BFGS); N – population size (GA, PSO); M – offspring size (GA); μ – mutation rate (GA); c_1, c_2 – cognitive/social coefficients (PSO); T_{init} – initial temperature (SA); α – cooling rate (SA). 28
4.2 Average errors for different optimization methods for the logistic equation 30
4.3 Average errors for different optimization methods 32
4.4 Best hyperparameter settings per algorithm for the diffusion equation 33
4.5 Average errors for different optimization methods 35
4.6 Best parameter settings and final losses for each algorithm on Burgers' equation. Abbreviations: Alg. – algorithm; LR – learning rate; Epo. – epochs; Loss – final loss; Hist. – history size (L-BFGS); N – population size (GA, PSO); M – offspring size (GA); μ – mutation rate (GA); c_1, c_2 – cognitive/social coefficients (PSO); T_{init} – initial temperature (SA); α – cooling rate (SA). 37
4.7 Average errors for different optimization methods 40

5.1 The following results were obtained by training a neural network within the time range $t \in [0, 5]$ and assessing its performance on an unseen time frame of $t \in (5, 10]$. For evaluation, a fixed number of collocation points, $Nt = 1000$, was employed, while the training phase explored the impact of varying the number of collocation points to observe performance differences. 48
5.2 The following results were obtained by training a neural network within the time range $t \in [0, 0.5]$ and assessing its performance on an unseen time frame of $t \in (0.5, 1]$. For evaluation, a fixed number of collocation points, $Nt = 100$, was employed, while the training phase explored the impact of varying the number of collocation points to observe performance differences. 51
5.3 The following results were obtained by training a neural network within the time range $t \in [0, 0.5]$ and assessing its performance on an unseen time frame of $t \in (0.5, 1]$. For evaluation, a fixed number of collocation points, $Nt = 100$, was employed, while the training phase explored the impact of varying the number of collocation points to observe performance differences. 53

5.4 Comparison of the methods based on key characteristics. “Generalisation Capability” indicates the ability to generalise beyond the training data, categorised as either “Improved" or “Limited". “Loss Evaluation" specifies the approach used: “Global" or “Local-based". “Computational Efficiency" refers to the level of computational resources required, categorised as either "High" or "Moderate". The table also notes whether the method handles solving non-linear Equation and solving equations in domains with a Complex Geometry. 56

6.1 Architecture of Different Models 60
6.2 Best hyperparameter tuning results for each algorithm (rounded to four significant digits). Abbreviations: Alg. – algorithm; LR – learning rate; Epo. – epochs; Loss – final loss; Hist. – history size (L-BFGS); N – population size (GA, PSO); M – offspring size (GA); μ – mutation rate (GA); c_1, c_2 – cognitive/social coefficients (PSO); T_{init} – initial temperature (SA); α – cooling rate (SA). 61
6.3 Average errors for different optimization methods on the logistic equation, where $u_0 = 0.6$ and $R = 1$ 63
6.4 Average errors for different optimization methods, where $u_0 = 0.7$ and $R = 0.5$ 64
6.5 Average errors for different optimization methods,Where $u_0 = 0.2$ and $R = 1$ 66
6.6 Architecture of Different Models 67
6.7 Average errors for different optimization methods Where $a = 1.14$,$b = 2.88$,$c = 1.80$ 69
6.8 Average errors for different optimization methods Where $a = 0.93$,$b = 3.58$,$c = 2.89$ 70
6.9 Average errors for different optimization methods Where $a = 0.93$,$b = 3.49$,$c = 2.93$ 72

List of Abbreviations

AD	**A**utomatic **D**ifferentiation
GA	**G**enetic **A**lgorithm
PSO	**P**article **S**warm **O**ptimisation
SA	**S**imulated **A**nealing
SGD	**S**tochastic **G**radient **D**escent
L-BGFS	**L**imited-Memory **B**royden–**F**letcher–**G**oldfarb–**S**hanno
PINNS	**P**hysics **I**nformed **N**eural **N**etworks
Deep**ONet**	**D**eep **O**perator **N**etwork
ANN	**A**rtificial **N**eural **N**etwork
FFNN	**F**eed-**F**orward Neural Network
RNN	**R**ecursive **N**eural **N**etwork
TWEANNS	**T**opology and **W**eight **E**volving **A**rtificial **N**eural **N**etworks
NE	**N**uero**E**volution
NEAT	**N**uero**E**volution of **A**ugmenting **T**opologies
CPPN	**C**ompositional **P**attern **P**roducing **N**etwork
PDEs	**P**artial **D**ifferential **E**quations
ODEs	**O**rdinary **D**ifferential **E**quations

Chapter 1

Introduction

Many industrial systems can be modelled by intricate and non-linear differential equations where analytical solutions are difficult to obtain, which makes it necessary to use numerical techniques to solve them. In the automotive industry, designing and testing vehicle suspensions is a crucial step in the development of a new car. Suspension systems are responsible for ensuring a smooth ride, good handling, and safety by controlling vehicle movement in response to bumps and turns on the road [1] [2]. The behaviour of the suspension system is described by differential equations that cannot be solved analytically. There are numerous numerical methods available that can return optimal solutions, but they often have significant drawbacks. For instance, solutions obtained with these methods are not reusable in different domains, and as the systems scale, the time and computational expense of finding a solution increase. However, with the advent of physics-informed neural networks (PINNs), it is now possible to design a generalised model governed by a differential equation that can be applied in multiple domains. This approach holds great promise as once the model is trained, sampling solutions becomes much faster due to the use of neural networks, which involve matrix multiplication passed through a non-linear activation function. Furthermore, the optimisation of these networks within popular packages such as PyTorch and TensorFlow further enhances their potential application in physics simulations, controller design, and inverse problems. Figure 1.1 shows how different professions balance accuracy and time taken to obtain solutions.

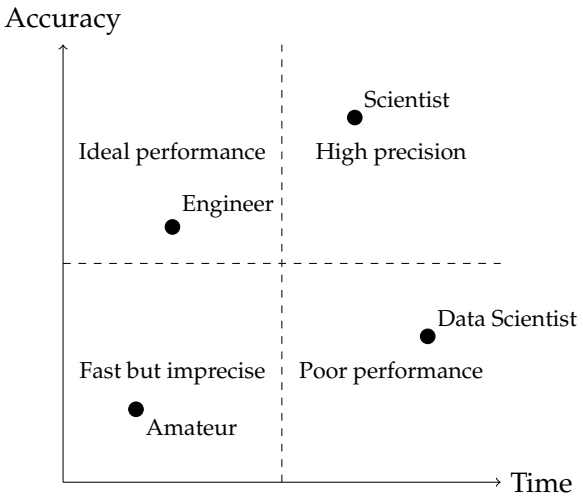


FIGURE 1.1: Engineers, scientists, and data scientists: how different professions balance accuracy and time in their work.

1.1 Aims, Objectives, and Research Questions

The significance of precise and fast solvers has been widely acknowledged in various industries. However, existing systems are hindered by computationally demanding,non-reusable solvers that require substantial computing resources to generate solutions. As our society progresses, industrial systems are becoming increasingly intricate, leading to more sophisticated governing equations and significantly increasing the resource requirements to solve them. The Finite Element Method and the Finite Volume Method are highly sought-after numerical techniques in industry for their exceptional accuracy. However, their implementation demands significant computational resources, prompting a search for faster, less computationally intensive alternatives.

1.1.1 Research Questions

The following research questions will be addressed throughout this study:

- How effective are gradient-based optimisation techniques compared to heuristic approaches in training machine learning models for solving PDEs/ODEs?
- What are the trade-offs between convergence speed, accuracy, and computational cost for different optimization strategies in physics-informed machine learning models?
- How do different neural network architectures perform in approximating solutions to differential equations?

1.1.2 Aims

Given the aforementioned questions, the aim of this study is to:

- To evaluate and compare the performance of gradient-based and heuristic optimization methods in training machine learning models for solving ordinary and partial differential equations.
- Analyze the trade-offs between convergence speed, solution accuracy, and computational resource usage in different optimisation strategies.

1.1.3 Objectives

Given the questions and objectives mentioned above, the objective of this study is to:

- Solve differential equations that govern the behaviour of industrial systems using Physics Based Machine Learning.
- Compare the accuracy of solutions obtained from different optimisation algorithms used in Physics Based Machine Learning.
- Determine the computational efficiency of training and sampling solutions from a Physics Based Machine Learning model.
- To explore how various network architectures impact different problems.

1.2 Model Overview

The following set of differential equations, comprising the logistic equation, Lorenz equations, diffusion equation, and Burgers equation, will be solved using various ANN architectures, which will be optimised using gradient- and heuristic-based methods.

1.2.1 Logistic Equation

The logistic equation (1.1) is a common differential equation used in population ecology to model the growth of a population with limited resources. r is a parameter , usually denoted as the growth rate, t is the independent variable, and $u(t)$ is the dependent variable, which is the solution to the differential equation. The exact solution to the differential equation is provided in equation (1.3). It describes the rate of change of a population over time as a function of the population's current size, the carrying capacity of the environment, and a growth rate parameter. The equation exhibits sigmoidal behaviour, meaning that the population initially grows rapidly but eventually levels off as it approaches the carrying capacity. The logistic equation has applications in many fields beyond ecology, including economics, epidemiology, and physics. An example of a solution obtained by solving the logistic equation can be seen in Figure 1.2. Further discussion of the logistic equation and its application can be found in [3].

$$\frac{du}{dt} = ru(t)(1 - u(t)) \tag{1.1}$$

$$u(0) = u_0 \tag{1.2}$$

$$u(t) = \frac{\frac{u_0}{u_0 - 1} \exp(rt)}{\frac{u_0}{u_0 - 1} \exp(rt) - 1} \tag{1.3}$$

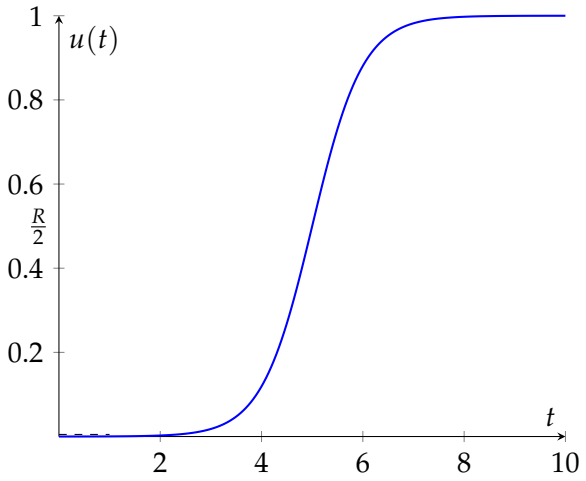


FIGURE 1.2: Example solution of the Logistic equation.

1.2.2 Lorenz System of Equations

$$\frac{dx}{dt} = \sigma(y - x) \tag{1.4}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{1.5}$$

$$\frac{dz}{dt} = xy - \beta z \tag{1.6}$$

The Lorenz system of equations (1.4 - 1.6) is a set of three non-linear differential equations that describe a simplified model of atmospheric convection. It was developed by Edward Lorenz in the 1960s as a mathematical representation of the behaviour of a fluid in a three-dimensional space. The system is characterised by its chaotic behaviour and exhibits a sensitive dependence on initial conditions. x, y, and z represent the state variables, and σ, ρ, and β are parameters that determine the behaviour of the system. The Lorenz system has applications in many fields beyond atmospheric science, including fluid dynamics, chaos theory, and non-linear dynamics. It has been used to model a wide range of phenomena, including the dynamics of chemical reactions, the behaviour of laser systems, and the formation of patterns in biological systems. The Lorenz system has also been studied extensively as a prototypical example of chaotic behaviour and as a benchmark problem for testing numerical algorithms and chaos control methods. An example of a solution obtained by solving the Lorenz equations can be seen in Figure 1.3. Further discussion of the logistic equation and its application can be found in [4] [5].

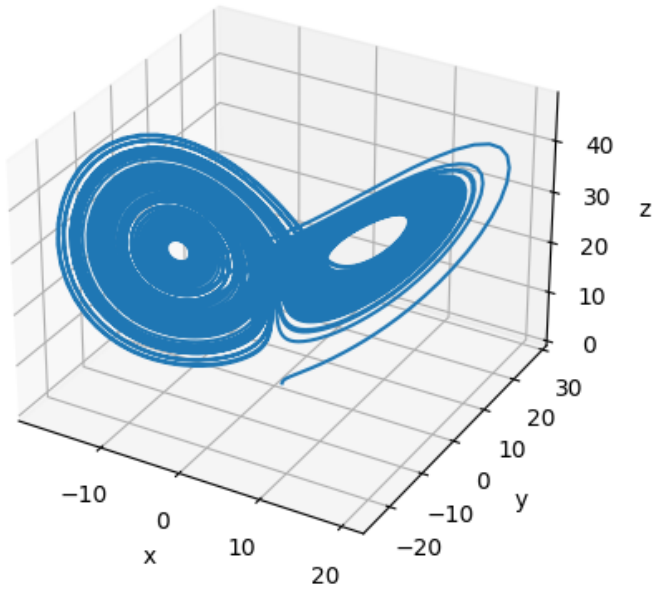


FIGURE 1.3: Example solution of the Lorenz system of equations.

1.2.3 Diffusion Equation

The diffusion equation (1.7) is a partial differential equation that describes how a quantity such as heat, mass, or concentration spreads or diffuses through space over time. $u(x,t)$ is the dependent variable that is the solution to equation (1.7).

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - e^{-t}\left(\sin(\pi x) - \pi^2 \sin(\pi x)\right) \tag{1.7}$$

subject to

$$u(x,0) = \sin(\pi x) \tag{1.8}$$

$$u(-1,t) = u(1,t) = 0 \tag{1.9}$$

The equation is a mathematical representation of the physical process of diffusion, which is the movement of particles from an area of high concentration to an area of low concentration due to random motion. The diffusion equation has many applications in science and engineering, including:

- Heat transfer: The diffusion of heat through a material can be modelled using the diffusion equation. This is important in many engineering applications, such as designing cooling systems for electronics or predicting the behaviour of materials in high-temperature environments [6].
- Mass transport: The diffusion equation is also used to model the movement of mass through a medium, such as the spread of a pollutant in the environment or the diffusion of a drug through tissue in the human body [7].
- Chemical reactions: The diffusion equation can be used to model the diffusion of reactants and products in chemical reactions, which is important in fields such as chemical engineering and materials science [8].
- Image processing: The diffusion equation has been applied in image processing to remove noise and sharpen images [9].

An example solution of the diffusion equation can be found in Figure 1.4. A detailed discussion on the diffusion equation and its application can be found in [10]

$$u(x,t) = e^{-t}\sin(\pi x) \tag{1.10}$$

The exact solution to the diffusion equation is given by equation (1.10)

1.2.4 Burgers' Equation

Burgers' equation (1.11) is a non-linear partial differential equation that describes the motion of fluids in which viscosity and non-linearity play an essential role. It was named after Dutch mathematician Johannes Burgers, who first introduced it in 1948. The equation has applications in various fields of physics and engineering, such as fluid dynamics [11], non-linear acoustics, traffic flow, and even in mathematical finance . In fluid dynamics, the equation is used to study the behaviour of fluids in which the velocity field exhibits turbulence or shock waves[12]. It is also used to study waves in shallow water, such as tsunamis [13]. In traffic flow modelling, the

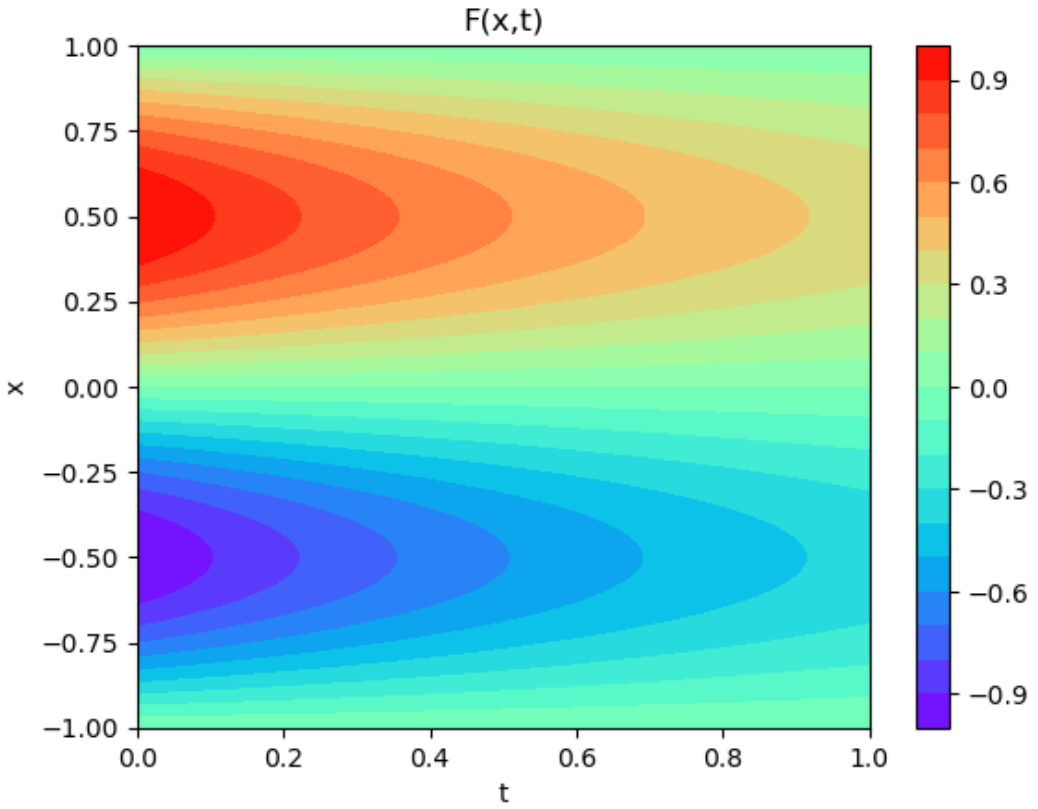


FIGURE 1.4: Example solution of the diffusion equation.

equation is used to model traffic jams and congestion [14]. In mathematical finance, it is used to model the dynamics of stock prices and other financial quantities[15]. Overall, Burgers' equation is a fundamental equation that describes the behaviour of many physical and mathematical systems, and its applications are numerous and diverse.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{1.11}$$

subject to

$$u(x,0) = \sin(x) \tag{1.12}$$

$$u(0,t) = u(\pi,t) = 0 \tag{1.13}$$

1.3 Method Overview

The objective of this study is to compare multiple neural network architectures that have been optimized using gradient-based methods and heuristics. The focus is on exploring the ability of these networks to learn the solution of a differential equation without the need for input data. To achieve our objective, we will first optimise the neural network architectures using gradient-based methods and heuristics. This will involve tuning the network's hyperparameters, such as the learning rate, regularisation strength, and number of layers. We will then evaluate the performance of each

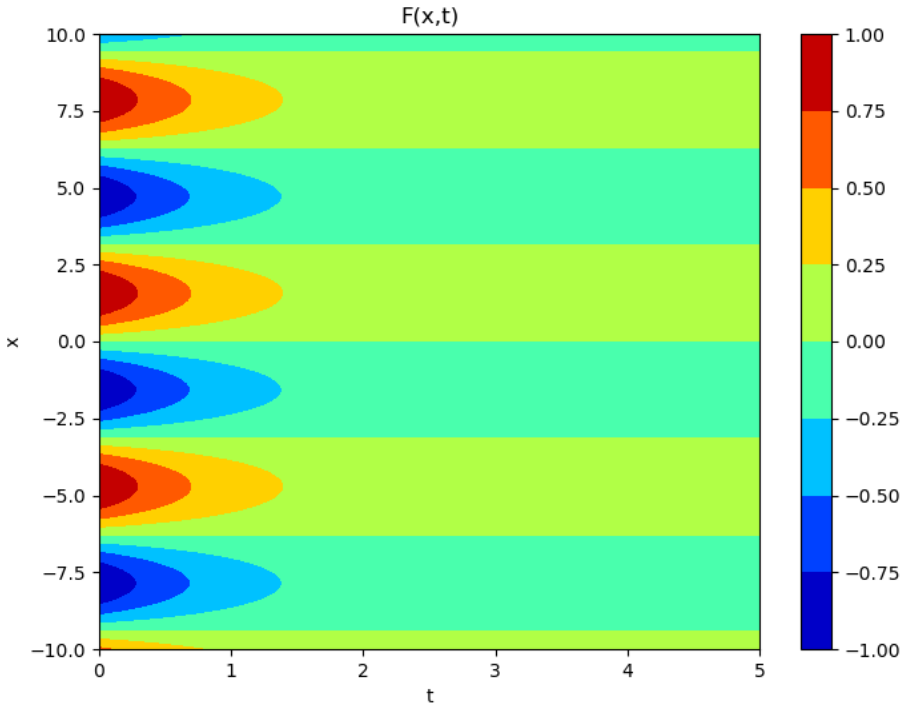


FIGURE 1.5: Example solution of Burgers' Equation.

optimized architecture by testing its ability to accurately approximate the solution of the differential equation under various parameters . The following Neural Network architectures will be used within this study:

- Feed-Forward Neural Networks (FFNNs)
- Deep Neural Operators (DNOs)
- Topology and Weight Evolving Artificial Neural Networks (TWEANNs).
- Extreme Learning Machines

Given the significant differences between architecture types, a direct comparison would be unreasonable; instead, we will focus on comparing the optimisation algorithms. In general, this study has the potential to provide information on the effectiveness of neural networks as a tool for solving differential equations. It may also shed light on the optimal design of neural network architectures for this purpose, and contribute to the development of new approaches for solving differential equations using deep learning techniques.

1.4 Dissertation Layout

In Chapter 1, the study's objective and the differential equations to be solved are presented. Chapter 2 provides background information on solving differential equations using both classical and ANN-based techniques. Chapter 3 offers a comprehensive comparison of gradient-based and heuristic methods, which will be used to optimise various neural network architectures. Chapter 4 discusses and compares

different optimisation algorithms in the context of the Physics-Informed Neural Network architecture. Chapter 5 Examines and contrasts a hybrid approach combining spectral methods with extreme learning machines against physics-informed neural networks (PINNs). Chapter 6 examines and compares the various optimisation algorithms when using the Deep Neural Operator architecture. Chapter 7 explores a novel approach to solving differential equations using Compositional Pattern Producing Networks.

1.5 Packages Used

For constructing and training the network, we used PyTorch in Python [16]. Samples were generated via Latin hypercube sampling using pyDOE [17].

Chapter 2

Background

A differential equation is a mathematical equation that involves an unknown function and its derivatives. It describes how a function changes over time on the basis of the relationship between the function and its derivatives with respect to the independent variables. Differential equations are used to derive analytical solutions that describe the behaviour of various systems. In situations where analytical solutions are not achievable, numerical methods are employed to approximate the solutions. The process of solving differential equations can provide crucial understanding into the behaviour of intricate systems and assist in predicting the potential state of the system. Differential equations provide a mathematical framework for modelling systems, while neural networks are frequently utilised in data-driven approaches. The vast availability of data has led to the utilisationof data-driven techniques, such as neural networks, to create effective models that describe the dynamics of a system. A neural network is a machine learning algorithm inspired by the human brain that consists of interconnected artificial neural networks organised into layers. It can learn from data and identify intricate patterns, making it suitable for various applications, such as image recognition and prediction tasks. The network's performance (accuracy,computational cost) depends on factors such as architecture, activation functions, and optimisation algorithms used during training. In the past, the derivatives of artificial neural networks (ANNs) were calculated to optimise the network weights using gradient-based optimisation techniques. With the same approach, it is now feasible to obtain the derivative of the network concerning the inputs, enabling networks to solve differential equations.

The chapter has been divided into three main sections. The first section covers the discussion of methods for approximating derivatives that are utilised in numerical solvers. The second section deals with the discussion of optimisation techniques for training neural networks, while the third section provides an overview of neural networks and their variants.

2.1 Differentiation

The subsequent section gives an overview of various numerical methods for approximating derivatives, which are essential to optimise the network and/or solve a differential equation.

We conducted systematic searches in Google Scholar to identify in-depth discussions of the preliminary concepts underpinning our study. Keywords such as "finite difference," "spectral differentiation," and "automatic differentiation" were used to locate relevant material. We then selected resources based on their citation counts

and our prior familiarity with the content, giving preference to sources that are both widely recognised and easy to follow.

2.1.1 Finite Difference

Finite difference is a numerical method for approximating derivatives in mathematical models. It replaces derivatives with finite differences obtained by computing the difference between nearby points. The method is widely used in science and engineering, but it can suffer from accuracy issues in certain situations. By definition, the derivative of a function $y = f(x)$ is given by equation (2.1). h is a small value representing the change in x.

$$y' = f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.1}$$

By utilizing Taylor Series, it is possible to approximate the function's value for a small change in x as seen in equations (2.2), (2.3).

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \dots \tag{2.2}$$

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \dots \tag{2.3}$$

There are three distinct methods for approximating derivatives using equations (2.1 - 2.3). The first method, known as the forward difference method and represented by Equation (2.4), is determined using Equation (2.2).

$$\frac{d^n f}{dx^n} = \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} f(x+ih) \tag{2.4}$$

The second scheme is the backward difference method which is given by equation (2.5). The formula is determined using equation (2.3).

$$\frac{d^n f}{dx^n} = \sum_{i=0}^{n} (-1)^{i} \binom{n}{i} f(x-ih) \tag{2.5}$$

The final scheme is the central difference given by equation (2.6) which is determined by adding equations (2.2) and (2.3).

$$\frac{d^n f}{dx^n} = \sum_{i=0}^{n} (-1)^{i} \binom{n}{i} f\left(x - \left(\frac{n}{2} - i\right) h\right) \tag{2.6}$$

A crucial point to keep in mind is that n represents the order of the derivative to be computed, and $\binom{n}{i}$ denotes the binomial coefficients of Pascal's triangle [18]. Compared to the forward and backward difference, the central difference has a greater degree of accuracy. More detailed information is available in [19].

2.1.2 Spectral Differentiation

Spectral differentiation is a numerical method used to calculate derivatives of functions represented as a sum of basis functions. It applies the derivative operator to the coefficients of the basis functions and can be more accurate and efficient than finite difference methods. Common basis functions include Fourier series, Chebyshev polynomials, and Legendre polynomials [20] [21]. The assumption made when

using spectral differentiation is the function $y(x)$ can be approximated by a combination of basis functions given by equation (2.7) and its derivative by equation (2.8).

$$y(x_j) \approx \sum_{i=0}^{N} y_j \phi_i(x_j) \tag{2.7}$$

$$y'(x_j) \approx \sum_{i=0}^{N} y_j \phi_i'(x_j) \tag{2.8}$$

N represents the number of basis functions that will be used to approximate $y(x)$ and x_j is the point on the index j. These points are usually evenly spaced and are referred to as collocation points. Gauss-Lobatto points are the most commonly used collocation points [22].Let $D_{jk} = \phi_k'(x_j)$, equations (2.7),(2.8) can be rewritten as a system of equations (2.9).

$$\begin{bmatrix} y'(x_0) \\ y'(x_1) \\ \vdots \\ y'(x_N) \end{bmatrix} = \begin{bmatrix} D_{0,0} & D_{0,1} & \dots & D_{0,N} \\ D_{1,0} & D_{1,1} & \dots & D_{1,N} \\ \vdots & \vdots & \dots & \vdots \\ D_{N,0} & D_{N,1} & \dots & D_{N,N} \end{bmatrix} \begin{bmatrix} y(x_0) \\ y(x_1) \\ \vdots \\ y(x_N) \end{bmatrix} \tag{2.9}$$

2.1.3 Automatic Differentiation

Automatic differentiation (AD) is a technique for computing derivatives of a function that is implemented by a computer programme. It is an alternative to numerical differentiation and symbolic differentiation. In AD, the derivative of a function is calculated by recursively applying the chain rule of calculus to the elementary operations (such as addition, multiplication, and division) that make up the function. This is done automatically by the computer, hence the name "automatic differentiation." AD is widely used in scientific computing, machine learning, and optimisation. It allows for efficient and accurate computation of derivatives, even for intricate functions that would be difficult or impossible to differentiate by hand [23]. It is important to note that the error using AD is similar to symbolic error. In order to compute derivatives, there are two primary methods: forward-mode and reverse-mode. Both approaches will be briefly discussed in the following subsection.

Foward-Mode

Forward mode Automatic Differentiation (AD) is a method for computing the derivative of a mathematical function.During the computation, the function and its derivative are evaluated simultaneously by propagating values and derivatives forward through the network of operations, starting from the input variables(an example of this can be seen in Figure 2.1). This is possible due to dual numbers [24].Dual numbers share similarities with intricate values but differ in that $i^2 = 0$. Consider two dual numbers: $u = f(x_j) + f'(x_j)i$ and $v = g(x_j) + g'(x_j)i$. By performing basic arithmetic operations such as addition, subtraction, multiplication, and division, the following results can be derived.

- $u \pm v : f(x_j) \pm g(x_j) + (f'(x_j) \pm g'(x_j))i$
- $u \times v : f(x_j) * g(x_j) + \left(f'(x_j) * g(x_j) + f(x_j) * g'(x_j)\right) i$
- $: u/v : \frac{f(x_j)}{g(x_j)} + \frac{f'(x_j)*g(x_j) - f(x_j)*g'(x_j)}{g(x_j)^2} i$

Using elementary operations on values in the dual plane as shown in the expressions aforementioned,the rules of differentiation can be derived. This is beneficial as it enables the simultaneous computation of both the function's value and its derivatives.Forward mode AD is primarily used for functions $f : R^n \rightarrow R^m$, where n is significantly smaller than m.

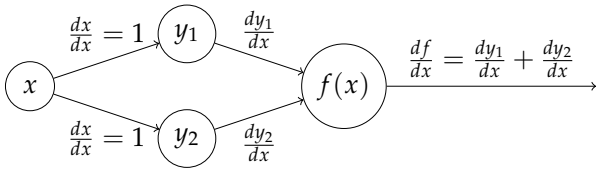


FIGURE 2.1: Diagram showing how forward mode automatic differentiation works for $f(x) = cos(x) + sin(x)$. $y_1 = cos(x)$ and $y_2 = sin(x)$

Reverse-Mode

The reverse mode AD rule computes the gradient of f with respect to each input x_i by traversing the computational graph in reverse order. Specifically, at each node i on the graph, the gradient of the graph output with respect to the output of node i is computed using the chain rule of calculus. The gradient is then propagated backward to the input of node i by multiplying it with the local gradient of node i with respect to its inputs. (An example of this can be seen in Figure 2.2). Reverse mode AD is primarily utilised for functions $f : R^n \rightarrow R^m$, where m is significantly smaller than n [23].

