Freie Universität Berlin

Optimizers for training Physics-informed Neural Networks

Master Thesis submitted in fulfillment of the requirements
for the degree M.Sc Mathematics

Submitted by:	Hend Naimi
matriculation number:	5562310
Supervisor:	Univ.-Prof. Dr. Volker John
Second Reviewer:	PD Dr. Alfonso Caiazzo

Department of Mathematics and Computer Science
Freie Universität Berlin

December 18, 2024

Freie Universität Berlin

Fachbereich Mathematik, Informatik und Physik

SELBSTSTÄNDIGKEITSERKLÄRUNG

Name:	Naimi	(BITTE nur Block- oder Maschinenschrift verwenden.)
Vorname(n):	Hend	
Studiengang:	Mathematik, M.Sc.	
Matr. Nr.:	5562310	

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende ___Masterarbeit___________ selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: 18.12.2024__ Unterschrift: __

Acknowledgments

My heartfelt gratitude goes first to Uni.-Prof. Dr. Volker John for his unwavering support and guidance throughout my research journey. From the time I took Numerics 2 course to the completion of this thesis. I am deeply grateful for the invaluable insights and encouragement he provided, which made this work possible. I would also like to extend my sincere appreciation to PD Dr. Alfonso Caiazzo for serving as the second supervisor of my thesis. My gratitude further goes to Marwa Zainelabdeen and Derk Frerichs-Mihov for their assistance and for facilitating the code and resources needed for this thesis.

To my beloved mother, Imen, Mariem and my brothers, thank you for your endless love and support throughout this journey. Your belief in me has been my greatest source of strength. Lastly, to the soul of my father, I dedicate this work to his memory and hope that he is proud of my achievements.

Contents

1 Introduction **3**
- 1.1 Motivation 3
- 1.2 Outline 4

2 The Artificial neural network **6**
- 2.1 Machine learning 6
- 2.2 Artificial neural network 8
  - 2.2.1 Neural network architecture 8
  - 2.2.2 Activation function 10
  - 2.2.3 Tuning hyperparameters 12
  - 2.2.4 Loss functional 12
- 2.3 Backpropagation process 13

3 Optimizers in training the neural network **17**
- 3.1 Stochastic gradient descent 17
  - 3.1.1 Gradient descent method 18
  - 3.1.2 SGD algorithm 20
- 3.2 Adaptive momentum estimation 21
  - 3.2.1 Adam Algorithm 21
  - 3.2.2 Nadam optimizer 25
- 3.3 Broyden–Fletcher–Goldfarb–Shanno 26
  - 3.3.1 Approximation of the Hessian matrix 27
  - 3.3.2 BFGS 28

4 PINNs for convection-diffusion-reaction problem **31**
- 4.1 Strong form of the convection-diffusion-reaction problem 31
- 4.2 Physics-informed neural networks 35
  - 4.2.1 Collocation points 37
  - 4.2.2 Standard loss functional 38
- 4.3 Hard-constrained PINNs 43

5 Numerical Studies **47**
- 5.1 Benchmark Problem Overview 48
  - 5.1.1 Circular interior layer 48
  - 5.1.2 Outflow layer 49

5.1.3 Hyperparameters for the training process . . . 50
5.1.4 Metrics for Model Implementation . . . 52
5.1.5 Challenges of implementing BFGS . . . 53
5.2 Results . . . 54
5.2.1 Analysis method . . . 54
5.2.2 Circular interior layer . . . 55
5.2.3 Outflow layer . . . 62

6 Conclusion **70**
6.1 Summary . . . 70
6.2 Implications for future work . . . 71

List of Figures

2.1 Overfitting in Machine Learning: Insights from Classification and Regression [Bab]. 7
2.2 Representation of an artificial neuron's components, including the weights, the net input, the activation function and the activation output [CdCV18]. . 9
2.3 Representation of simple neural network architecture with two hidden layers with 4 neurons per layer, an input layer with 3 nodes and output layer with single node. 10
2.4 Sigmoid (2.3) and Tanh (2.4) activation functions [Ajm23]. 11
2.5 GeLU (2.5) and Swish (2.2) activation functions [Ajm23]. 11

3.1 Representation of steepest descent direction for a function of two variables [NW06]. 18

4.1 Schematic of a physics-informed neural network (PINN) [CMW+21] 36
4.2 Domain, direction of convection field and exact solution of 4.2.2. Lines with ticks indicate Dirichlet boundary Ω_D and lines without ticks indicate Neumann boundary Ω_N [FM23]. 40
4.3 Representation of the FNN model's layers, nodes, number of parameters and output shape used in training PINN to approximate Example 4.2.2. 41
4.4 Training points [FM23] and history of loss functional values for PINN approximation to Example 4.2.2. 42
4.5 PINN approximated solution u_N and point-wise error for Example 4.2.2. . . 43
4.6 Hard-constrained PINN approximated solution u_N and point-wise error for Example 4.3. 45
4.7 History of loss functional values for hard-constrained PINN approximation to Example 4.3. 45

5.1