To be more precise, let y_i be the output of node i in the computational graph, and let $w = f(x_1, x_2, \ldots, x_n)$ be the final output of the graph. Then, the reverse mode AD rule can be written as:

$$\frac{\partial w}{\partial x_i} = \sum_j \frac{\partial w}{\partial y_j} \frac{\partial y_j}{\partial x_i},$$

where the sum is taken over all nodes j that depend on x_i, that is, all nodes that are connected to node i in the graph.

Intuitively, this rule says that the gradient of the output w with respect to an input x_i can be calculated by summing the contributions of all nodes in the graph that depend on x_i. Each contribution is the product of two factors: the gradient of the output with respect to the output of the node, and the local gradient of the node with respect to the input x_i.

For instance, using $y_1 = \cos(x)$ and $y_2 = \sin(x)$, we can apply the reverse mode AD rule to compute the gradient of $f(x) = \cos(x) + \sin(x)$ with respect to its input x:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial f}{\partial y_2} \frac{\partial y_2}{\partial x},$$

From the output, the following is obtained:

$$\frac{\partial f}{\partial y_1} = 1 \qquad \frac{\partial f}{\partial y_2} = 1$$

Using the chain rule,it is possible to compute the local gradients of y_1 and y_2 with respect to x as follows:

$$\frac{\partial y_1}{\partial x} = -\sin(x) \qquad \frac{\partial y_2}{\partial x} = \cos(x)$$

Substituting these values into the reverse-mode AD equation, the following is obtained:

$$\frac{\partial f}{\partial x} = 1 \cdot (-\sin(x)) + 1 \cdot \cos(x) = \cos(x) - \sin(x)$$

Therefore, the gradient of f with respect to its input x is $\cos(x) - \sin(x)$.

Note that this is equivalent to the result that would be obtained by taking the derivative of f directly: $f'(x) = -\sin(x) + \cos(x)$. However, reverse mode AD can be more efficient for functions with many inputs or intricate computational graphs, as it allows us to reuse the local gradients computed during the backwards pass.

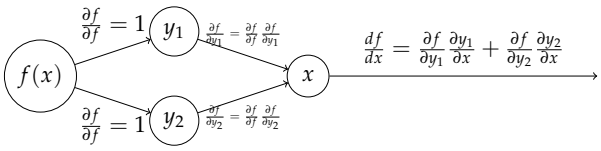


FIGURE 2.2: Diagram showing how reverse mode automatic differentiation works for $f(x) = cos(x) + sin(x)$. $y_1 = cos(x)$ and $y_2 = sin(x)$

2.2 Chebyshev Psuedo Spectral Method

Spectral methods are numerical techniques that transform differential equations or data into the spectral domain using eigenvalues and eigenvectors [25]. Spectral methods provide accurate approximations by representing the solution or data as a combination of orthogonal basis functions, such as Fourier or Chebyshev polynomials. They are used in various fields like fluid dynamics, weather prediction, and signal processing [25]. The S-PIELM algorithm is based on the Chebyshev spectral method. Hence, more emphasis is placed on it in this study. The Chebyshev spectral method approximates functions using a series of Chebyshev orthogonal polynomials. The Chebyshev polynomials $T_N(z)$ of order N are defined by equation (2.10).

$$T_N(z) = cos(Ncos^{-1}(z)) \qquad N \in \mathcal{N} \tag{2.10}$$

Let $u_C(z)$ denote the Chebyshev approximation of the function $u(z)$ at $z = z_i$, then $u_C(z)$ can be defined by equation (2.11), where $L_i(z)$ Lagrange polynomials based on the Chebyshev polynomials given by equation (2.12). Where $c_0 = c_N = 2, c_i = 1$ and $i = 1, \ldots, N-1$.

$$u_C(z) = \sum_{i=0}^{N} u_i L_i(z) \tag{2.11}$$

$$L_i(z) = \frac{(-1)^{w+1}(1-z^2)T_N(z)}{c_i N^2 (z - z_i)} \qquad i = 0, \ldots, N \tag{2.12}$$

The points z_i are denoted as collocation points, which are sampled using equation 2.13. in order to convert the collocation points into a domain from a to b the equation (2.14) is used.

$$\{z_i\} = \left\{ cos\left(\frac{\pi i}{N}\right) \right\}_{i=0}^{N} \tag{2.13}$$

$$x_i = \frac{b-a}{2} z_i + \frac{b+a}{2} \tag{2.14}$$

2.2.1 Approximating Derivatives

Taking equation 2.11 into account, it is also possible to approximate the derivative of the function. By taking the derivative at a specific point z_i, the function's derivative can be approximated using equation (2.15).

$$\frac{d}{dz}\left[u_C(z)\right] = \sum_{j=0}^{N} u_j \frac{dL_j(z_i)}{dz} = \sum_{j=0}^{N} D_{ij} u_j \tag{2.15}$$

$D_{ij} = \frac{dL_j(z_i)}{dz}$ is an element of the differentiation matrix that calculates the derivative of the jth Lagrange polynomial at the ith data point. The differentiation matrix D is then computed using equation (2.16).

$$D_{ij} = \begin{cases} \frac{c_i(-1)^{j+i}}{c_j(z_j - z_i)}, & i \neq j \\ -\frac{z_i}{2(1-z_i^2)}, & (i=j) \neq 0, N \\ \frac{2N^2+1}{6}, & i = j = 0 \\ -\frac{2N^2+1}{6}, & i = j = N \end{cases} \tag{2.16}$$

Thus, it is possible to approximate the derivatives of a function at specific points using $\vec{u}' = D\vec{u}$, where $\vec{u} = [u_0, u_1, \ldots, u_N]^T$.

2.2.2 Solution Method

Consider an example which aims to solve the given differential equation

$$u' = 2u, u(0) = 3$$

with the help of Chebyshev pseudospectral methods.

The given equation can be transformed into a spectral coefficient formulation. The differential operator D, a linear transformation operator defined by $Du = u'$, and the operator for the right-hand side $f(u) = 2u$ are taken into account. The system of equations can then be rewritten as $Lu = f(u)$.

Discretizing the solution u at the collocation points x_i yields $u_i = u(x_i)$ and $f_i = f(u_i)$. Thus, using the established expressions for the differential operators acting on the Chebyshev polynomials, the discretized system at each grid point x_i is as follows:

$$\sum_{j=0}^{N} D_{ij} u_j = f_i \quad i = 0, \ldots, N \tag{2.17}$$

where D_{ij} is the earlier defined differentiation matrix.

The initial condition $u_0 = 3$ is then applied and the above system of equations can be resolved with a numerical solution method like Gauss-Seidel or Conjugate Gradient methods for each time step until the intended accuracy is accomplished or maximum iterations reached.

In the end, a numerical solution for the original differential equation

$$u' = 2u, u(0) = 3$$

at any point within the domain of $[-1, 1]$ or the mapped domain is given by the Chebyshev approximation $u_C(z) = \sum_{i=0}^{N} u_i D_i(z)$, using the conversion equation (2.14).

2.3 Optimisation Algorithms

In this study, gradient-based and heuristic-based methods will be used to optimise the networks. the following section discusses Each optimisation method briefly ad highlights some algorithms that will be used.

We conducted a systematic search of Google Scholar to identify foundational papers on various optimization algorithms. Our query terms included "stochastic gradient descent," "Adam optimization," "L-BFGS," "genetic algorithm," "particle swarm optimization," and "simulated annealing." From the results, we then selected only those works that originally introduced each algorithm.

2.3.1 Gradient based methods

In order to reach an optimal point, gradient-based optimisation algorithms use the derivatives of the function to calculate the direction that leads to the optimal point. Gradient-based methods are popular because of their efficiency. They often perform faster than other optimisation methods and can be easily adjusted for improved performance compared to other methods[26]. However, there are some drawbacks associated with these methods. For instance, they may get stuck in local optimal points, are sensitive to the initial point used, and can be computationally expensive depending on the application. Furthermore, gradient-based methods struggle with non-differentiable functions.

The following gradient-based optimisation algorithms will be used in this study if applicable:

- Stochastic Gradient Descent (SGD)
- Adam Optimisation [27]
- Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BGFS) [28]

SGD and Adam are first-order optimisation algorithms that use only the first derivative of the objective function to determine the optimal point. Adam optimisation is faster to converge than SGD as it incorporates the momentum concept, which uses past and current gradient information to update the algorithm. On the other hand, L-BFGS is a second-order optimisation algorithm that approximates the Hessian using past gradient information, similar to Adam optimisation.

2.3.2 Heuristic based methods

Heuristic-based optimisation algorithms do not rely on derivatives to find the optimal solution. Instead, they use trial-and-error techniques or problem-specific knowledge to search for the best solution. Heuristic-based methods are often used as alternatives to gradient-based methods because of their ability to handle non-differentiable

functions and their flexibility in dealing with real-world constraints and noisy data. Another major benefit is that they do not require explicit mathematical formulations, making them suitable for black-box optimisation of neural networks. Heuristic-based methods lack theoretical guarantees, resulting in slower convergence compared to gradient-based methods. Additionally, the random nature of these algorithms leads to non-reproducible results, making the interpretability of solutions a challenge.

The Heuristic based algorithms that will be used in this study

- Genetic Algorithm (GA) [29]
- Particle Swarm Optimisation (PSO) [30]
- Simulated Annealing (SA) [31]

2.4 Network Architecture and Algorithms

This section discusses both the network architectures and the algorithms used to optimise them in the context of their application. Additionally, a general overview of physics-informed neural networks is provided.

We carried out a systematic Google Scholar search to identify the seminal papers on key neural network architectures. Search terms included "artificial neural networks," "neuroevolution," "compositional pattern producing networks," and "physics-informed neural networks." From the retrieved studies, we then selected only those that first introduced each architecture.

2.4.1 Artificial Neural Network

Artificial neural networks (ANNs) are a type of machine learning algorithm that is modelled after the biological neural networks that make up the human brain. ANNs are made up of interconnected nodes, that process and transmit information through a network of weighted connections (as seen in Figure 2.3). The basic building block of an ANN is a neuron, which takes a set of input values and produces an output value. The neuron calculates a weighted sum of the inputs and applies an activation function to produce the output. The mathematical representation of a neuron can be written as Equation (2.18).

$$y_j = \sigma\left(\sum_i^N w_{i,j} x_i + b_j\right) \tag{2.18}$$

where y_j is the output of the neuron j, x_i are the inputs, $w_{i,j}$ are the weights that determine the importance of each input, b_j is the bias term that shifts the output to the left or right, and σ is the activation function that introduces non-linearity into the output. A deep neural network can be created by connecting multiple layers, each consisting of several nodes, and passing the output from one layer to the next.

The development of ANNs has a long history dating back to the 1940s, when the first artificial neuron model was proposed by Warren McCulloch and Walter Pitts. However, progress was slow due to limited computing power and a lack of efficient training algorithms [32].

In the 1980s, with the introduction of a backpropagation algorithm (also known as Reverse Mode AD), ANNs experienced a resurgence of interest and became increasingly popular as a machine learning tool. Backpropagation is a supervised learning algorithm that adjusts the weights of the connections between nodes in response to errors in the network's output, allowing it to learn from labeled data[33].

Since then, ANNs have been applied to a wide range of tasks, including image and speech recognition, natural language processing, and even game playing. They have also been combined with other machine learning techniques such as reinforcement learning and genetic algorithms to create more powerful and flexible models.

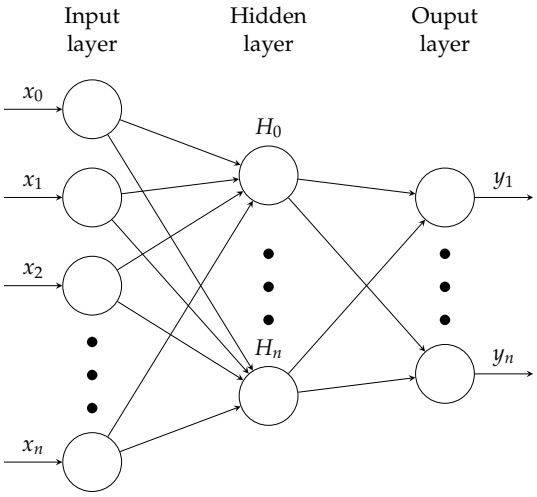


FIGURE 2.3: Example of an ANN

2.4.2 Nuero-Evolution

Neuro-evolution, as discussed in [34], is a subfield of computational intelligence that uses an artificial neural network (ANN) to learn the dynamics of a given system. It draws inspiration from Darwinism [35], which involves populations mating and interacting to promote the development of species better suited for their environment. These concepts are adapted for use in computational models.To find an effective ANN, the method of choice for optimisation metaheuristics varies, but the general process remains the same. A randomly generated network population is initialised and the effectiveness of each network is evaluated using a fitness function. Based on the fitness value, a subset of networks in the population will generate a new set of ANNs that will replace underperforming ones in the next iteration (commonly referred to as a "generation" in the relevant literature). Metaheuristics such as genetic algorithm, differential evolution, particle swarm optimisation, and simulated annealing are commonly employed.

Compared to standard neural-network-based techniques, neuro-evolution differs in that the topology of the network is not fixed (as illustrated in Figure 2.3) and evaluations are typically conducted after a fixed period of exploration. An example of a neural network with a changing topology is shown in Figure 2.4. The structure of these metaheuristics enables distribution of computation across multiple machines, making scaling these algorithms computationally efficient.

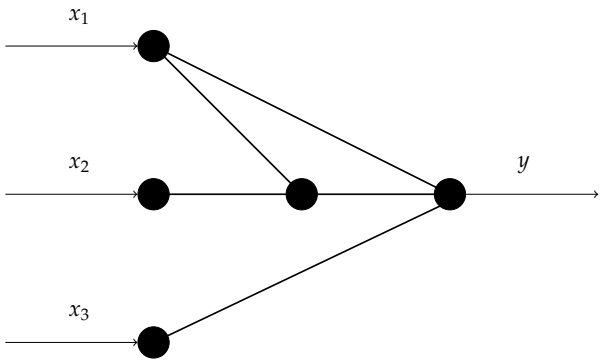


FIGURE 2.4: Example of an ANN without a fixed topology

2.4.3 Compositional Pattern Producing Network

A Compositional Pattern-Producing Network (CPPN) is a type of artificial neural network that generates intricate patterns based on a set of inputs. These networks are an extension to Neuroevolution, with the possiblity of each activation function between neurons may be different. One of the key features of CPPNs is their ability to generate patterns with fractal-like properties. By varying the weights of the connections between nodes, CPPNs can produce patterns that exhibit self-similarity at different scales [36] . This makes them useful for generating intricate and intricate designs that resemble those found in nature. CPPNs have been used in various applications, such as generating artwork, designing 3D shapes, and creating virtual creatures for video games.

2.4.4 Physics Informed Nueral Network

Physics Informed Neural Networks (PINNS) are a type of artificial neural network (ANNs) that aims to interpolate data while satisfying a governing differential equation. This is typically achieved by creating a loss function that minimizes the error within the dataset and reduces the error when applying the solution to the differential equation [37].

The paper proposed by Rassi et al. [38] creates a loss function, the residual of the differential equation. Consider equation 2.19 with the initial condition 2.20, boundary conditions 2.21,2.22 and N being an operator on u which could be linear or nonlinear, the residual of the PDE is defined as

$$\mathcal{L}(u) = \frac{\partial u}{\partial t} + N(u(x,t)) - R(x,t).$$

By denoting $u_{net}(x,t)$ as the output of the network given inputs x,t, the aim of the proposed method was to minimise equation (2.23). the boundary conditions of the differential equations are treated as soft constraints within this setting. The partial derivatives of the neural network with respect to its input (in this case, x,t) can be computed by using automatic differentiation, which computes the derivative in smaller sections using the chain rule and propagates that information backward in the network to calculate the previous derivatives [23].

$$\frac{\partial u}{\partial t} + N(u(x,t)) = R(x,t) \tag{2.19}$$

$$u(x,0) = f(x) \tag{2.20}$$

$$u(0,t) = g(t) \tag{2.21}$$

$$u(L,t) = h(t) \tag{2.22}$$

$$||\mathcal{L}(u_{net})||^2 + ||u_{net}(x,0) - f(x)||^2 + ||u_{net}(0,t) - g(t)||^2 + ||u_{net}(L,t) - h(t)||^2 \tag{2.23}$$

Chapter 3

Gradient Based Methods in Comparision to Heuristic Based Methods

This chapter provides a detailed explanation of how neural networks can be trained to solve differential equations using gradient based and heuristic based optimisation algorithms. The methods and algorithms proposed for this purpose will be discussed comprehensively. The methods discussed within this chapter are used in Chapters 4,6,7 in order to optimise the respective network architectures.

3.1 Gradient Optimisation Algorithms

Gradient-Based Optimisation algorithms are a class of methods that make use of derivative information in order to find optimal solutions. The upcoming section will delve into the gradient-based optimisation algorithms that are intended to be used for network optimisation.

3.1.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a popular optimisation algorithm used to train machine learning models. At each iteration, it randomly selects a subset of the training data (a mini-batch) and computes the gradient of the loss function with respect to the model parameters using only this subset. It then updates the model parameters in the opposite direction of the gradient to minimise the loss function. The process is repeated until convergence. The algorithmic process of the method is given in Algorithm 1. The input $\mathcal{D}$ denotes the training dataset, $f_\theta(x)$ is the model's prediction for input x with parameters θ, ∇_θ denotes the gradient with respect to θ. The algorithm updates the model parameters by subtracting the gradient estimate scaled by the learning rate α.

Algorithm 1 Pseudocode for Stochastic Gradient Descent

procedure SGD($\mathcal{D}$, η, θ_0)
    $\theta \leftarrow \theta_0$
    while termination criterion not met **do**
        Randomly sample a minibatch of m examples from $\mathcal{D}$.
        Calculate the gradient of the minibatch: $\nabla J_{\text{minibatch}}(\theta)$
        Update the parameters: $\theta \leftarrow \theta - \eta \nabla J_{\text{minibatch}}(\theta)$
    return θ

3.1.2 ADAM

Adam optimisation is a gradient-based optimisation algorithm that is commonly used to train deep learning models. It is a combination of two other optimization algorithms, namely, RMSprop and momentum. Adam uses adaptive learning rates for each parameter and can dynamically adjust these learning rates during training. The algorithm is presented in Algorithm 2.

Algorithm 2 Pseudocode for Adam optimisation

1: **procedure** ADAM
2: Initialize parameters w and first and second moment vectors m and v
3: Set hyperparameters: learning rate α, first moment decay rate β_1, second moment decay rate β_2, and small constant ϵ
4: **while** termination criterion not met **do**
5: Get minibatch of m training examples and corresponding targets y
6: Compute gradient $\nabla_w L(w, y)$ on minibatch
7: Update first and second moment estimates:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla_w L(w, y)$$
$$v \leftarrow \beta_2 v + (1 - \beta_2)\nabla_w L(w, y)^2$$

8: Compute bias-corrected first and second moment estimates:

$$\hat{m} \leftarrow \frac{m}{1 - \beta_1^t}$$
$$\hat{v} \leftarrow \frac{v}{1 - \beta_2^t}$$

9: Update parameters:

$$w \leftarrow w - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

3.1.3 L-BGFS

Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) is a gradient-based optimisation algorithm used to solve unconstrained non-linear optimisation problems. It is a quasi-Newton method that approximates the Hessian matrix of the objective function using a limited amount of memory and then applies gradient descent to update the solution. The process is highlighted in Algorithm 3. The objective function is $f(x)$, x_0 is the initial solution, m is the maximum number of updates to store in memory, and ϵ is the termination threshold. The algorithm iteratively updates the solution by computing a descent direction using an approximation of the Hessian matrix, and then performs a backtracking line search to find a suitable step size. The memory is updated with the most recent updates to the solution and gradient, and the Hessian approximation is updated using these values. The algorithm terminates when the gradient norm falls below the threshold.

Algorithm 3 L-BFGS Algorithm

procedure L-BFGS($f(x), x_0, m, \epsilon$)
  Initialize $x_k = x_0$
  Initialize $\nabla f_k = \nabla f(x_k)$
  Initialize $H_k = I$
  Initialize s_i, y_i, ρ_i for $i = k-m, \ldots, k-1$ (empty at $k = 0$)
  Initialize $i = k$
  while termination criterion not met **do**
    $d_k \leftarrow -H_k \nabla f_k$
    $\alpha_k \leftarrow$ backtracking line search to find suitable α_k
    $s_k \leftarrow \alpha_k d_k$
    $x_{k+1} \leftarrow x_k + s_k$
    $\nabla f_{k+1} \leftarrow \nabla f(x_{k+1})$
    $y_k \leftarrow \nabla f_{k+1} - \nabla f_k$
    $\rho_k \leftarrow \frac{1}{y_k^T s_k}$
    $H_k y_k \leftarrow s_k$
    for $j = k-1, k-2, \ldots, k-m$ **do**
      $\alpha_j \leftarrow \rho_j s_j^T H_j \nabla f_j$
      $\nabla f_{j+1} \leftarrow \nabla f_j + \alpha_j y_j$
      $s_j \leftarrow s_j - \alpha_j d_j$
    $H_k \leftarrow (y_k^T s_k)/(y_k^T y_k) I + (\sum_{j=k-m}^{k-1} \rho_j s_j s_j^T)$
    $i \leftarrow i + 1$
  return x_k

3.2 Hueristic Optimisation Algorithms

Hueristic Optimisation algorthms are a class of methods that make use of trial & error in order to find a ssuboptimal solution. This section will dive into the heuristic-based optimisation algorithms that are intended to be used for network optimisation.

3.2.1 Genetic Algorithm

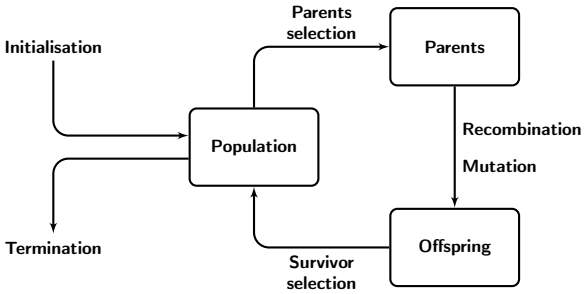


FIGURE 3.1: Genetic Algorithm Flowchart

Genetic Algorithm (GA) is a popular metaheuristic that mimics natural selection to find the global optima. The algorithm starts with a randomly initialised population of candidate solutions, which evolve over generations; the flow diagram of the algorithm is shown in Figure 3.1). The fittest candidates are carried over to the next

generation, whereas new candidates are created through crossover. Occasionally, solutions may be mutated for better results [39].

Selection

In GA, the selection of good candidate solutions is crucial to generate new solutions. Two commonly used selection operators are Roulette and Tournament Selection [39]. Roulette Selection samples a random solution based on fitness values, but requires all fitness values to be non-negative. Tournament selection selects the two best solutions from a sample of the population to use as parents for crossover, addressing the limitations of roulette selection.

Crossover

Crossover is an operator that generates new candidate solutions by combining subsets of the parent solutions. A commonly used crossover operator is single-point crossover. As seen in Figure 3.2, an index i is selected, and from this index to the index of the last gene, the values of the genes are swapped, resulting in the formation of two new candidate solutions.

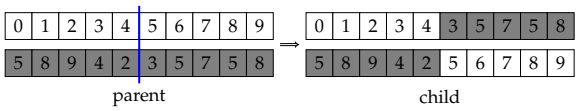


FIGURE 3.2: Example of crossover in GA

Mutation

The mutation operator perturbs candidate solutions to diversify the population search space, an example being bit flipping in binary encoding problems [39]. Mutation occurs at a small rate, similar to the chance of natural biological organisms having their genetic information mutated for better or worse solutions. Figure 3.3 provides an example of applying this operator to a candidate solution.

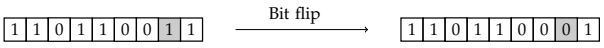


FIGURE 3.3: Visualisation of the bit-flip.

GA Algorithm

In this research, GA is used to optimise the different ANNs. The general algorithmic process of the algorithm is given by Algorithm 4.

3.2.2 Particle Swarm Optimisation

PSO is a swarm-based metaheuristic that uses social behaviour to search for optimal solutions, with each particle's movement influenced by its local and global best solutions [39]. Position update is determined using equation (3.1).

$$x_{t+1}^i = x_t^i + c_1 r_1 (p_t^i - x_t^i) + c_2 r_2 (g_t - x_t^i). \tag{3.1}$$

Algorithm 4 Pseudocode for Genetic Algorithm

```
procedure GA
    for each agent i = 1, ..., N do
        Initialise agent_i's gene x_i within search space
        Assign a fitness value to agent_i
        Add agent to population set
    while termination criterion not met do
        Select candidates within population for crossover
        Generate new candidate solutions using selected candidates
        Update the population of candidate solutions
        Mutate subset of population
        Update the fitness scores of the candidate solutions
    return Fittest member within the population
```

The agent's personal best solution and the global best solution are indicated by p_t^i and g_t, respectively, with t representing the iteration count and i representing the agent number. The coefficients r_1 and r_2 are uniformly sampled values between $[0, 1)$, and the weight parameters are denoted by c_1 and c_2. Figure 3.4 provides a visual representation of how PSO updates the agent's position. The values of c_1 and c_2 used in the experiments will be set to 2.

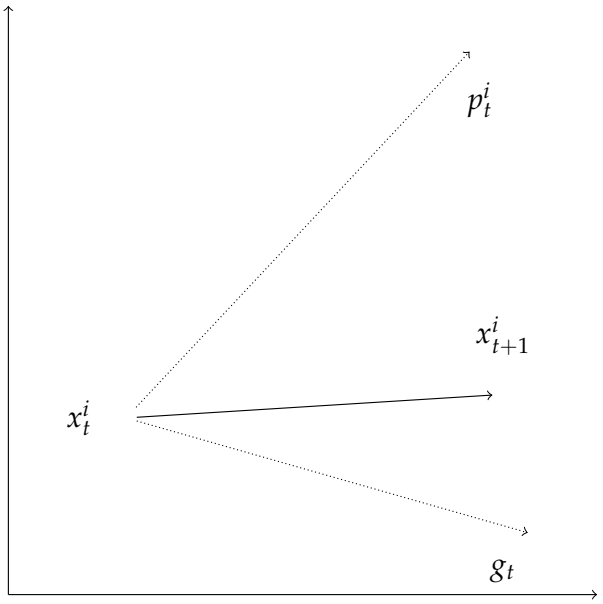


FIGURE 3.4: PSO Update

PSO Algorithm

PSO is used in varying frameworks to determine the optimal network, therefore, a general overview of the implementation of the algorithm is given in Algorithm 5.

Algorithm 5 Pseudo-code for Particle Swarm Optimisation

procedure PSO
  for each agent $i = 1, \ldots, N$ **do**
    Initialise agents position x_i within search space
    Set agents personal best position p_i to current position
    if $fitness(p_i) < fitness(g)$ **then**
      $g \leftarrow p_i$
  while termination criterion not met **do**
    for each agent $i = 1, \ldots, M$ **do**
      Update agents position using equation 3.1
      if $fitness(x_i) < fitness(p_i)$ **then**
        $p_i \leftarrow x_i$
      if $fitness(p_i) < fitness(g)$ **then**
        $g \leftarrow p_i$
  return g

3.3 Automatic Differentiation Approach

The next section describes a method for solving a differential equation using automatic differentiation. The process is described in Algorithm 6, which involves computing the weight gradients of an artificial neural network using reverse mode automatic differentiation and computing the derivative of the network with respect to its input using forward mode automatic differentiation. Gradients with respect to the inputs are used to calculate the loss function $\mathcal{L}$.

Algorithm 6 Solving Differential Equations using gradient information

procedure TRAIN(model, initial/boundary conditions, differential equation, optimizer, learning rate)
  Initialize model parameters θ
  Define input data **X**: initial/boundary conditions and differential equation
  while termination criterion not met **do**
    Sample random points from domain to enforce differential equation
    Sample random points from boundary to enforce boundary conditions
    Compute loss function $\mathcal{L}$ using input data and model predictions
    Compute gradients of loss function w.r.t model parameters using AD
    Update model parameters using optimizer and learning rate
  return trained model as solution to differential equation

Chapter 4

Physics Informed Neural Networks

In this chapter, we provide a detailed discussion and implementation of Physics-Informed Neural Networks (PINNs). The optimisation algorithms discussed in Chapter 3 are then used to optimise and measure the solution quality of these PINNs. This chapter draws heavily on Raissi's seminal paper introducing PINNs [37].

4.1 Introduction

Physics-Informed Neural Networks represent a ground-breaking advancement in scientific machine learning. Unlike traditional neural networks that rely solely on data for training, PINNs incorporate physical laws directly into their learning process. This integration allows PINNs to leverage the underlying physics governing a problem, enhancing their predictive power and generalisation capabilities, especially in scenarios where data may be scarce or noisy.
PINNs embed differential equations, which describe physical phenomena such as fluid dynamics, heat transfer and wave propagation, directly into the neural network's architecture. This embedding allows PINNs to enforce physical constraints and ensure that the network's outputs are consistent with known scientific principles. This approach not only improves prediction accuracy but also provides physically meaningful solutions, which are essential in many engineering and scientific applications.
The training process of a PINN involves minimising a loss function that accounts for both the data and the physics. This is achieved by defining a composite loss function that includes terms for data mismatch and the residuals of the differential equations. By simultaneously minimising these terms, the neural network learns to approximate the solution to the differential equation while fitting the observed data.

Loss Function in PINNs

The loss function in a Physics-Informed Neural Network typically comprises two main components: the data loss and the physics loss. The data loss, $\mathcal{L}_{\text{data}}$, measures the discrepancy between the network's predictions and the observed data. The physics loss, $\mathcal{L}_{\text{physics}}$, quantifies the residuals of the differential equations, ensuring that the network's predictions satisfy physical laws.
Let $\mathbf{u}(\mathbf{x}, t; \mathbf{w})$ be the neural network's prediction for the physical quantity of interest, where $\mathbf{x}$ represents the spatial coordinates, t is time, and $\mathbf{w}$ denotes the network's parameters. The loss function $\mathcal{L}$ can be expressed as:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{physics}},$$

where

$$\mathcal{L}_{\text{data}} = \frac{1}{N}\sum_{i=1}^{N} \left|\mathbf{u}(\mathbf{x}_i, t_i; \mathbf{w}) - \mathbf{u}_{\text{true}}(\mathbf{x}_i, t_i)\right|^2,$$

and

$$\mathcal{L}_{\text{physics}} = \frac{1}{M}\sum_{j=1}^{M} \left|\mathcal{F}(\mathbf{u}(\mathbf{x}_j, t_j; \mathbf{w}))\right|^2.$$

Here, $\mathbf{u}_{\text{true}}$ denotes the observed data, $\mathcal{F}$ represents the differential operator defined by physical laws, and N and M are the number of data points and collocation points, respectively.
By minimising this composite loss function, the PINN learns to approximate the solution to the differential equations while fitting the observed data, providing a powerful tool for solving intricate scientific problems.

4.2 Methodology

This section provides an in-depth look at the implementation details of the networks. The inputs and outputs of the network were determined by the number of independent and dependent variables. To ensure comparability among the networks, the number of hidden nodes was fixed at 200. This configuration was chosen to maintain an equivalent number of parameters across all methods. Experiments for each equation were conducted by first sampling points, followed by training the model (i.e minimising the loss function along the points). once training is complete the model is then benchmarked against the ground truth solution.

4.2.1 Point Sampling

During training, points on the boundary were sampled uniformly where applicable, whilst points within the domain were sampled using Lattice hypercube sampling.

4.2.2 Hyperparameter tuning

In order to acheive results in a meaningful timeframe, we employed a combination of trial-and-error and grid search to determine the optimal hyperparameters.

4.3 Results

In the following section PINNs are evaluted over the differential equations mentioned above.

4.3.1 Logistic Equation

The results of the different optimisation techniques used when learning the solution to the logistic equation are presented, followed by a discussion. The results from hyperparameter optimisation can be found in Table 4.1

Alg.	LR	Epo.	Loss	Hist.	N	M	μ	c_1	c_2	T_{init}	α
SGD	0.01	5000	2.8878×10^{-4}	—	—	—	—	—	—	—	—
Adam	0.01	5000	3.3093×10^{-4}	—	—	—	—	—	—	—	—
LBFGS	0.10	500	7.7727×10^{-8}	20	—	—	—	—	—	—	—
GA	—	200	1.2003×10^{-1}	—	50	20	0.10	—	—	—	—
PSO	—	50	3.9282×10^{-5}	—	100	—	—	2.0	2.0	—	—
SA	—	—	3.6393×10^{-1}	—	—	—	—	—	—	10000	0.80

TABLE 4.1: Results of hyperparameter Tuning on the logistic equation (rounded to four significant digits). Abbreviations: Alg. – algorithm; LR – learning rate; Epo. – epochs; Loss – final loss; Hist. – history size (L-BFGS); N – population size (GA, PSO); M – offspring size (GA); μ – mutation rate (GA); c_1, c_2 – cognitive/social coefficients (PSO); T_{init} – initial temperature (SA); α – cooling rate (SA).

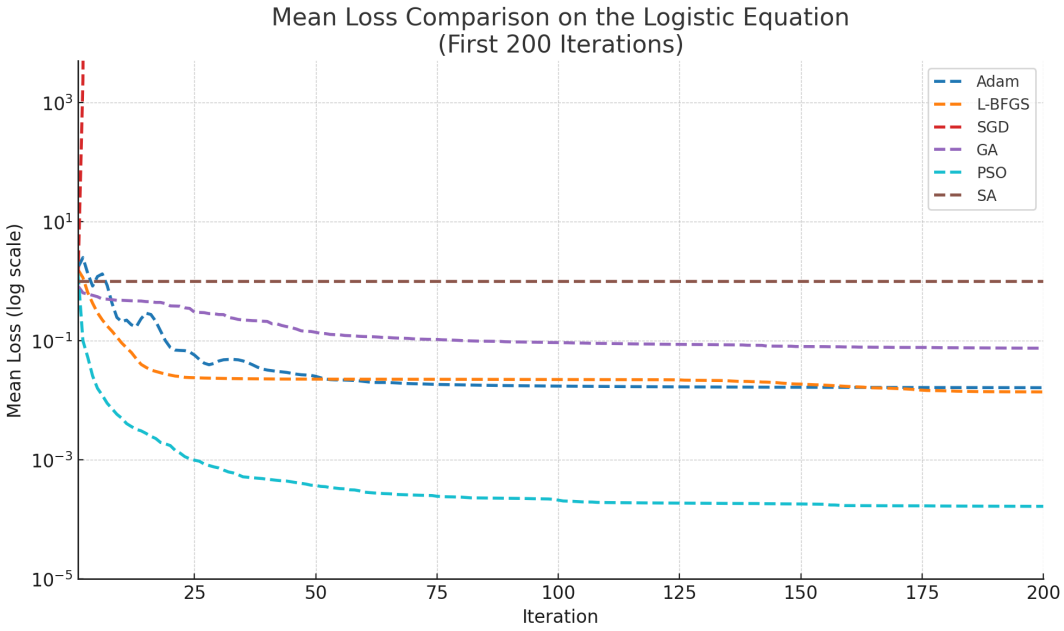


FIGURE 4.1: Convergence Analysis of Different optimisation techniques on the logistic equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.1 and measuring the mean loss

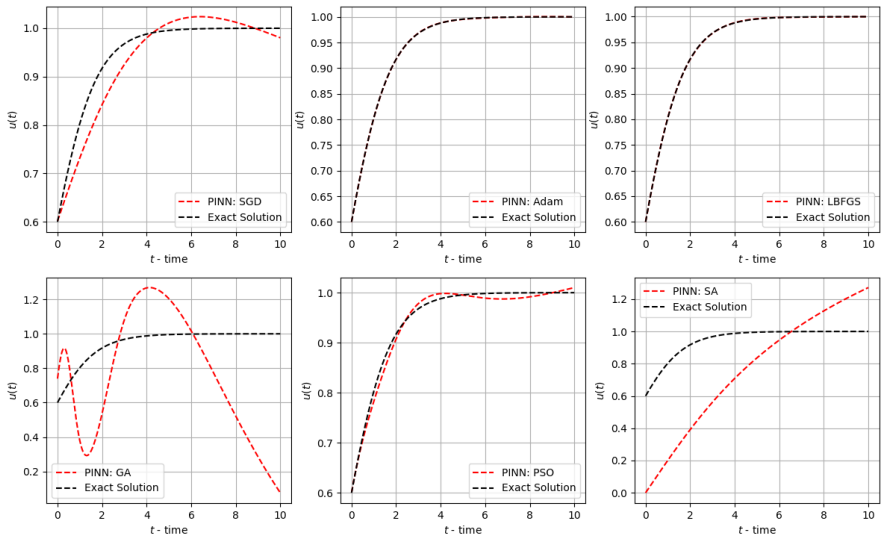


FIGURE 4.2: Comparision Analysis of Different optimisation techniques on the logistic equation

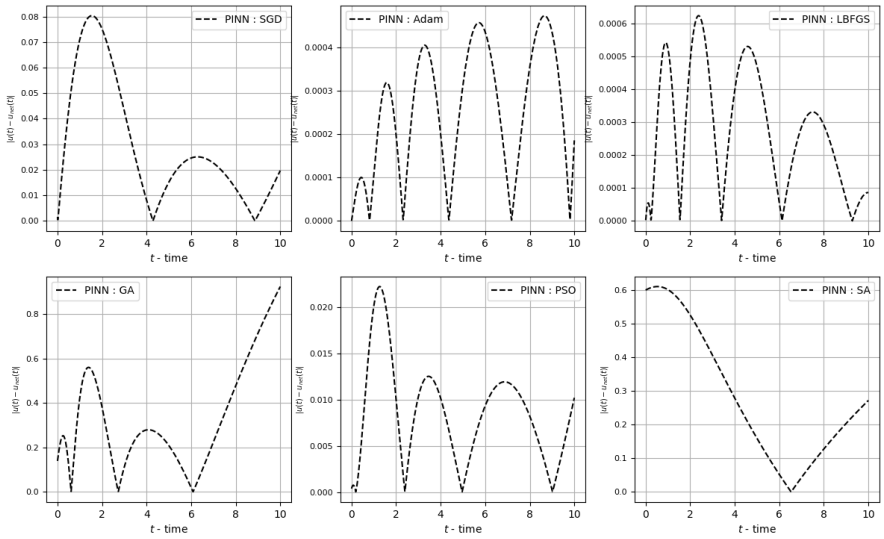


FIGURE 4.3: Error Analysis of Different optimisation techniques on the logistic equation

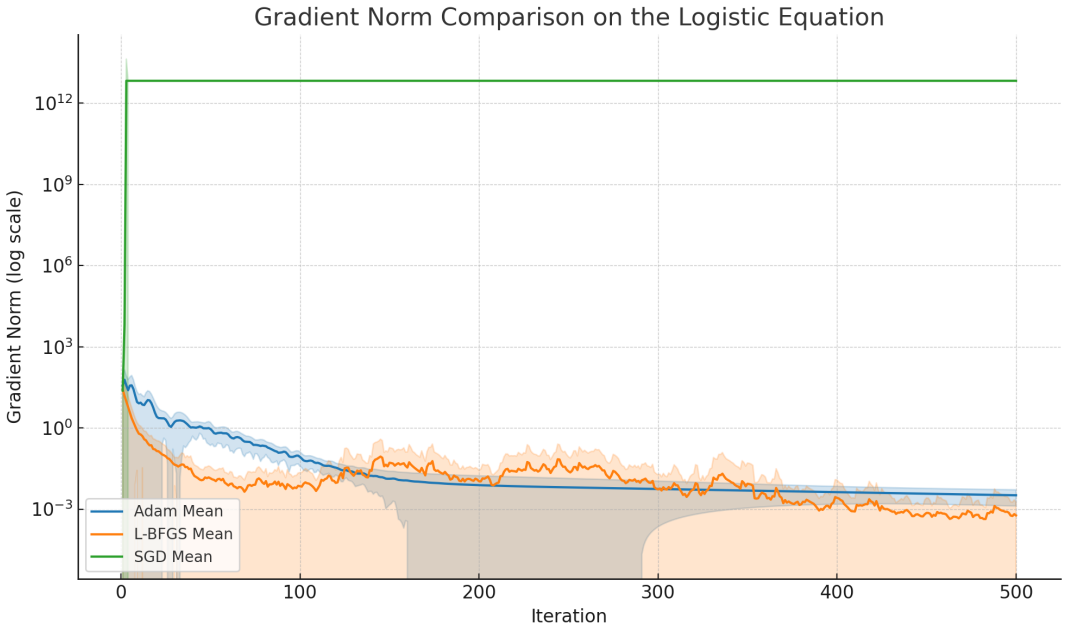


FIGURE 4.4: Gradient Analysis of Different gradient based optimisation techniques on the logistic equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.1 and measuring the norm of the gradient vector

Method	Average Error
SGD	0.02907
Adam	0.00025
LBFGS	0.0003
GA	0.33359
PSO	0.00856
SA	0.28258

TABLE 4.2: Average errors for different optimization methods for the logistic equation

Figures 4.1–4.3 summarize the convergence and error behaviour of six optimization methods applied to the logistic PINN. SGD converges the slowest, while Adam and LBFGS exploit adaptive- and quasi-Newton updates to reach low loss in under 500 iterations. Heuristic methods (GA, PSO, SA) either stall or fluctuate without reliably reducing the error.
Table 4.2 reports the average pointwise error over $t \in [0, 10]$. LBFGS (0.00030) and Adam (0.00025) achieve the smallest errors—nearly two orders of magnitude below SGD (0.02907). PSO (0.00856) performs moderately, whereas GA (0.33359) and SA (0.28258) lag significantly. These results confirm that gradient-based optimizers are best suited for training PINNs on smooth differential equations. Table 4.4 shows the gradient magnitudes over 30 trials of the logistic equation. We observe that SGD experienced exploding gradients—likely because its fixed step size can cause it to overshoot on steep slopes (especially when t is near zero). In contrast, L-BFGS exhibited vanishing gradients: by using an approximate Hessian and adaptive step sizing, it takes progressively smaller steps as it approaches the optimum, driving

the gradient toward zero.Adam did not suffer from vanishing or exploding gradients, likely because its built-in momentum and oscillatory update behaviour help stabilize the updates.

4.3.2 Lorenz System of Equations

Below are the results of the different optimisation techniques used when learning the solution to the Lorenz system of equations, followed by a discussion.

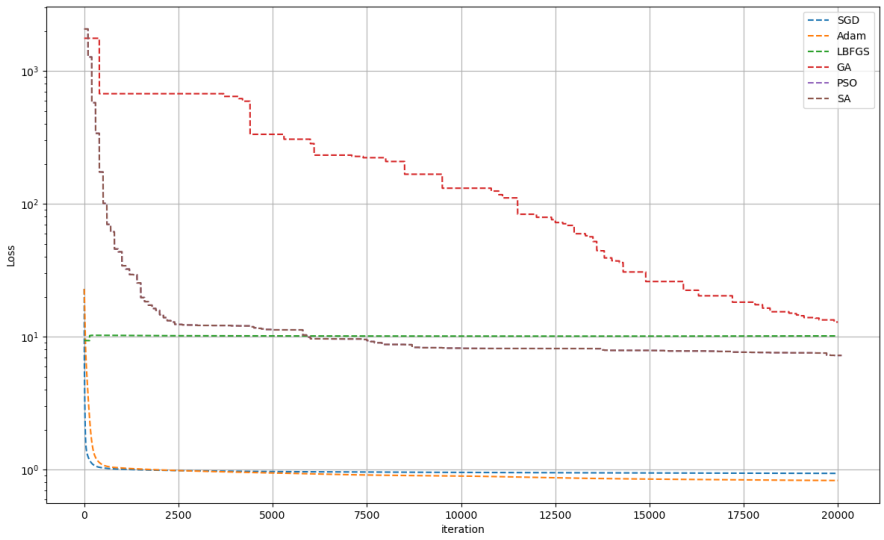


FIGURE 4.5: Convergence Analysis of Different optimisation techniques on the Lorenz system of equations

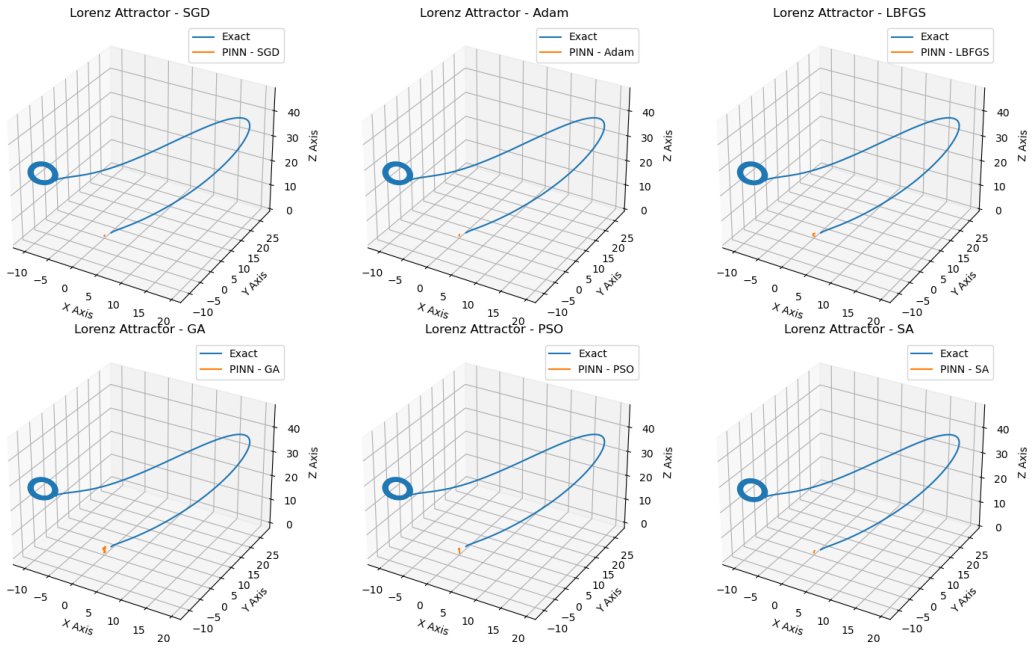


FIGURE 4.6: Comparision Analysis of Different optimisation techniques on the Lorenz system of equations

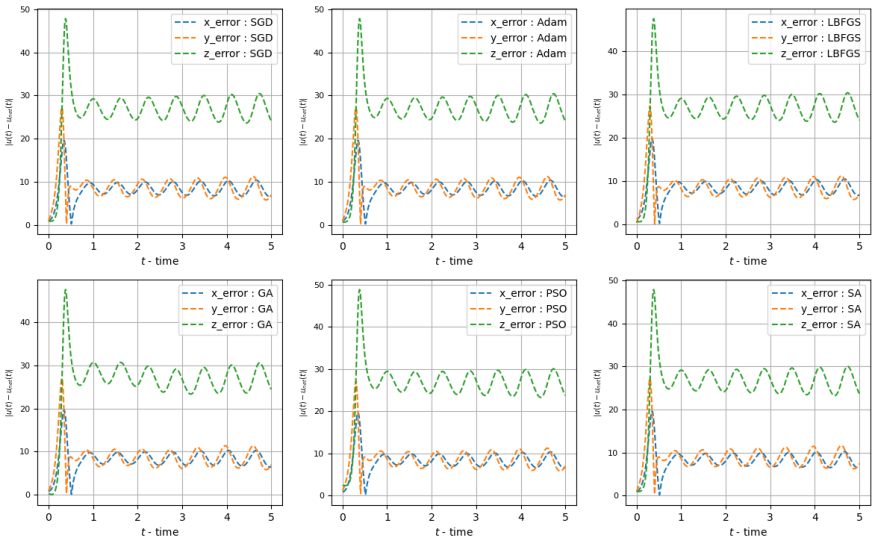


FIGURE 4.7: Error Analysis of Different optimisation techniques on the Lorenz system of equations

Method	Average Error
SGD	14.32755
Adam	14.33211
LBFGS	14.28024
GA	14.37748
PSO	14.32636
SA	14.33094

TABLE 4.3: Average errors for different optimization methods

The analysis of the convergence rates in Figure 4.1 shows that both SGD and Adam methods achieve the fastest convergence. However, as depicted in Figures 4.7 and 4.6, all methods exhibit significantly high error rates. Particularly in Figure 4.6, it is evident that the model generates a trivial solution. This can be attributed to the intricate dynamics of the Lorenz system, which complicates the task of deriving a non-trivial solution.

Table 4.3 indicates that the errors associated with all the optimization algorithms were approximately 14.3. This uniformity in error could suggest that each of the six algorithms was converging towards a trivial solution. While technically a solution, it does not offer insight into the system's dynamics.

4.3.3 Diffusion Equation

Below are the results of the different optimisation techniques used when learning the solution to the diffusion equation, followed by a discussion.
The results of hyperparameter tuning is shown in Table 4.4. in this case the history length of lbgfs was not tuned as it dramatically increased computational time within the given problem.

Algorithm	lr	Epoch	Final Loss	N	M	μ	c_1	c_2	Init Temp	α
SGD	0.01	5000	0.1421054751	—	—	—	—	—	—	—
adam	0.01	5000	0.0004889534	—	—	—	—	—	—	—
lbfgs	0.10	500	7.8140793×10^{-6}	—	—	—	—	—	—	—
GA	—	200	0.6474032998	50	20	0.1	—	—	—	—
PSO	—	200	6.2352385521	100	—	—	2.0	2.0	—	—
SA	—	200	16.0483398438	—	—	—	—	—	1000	0.8

TABLE 4.4: Best hyperparameter settings per algorithm for the diffusion equation

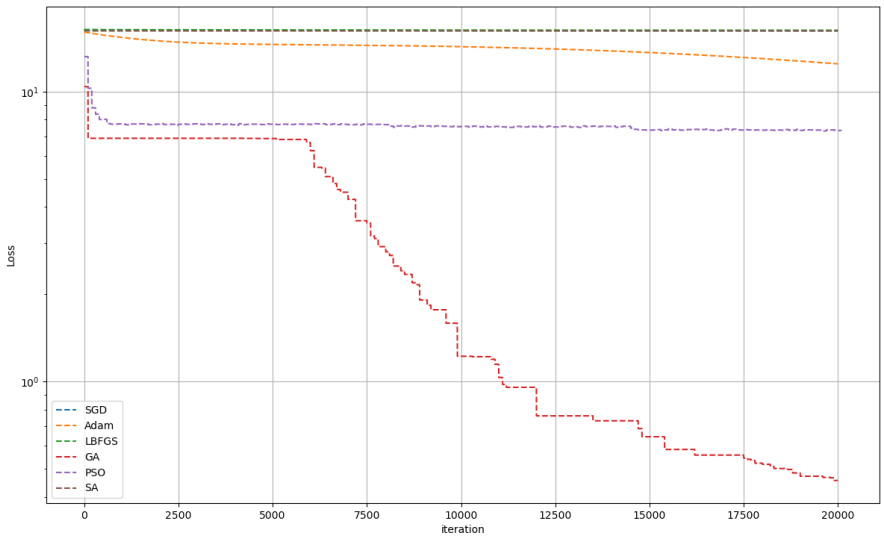


FIGURE 4.8: Convergence Analysis of Different optimisation techniques on the diffusion equation

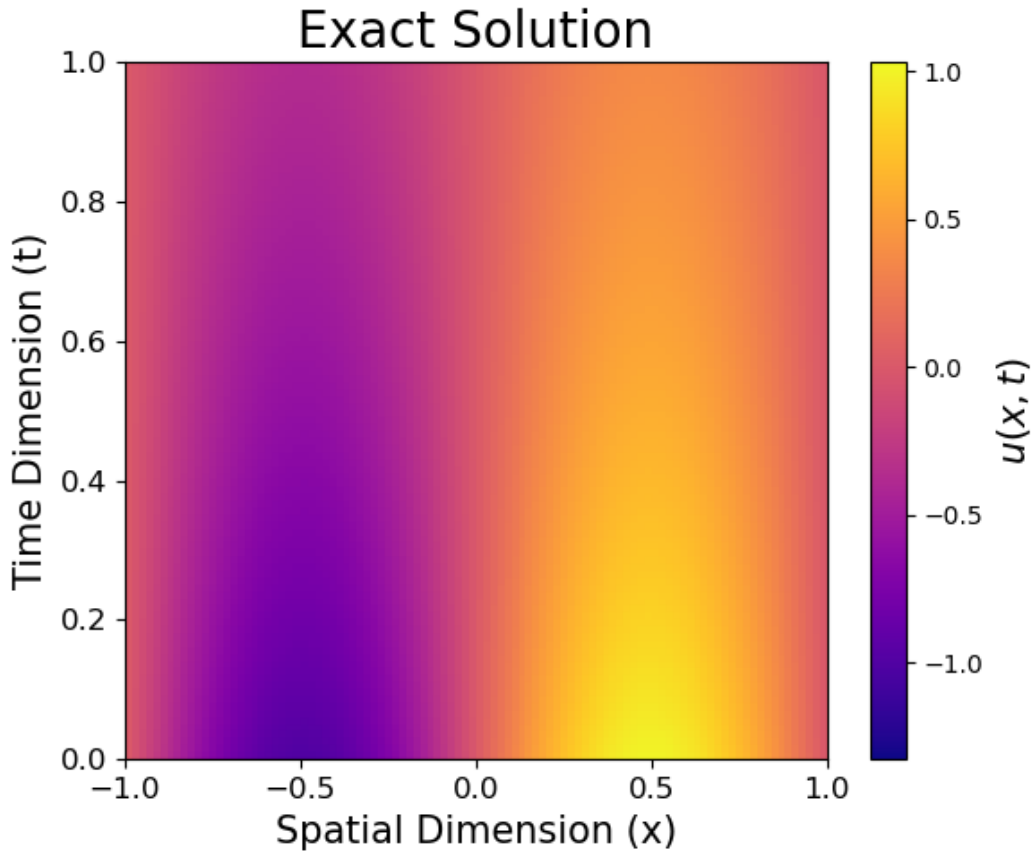


FIGURE 4.9: Ground Truth Solution to the Diffusion equation

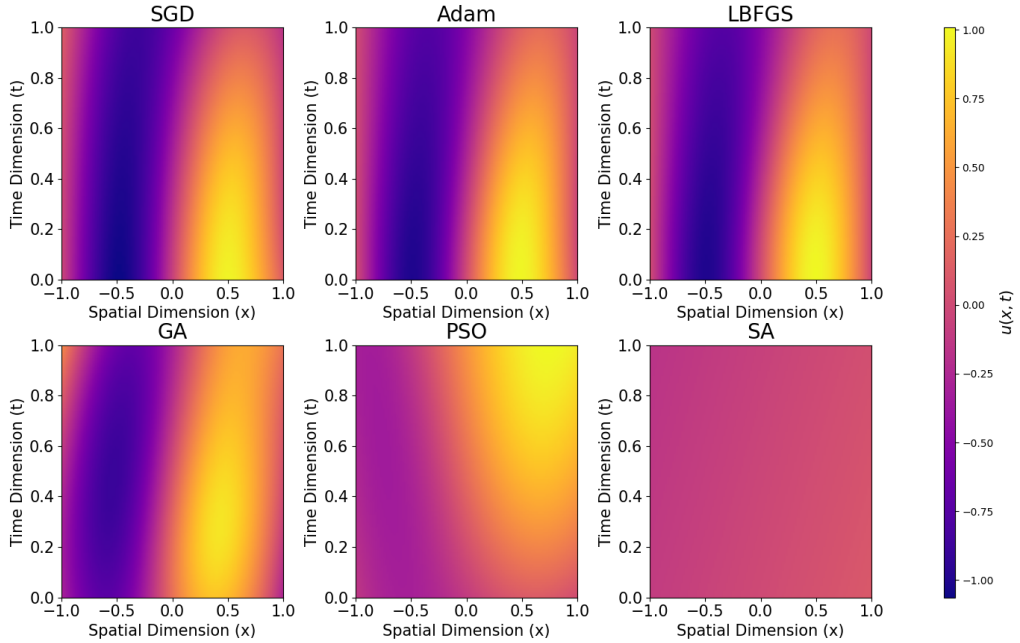


FIGURE 4.10: Comparision Analysis of Different optimisation techniques on the diffusion equation

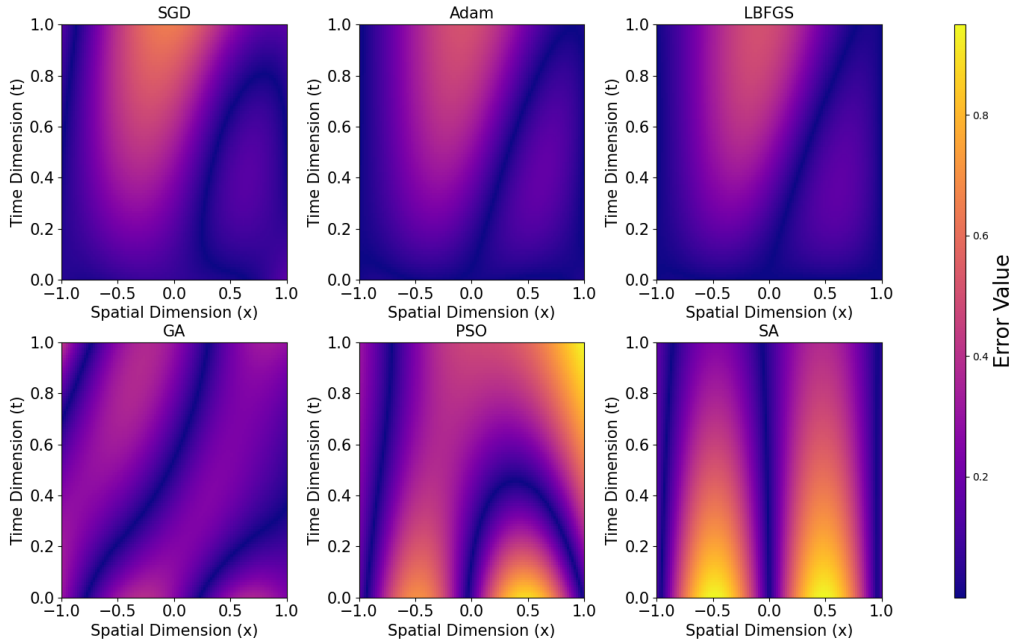


FIGURE 4.11: Error Analysis of Different optimisation techniques on the diffusion equation

Method	**Average Error**
SGD	0.19709
Adam	0.16058
LBFGS	0.16025
GA	0.19045
PSO	0.32412
SA	0.35643

TABLE 4.5: Average errors for different optimization methods

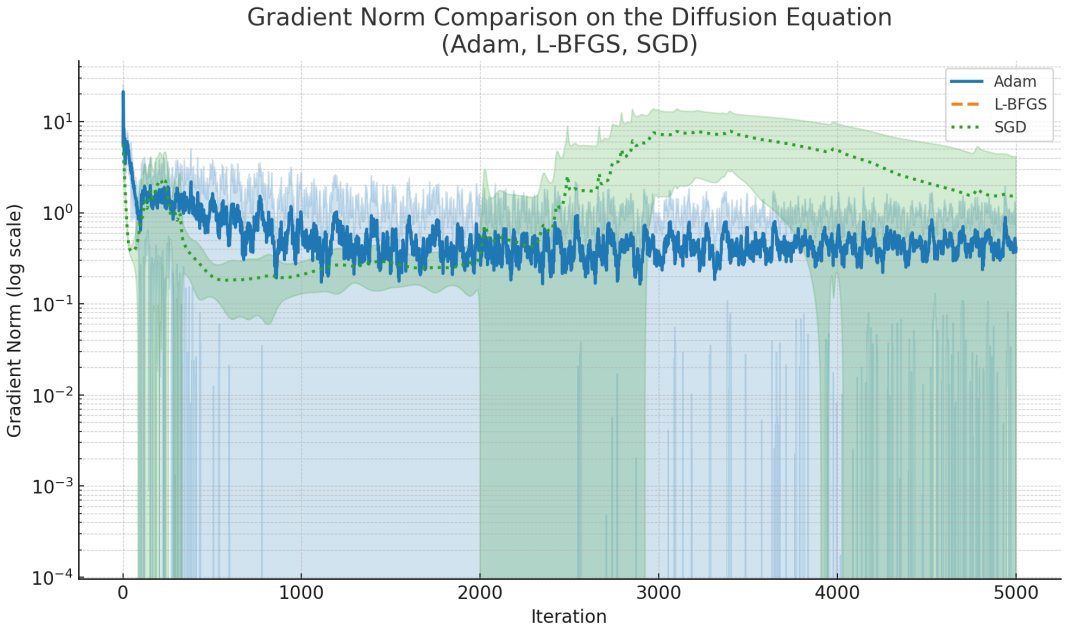


FIGURE 4.12: Gradient Analysis of Different gradient based optimisation techniques on burgers equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.4 and measuring the norm of the gradient vector. L-BFGS results returned NaN consistently across the trials; hence, the gradient results were ommitted

Looking at the convergence curves in Figure 4.8, the gradient-based optimizers (SGD, Adam, LBFGS) still lead in speed by directly following the loss gradient, while the Genetic Algorithm (GA) achieves a comparable convergence rate thanks to its population diversity, but does not surpass them.

Figure 4.10 reveals that only Simulated Annealing (SA) repeatedly collapses to the trivial zero solution. SA's reliance on a single-solution trajectory and temperature-driven random walks makes it prone to becoming trapped in the zero-loss basin, unlike population methods.

From Figure 4.11, LBFGS attains the lowest final loss by using approximate second-order information, which provides precise descent directions and helps escape shallow minima. GA follows closely—its pseudo-random variation and uniform initial sampling allow it to rival gradient methods despite lacking curvature estimates. SGD's noisy updates offer moderate performance and occasional escapes from local minima, while Adam's adaptive step-sizes can cause oscillations around the optimum, resulting in a slightly higher error.

Table 4.5 quantifies these effects: LBFGS achieves the best average error, GA matches it closely, PSO and SGD occupy the middle ground, Adam is somewhat worse due to sensitivity to learning-rate scheduling, and SA performs worst by converging to the trivial solution.

Observing the Results in Figure 4.12, Adam's built-in momentum and adaptive step sizes keep its gradients in a stable, moderate band, while plain SGD's fixed global learning rate overshoots steep regions and produces large, noisy swings before slowly settling. L-BFGS, however, repeatedly hits NaNs because its quasi-Newton Hessian approximation becomes ill-conditioned on the PDE's stiff, highly curved landscape—when the line search cannot find a valid descent direction due to

a singular curvature estimate, the update breaks down and yields undefined (NaN) values.

4.3.4 Burgers' Equation

Below are the results of the different optimisation techniques used when learning the solution to Burgers' equation, followed by a discussion.

Alg.	LR	Epo.	Loss	Hist.	N	M	μ	c_1	c_2	T_{init}	α
SGD	0.01	5000	2.9409×10^{-2}	—	—	—	—	—	—	—	—
Adam	0.01	5000	3.8046×10^{-4}	—	—	—	—	—	—	—	—
LBFGS	0.10	100	2.2308×10^{-5}	—	—	—	—	—	—	—	—
GA	—	50	7.2328×10^{-1}	—	100	50	0.30	—	—	—	—
PSO	—	200	8.1341×10^{-3}	—	100	—	—	2.0	1.0	—	—
SA	—	200	1.1429×10^{-1}	—	—	—	—	—	—	1000	0.80

TABLE 4.6: Best parameter settings and final losses for each algorithm on Burgers' equation. Abbreviations: Alg. – algorithm; LR – learning rate; Epo. – epochs; Loss – final loss; Hist. – history size (L-BFGS); N – population size (GA, PSO); M – offspring size (GA); μ – mutation rate (GA); c_1, c_2 – cognitive/social coefficients (PSO); T_{init} – initial temperature (SA); α – cooling rate (SA).

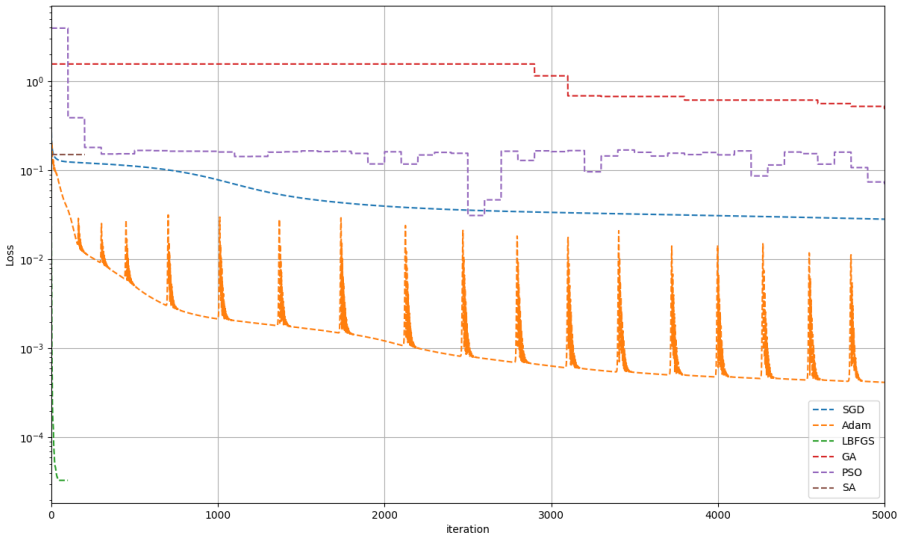


FIGURE 4.13: Convergence Analysis of Different optimisation techniques on Burgers' equation

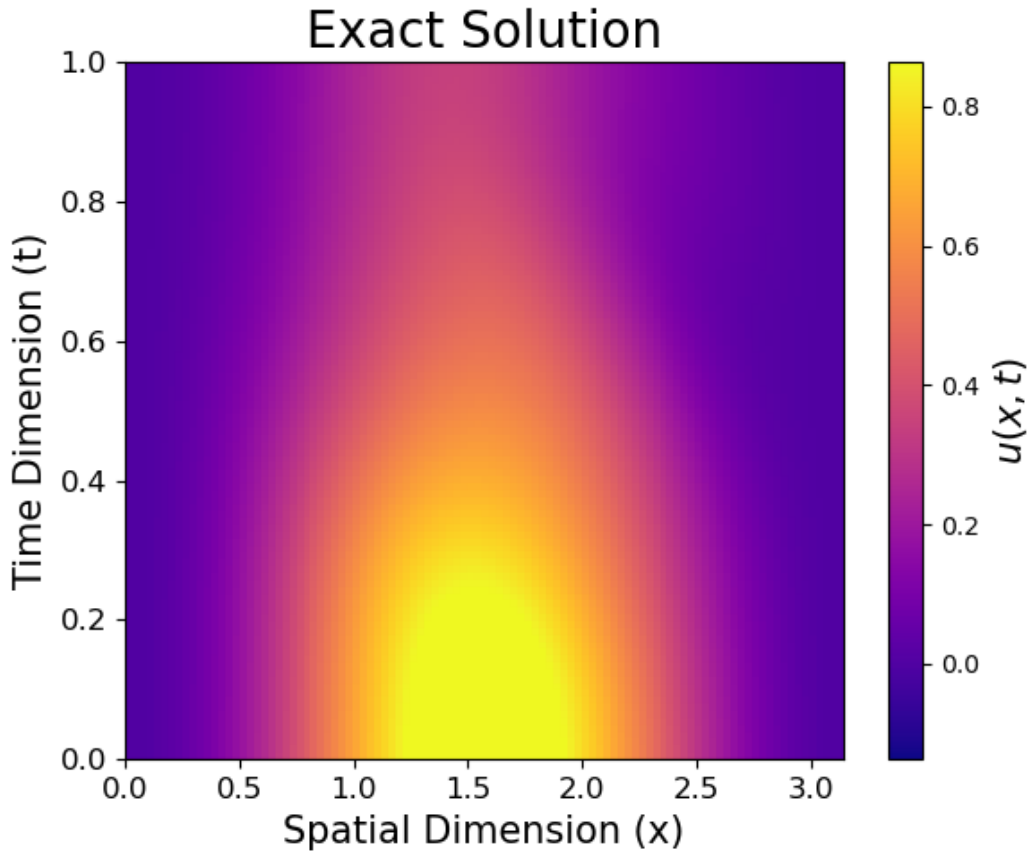


FIGURE 4.14: Ground Truth Solution to Burgers' equation

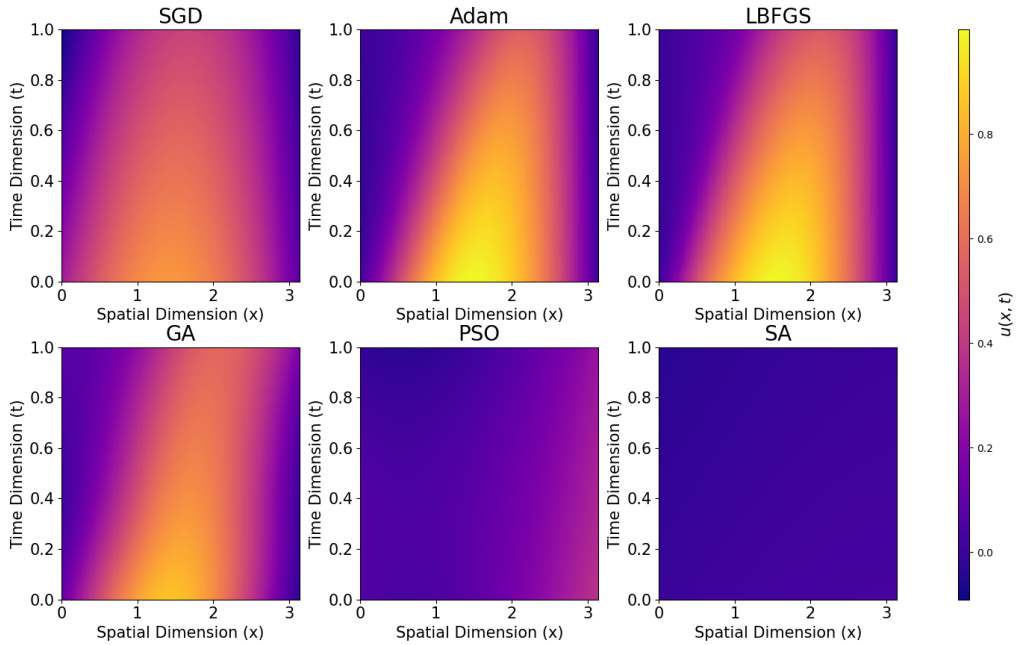


FIGURE 4.15: Comparision Analysis of Different optimisation techniques on Burgers' equation

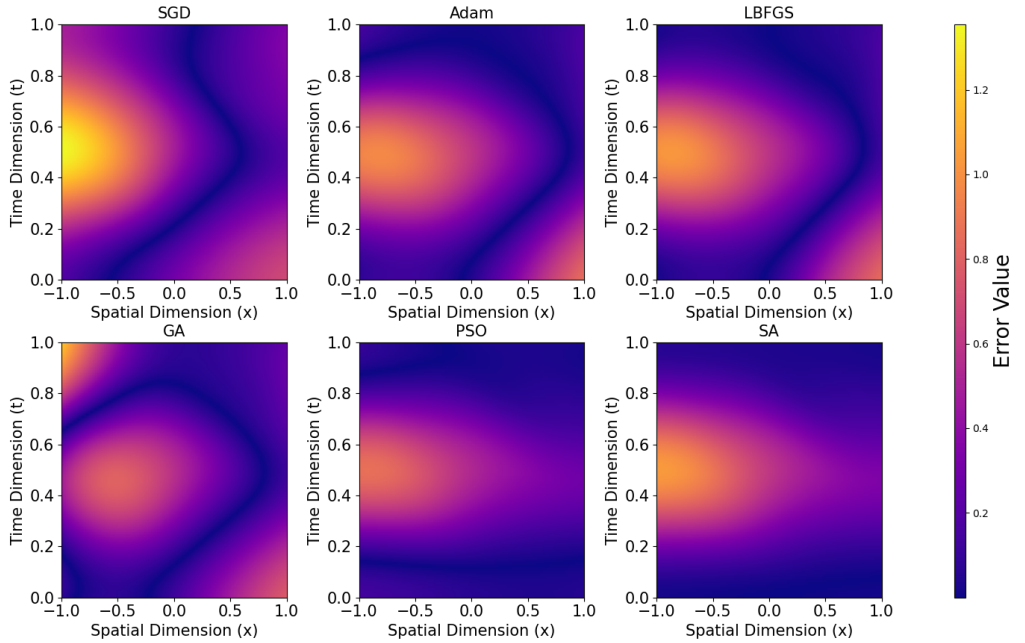


FIGURE 4.16: Error Analysis of Different optimisation techniques on Burgers' equation

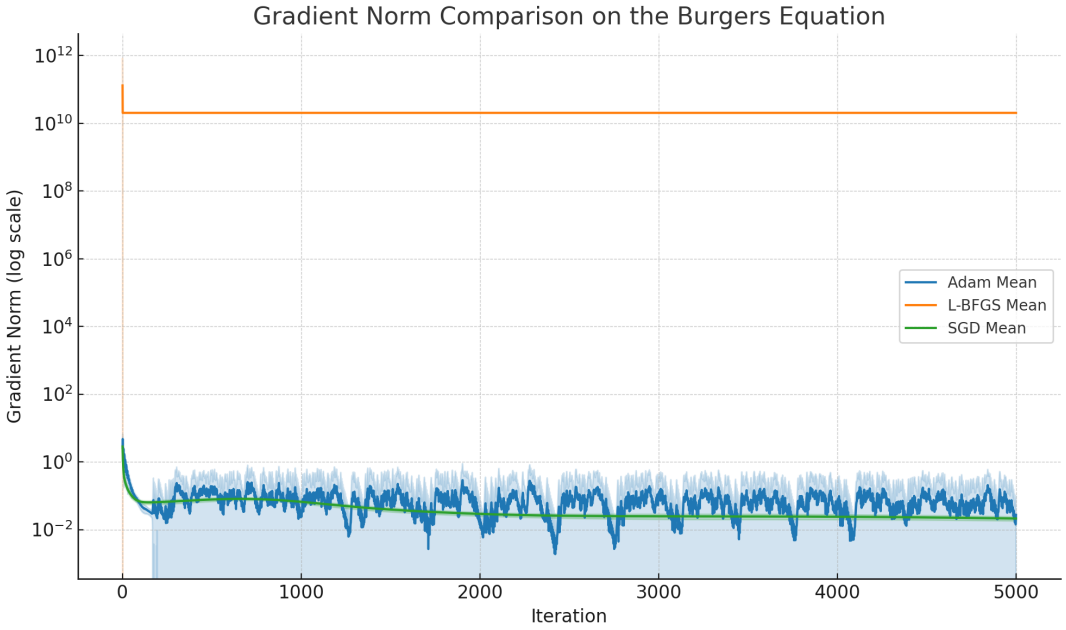


FIGURE 4.17: Gradient Analysis of Different gradient based optimisation techniques on burgers equation. the results were obtained by running 30 experiments using the hyperparameters obtained in Table 4.6 and measuring the norm of the gradient vector

As Figure 4.13 shows, the heuristic methods (GA and PSO) unexpectedly collapsed to trivial, zero-everywhere solutions, whereas the gradient-based algorithms (SGD, Adam, LBFGS) maintained meaningful descent and converged to non-zero profiles. This counter-intuitive outcome appears to be driven by two factors: first, the lack of gradient guidance in population-based search caused premature convergence to a single, suboptimal basin; and second, the random perturbations in GA

Method	Average Error
SGD	0.39339
Adam	0.31057
LBFGS	0.32172
GA	0.29099
PSO	0.24732
SA	0.31129

TABLE 4.7: Average errors for different optimization methods

and PSO were insufficient to escape that basin once all candidate solutions had decayed toward zero.
In Figure 4.15, the zero-solution bias of heuristic methods is made explicit—both GA and PSO, like Simulated Annealing (SA), repeatedly settle on the trivial solution across trials. By contrast, the gradient-based optimizers leverage local curvature information to avoid the all-zero trap, producing a richer approximation of the true solution.
The error-analysis in Figure 4.16 confirms these observations: LBFGS achieves the lowest final loss, followed by Adam and then SGD, while GA and PSO incur the highest errors. The surprise reversal—heuristics faring worse than their gradient-based counterparts—highlights the critical importance of directional information when training neural approximators on PDE-constrained tasks.

Figure 4.17 shows the gradient norms over 30 trials on the Burgers equation. Here, L-BFGS's gradients sit at a huge, nearly constant value—its curvature-based updates are too aggressive on this stiff problem, so the gradients effectively "explode" and never settle down. In contrast, SGD's gradient norm steadily shrinks over time, reflecting its fixed step size taking consistent, if slow, steps toward smaller updates without wild spikes. Adam stays in the middle: its gradients hover in a moderate band with mild up–and-down oscillations, as its adaptive rates and momentum together keep the updates both stable and responsive.

4.4 Conclusion and Discussion

Overall, across all four test problems, gradient-based optimizers consistently outperformed heuristic methods in both convergence speed and final accuracy. In the logistic PINN, L-BFGS and Adam reached low losses within 500 iterations and attained average errors of 3×10^{-4} and 2.5×10^{-4} respectively—nearly two orders of magnitude better than SGD (2.9×10^{-2}) and dramatically superior to GA (0.33) or SA (0.28). Similarly, for the diffusion and Burgers equations, L-BFGS achieved the lowest errors, with Adam and SGD following, while GA, PSO, and SA often collapsed to the trivial zero solution or fluctuated without reliable error reduction. The only notable exception was that GA approached gradient-based performance on the diffusion problem due to its diverse population sampling, but still did not surpass second-order methods. In contrast, all six algorithms converged to a trivial solution on the Lorenz system—highlighting that when the problem landscape admits a zero-loss basin, even gradient methods cannot guarantee a non-trivial solution without additional problem structure.

The trade-offs among these optimizers hinge on the balance between convergence speed, final accuracy, and computational overhead:

- **L-BFGS**: delivers the fastest, most accurate convergence by using an approximate Hessian, but at the expense of higher per-step memory and computation—and it can break down (yield NaNs) on very stiff or highly curved loss surfaces.
- **Adam**: offers a robust middle ground: its momentum and adaptive step sizing stabilize updates (avoiding both vanishing and exploding gradients) with only modest additional cost over SGD, at the price of slight oscillations around the optimum and sensitivity to learning-rate scheduling.
- **SGD**: is the cheapest per iteration but suffers from slow convergence, noisy or exploding gradients, and generally higher error floors unless carefully tuned.
- **Heuristic methods** (GA, PSO, SA): incur significant computational cost due to population evaluations or random walks

In summary, derivative-based schemes are the most effective for physics-informed neural models on smooth differential equations, while the choice among them depends on whether one prioritizes the ultimate accuracy and speed of L-BFGS, the stability of Adam, or the simplicity of SGD—and generally disfavors expensive heuristics for these tasks.

These observations suggest a hybrid strategy: initialize or periodically augment a gradient-driven training schedule with population-based search steps. Such a combination can harness the fast descent of gradient methods together with the exploratory robustness of heuristics, potentially achieving both rapid convergence and meaningful, nontrivial solutions.

Chapter 5

Spectral Based Physics Informed Extreme Learning Machine

Solving partial differential equations using neural networks—specifically, Physics-Informed Neural Networks (PINNs)—has recently led to substantial improvements, unlocking aspects of numerical solutions that were previously difficult or impossible to achieve. This chapter introduces a variant of PINNs, the Spectral Physics-Informed Extreme Learning Machine (S-PIELM), tailored to solve stationary and time-dependent, linear and non-linear differential equations. The study reveals that S-PIELM achieves comparable or superior accuracy to PINNs across diverse problem domains while demanding fewer computational resources.
Spectral-ELM approaches integrate orthogonal bases with the analytical training framework of extreme learning machines to achieve spectral accuracy and rapid convergence when solving differential equations [40]. Variants of the ELM algorithm have been adapted to compute high-dimensional PDE solutions efficiently by fixing random input-to-hidden weights and solving only for output weights, thereby avoiding the solution of non-sparse systems [41]. Additionally, the synergy of machine-learning-based spectral expansions with learned basis functions has been shown to handle complex domains in time-dependent PDEs with high fidelity [42].

5.1 Mathematical Derivation of Proposed Method

The S-PIELM algorithm represents a hybrid of the Spectral Method and the Extreme Learning Machine (ELM) Algorithm. The algorithm's derivation revolves around substituting the solution approximation in equation (2.11). This approximation entails a combination of known solution values at equal-spaced points multiplied by a basis function. The algorithm transitions to an approximation strategy utilising the neural network outputs multiplied by a basis function, as defined by equation (5.1). In this context, $\widetilde{u}_i$ signifies the neural network's output corresponding to the input z_i.

Drawing upon the foundational principles of the ELM algorithm, we find that $\widetilde{u}_i$ is given by $\vec{H}(z_i)^T\vec{\beta}$. Here, $\vec{H}_{n\times 1}$ and $\vec{\beta}_{n\times 1}$ denote the dimensions of these vectors, where n represents the count of hidden nodes within the ELM architecture.

$$u(z) \approx \sum_{i}^{N} \widetilde{u}_i L_i(z) \tag{5.1}$$

$$\approx \sum_{i=0}^{N} \vec{H}(z_i)^T \vec{\beta} L_i(z) \tag{5.2}$$

The derivatives of $u(z)$ are approximated in a similar manner to equation (2.15), resulting in equation (5.3)

$$\left.\frac{du}{dz}\right|_{z=z_i} \approx \sum_{j=0}^{N} \widetilde{u}_j \frac{dL_j(z_i)}{dz} \tag{5.3}$$

$$\approx \sum_{j=0}^{N} \vec{H}(z_j)^T \vec{\beta} \frac{dL_j(z_i)}{dz} \tag{5.4}$$

$$\approx \sum_{j=0}^{N} D_{ij} \vec{H}(z_j)^T \vec{\beta} \tag{5.5}$$

The derivatives across all collocation points can be rewritten in the form of (5.6)

$$\mathbf{U}' = D\mathbf{H}(\vec{z})\vec{\beta} \tag{5.6}$$

Here $\mathbf{U}'$ is defined as $[u'_0, u'_1, \ldots, u'_N]$ and $\mathbf{H}(\vec{z})$ is defined as $\left[\vec{H}(z_0), \vec{H}(z_1), \ldots, \vec{H}(z_N)\right]^T$ with dimensions $n \times N$, where n is the number of hidden nodes and N the number of collocation points. In the ELM paper introduced by Huang et al. [43], the weights of the hidden layer are fixed and, hence, the only unknown in equations (5.1) , (5.3) and (5.6) is $\vec{\beta}$. By utilising the approximation (5.6) with the given differential equation, the objective is to transform the system into the concise form denoted by equation (5.7).

$$\mathbf{A}\mathbf{H}(\vec{z})\vec{\beta} = \mathbf{b} \tag{5.7}$$

Since this method aims to reduce the residual of the differential equation, $\mathbf{A}$ denotes the matrix representation of the residual of the differential equation and $\mathbf{b}$ is an $N \times 1$ vector of zeros expect at the boundary and initial condition collocation points (see [25] for a review on spectral methods). Assuming the initial and boundary conditions have been applied, the optimal weights $\vec{\beta}$ are then computed using equation (5.8), which is obtained by solving equation (5.7) using the Moore Penrose pseudo inverse, denoted by †.

$$\vec{\beta} = (\mathbf{A}\mathbf{H}(\vec{z}))^{\dagger}\mathbf{b} \tag{5.8}$$

5.2 Solution Method

In the following subsection gives a detailed solution method for each of the models introduction in section 1.2.

5.2.1 Logistic Equation

Before applying the proposed approach, it is necessary to linearize the logistic equation, which is a non-linear differential equation [44]. By using Taylor series linearisation equation (5.9) is obtained. The solution is then updated iteratively until a convergence criterion is met. u^k is the current known solution with u^0 being an initial guess.In this case $\vec{\beta}$ will be randomly initialised and the initial solution will be sampled from the network. the aim is then to solve (5.9) for u^{k+1}.

$$\frac{du^{k+1}}{dt} = ru^{k+1} - 2ru^k u^{k+1} + r(u^k)^2 \tag{5.9}$$

By denoting $u = u^{k+1}$ and $p = u^k$ equation (5.9) can be rewritten as

$$\frac{du}{dt} = ru - 2rpu + rp^2 \tag{5.10}$$

$$= [r - 2rp]\, u + rp^2 \tag{5.11}$$

Which is linear in terms of u.By assuming $\mathbf{U}$ is the solution to equation (5.10) across a set of N collocation points, equation (5.10) can be rewritten in matrix form, given by equation (5.12), where D is the $N \times N$ differentiation matrix, $\mathbf{I}$ is the $N \times N$ identity matrix, $\mathbf{P}$ is a $N \times N$ diagonal matrix whose entries are from the values of p and $\widetilde{\mathbf{P}}$ an $N \times 1$ matrix where $\widetilde{\mathbf{P}}_i = p_i^2$. Which can be rewritten as equation (5.13), which incorporates the weights of the extreme learning machine. It is important to note that the differentiation matrix is scaled by $\frac{2}{b-a}$ beforehand, i.e. $D = \frac{2}{b-a}D$ to scale the differentiation matrix in the domain of the differential equation.

$$[D - r\mathbf{I} + 2r\mathbf{P}]\, \mathbf{U} = r\widetilde{\mathbf{P}} \tag{5.12}$$

$$[D - r\mathbf{I} + 2r\mathbf{P}]\, \mathbf{H}(\vec{t})\vec{\beta} = r\widetilde{\mathbf{P}} \tag{5.13}$$

By denoting $\mathbf{A} = D - r\mathbf{I} + 2r\mathbf{P}$ and $\mathbf{b} = r\widetilde{\mathbf{P}}$, the system of equations simplify to equation (5.7). In order to apply the initial condition, the last row in $\mathbf{A}$ and $\mathbf{b}$ need to be replaced as the points are generated in reverse order when using the guass lobatto points as seen in equation (5.14). Once the boundary condition has been included in the system, the optimal weights $\vec{\beta}$ are computed using equation (5.8). The pseudocode to solve the logistic equation is provided in algorithm 7.

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \dots & A_{0,N_t} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \mathbf{H}(\vec{t})\vec{\beta} = \begin{bmatrix} b_0 \\ \vdots \\ u_0 \end{bmatrix} \tag{5.14}$$

5.2.2 Diffusion Equation

The diffusion equation is linear, Hence no training loop or linearisation is required. the optimal weights of the network will be determined at once. Since there are 2 independent variables, the approximation begins around t and then x. starting with t we have equation (5.15)

$$u(x,t) \approx \sum_{j=0}^{Nt} \mathbf{H}(x,t_j)\vec{\beta}L_j(t) \tag{5.15}$$

Algorithm 7 Solving Logistic Equation using S-PIELM approach

procedure TRAIN(u0,a,b,N,n)
    Create a single layer neural network
    Create random weights **W** between the input nodes and hidden nodes
    Create random weights $\vec{\beta}$ between the hidden nodes and output nodes
    Generate $\vec{z}$ and $\vec{t}$ using equations (2.13) and (2.14) respectively
    Generate D using equation (2.16)
    $D = \frac{2}{b-a}D$
    while Convergence criterion not met **do**
        Generate current guess $\vec{p} = \mathbf{H}(\vec{t})\vec{\beta}$
        Generate **A**,**b** using equation (5.13)
        Apply the boundary conditions to **A**,**b**
        Update $\vec{\beta}$ using equation (5.8)

Taking the derivative with respect to t at t_i we have equation (5.16), where d_{ij} is the differentation value with respect to time. this is determined using equation (2.16).

$$\left.\frac{\partial u}{\partial t}\right|_{t=t_i} \approx \sum_{j=0}^{Nt} \mathbf{H}(x,t_j)\vec{\beta}\left[\left.\frac{dL_j}{dt}\right|_{t=t_i}\right] \tag{5.16}$$

$$\approx \sum_{j=0}^{Nt} d_{ij}\mathbf{H}(x,t_j)\vec{\beta} \tag{5.17}$$

By denoting $f(x,t) = e^{-t}\left(\sin(\pi x) - \pi^2 \sin(\pi x)\right)$ the diffusion equation can be rewritten as (5.18).

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - f(x,t) \tag{5.18}$$

Using the results obtained from equation (5.16) at a specific $t = t_i$, the ELM approximation can be fit to the differential equation, resulting in equation (5.19).

$$\sum_{j=0}^{Nt} d_{ij}\mathbf{H}(x,t_j)\vec{\beta} = \frac{\partial^2}{\partial x^2}\left[\mathbf{H}(x,t_i)\vec{\beta}\right] - f(x,t_i) \tag{5.19}$$

Since the intial condition is known, equation (5.19) can be written as (5.20), where U_{N_t} is the initial condition across the collocation points in x.

$$\frac{\partial^2}{\partial x^2}\left[\mathbf{H}(\vec{x},t_i)\vec{\beta}\right] - \sum_{j=0}^{Nt-1} d_{ij}\mathbf{H}(\vec{x},t_j)\vec{\beta} = f(\vec{x},t_i) - d_{i,N_t}U_{N_t} \tag{5.20}$$

Writing equation (5.20) in matrix form using the differentiation matrix in x, D results in equation (5.21)

$$D^2\mathbf{H}(\vec{x},t_i)\vec{\beta} - \sum_{j=0}^{Nt-1} d_{ij}\mathbf{H}(\vec{x},t_j)\vec{\beta} = f(\vec{x},t_i) - d_{i,N_t}U_{N_t} \tag{5.21}$$

$$\left[D^2 - \sum_{j=0}^{Nt-1} d_{ij}\mathbf{I}\right]\mathbf{H}(\vec{x},t_i)\vec{\beta} = f(\vec{x},t_i) - d_{i,N_t}U_{N_t} \tag{5.22}$$

The result from equation (5.21) can be reduced to the form seen in equation (5.7), where the submatrices in $\mathbf{A}$ are defined using equation (5.23) and the sub vectors of $\mathbf{b}$ are defined using equation (5.24).

$$A_{ij} = \begin{cases} D^2 - d_{ii}\mathbf{I}, & i = j \\ -d_{ij}\mathbf{I}, & i \neq j \end{cases} \tag{5.23}$$

$$\mathbf{b}_i = f(\vec{x},t_i) - d_{i,N_t}U_{N_t} \tag{5.24}$$

the boundary conditions should be applied on the submatrices $\mathbf{A}_{ii}$ and $\mathbf{b}_i$. The pseudocode to solve the diffusion equation is provided in algorithm 9

Algorithm 8 Solving Diffusion Equation using S-PIELM approach

procedure TRAIN(x_0,x_{N_t},t_0,t_{N_t},N_x,N_t)
    Create a single layer neural network
    Create random weights $\mathbf{W}$ between the input nodes and hidden nodes
    Create random weights $\vec{\beta}$ between the hidden nodes and output nodes
    Generate D and d using equation (2.16)
    $d = \frac{2}{t_0 - t_{N_t}} d$
    $D = \frac{2}{x_0 - x_{N_x}} D$
    Generate $\mathbf{A}$,$\mathbf{b}$ using equation (5.21)
    Apply the intial and boundary conditions to $\mathbf{A}$,$\mathbf{b}$
    solve for $\vec{\beta}$ using equation (5.8)

5.2.3 Burgers Equation

Burgers equation is a non-linear partial differential equation, Hence linerization is required before applying the proposed approach [44]. By using the Taylor Series linearisation, equation (5.25) is obtained. The solution is then updated iteratively, until a convergence criterion is met. u_r is the current known solution with u_0 being the initial guess which is obtained by randomly initialising $\vec{\beta}$. The aim is then to solve equation (5.25) for u_{r+1}

$$\frac{\partial u_{r+1}}{\partial t} + \frac{\partial u_r}{\partial x}u_{r+1} + u_r\frac{\partial u_{r+1}}{\partial x} - u_r\frac{\partial u_r}{\partial x} = v\frac{\partial^2 u_{r+1}}{\partial x^2} \tag{5.25}$$

By denoting $u = u_{r+1}$ and $p = u_r$ equation (5.26) is obtained. p' is the derivative of p with respect to x

$$\frac{\partial u}{\partial t} - v\frac{\partial^2 u}{\partial x^2} + p\frac{\partial u}{\partial x} + p'u = pp' \tag{5.26}$$

Since there are 2 independent variables, the approximation begins around t and then x. starting off with t we have equation (5.27)

$$u(x,t) \approx \sum_{j=0}^{Nt} \mathbf{H}(x,t_j)\vec{\beta}L_j(t) \tag{5.27}$$

By taking the derivative with respect to t at t_i we have equation (5.28), where d_{ij} is the differentation value with respect to time. this is determined using equation (2.16).

$$\left.\frac{\partial u}{\partial t}\right|_{t=t_i} \approx \sum_{j=0}^{Nt} \mathbf{H}(x,t_j)\vec{\beta}\left[\left.\frac{dL_j}{dt}\right|_{t=t_i}\right] \tag{5.28}$$

$$\approx \sum_{j=0}^{Nt} d_{ij}\mathbf{H}(x,t_j)\vec{\beta} \tag{5.29}$$

Using the results obtained from equation (5.28) at a specific $t = t_i$, the ELM approximation can be fit into the differential equation, resulting in equation .

$$\sum_{j=0}^{Nt} d_{ij}\mathbf{H}(x,t_j)\vec{\beta} - v\frac{\partial^2}{\partial x^2}\left[\mathbf{H}(x,t_i)\vec{\beta}\right] + p\frac{\partial}{\partial x}\left[\mathbf{H}(x,t_i)\vec{\beta}\right] + p\mathbf{H}(x,t_i)\vec{\beta} = p'p \tag{5.30}$$

Since the intial condition is known, equation (5.30) can be written as equation (5.31), where U_{N_t} is the initial condition across the collocation points in x.

$$\left[vD^2 - pD - p' - \sum_{j=0}^{Nt-1} d_{ij}\mathbf{I}\right]\mathbf{H}(\vec{x},t_i)\vec{\beta} = d_{i,N_t}U_{N_t} - pp' \tag{5.31}$$

The result from equation (5.31) can be reduced to the form seen in equation (5.7), where the submatrices in $\mathbf{A}$ are defined using equation (5.32) and the sub vectors of $\mathbf{b}$ are defined using equation (5.33).

$$A_{ij} = \begin{cases} vD^2 - pD - p' - d_{ii}\mathbf{I}, & i = j \\ -d_{ij}\mathbf{I}, & i \neq j \end{cases} \tag{5.32}$$

$$\mathbf{b}_i = d_{i,N_t}U_{N_t} - p_i p_i' \tag{5.33}$$

Algorithm 9 Solving Burgers Equation using S-PIELM approach

procedure TRAIN(x_0,x_{N_t},t_0,t_{N_t},N_x,N_t)
    Create a single layer neural network
    Create random weights $\mathbf{W}$ between the input nodes and hidden nodes
    Create random weights $\vec{\beta}$ between the hidden nodes and output nodes
    Generate D and d using equation (2.16)
    $d = \frac{2}{t_0 - t_{N_t}}d$
    $D = \frac{2}{x_0 - x_{N_x}}D$
    while Convergence criterion not met **do**
        Generate current guess $\vec{p} = \mathbf{H}(\vec{x},\vec{t})\vec{\beta}$
        Generate $\mathbf{A}$,$\mathbf{b}$ using equation (5.31)
        Apply the intial and boundary conditions to $\mathbf{A}$,$\mathbf{b}$
        solve for $\vec{\beta}$ using equation (5.8)

5.3 Results

To assess the computational efficiency and precision of the proposed method, the networks were trained in half of the evaluation domain and subsequently validated throughout the entire domain. This process aimed to evaluate the network's ability to generalise to unseen data. Different numbers of collocation points were employed during training to investigate the influence of collocation point density on the neural network's performance. The following results cannot be directly compared with standard numerical techniques because these standard techniques require refinement (either through mesh refinement or higher-order approximations) to achieve increasingly accurate numerical simulations, whereas network-based techniques do not rely on meshes during evaluation and standard numerical techniques are not conducive to fitting initial observed data and extrapolating based on equation dynamics.

For each method, a network with a single layer of 200 hidden nodes was used in order to access the performance of the proposed method. 200 nodes were chosen to strike a balance between accuracy and computational performance speed. The $tanh(z)$ function, was chosen within this experiment for its zero-centered nature facilitating the expression of negative values and is smooth, a critical feature since the network's derivatives will be utilized in the loss function. The weights of both networks were sampled using Xavier initialisation for PINNs **datta2020survey**, and for S-PIELM the weights were sampled from a standard normal distribution.

5.3.1 Logistic Equation Results

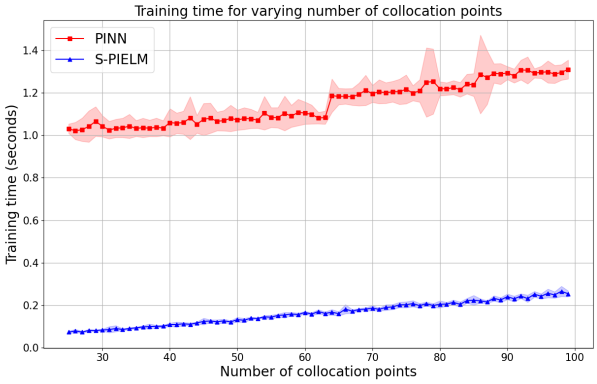
In the following section comparisions between S-PIELM and PINN algorithms will be made based on the networks accuracy on extrapolation in time on the logistic equation.

Logistic Equation Error Comparision				
	S-PIELM ($N_t = 50$)	S-PIELM ($N_t = 100$)	PINN ($N_t = 50$)	PINN ($N_t = 100$)
$t = 5$	0.00013	0.000607	0.000175	0.000234
$t = 6$	0.000107	0.000728	0.004284	0.004488
$t = 7$	8.8e-05	0.000835	0.011699	0.012100
$t = 8$	7.8e-05	0.000924	0.021601	0.022231
$t = 9$	7.1e-05	0.000974	0.033059	0.033937
$t = 10$	5.3e-05	0.000944	0.045186	0.046312

TABLE 5.1: The following results were obtained by training a neural network within the time range $t \in [0, 5]$ and assessing its performance on an unseen time frame of $t \in (5, 10]$. For evaluation, a fixed number of collocation points, $Nt = 1000$, was employed, while the training phase explored the impact of varying the number of collocation points to observe performance differences.

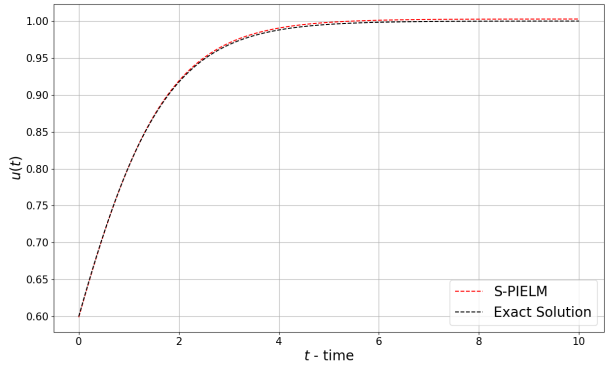


(A) Convergence of the 2 methods on logistic equation

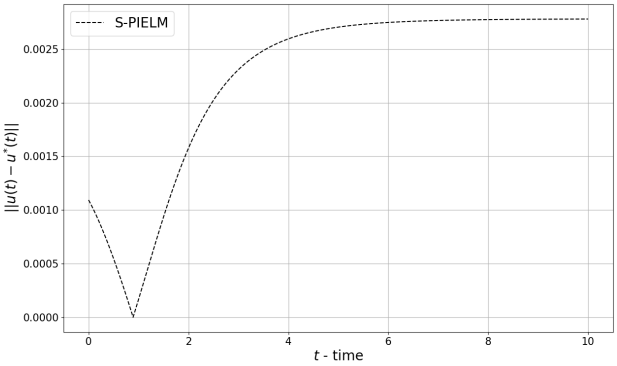


(B) Training time comparison of the 2 algorithms

FIGURE 5.1: Performance Analysis of PINNs vs. S-PIELM on the Logistic Equation: Averaged Results and Standard Deviation over 30 Trials.

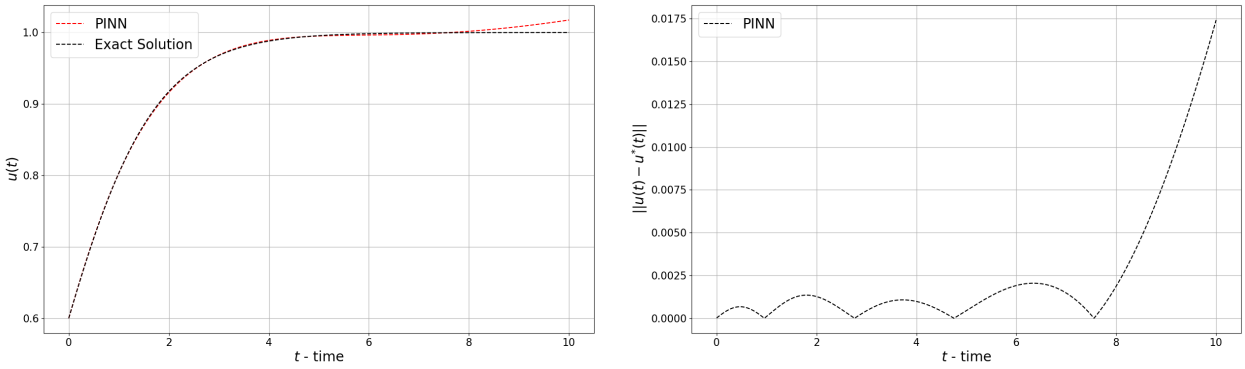


(A) Evalutation of S-PIELM on logistic equation



(B) Error of S-PIELM on the logistic equation

FIGURE 5.2: Performance Evalutation of S-PIELM on the logistic equation. The initial condition $u_0 = 0.6$ and $r = 1$.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$



(A) Evalutation of PINN on the logistic equation

(B) Error of PINN on the logistic equation

FIGURE 5.3: Performance evaluation of PINN on the logistic equation. The initial condition $u_0 = 0.6$ and $r = 1$. The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$

The results indicate that the Physics-Informed Neural Network (PINN) method exhibits limited generalisation ability when extrapolating beyond the training domain when considering the results obtained in Table 5.1. As shown in Figures 5.2a and 5.3a, the PINN method performs well on the data within its training range but experiences a sharp increase in error when extrapolating beyond that range (Figure 5.3b). In contrast, the proposed method does not exhibit such behaviour as seen in Figure 5.2b.

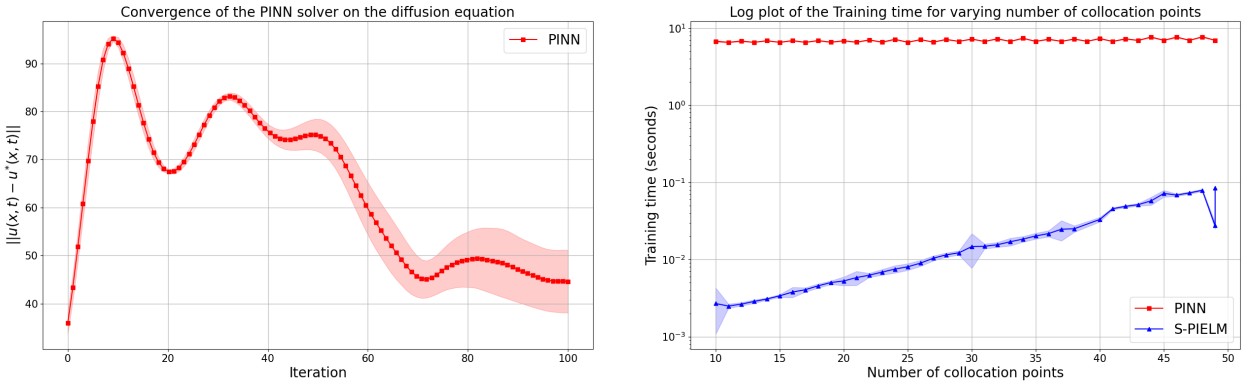
Upon analysis of the data presented in Figures 5.1a and 5.1b, it becomes apparent that the S-PIELM methodology exhibits superior accuracy for both observed and unseen datasets, while requiring less computational resource usage compared to the PINN method. A critical examination of Figure 5.1a reveals that initially, S-PIELM demonstrates a less accurate approximation of the solution. However, it rapidly converges towards the optimal solution. This is in stark contrast to the PINN method, which, despite starting with a more accurate approximation post the initial iteration, exhibits a notably slower convergence rate relative to the S-PIELM method.

5.3.2 Diffusion Equation

In the following section, comparisons between S-PIELM and PINN algorithms will be made based on the network accuracy in extrapolation time on the diffusion equation.

Diffusion Equation Error Comparision				
	S-PIELM ($N_t = N_x = 20$)	S-PIELM ($N_t = N_x = 50$)	PINN($N_t = N_x = 20$)	PINN($N_t = N_x = 50$)
$t = 0.5$	1.1277e-06	8.3040e-06	0.286873	0.261218
$t = 0.6$	0.000156	0.000969	0.332587	0.306016
$t = 0.7$	0.001509	0.009923	0.382886	0.356571
$t = 0.8$	0.006661	0.049196	0.440946	0.411917
$t = 0.9$	0.019837	0.164976	0.503173	0.471547
$t = 1$	0.042622	0.396286	0.561454	0.527796

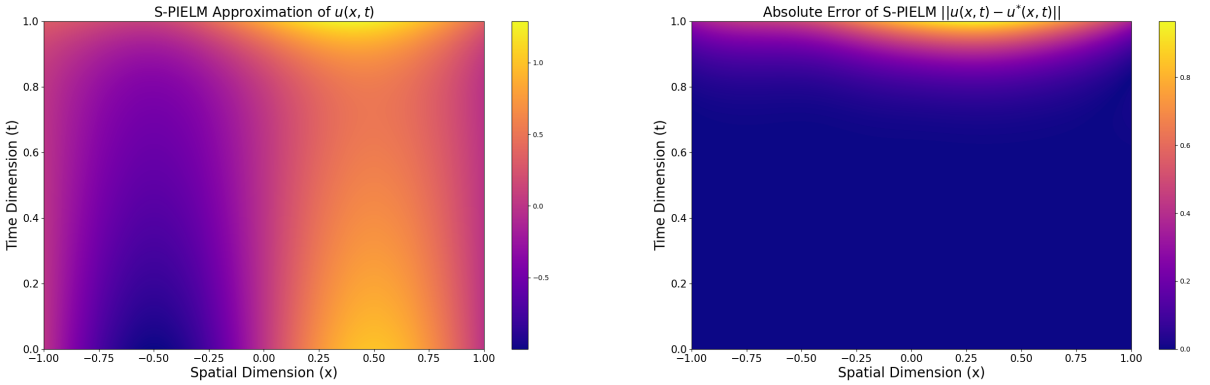
TABLE 5.2: The following results were obtained by training a neural network within the time range $t \in [0, 0.5]$ and assessing its performance on an unseen time frame of $t \in (0.5, 1]$. For evaluation, a fixed number of collocation points, $Nt = 100$, was employed, while the training phase explored the impact of varying the number of collocation points to observe performance differences.



(A) Convergence on the diffusion equation

(B) Training time comparision on the diffusion equation

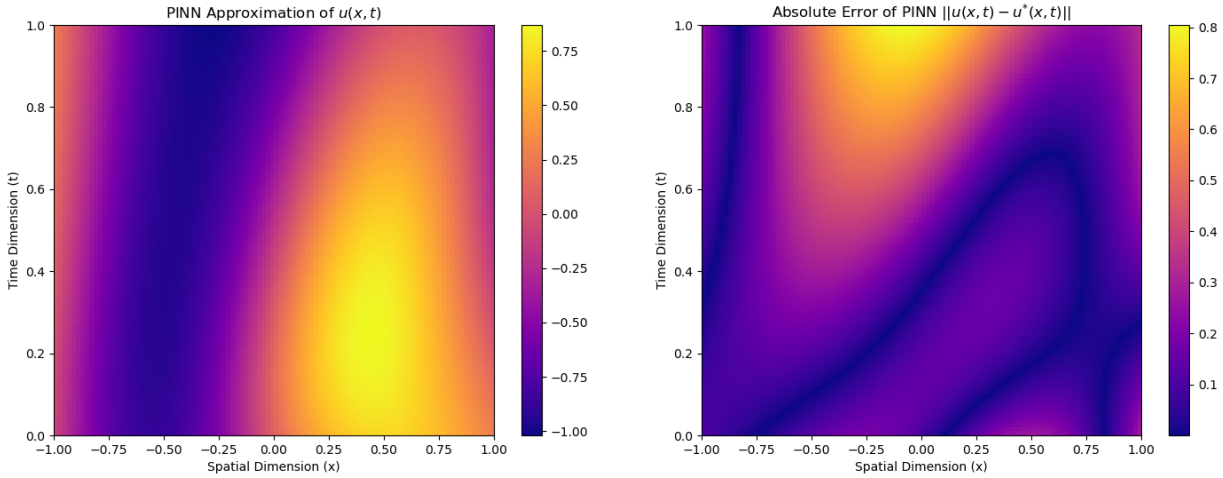
FIGURE 5.4: Performance Analysis of PINNs vs. S-PIELM on the Diffusion Equation: Averaged Results and Standard Deviation over 30 Trials. S-PIELM was omitted from the convergence plot as the optimal weights were determined in a single iteration



(A) Evalutation of S-PIELM on the diffusion equation

(B) Error of S-PIELM on the diffusion equation.

FIGURE 5.5: Performance of the S-PIELM on the diffusion equation in comparision to the exact solution. The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. $u(x,t)$ is the exact and $u^*(x,t)$ is the approximation



(A) Evalutation of PINN on the diffusion equation

(B) Error of PINN on the diffusion equation.

FIGURE 5.6: Performance of the PINN on the diffusion equation in comparision to the exact solution.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. $u(x,t)$ is the exact and $u^*(x,t)$ is the approximation

The results indicate that the PINN method exhibits limited generalisation capability in comparison to the S-PIELM approach when considering the results obtained in Table 5.2 and Figures 5.5 , 5.6. Figures 5.5 , 5.6 Indicate that the PINN method performs well on the data within its training range but experiences a dramatic increase in error when extrapolating beyond that range $t > 0.5$ (Figure 5.6b). On the contrary, the proposed method does not exhibit a sharp increase in error until $t > 0.8$, as seen in Figure 5.5b.

Upon analysis of the data presented in Figures 5.4a and 5.4b, it becomes apparent that the S-PIELM methodology exhibits superior accuracy for both observed and unseen datasets, while requiring less computational resource usage compared to the PINN method. A critical examination of Figure 5.4b,5.4a indicates that there is a large reduction in the required computational resources required as the S-PIELM

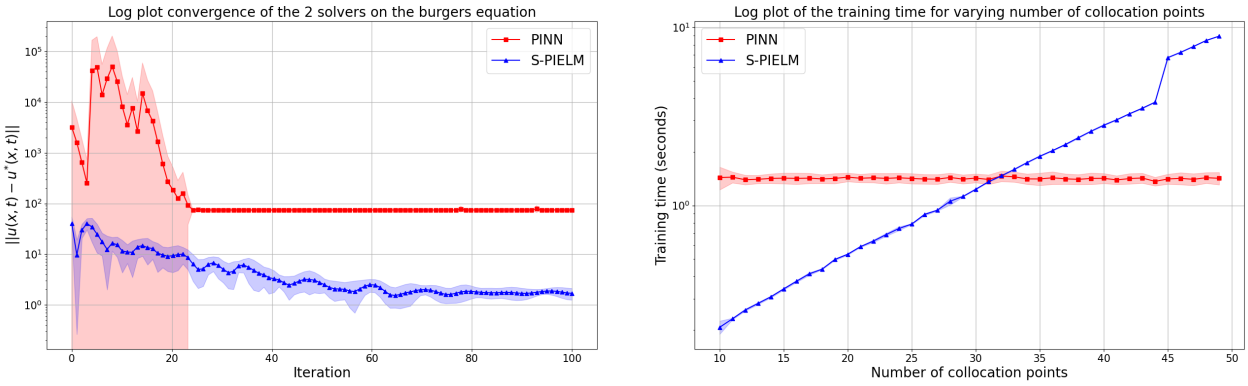
requires one iteration in order to determine the optimal weights whilst the PINN approach still requires multiple iterations in order to acheive a satisfactory approximation of the diffusion equation.

5.3.3 Burgers' Equation

In the following section, comparisons between S-PIELM and PINN algorithms will be made based on the network accuracy in extrapolation time on Burgers' equation.

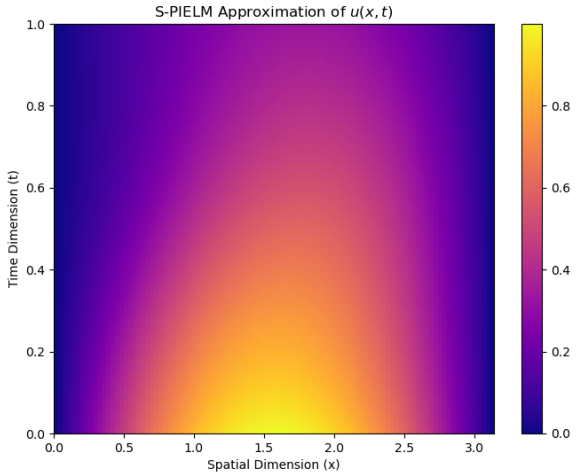
Burgers' Equation Residual Error Comparision				
	S-PIELM ($N_t = N_x = 20$)	S-PIELM ($N_t = N_x = 50$)	PINN($N_t = N_x = 20$)	PINN($N_t = N_x = 50$)
$t = 0.5$	0.246602	0.282473	0.177029	0.056407
$t = 0.6$	0.638267	0.59328	0.585154	0.436997
$t = 0.7$	1.187690	1.080476	0.883564	0.763992
$t = 0.8$	1.673464	1.528686	1.071504	1.026635
$t = 0.9$	2.024572	1.915660	1.168426	1.227463
$t = 1$	2.206407	2.138484	1.196172	1.355656

TABLE 5.3: The following results were obtained by training a neural network within the time range $t \in [0, 0.5]$ and assessing its performance on an unseen time frame of $t \in (0.5, 1]$. For evaluation, a fixed number of collocation points, $Nt = 100$, was employed, while the training phase explored the impact of varying the number of collocation points to observe performance differences.

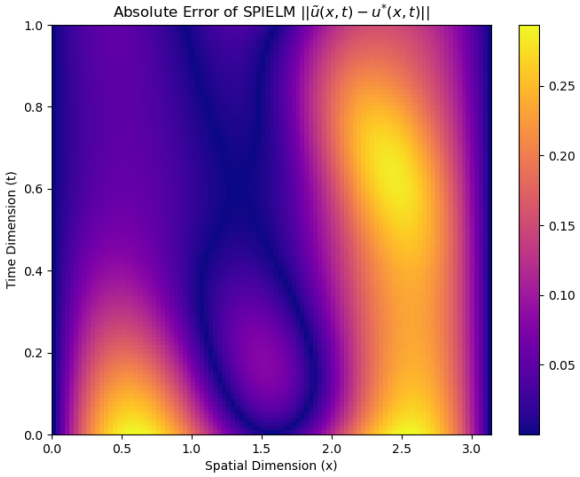


(A) Convergence of the both methods on Burgers' equation (B) Training time comparision both methods on Burgers' equation

FIGURE 5.7: Performance Analysis of PINNs vs. S-PIELM on Burgers' Equation: Averaged Results and Standard Deviation over 30 Trials.

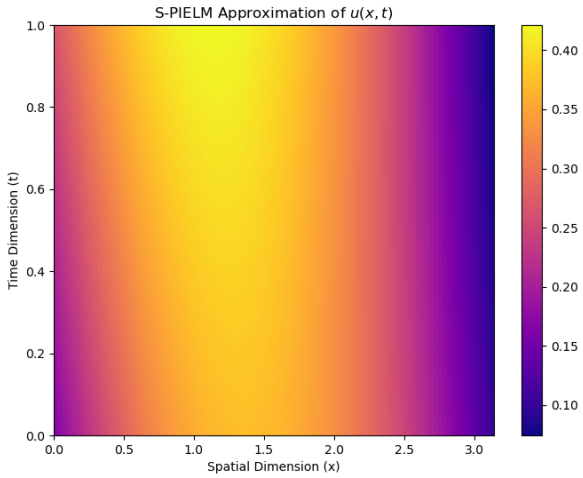


(A) Evalutation of S-PIELM on Burgers' equation

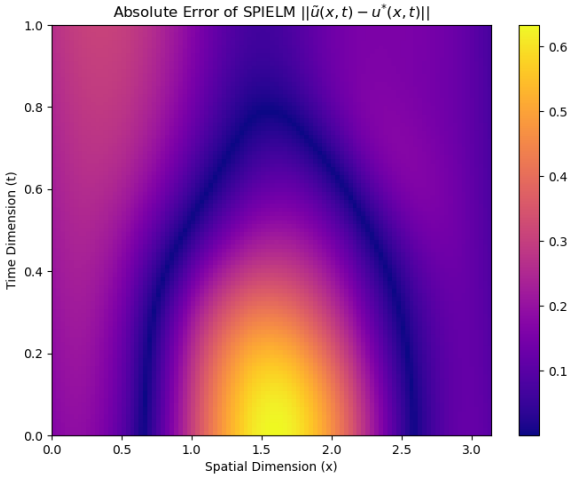


(B) Error of S-PIELM on Burgers' equation.

FIGURE 5.8: Performance of the S-PIELM on Burgers' equation in comparision to the exact solution.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. The error computed is by comparing the numerical solution using spectral methods $\tilde{u}(x,t)$, with the S-PIELM approximation $u^*(x,t)$.



(A) Evalutation of PINN on Burgers' equation



(B) Error of PINN on Burgers' equation.

FIGURE 5.9: Performance of the PINN on Burgers' equation in comparision to the exact solution.The model was trained on the domain $t \in [0, 0.5]$ and is extrapolated in the interval $t \in (0.5, 1]$. The error computed is by comparing the numerical solution using spectral methods $\tilde{u}(x,t)$, with the PINN approximation $u^*(x,t)$.

The findings indicate that the Physics-Informed Neural Network (PINN) method demonstrates limited generalisation capabilities, as evidenced by the data presented in Table 5.3. In Figure 5.7a, the PINN method initially exhibits a lower error, which gradually increases and then stabilizes. This pattern might stem from the PINN method's vulnerability to trivial solutions, wherein it initially settles on a simple solution and subsequently, albeit slowly, shifts towards a more complex solution. Moreover, Figure 5.7b illustrates that as the number of collocation points surpasses

30, the computational demands of the proposed method escalate significantly. This increase in complexity is due to the requirement of calculating the inverse of a matrix in each iteration, a process known to be computationally intensive.

When comparing the error graphs in Figures 5.8b and 5.9b, it is evident that the proposed method outperforms the PINN method by nearly a factor of two under identical iteration constraints. This superior performance can be attributed to the slower convergence rate of the PINN method. However, when increasing the iterations to 10000, the PINN method has the potential to achieve comparable levels of accuracy within the training domain.

5.3.4 Discussion of results

Upon analysing the results obtained from both the logistic and diffusion equation cases, it becomes evident that the Physics-Informed Neural Network (PINN) method exhibits limited generalisation capability. In the context of the logistic equation (Table 5.1), the PINN method performs adequately within its training range, but encounters a sharp increase in error when venturing beyond that range, as observed in Figures 5.2a and 5.3a. Conversely, the proposed method, an extension of the pseudospectral technique, displays a more consistent behaviour and avoids such undesirable extrapolation tendencies. The key differentiator lies in the loss evaluation approach: While the PINN approach relies on a local-based method that scrutinises individual points' compliance with the differential equation, the proposed method employs a global loss evaluation that takes all collocation points into consideration. As a result, the proposed method demonstrates a better capacity for reasonable extrapolation error mitigation.

Similarly, in the context of the diffusion equation (Table 5.2), the PINN method encounters challenges in generalising beyond the training data, as shown in Figures 5.8a,5.9a,5.5a and 5.6a. Similarly,The proposed method demonstrates its capability in maintaining consistent performance across extrapolation scenarios. As with the logistic equation, the divergence in performance can be attributed to the different loss evaluation strategies employed by each method.

The main drawback with the proposed method is evident when applying the technique in domains with complex geometries.

In conclusion, the proposed S-PIELM method emerges as a promising alternative, exhibiting higher accuracy not only on observed data but also on unobserved data. Remarkably, this superior performance is achieved with the same or fewer computational resources as the PINN method. The ability of the proposed method to address the challenges of extrapolation while maintaining computational efficiency makes it a valuable contender for practical applications in solving partial differential equations. A summary of each method is given in Table 5.4.

5.4 Conclusion and Future Work

This research sought to present an alternative approach to solving differential equations by using spectral-enhanced neural networks. This method stands as an alternative to the conventional technique of automatic differentiation. Earlier investigations have demonstrated the feasibility of employing automatic differentiation within neural networks to solve differential equations. However, this traditional method necessitates an extensive number of collocation points and prolonged training periods, while also exhibiting limited accuracy in extrapolations beyond the

Summary of Methods					
	Generalisation Capability	Loss Evaluation	Computational Efficiency	non-linear Equation	Complex Geometry
S-PIELM	Improved	Global	High	Yes	No
PINN	Limited	Local-based	Moderate	Yes	Yes

TABLE 5.4: Comparison of the methods based on key characteristics. "Generalisation Capability" indicates the ability to generalise beyond the training data, categorised as either "Improved" or "Limited". "Loss Evaluation" specifies the approach used: "Global" or "Local-based". "Computational Efficiency" refers to the level of computational resources required, categorised as either "High" or "Moderate". The table also notes whether the method handles solving non-linear Equation and solving equations in domains with a Complex Geometry.

training domain. On the contrary, the findings of this study indicate the potential to develop a model with reasonable accuracy, requiring fewer collocation points and reduced training time. Furthermore, this approach enhances the capability for model generalization beyond the confines of the training domain.

Based on the analysis of the results obtained, it is evident that the Physics-Informed Neural Network (PINN) method demonstrates limited generalisation capability. While the PINN method performs reasonably well within its training range, it struggles when extrapolated beyond that range, leading to a significant increase in error. In contrast, the proposed Spectral Physics Informed Extreme Learning Machine (S-PIELM) showcases improved generalisation by maintaining a more consistent performance across extrapolation scenarios. This advantage is attributed to the global loss evaluation approach used in S-PIELM, which takes into account all collocation points involved, in contrast to the local-based evaluation employed by PINN.

The comparison also revealed that S-PIELM achieves acomparable or even superior accuracy compared to PINN across various problem domains while demanding fewer computational resources. This finding positions S-PIELM as a promising alternative for solving partial differential equations, and with its low resource demand may make it particularly suitable for inverse problems.

However, it is essential to acknowledge that the proposed method encounters challenges when applied in domains with complex geometries. This limitation warrants further investigation and potential improvements to address the complexities introduced by intricate spatial configurations (i.e using domain decomposition).

In conclusion, the proposed S-PIELM method demonstrates enhanced generalisation capability and computational efficiency compared to the PINN method. The ability to handle both stationary and time-dependent linear and non-linear differential equations enhances its practical utility. Additionally, possible future work could focus on refining the method's performance in domains with complex geometries, allowing for broader applications in real-world scenarios.

The summarized characteristics of each method are presented in Table 5.4. Overall, the findings underscore the potential of S-PIELM as a promising and efficient solution for solving partial differential equations in various problem domains, laying the groundwork for further advancements in the field.

Chapter 6

Deep Neural Operators

In the following chapter, a detailed discussion and implementation of Deep Neural Operators for solving Differential Equations is provided. The optimisation algorithms discussed in Chapter 3 are used to optimise and measure the solution quality of Deep Neural Operators

6.1 Introduction

Deep Neural Operators (DNOs) are a class of machine learning models designed to learn mappings between infinite-dimensional spaces, specifically focusing on solving parameterized partial differential equations (PDEs). Traditional neural networks often struggle with tasks that require the handling of continuous and high-dimensional data, such as those found in fluid dynamics, material science, and other physics-based fields. DNOs address these challenges by using neural networks to approximate operators directly, rather than merely functions.

The core idea behind DNOs is to represent intricate physical systems as learnable operators, which can predict system behaviourad under various conditions without needing to retrain from scratch for each new scenario. This approach leverages the compositional structure of neural networks to efficiently learn and generalize across a range of input functions and parameters. As a result, Deep Neural Operators offer a promising tool for accelerating simulations and analyses in computational science and engineering, providing significant computational savings and enabling more robust predictions in intricate systems. [45]

Background on Green's Functions

Green's functions are a fundamental concept used to solve boundary value problems for linear differential equations. Essentially, a Green's function $G(x,s)$ for a linear operator L acting on a function $u(x)$ is defined such that:

$$LG(x,s) = \delta(x-s),$$

where $\delta(x-s)$ is the Dirac delta function, which is zero everywhere except at $x=s$ and integrates to 1 over the entire real line.

The solution $u(x)$ to the differential equation

$$Lu(x) = f(x),$$

with boundary conditions can then be expressed as:

$$u(x) = \int G(x,s)f(s)\,ds,$$

where $f(x)$ is a known function, and the integral is taken over the domain of x.

Example: One-Dimensional Poisson's Equation

Consider the one-dimensional Poisson's equation:

$$\frac{d^2u}{dx^2} = -\rho(x),$$

with boundary conditions $u(0) = u(L) = 0$ where L is the length of the domain and $\rho(x)$ is a given charge density.

The Green's function for this equation under the specified boundary conditions is:

$$G(x,s) = \begin{cases} \frac{(x-s)L+x(L-s)}{L} & \text{for } x < s \\ \frac{(s-x)L+s(L-x)}{L} & \text{for } x > s \end{cases}$$

Using Green's function, the solution $u(x)$ can be found by:

$$u(x) = \int_0^L G(x,s)\rho(s)\,ds.$$

This integral effectively "spreads" the effect of the charge density $\rho(s)$ throughout the domain according to Green's function, respecting the boundary conditions.

Greens functions in relation to Deep Neural Operators

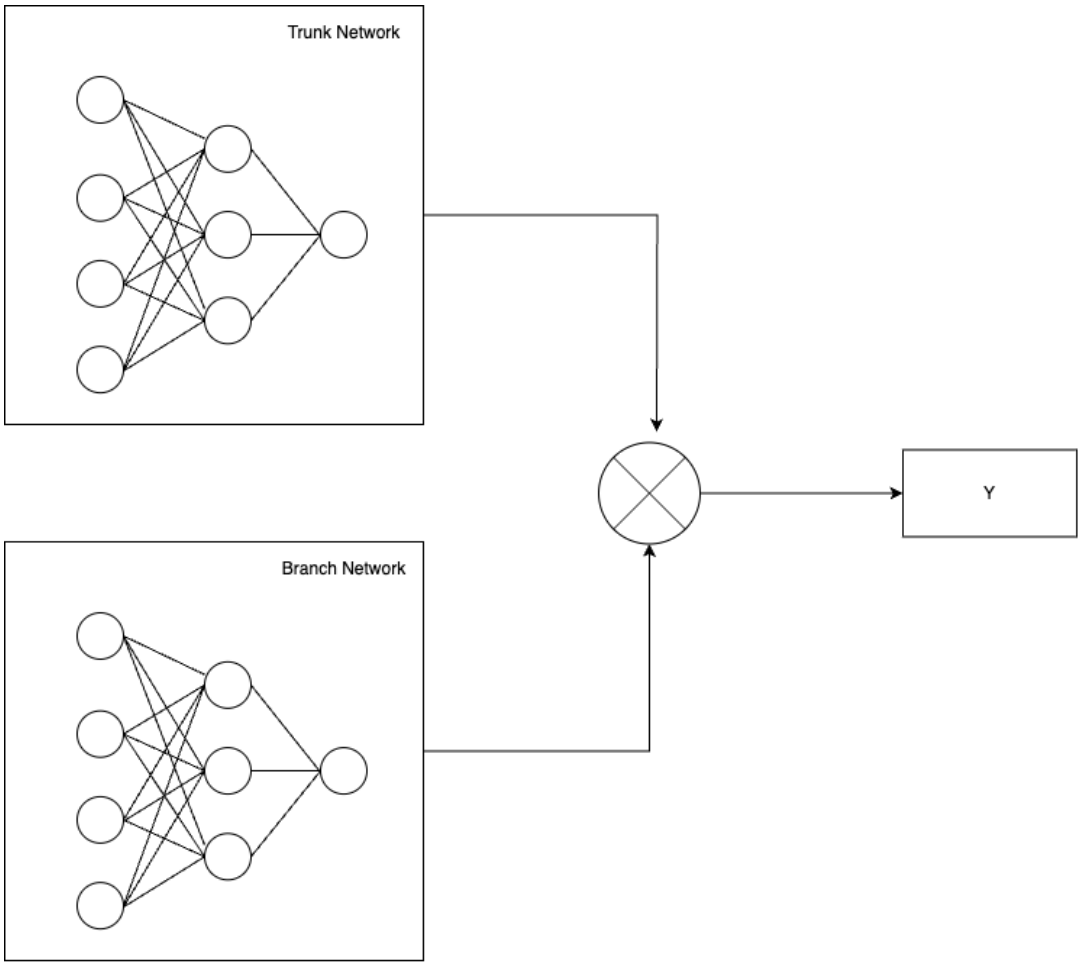


FIGURE 6.1: Diagram of the Deep Neural Operator Architecture

The concept of Deep Neural Operators is inspired by Green's functions. Both the trunk and branch networks can utilize any neural network architecture, such as MLPs or Transformers. In this study, an MLP is used.

The branch network aims to learn the Green's function and maps it to a low-dimensional vector, while the trunk network maps the initial condition to another low-dimensional vector. The dot product (represented by a circle with an X in the middle in Figure 6.1) between these two vectors is used to make a prediction. The Deep Neural Operator aims to learn the functional of the differential equation, which allows for changing parameters without the need of retraining.

6.2 Methodology

To train the model, specific data is needed. This data is derived by solving the differential equation using a range of parameters. The spectral collocation method is employed to generate the training data. The logistic equation was trained using MLPs with a width of 200 and a single hidden layer, whereas the Burgers equation utilized a width of 400 with four hidden layers. Training for the logistic equation

was limited to 1000 epochs to prevent overfitting, while the Burgers equation was capped at 20000 epochs. Experiments consisted of firstly generating data for DNO to be trained on. Once the data is available, the network is then trained on that data. afterwards the model is evaluted on unseen data/new parameters.

6.3 Results

In the subsequent sections, DeepONets are analyzed using the logistic equation and Burgers' equation. These equations were selected because they are non-linear and contain parameters that influence the system's trajectory.

6.3.1 Logistic Equation

Below are the results of the different optimisation techniques used when learning the solution to the logistic equation, followed by a discussion. Network archtecture for each of the algorithms is provided in Table 6.1. Where

n branch: the number of inputs into the branch network, in this case it is u_0 and R

n trunk number of inputs into the trunk network, in this case it is just t

p**:** the output size of the branch and trunk network

depth: the number of hidden layers for the branch and trunk network

width: the number of nodes within each layer

iterations: number of iterations for the algorithm

Population: the number of networks being updated at each iteration

Model	n_branch	n_trunk	p	depth	width	Iterations	Population
SGD	2	1	50	2	400	1000	1
Adam	2	1	50	2	400	1000	1
LBFGS	2	1	40	2	200	1000	1
SA	2	1	40	2	200	1000	1
GA	2	1	40	2	200	100	20
PSO	2	1	40	2	200	100	20

TABLE 6.1: Architecture of Different Models

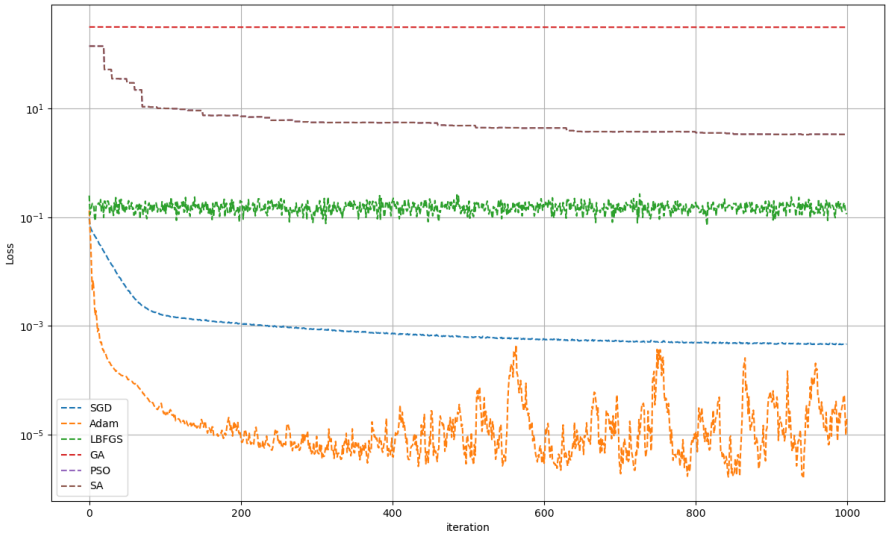


FIGURE 6.2: Convergence Analysis of Different optimisation techniques on the logistic equation

Alg.	LR	Epo.	Loss	Hist.	N	M	μ	c_1	c_2	T_{init}	α
SGD	0.01	200	1.019×10^{-3}	—	—	—	—	—	—	—	—
Adam	0.001	200	5.724×10^{-4}	—	—	—	—	—	—	—	—
LBFGS	0.01	100	5.205×10^{-4}	—	—	—	—	—	—	—	—
GA	—	100	1.175×10^{-3}	—	50	20	0.30	—	—	—	—
PSO	—	100	4.664×10^{-4}	—	100	—	—	2.0	2.0	—	—
SA	—	200	1.820×10^{-1}	—	—	—	—	—	—	10000	0.95

TABLE 6.2: Best hyperparameter tuning results for each algorithm (rounded to four significant digits). Abbreviations: Alg. – algorithm; LR – learning rate; Epo. – epochs; Loss – final loss; Hist. – history size (L-BFGS); N – population size (GA, PSO); M – offspring size (GA); μ – mutation rate (GA); c_1, c_2 – cognitive/social coefficients (PSO); T_{init} – initial temperature (SA); α – cooling rate (SA).

$u_0 = 0.6$ **and** $R = 1$

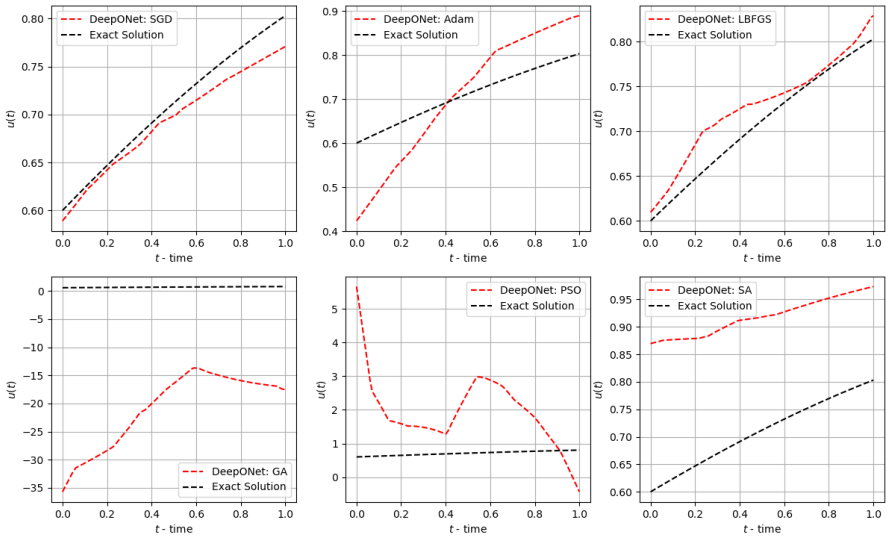


FIGURE 6.3: Comparision of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.6$ and $R = 1$

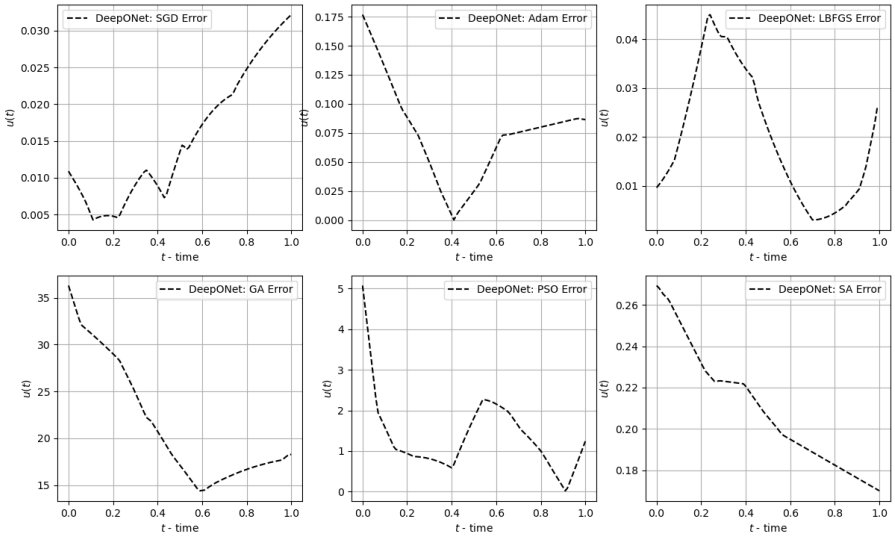


FIGURE 6.4: Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.6$ and $R = 1$

Method	Average Error
SGD	0.01538
Adam	0.07388
LBFGS	0.01987
GA	21.41559
PSO	1.35478
SA	0.20957

TABLE 6.3: Average errors for different optimization methods on the logistic equation, where $u_0 = 0.6$ and $R = 1$

$u_0 = 0.7$ **and** $R = 0.5$

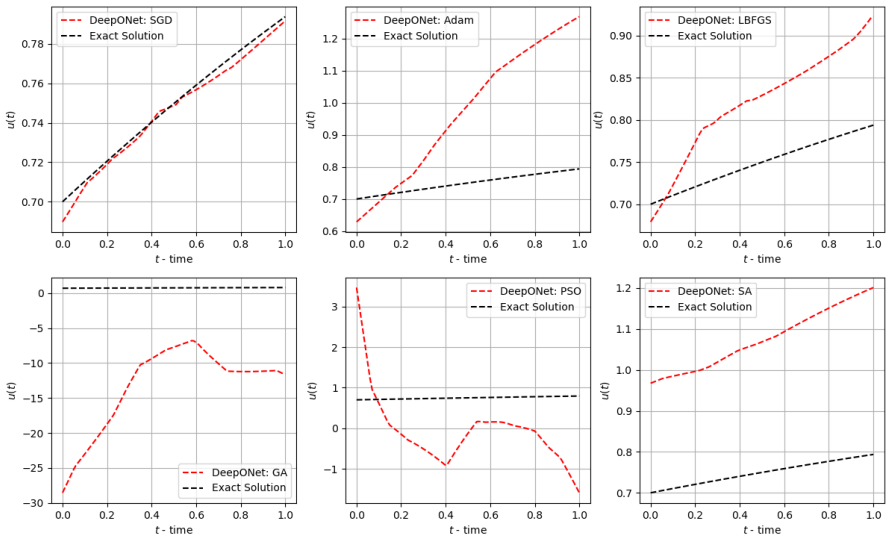


FIGURE 6.5: Comparision of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.7$ and $R = 0.5$

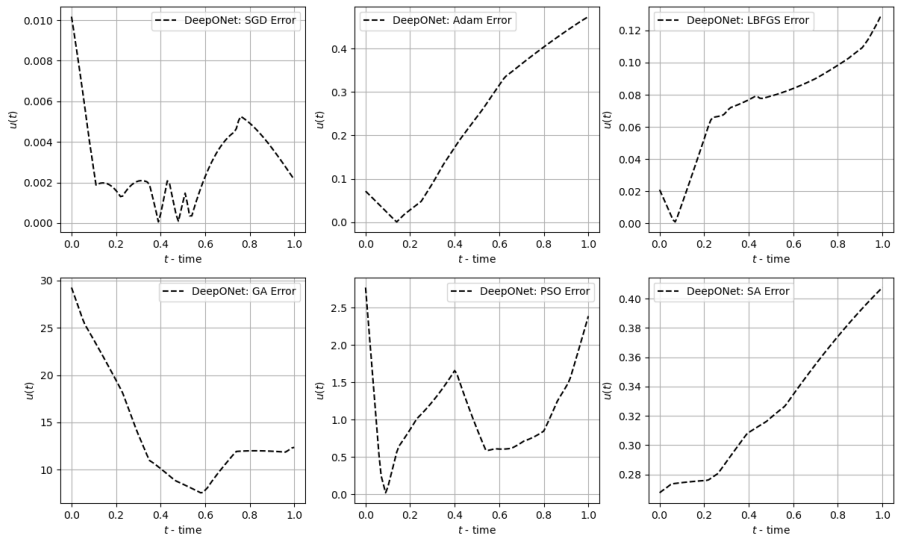


FIGURE 6.6: Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets. Where $u_0 = 0.7$ and $R = 0.5$

Method	**Average Error**
SGD	0.00295
Adam	0.23428
LBFGS	0.07412
GA	13.89046
PSO	1.06402
SA	0.32593

TABLE 6.4: Average errors for different optimization methods, where $u_0 = 0.7$ and $R = 0.5$

$u_0 = 0.2$ **and** $R = 1$

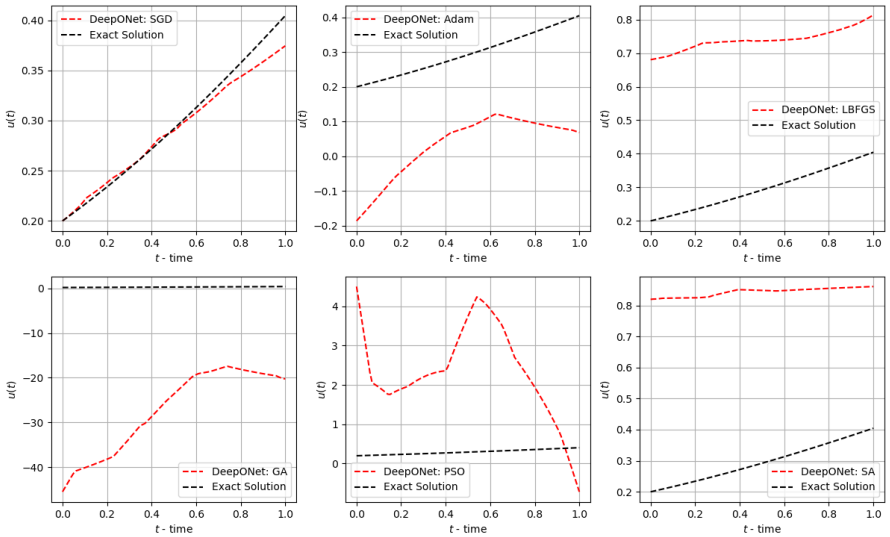


FIGURE 6.7: Comparision of Different optimisation techniques on the logistic equation using DeepONets.Where $u_0 = 0.2$ and $R = 1$

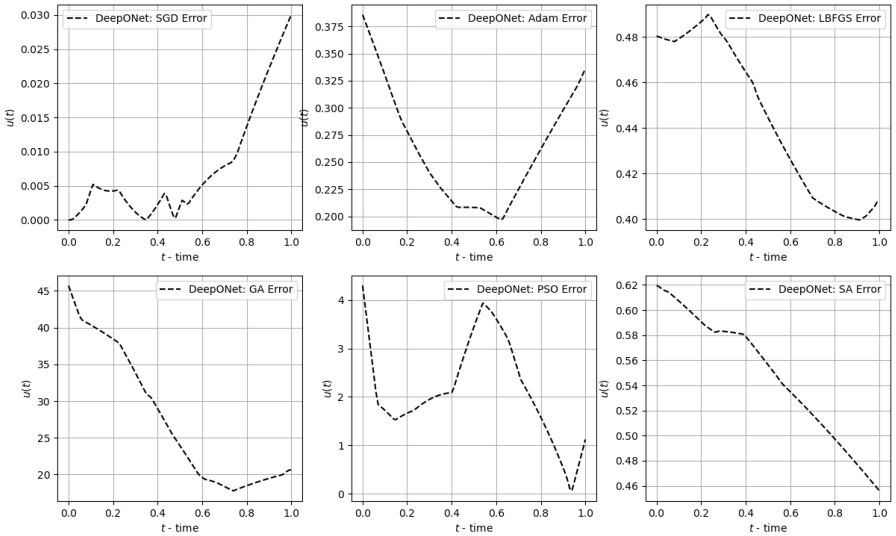


FIGURE 6.8: Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets. Where $u_0 = 0.2$ and $R = 1$

The results shown in Figure 6.2 reveal that gradient-based methods converge faster than heuristic methods. This could be because the Mean Squared Error, which measures the difference between actual and predicted values, is a convex function. Consequently, heuristic methods perform worse, as shown in Figures 6.4, 6.6, and 6.8.

Method	Average Error
SGD	0.00767
Adam	0.26174
LBFGS	0.44385
GA	27.41281
PSO	2.13967
SA	0.54785

TABLE 6.5: Average errors for different optimization methods,Where $u_0 = 0.2$ and $R = 1$

We also adjusted the parameters to see how they affect the network's performance. Figures 6.3, 6.5, and 6.7 indicate that SGD results in lower errors compared to Adam and L-BFGS. This might be because the latter two optimization algorithms lead to overfitting, resulting in a lower convergence error shown in Figure 6.2.

6.3.2 Burgers' Equation

Below are the results of the different optimisation techniques used when learning the solution to the logistic equation, followed by a discussion.

It is important to note that due to memory constraints smaller model sizes were used when training using Genetic Algorithm and Particle Swarm Optimisation. Table 6.6 provides specific details on the network architecture, population size and training. Each column is described as the following:

n branch: the number of inputs into the branch network, in this case it is 201 points sampled from the initial condition

n trunk number of inputs into the trunk network, in this case it is just x and t

p**:** the output size of the branch and trunk network

depth: the number of hidden layers for the branch and trunk network

width: the number of nodes within each layer

iterations: number of iterations for the algorithm

Population: the number of networks being updated at each iteration

The initial conditions where sampled from the following equation

$$asin(bx) + c,$$

where the variables were from $(0.1, 2)$, $(0.1, 4)$, $(0.1, 3)$ for a,b and c respectively.

Model	**n_branch**	**n_trunk**	**p**	**depth**	**width**	**Iterations**	**Population**
SGD	201	2	50	4	400	20000	1
Adam	201	2	50	4	400	20000	1
LBFGS	201	2	50	4	400	20000	1
SA	201	2	50	4	400	20000	1
GA	201	2	50	2	400	100	20
PSO	201	2	50	2	400	100	20

TABLE 6.6: Architecture of Different Models

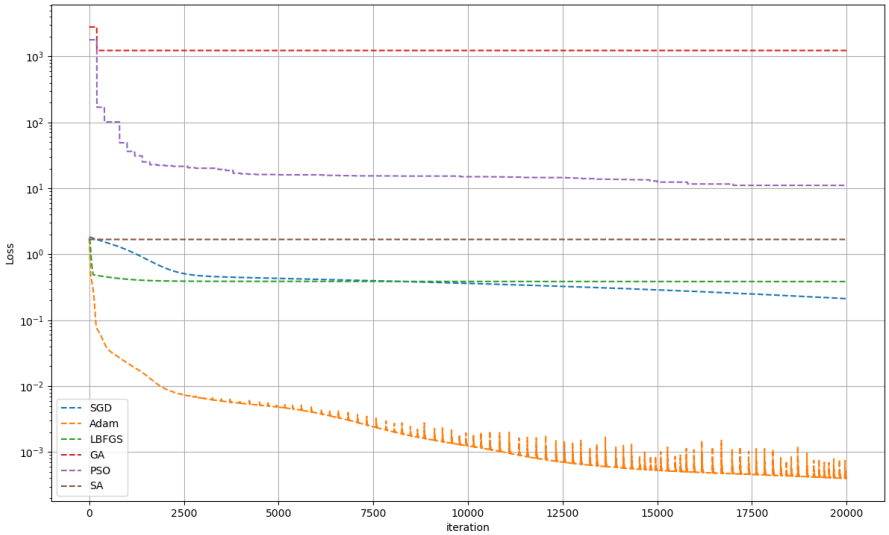


FIGURE 6.9: Convergence Analysis of Different optimisation techniques on Burgers' Equation

$a = 1.14$, $b = 2.88$, $c = 1.80$

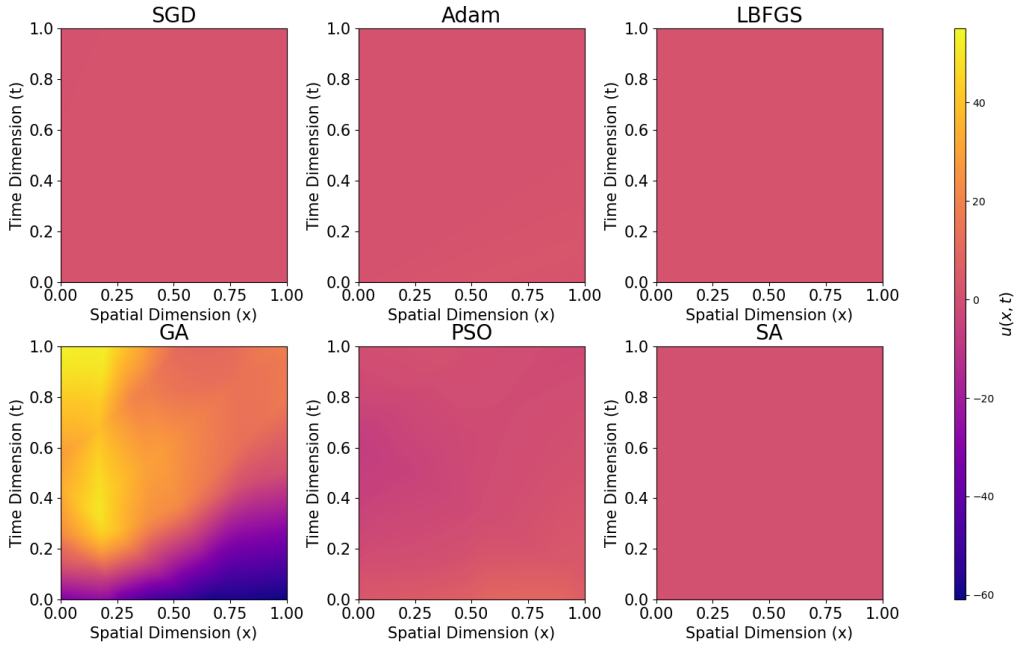


FIGURE 6.10: Comparision of Different optimisation techniques on the Burgers' equation using DeepONets.Where $a = 1.14$,$b = 2.88$,$c = 1.80$

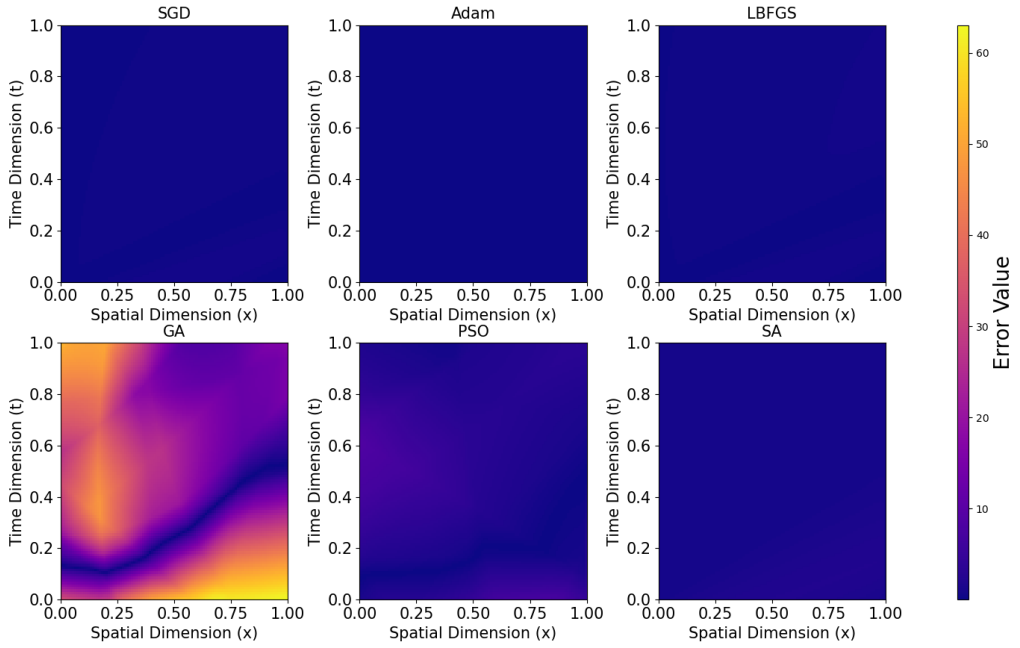


FIGURE 6.11: Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $a = 1.14$,$b = 2.88$,$c = 1.80$

Method	Average Error
SGD	0.3162
Adam	0.0124
LBFGS	0.3663
GA	24.3345
PSO	2.7431
SA	0.9717

TABLE 6.7: Average errors for different optimization methods Where $a = 1.14$,$b = 2.88$,$c = 1.80$

$a = 0.93$, $b = 3.58$, $c = 2.89$

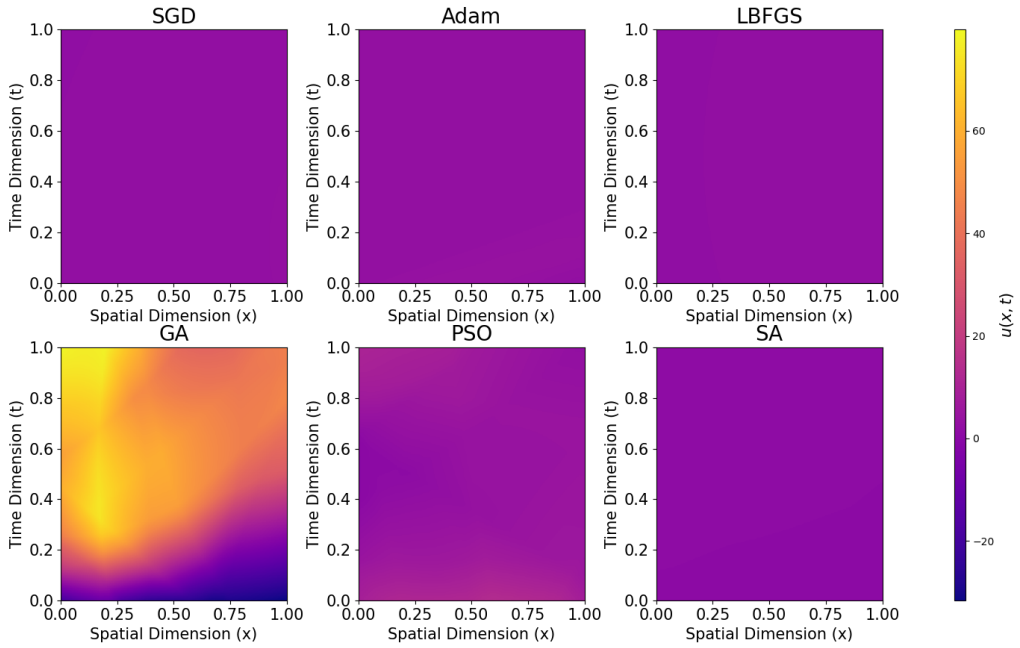


FIGURE 6.12: Comparision of Different optimisation techniques on the Burgers' equation using DeepONets.Where $a = 0.93$,$b = 3.58$,$c = 2.89$

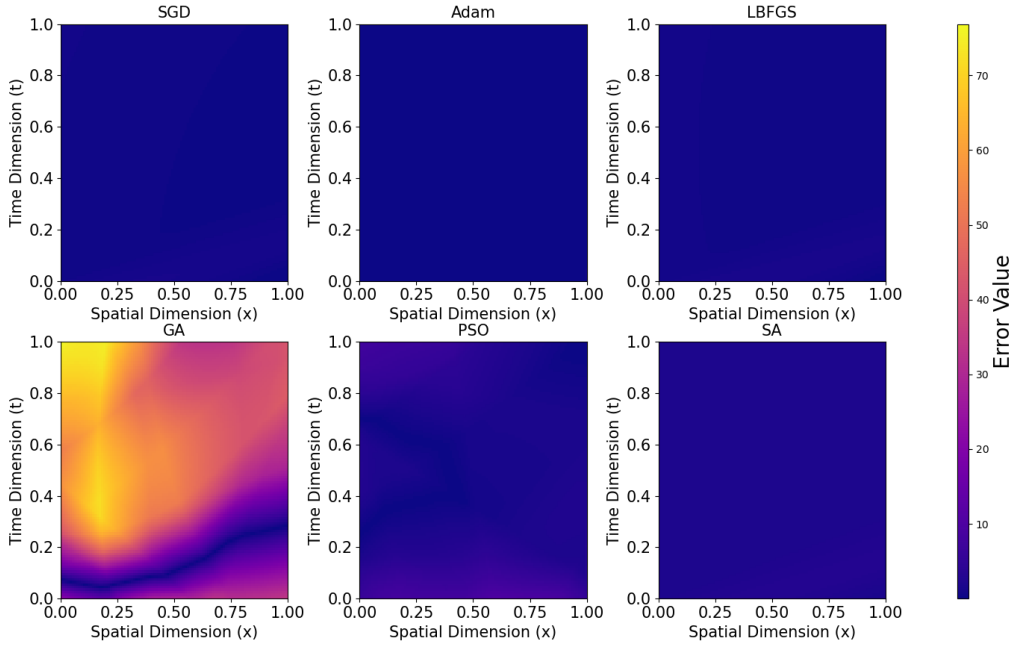


FIGURE 6.13: Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $a = 0.93$,$b = 3.58$,$c = 2.89$

Method	**Average Error**
SGD	0.4269
Adam	0.0086
LBFGS	0.5902
GA	40.4490
PSO	2.5995
SA	1.9632

TABLE 6.8: Average errors for different optimization methods Where $a = 0.93$,$b = 3.58$,$c = 2.89$

$a = 0.93$, $b = 3.49$, $c = 2.93$

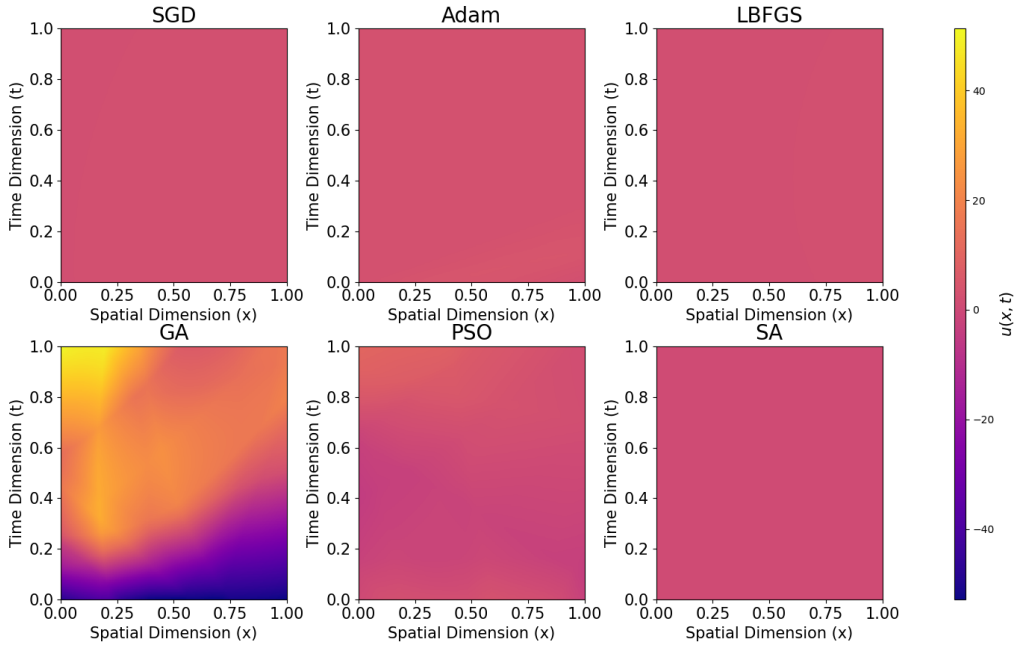


FIGURE 6.14: Comparision of Different optimisation techniques on the Burgers' equation using DeepONets.Where $a = 0.93$,$b = 3.49$,$c = 2.93$

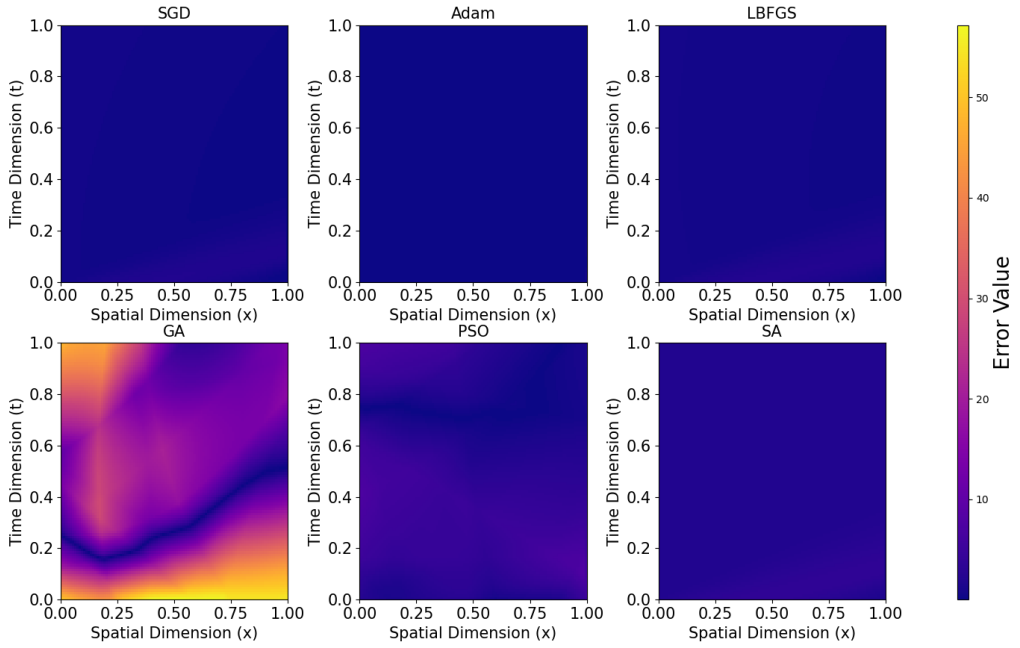


FIGURE 6.15: Comparision of the error of Different optimisation techniques on the logistic equation using DeepONets.Where $a = 0.93$,$b = 3.49$,$c = 2.93$

As depicted in Figure 6.9, the Adam optimization algorithm exhibited the quickest convergence compared to other algorithms. L-BFGS had slightly inferior performance relative to Adam and SGD, possibly because parameter choices can lead

Method	Average Error
SGD	0.4499
Adam	0.0103
LBFGS	0.6697
GA	19.9529
PSO	3.1863
SA	2.0566

TABLE 6.9: Average errors for different optimization methods Where $a = 0.93$,$b = 3.49$,$c = 2.93$

to instability, thereby complicating the approximation of second derivative information. The population-based heuristic demonstrated notably slow convergence, likely due to smaller network sizes. Despite the high computational demands of training individual deep neural operator networks, employing a population of networks significantly increases computational costs. However, as illustrated in the figure, this strategy proved to be inefficient, performing poorly compared to gradient-based alternatives.

Considering Figures 6.10, 6.12, and 6.14, all values converged uniformly, possibly attributed to the complexities of training the Burgers' equation. Reviewing Tables 6.7, 6.8, and 6.9, Adam consistently outperformed other methods in all three test scenarios, with SGD and L-BFGS alternating for second place. The heuristic-based methods demonstrated inferior performance compared to gradient-based methods, largely due to the smaller network sizes.

6.4 Conclusion and Discussion

The analyses of the experimental outcomes, as shown in Figure 6.2, indicate that gradient-based optimization methods outpace heuristic approaches in terms of convergence speed. This advantage likely stems from the Mean Squared Error (MSE) metric, a convex function, providing a clearer path to minimization for gradient-based methods. Figures 6.4, 6.6, and 6.8 further corroborate the underperformance of heuristic methods under these conditions. Parameter adjustments within the network also reveal that stochastic gradient descent (SGD) typically yields lower errors than both Adam and L-BFGS, as demonstrated in Figures 6.3, 6.5, and 6.7. The latter algorithms may induce overfitting, a notion supported by the lower convergence errors observed in Figure 6.2.

In the examples involving Burgers' equation, as depicted in Figure 6.9, Adam optimization proved to be the fastest at converging among the tested algorithms. L-BFGS displayed slightly less efficient performance than Adam and SGD, possibly due to the instability introduced by certain parameter settings which complicates the approximation of second derivative information. Furthermore, the heuristic-based approach showed considerably slower convergence, likely a consequence of using reduced network sizes. Although training individual deep neural operator networks demands significant computational resources, deploying a population of networks dramatically escalates these costs. Yet, as illustrated in Figure 6.9, this approach does not justify the trade-off, as it performs poorly compared to gradient-based methods.

An examination of Figures 6.10, 6.12, and 6.14 reveals a uniform convergence across scenarios, which may be attributed to the inherent challenges of training the Burgers' equation. Comparative analyses, as indicated in Tables 6.7, 6.8, and

6.9, consistently show Adam as the superior performer in all three test cases, with SGD and L-BFGS alternating for the runner-up position. The less effective results of heuristic-based methods, primarily due to their smaller network sizes, underscore the efficiency of gradient-based methods in intricate optimization scenarios.

Chapter 7

Compositional Pattern Producing Networks

In the following chapter a detailed discussion and implementation of Compositional Pattern Producing Networks to solving Differential Equations is provided. The Genetic Algorithm discussed in Chapter 3 is used to optimise and measure the solution quality of Compositional Pattern Producing Networks

7.1 Introduction

Compositional Pattern Producing Networks (CPPNs) are a type of artificial neural network that are particularly effective at generating intricate patterns and images. They were developed by Ken Stanley as an extension of the idea of evolving neural networks through genetic algorithms, under the concept of "NeuroEvolution of Augmenting Topologies" (NEAT). Unlike traditional neural networks that usually produce a single output, CPPNs are designed to output an entire dataset, often in the form of an image or a pattern.

The unique characteristic of CPPNs is their ability to exploit geometric properties such as symmetry, repetition, and variation, which are commonly found in natural organisms. They achieve this by using a variety of activation functions throughout the network, rather than a single type of activation function. This allows CPPNs to create intricate and diverse patterns that are not easily achievable with conventional image-generation techniques.

In this Chapter the effectiveness of CPPNs in solving differential equations is explored.

Loss Function in CPPNs

The loss function in a Compositional Pattern Producing Network will comprise of two main components: the data loss and the physics loss. The data loss, $\mathcal{L}_{\text{data}}$, measures the discrepancy between the network's predictions and the observed data. The physics loss, $\mathcal{L}_{\text{physics}}$, quantifies the residuals of the differential equations, ensuring that the network's predictions satisfy the physical laws.

Let $\mathbf{u}(\mathbf{x}, t; \mathbf{w})$ be the neural network's prediction for the physical quantity of interest, where $\mathbf{x}$ represents the spatial coordinates, t is time, and $\mathbf{w}$ denotes the network's parameters. The loss function $\mathcal{L}$ can be expressed as:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{physics}},$$

where,

$$\mathcal{L}_{\text{data}} = \frac{1}{N}\sum_{i=1}^{N} \left|\mathbf{u}(\mathbf{x}_i, t_i; \mathbf{w}) - \mathbf{u}_{\text{true}}(\mathbf{x}_i, t_i)\right|^2,$$

and,

$$\mathcal{L}_{\text{physics}} = \frac{1}{M}\sum_{j=1}^{M} \left|\mathcal{F}(\mathbf{u}(\mathbf{x}_j, t_j; \mathbf{w}))\right|^2.$$

Here, $\mathbf{u}_{\text{true}}$ denotes the observed data, $\mathcal{F}$ represents the differential operator defined by the physical laws, and N and M are the number of data points and collocation points, respectively.

By minimizing this composite loss function, the CPPN learns to approximate the solution to the differential equations while fitting the observed data, providing a powerful tool for solving intricate scientific problems.

7.2 Methodology

Compositional Pattern Producing Networks (CPPNs) which was introduced by Stanley [46] is an adaptive learning mechanism which builds on the disadvantages of Nueroevolution. The universal function approximation theorem states that a network with a large enough hidden layer can be used to approximate any function. The issue with predefined network dimensions is it may be using more nodes than required, which in turn increases the computational expense. CPPN addresses these issues by starting off with a network of input and output nodes only, then adding complexity to the network only when required. The fitness value of candidate networks is determined by using the residual of the differential equation. In a similar fashion to PINNs, points on the boundary were sampled uniformly where applicable, whilst points within the domain were sampled using Lattice hypercube sampling. Experiments for each equation consisted of point sampling, followed by training then by evaulating the population and best solution at each iteration. right at the end the performance of the model against the ground truth is evaluated.

7.2.1 Representation of Genomes

As mentioned above there is no specified structure within the possible network architecture which requires a new way of representing the network. An example of this representation can be seen in figure 7.2 and figure 7.1. Each genome representation will consist of node genes, which give a description of the node type of the specified node and edge genes which detail information on edges between nodes. The following information is derived from the edge genes

- In: denotes where the signal will be travelling from along that edge
- Out: denotes where the signal will be travelling to
- Weight: denotes the weight parameter along that edge
- Enabled/Disabled: denotes whether the edge is active or not
- Innov: is the innovation number which is a unique number assigned to each edge which will be used during crossover

Node Genes	Node 1 Type: Input	Node 2 Type: Input	Node 3 Type: Input	Node 4 Type: Output	Node 5 Type: Hidden	
Edge Genes	In: 1 Out: 4 Weight: 0.7 Enabled Innov 1	In: 2 Out: 4 Weight: -0.5 Disabled Innov 2	In: 3 Out: 4 Weight: 0.5 Enabled Innov 3	In: 2 Out: 5 Weight: 0.2 Enabled Innov 4	In: 5 Out: 4 Weight: 0.4 Enabled Innov 5	In: 1 Out: 5 Weight: 0.6 Enabled Innov 6

FIGURE 7.1: Genome representation of the network in 7.2

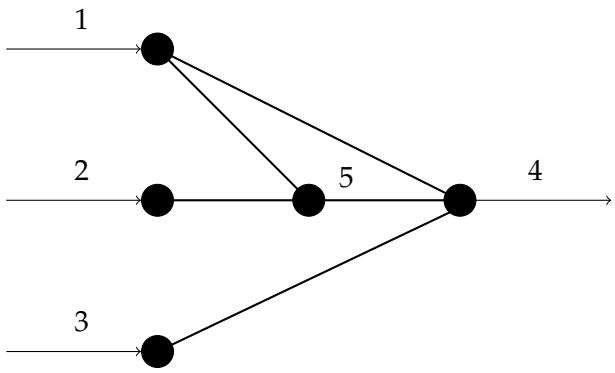


FIGURE 7.2: Network representation of genome in 7.1

7.2.2 Mutations

Within the network architecture, there exist 2 types of mutations, edge and node mutations. Edge mutations occur by adding new edges between nodes, mutating the weight value along the edge, disabling existing edges and enabling disabled edges. The type of edge mutation is determined by randomly a random value and setting up a probability distribution of each type of edge mutation that can occur. Node mutations occur by adding or removing hidden nodes within the network. Similarly to the edge mutation, The type of node mutation is determined by sampling a random value and setting up a probability distribution of each type of node mutation that can occur.

7.2.3 Crossover

As mentioned above, the candidate networks in the population have different topological structures, which makes using traditional crossover functions relatively difficult. By using the innovation number we can generate new candidate networks by matching the edge genes with the same innovation numbers and applying crossover operators. Genes that do not match can be classified as either disjoint or excess genes. Referencing Figure 7.3, The genes with innovation numbers 4 and 5 can be classified as disjoint edges and the gene with innovation number 7 can be classified as an excess gene. Given there are matching genes, the gene that would be selected would be determined randomly. For excess and disjoint genes, the candidate network with the better fitness value will determine whether the genes will be accepted. For example, assuming the second network has a higher fitness, the genes with innovations 4 and 5 will be selected whilst the gene with innovation number 7 will be rejected.

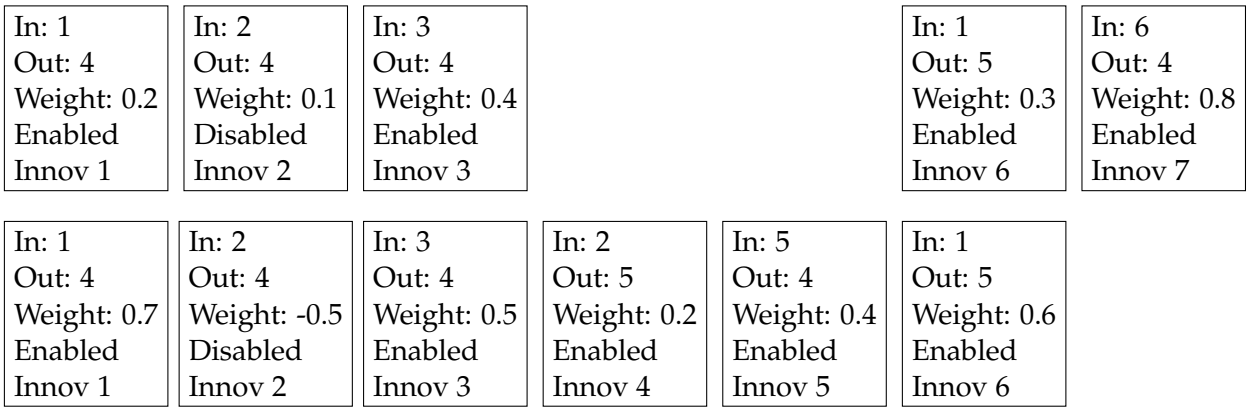


FIGURE 7.3: Crossover between 2 networks

7.2.4 Speciation

The main drawback with Topology augmenting Networks is its ability to differentiate between networks of different topologies. Since larger networks take longer to optimise and there is no unique network representation of approximating the solution to the differential equation, networks within the population are grouped based on their topological similarities. Given two networks, we can denote E as the number of excess edges, D the number of disjoint edges between the 2 networks and $\overline{W}$ the average weight difference between matching genes. The equation used to determine the similarity between networks is,

$$\delta = \frac{c1E}{N} + \frac{c2D}{N} + c3\overline{W}. \tag{7.1}$$

The constants $c1, c2, c3$ are weight parameters which allow for adjusting the significance of each term. The variable N is the maximum number of genes between the 2 candidate networks. If the similarity value computed using equation (7.1) is less than some threshold value δ_{th}, it is possible to conclude that the 2 networks are similar. Once the networks have been grouped together, the candidate network within each species group is sorted and the bottom half of the networks within each group is eliminated. The remainder of networks is then used to generate new candidate networks using the crossover operator.

7.2.5 Algorithm

A general algorithmic process of CPPN is given in Algorithm 10. In the case of solving differential equations, the fitness of the network would be the residual loss of the model on the differential equation:

$$\mathcal{L} = \mathcal{L}_{\text{data}} + \mathcal{L}_{\text{physics}}$$

Algorithm 10 Pseudocode for CPPN

```
procedure CPPN
    for each agent i = 1, ..., N do
        Initialise agent_i's genome with input and output nodes only
        Randomly create connections between the input and output nodes
        Assign a fitness value to agent_i
        Add agent to population set
    while termination criterion not met do
        Divide agents into subspecies using speciation
        Sort the agents within each subspecies based on their fitness
        Eliminate bottom half of agents within each subspecies from population
        Select agents within population for crossover
        Generate new agents using selected candidates
        Update the population of agents
        Mutate subset of population
        Update the fitness scores of the agents
    return Fittest member within the population
```

7.3 Results

In the following section CPPNs are evaluted over the differential equations mentioned above

7.3.1 Logistic Equation

In the following subsection, the solution results on the logistic equation is examined and discussed within this section.

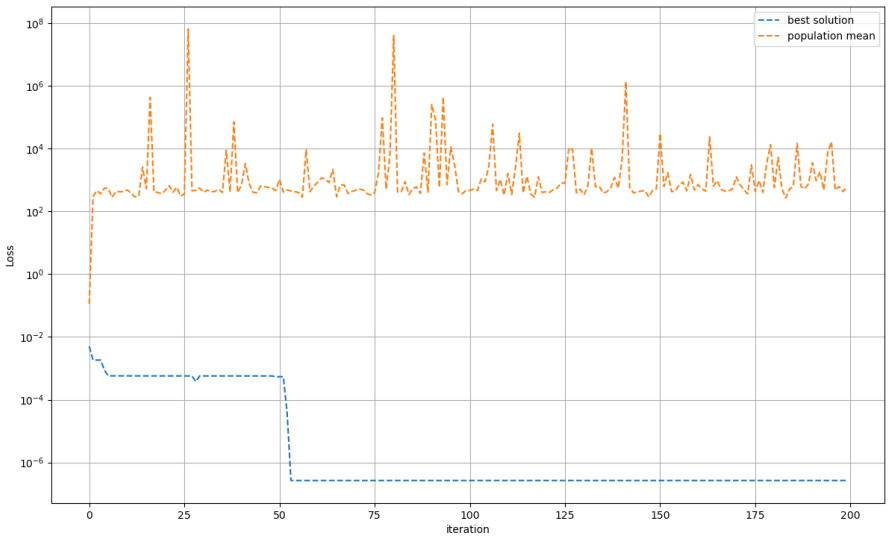


FIGURE 7.4: Convergence Analysis of the CPPN algorithm on the logistic equation

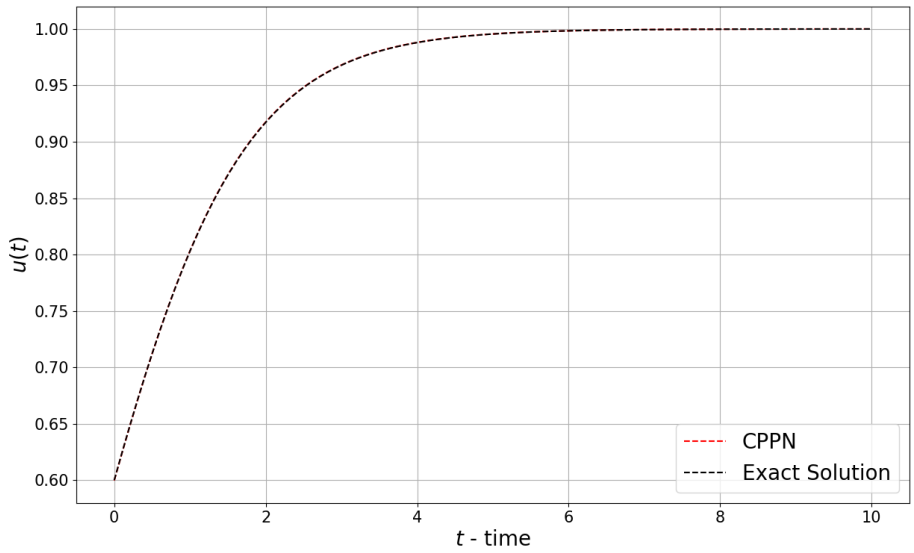


FIGURE 7.5: Comparision of the CPPN algorithm on the logistic equation

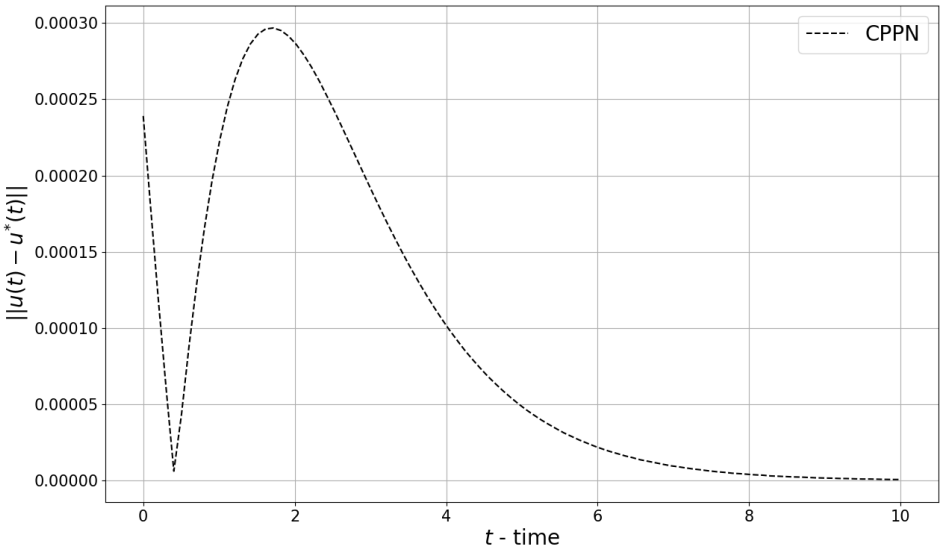


FIGURE 7.6: Error Analysis of the CPPN algorithm on the logistic equation

As observed in Figure 7.4, the best solution converges significantly faster compared to the average of the population. The persistently high loss among the population likely contributes to this swift convergence of the best solution, since the CPPN framework strongly promotes exploration. This exploration process uncovers both superior and inferior solutions, resulting in a higher mean loss compared to that of the best solution. Consequently, as depicted in Figures 7.5,7.6, this approach leads to a highly accurate solution. The enhanced accuracy can be attributed to the use

of distinct activation functions across neurons, enabling a more effective fit to the curve.

7.3.2 Lorenz Equation

In the following subsection, the solution results on the lorenz system of equations is examined and discussed within this section.

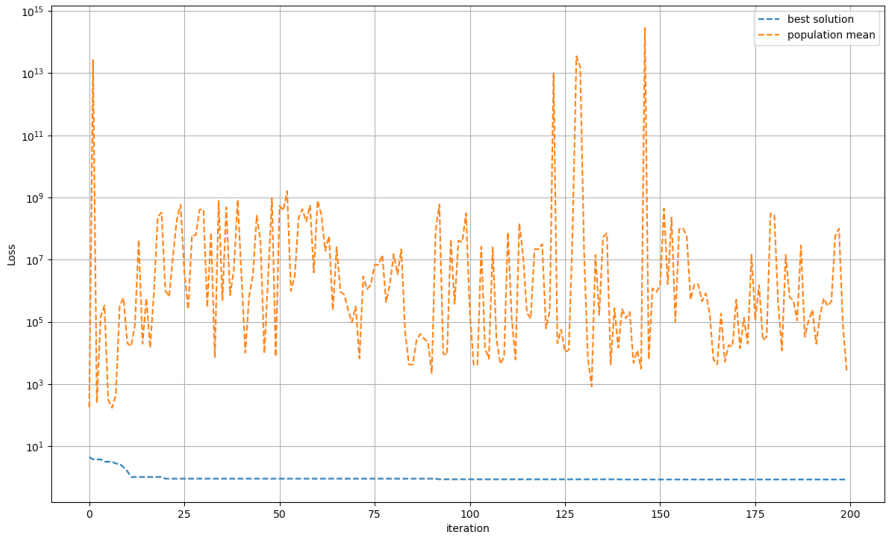


FIGURE 7.7: Convergence Analysis of the CPPN algorithm on Lorenz System of Equations

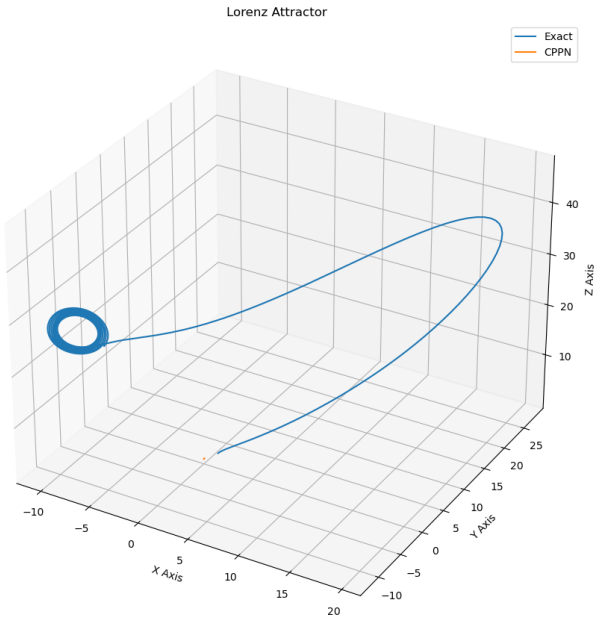


FIGURE 7.8: Comparision of the CPPN algorithm on Lorenz System of Equations

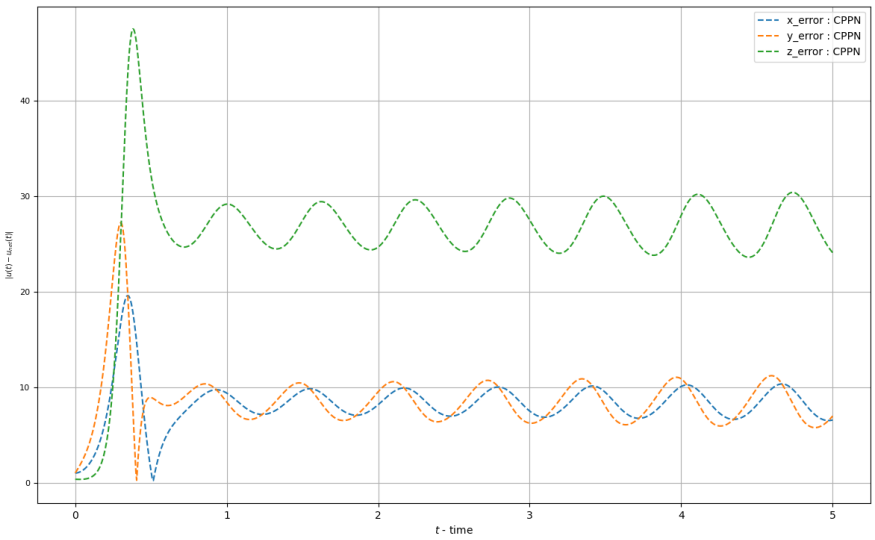


FIGURE 7.9: Error Analysis of the CPPN algorithm on Lorenz System of Equations

Referring to Figure 7.7, there is a noticeable rapid decrease in the convergence of the best solution compared to the average population. This high average loss among the population likely contributes to the swift improvement of the best solution, as the CPPN framework strongly promotes exploration. This exploratory process uncovers both effective and ineffective solutions, resulting in a higher average loss compared to the best solution. Upon reviewing Figures 7.8, 7.9, it becomes evident that the model converged to a trivial solution. Although technically a solution to the Lorenz system equations, a trivial solution offers no insights into the system's dynamics, rendering the model less valuable.

7.3.3 Diffusion Equation

In the following subsection, the solution results on the diffusion equation is examined and discussed within this section.

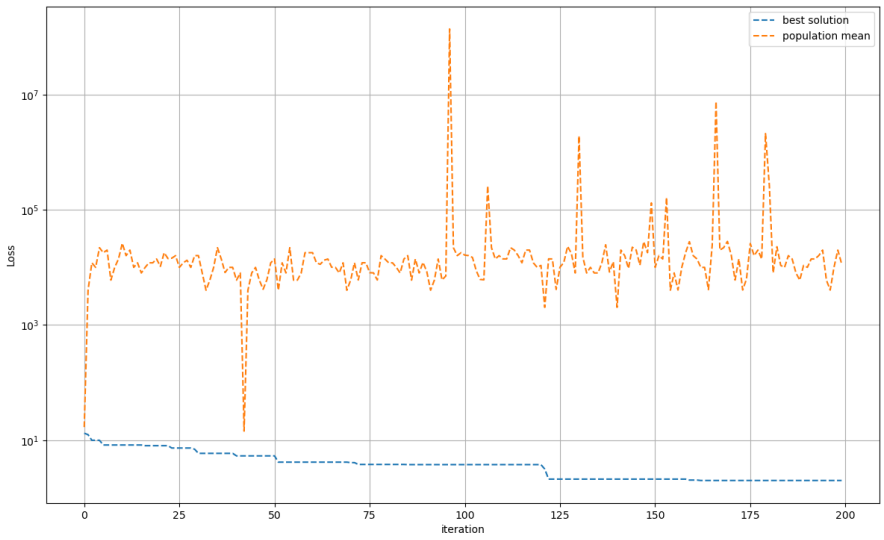


FIGURE 7.10: Convergence Analysis of the CPPN algorithm on the Diffusion equation

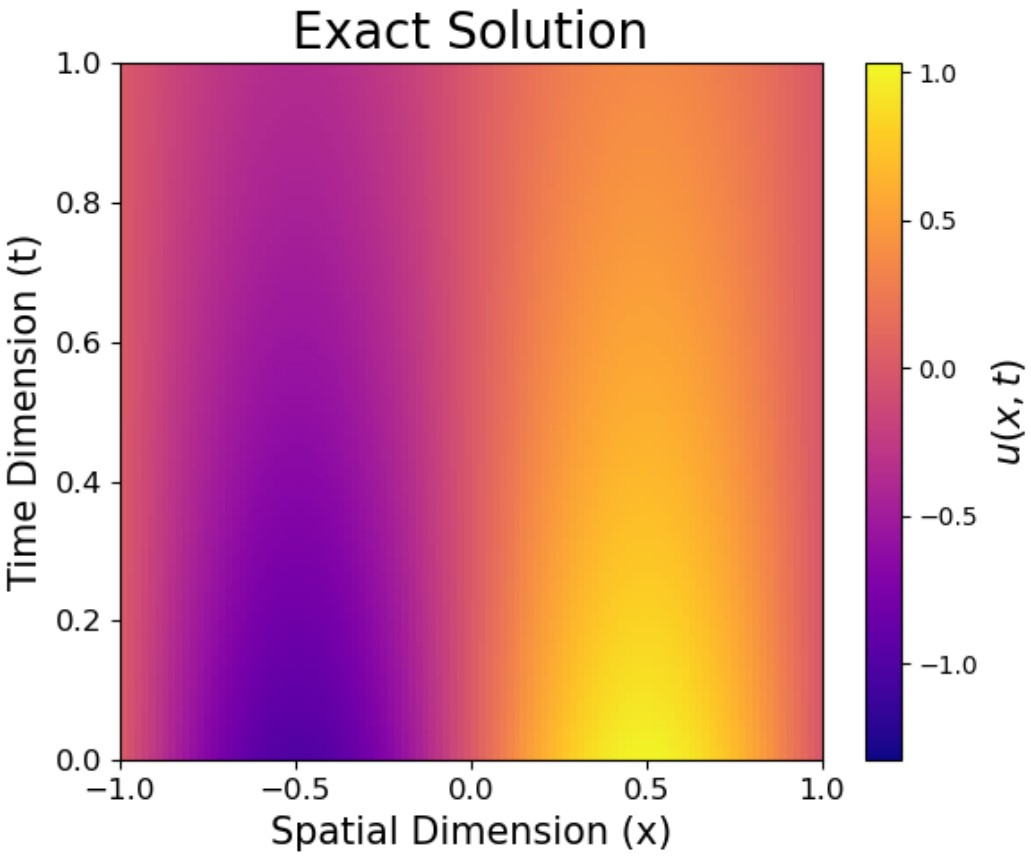


FIGURE 7.11: Ground Truth Solution to the Diffusion equation

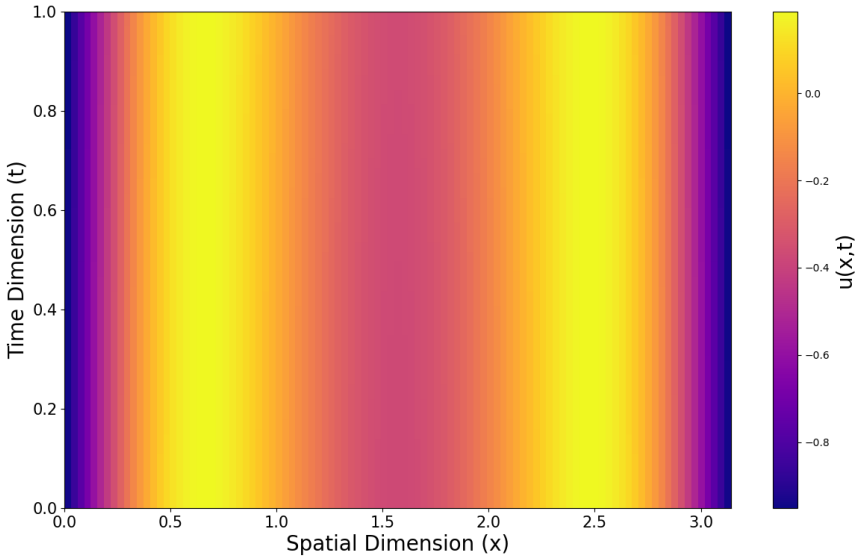


FIGURE 7.12: Comparision of the CPPN algorithm on the Diffusion equation

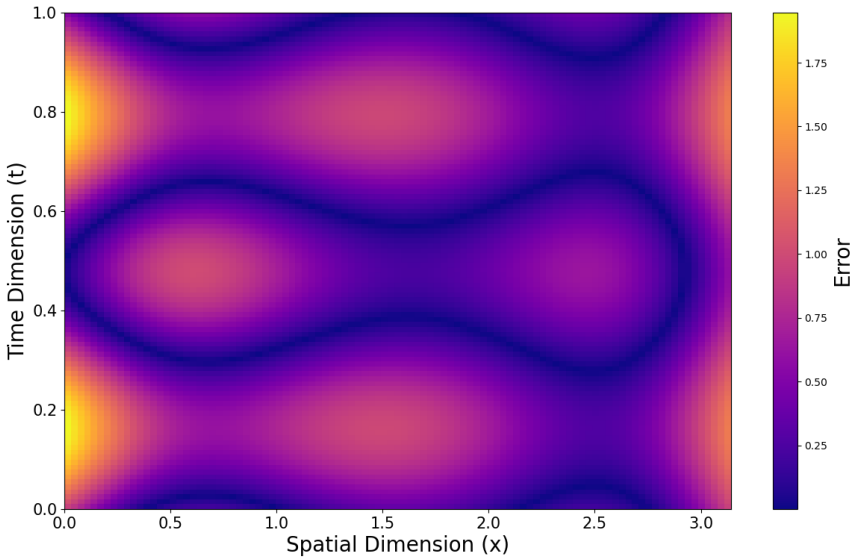


FIGURE 7.13: Error Analysis of the CPPN algorithm on the Diffusion equation

7.3.4 Burgers' Equation

In the following subsection, the solution results on Burgers' equation is examined and discussed within this section.

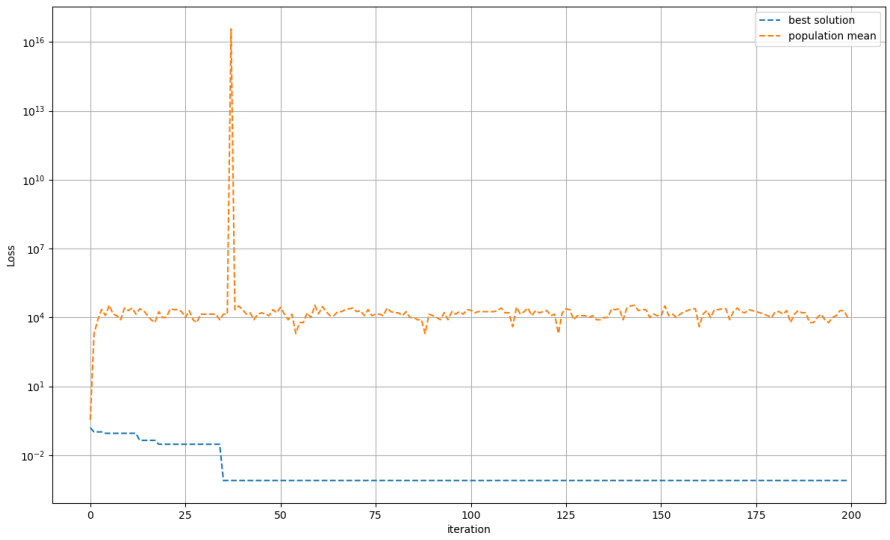


FIGURE 7.14: Convergence Analysis of the CPPN algorithm on Burgers Equation

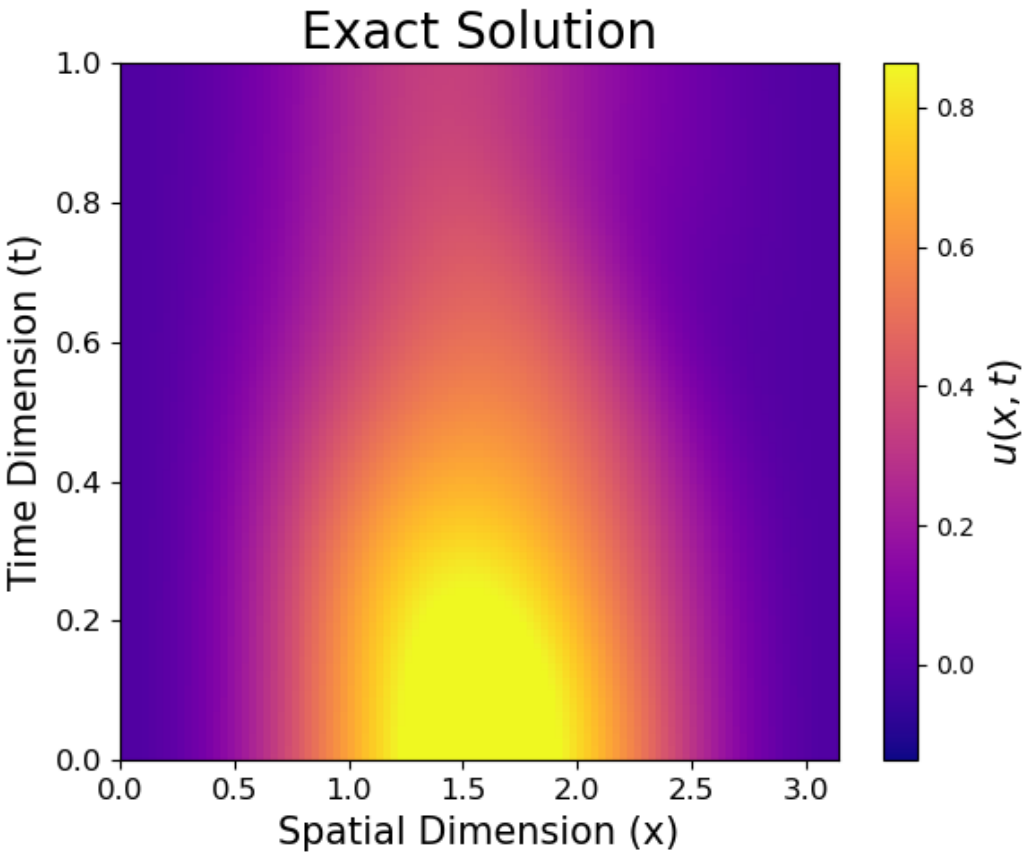


FIGURE 7.15: Ground Truth Solution to Burgers' equation

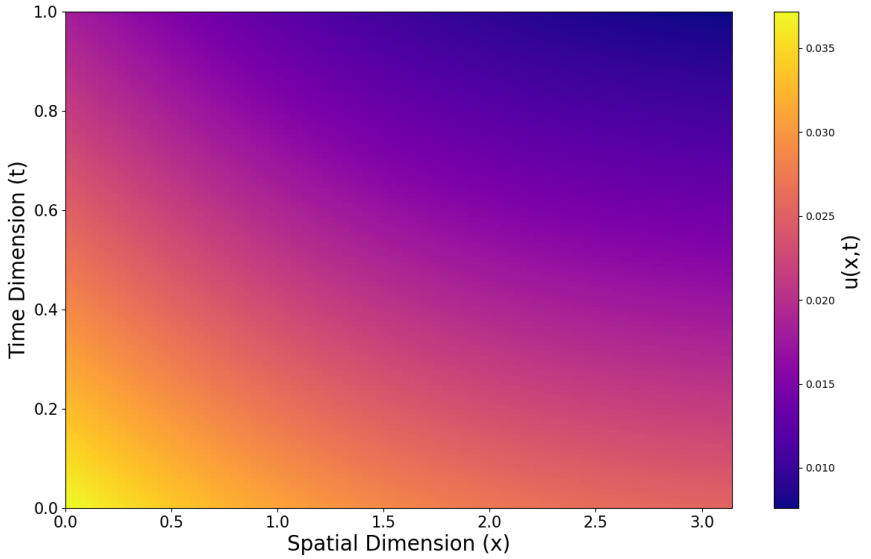


FIGURE 7.16: Comparision of the CPPN algorithm on Burgers Equation

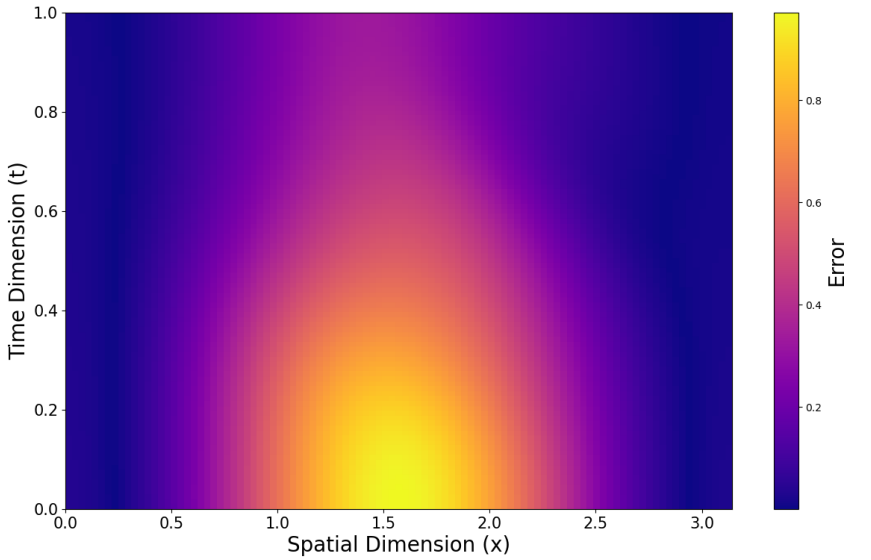


FIGURE 7.17: Error Analysis of the CPPN algorithm on Burgers Equation

As shown in Figure 7.14, there is a marked rapid decline in the convergence rate of the optimal solution compared to the average population. The prevalent high average loss among the population likely accelerates the enhancement of the optimal solution, due to the CPPN framework's strong emphasis on exploration. This exploratory approach reveals both effective and ineffective solutions, leading to a higher average loss relative to the optimal solution. Further examination of Figures 7.16, 7.17 reveals that the model settled on a trivial solution. Although this solution satisfies Burgers' equation, it fails to provide meaningful insights into the system's

dynamics, thereby diminishing the model's utility. Additionally, depending on the parameters used in the differential equation, achieving a solution can be challenging due to the potential formation of shocks.

7.4 Conclusion and Discussion

In summary, the exploration promoted by the CPPN framework plays a pivotal role in both the convergence rates and the quality of solutions obtained, as evidenced in Figures 7.4, 7.7, and 7.14. The framework's tendency to encourage exploration results in a wide variance of solution quality, with the best solutions outperforming the average population markedly. This divergence is primarily due to the high average loss among the population, which propels the swift enhancement of the best solutions.

Notably, while this approach leads to highly accurate solutions in some instances, as highlighted by the results for the curve fitting in Figures 7.5, 7.6, it also results in trivial solutions in more intricate systems such as those described by the Lorenz and Burgers' equations (Figures 7.8, 7.9, 7.16, 7.17). These trivial solutions, while satisfying the respective system equations, fail to offer valuable insights into the underlying dynamics, ultimately limiting the practical utility of the model in these scenarios.

Therefore, while the CPPN framework effectively facilitates rapid convergence and can yield highly accurate solutions, its application must be carefully considered in the context of the problem's complexity and the potential for obtaining trivial solutions that do not enhance our understanding of the system dynamics.

Chapter 8

Conclusion

This thesis has demonstrated that gradient-based optimization techniques outperform heuristic and extreme learning approaches for training machine learning models to solve ODEs and PDEs, primarily because they exploit derivative information to guide updates. In the PINN experiments, L-BFGS leveraged an approximate Hessian to take quasi-Newton steps, achieving errors on the order of 10^{-4} in under 500 iterations; Adam combined momentum with adaptive step sizes to stabilize learning and avoid vanishing or exploding gradients, resulting in similarly low errors; by contrast, vanilla SGD—lacking momentum and second-order curvature—converged more slowly and settled at higher error floors.
The SPIELM chapter showed that randomly initialized hidden layers with analytically determined output weights enable extremely fast, one-shot training. SPIELM achieved comparable accuracy to PINNs while requiring only a single matrix solve, but its fixed random features limited expressive power on highly non-linear or stiff problems like Burgers', where gradient methods retained an advantage.
DeepONet experiments corroborated these findings: convex MSE loss surfaces allowed gradient methods to follow clear descent directions, with SGD's simplicity yielding robust generalization and Adam/L-BFGS occasionally overfitting when their adaptive mechanisms oversteered.
The CPPN framework highlighted the power of structured, generative network encodings: its population-based exploration uncovered some highly accurate outlier solutions by sampling diverse topologies, but also produced trivial basins in complex systems (e.g., Lorenz, Burgers), underscoring that exploration without gradient cues cannot reliably avoid zero-loss traps in intricate landscapes.
Overall, L-BFGS is ideal when memory permits and curvature information is trustworthy; Adam strikes a balance by smoothing noisy gradients and adapting learning rates; SGD remains useful for its low per-iteration cost but demands careful tuning; SPIELM offers unmatched training speed via non-iterative learning; and heuristic methods, while computationally expensive, provide robust global search capabilities, gradient-free operation, and parallelizable evaluation pipelines that excel on non-differentiable or noisy loss surfaces. These mechanistic insights explain why gradient-driven schemes deliver faster convergence and higher accuracy on smooth problems, and why a hybrid strategy—alternating gradient updates with occasional population-based jumps—holds promise for avoiding local minima and trivial solutions.

Recommendations for Future Work

- **Advanced sampling schemes:**Use uniform residual sampling and Latin Hypercube Sampling to ensure diverse training points and prevent models from gravitating toward trivial residual minima.

- **Gradient-penalized residuals:**Add a term penalizing the gradient norm of the PDE residual, which spikes near trivial solutions and thus steers training away from zero-loss basins [47].
- **Genetic encoding of parameters:**Employ autoencoder or wavelet-based encodings to represent network weights in a lower-dimensional space, reducing the cost of evaluating large populations in heuristic search [48], [49].
- **Hybrid optimization schedules:**Alternate batches of gradient-based steps (for rapid descent) with population-based perturbations (for global exploration), combining curvature-aware updates with stochastic escapes from local minima.
- **Meta-learning optimizers:**Investigate algorithms that adaptively tune their own hyperparameters—such as learning rates or momentum coefficients—based on observed loss curvature, improving robustness across diverse PDE/ODE problems.
- **Domain decomposition and multi-fidelity modeling:**Split complex domains into subregions or employ coarse-to-fine fidelity hierarchies to reduce optimization complexity and guide solvers through progressively detailed solution landscapes.

Bibliography

[1] H. Du, W. Li, and N. Zhang, "Semi-active control of an integrated full-car suspension with seat suspension and driver body model using er dampers," *International journal of vehicle design*, vol. 63, no. 2-3, pp. 159–184, 2013.

[2] B. Eickhoff, J. Evans, and A. Minnis, "A review of modelling methods for railway vehicle suspension components," *Vehicle System Dynamics*, vol. 24, no. 6-7, pp. 469–496, 1995.

[3] M. Morisita, "The fitting of the logistic equation to the rate of increase of population density," *Researches on Population Ecology*, vol. 7, no. 1, pp. 52–55, 1965.

[4] A. Y. Pogromsky, G Santoboni, and H Nijmeijer, "An ultimate bound on the trajectories of the lorenz system and its applications," *Nonlinearity*, vol. 16, no. 5, p. 1597, 2003.

[5] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of atmospheric sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[6] A Díaz, J. Alegre, and I. Cuesta, "Coupled hydrogen diffusion simulation using a heat transfer analogy," *International Journal of Mechanical Sciences*, vol. 115, pp. 360–369, 2016.

[7] Y. Kim, H. Jeon, and H. Othmer, "The role of the tumor microenvironment in glioblastoma: A mathematical model," *IEEE Transactions on Biomedical Engineering*, vol. 64, no. 3, pp. 519–527, 2016.

[8] J. M. Chadam and H.-M. Yin, "A diffusion equation with localized chemical reactions," *Proceedings of the Edinburgh Mathematical Society*, vol. 37, no. 1, pp. 101–118, 1994.

[9] S. Henn, "A multigrid method for a fourth-order diffusion equation with application to image processing," *SIAM Journal on Scientific Computing*, vol. 27, no. 3, pp. 831–849, 2005.

[10] A Kolmogoroff, I Petrovsky, and N Piscounoff, "Study of the diffusion equation with growth of the quantity of matter and its application to a biology problem," in *Dynamics of curved fronts*, Elsevier, 1988, pp. 105–130.

[11] J. M. Burgers, "A mathematical model illustrating the theory of turbulence," *Advances in applied mechanics*, vol. 1, pp. 171–199, 1948.

[12] J. D. Cole, "On a quasi-linear parabolic equation occurring in aerodynamics," *Quarterly of applied mathematics*, vol. 9, no. 3, pp. 225–236, 1951.

[13] Y. Tsuji, T. Yanuma, I. Murata, and C. Fujiwara, "Tsunami ascending in rivers as an undular bore," *Natural Hazards*, vol. 4, pp. 257–266, 1991.

[14] M. J. Lighthill and G. B. Whitham, "On kinematic waves ii. a theory of traffic flow on long crowded roads," *Proceedings of the royal society of london. series a. mathematical and physical sciences*, vol. 229, no. 1178, pp. 317–345, 1955.

[15] J. Lindgren, "Efficient markets and contingent claims valuation: An information theoretic approach," *Entropy*, vol. 22, no. 11, p. 1283, 2020.

[16] A Paszke, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[17] A. Lee, *pyDOE: Experimental design for python*, Python Package Index (PyPI), Version 0.3.8, released July 13, 2015. [Online]. Available: https://pypi.org/project/pyDOE/.

[18] G. S. Call and D. J. Velleman, "Pascal's matrices," *The American Mathematical Monthly*, vol. 100, no. 4, pp. 372–376, 1993.

[19] H. Levy and F. Lessman, *Finite difference equations*. Courier Corporation, 1992.

[20] A. Solomonoff, "A fast algorithm for spectral differentiation," *Journal of Computational Physics*, vol. 98, 1992.

[21] J. A. Weideman and S. C. Reddy, "A matlab differentiation matrix suite," *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 4, pp. 465–519, 2000.

[22] R Baltensperger and J.-P. Berrut, "The errors in calculating the pseudospectral differentiation matrices for cebysev-gauss-lobatto points," *Computers & Mathematics with Applications*, vol. 37, no. 1, pp. 41–48, 1999.

[23] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in pytorch," 2017.

[24] A. Griewank *et al.*, "On automatic differentiation," *Mathematical Programming: recent developments and applications*, vol. 6, no. 6, pp. 83–107, 1989.

[25] L. N. Trefethen, *Spectral methods in MATLAB*. SIAM, 2000.

[26] R. Haupt, "Comparison between genetic and gradient-based optimization algorithms for solving electromagnetics problems," *IEEE Transactions on Magnetics*, vol. 31, no. 3, pp. 1932–1935, 1995.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] A. S. Berahas and M. Takáč, "A robust multi-batch l-bfgs method for machine learning," *Optimization Methods and Software*, vol. 35, no. 1, pp. 191–219, 2020.

[29] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: Optimizing connections and connectivity," *Parallel computing*, vol. 14, no. 3, pp. 347–361, 1990.

[30] R. Eberhart and J. Kennedy, "Particle swarm optimization," in *Proceedings of the IEEE international conference on neural networks*, Citeseer, vol. 4, 1995, pp. 1942–1948.

[31] R. S. Sexton, R. E. Dorsey, and J. D. Johnson, "Optimization of neural networks: A comparative analysis of the genetic algorithm and simulated annealing," *European Journal of Operational Research*, vol. 114, no. 3, pp. 589–601, 1999.

[32] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[34] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evolutionary intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[35] R. Dawkins *et al.*, "Universal darwinism," *Evolution from molecules to men*, pp. 403–425, 1983.

[36] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic programming and evolvable machines*, vol. 8, pp. 131–162, 2007.

[37] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations," *arXiv preprint arXiv:1711.10561*, 2017.

[38] 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*, vol. 378, pp. 686–707, 2019.

[39] A. P. Engelbrecht, *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[40] P. Yin, S. Ling, and W. Ying, "Chebyshev spectral neural networks for solving partial differential equations," *arXiv preprint arXiv:2407.03347*, 2024.

[41] Y. Wang and S. Dong, "An extreme learning machine-based method for computational pdes in higher dimensions," *Computer Methods in Applied Mechanics and Engineering*, vol. 418, p. 116 578, 2024.

[42] B. Meuris, S. Qadeer, and P. Stinis, "Machine-learning-based spectral methods for partial differential equations," *Scientific Reports*, vol. 13, no. 1, p. 1739, 2023.

[43] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1-3, pp. 489–501, 2006.

[44] O. Otegbeye, "On decoupled quasi-linearization methods for solving systems of nonlinear boundary value problems.," Ph.D. dissertation, 2014. [Online]. Available: http://hdl.handle.net/10413/12114.

[45] L. Lu, P. Jin, and G. E. Karniadakis, "Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators," *arXiv preprint arXiv:1910.03193*, 2019.

[46] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[47] R. Leiteritz and D. Pflüger, "How to avoid trivial solutions in physics-informed neural networks," *arXiv preprint arXiv:2112.05620*, 2021.

[48] P. Templier, E. Rachelson, and D. G. Wilson, "A geometric encoding for neural network evolution," in *Proceedings of the genetic and evolutionary computation conference*, 2021, pp. 919–927.

[49] S. Van Steenkiste, J. Koutník, K. Driessens, and J. Schmidhuber, "A wavelet-based encoding for neuroevolution," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 517–524.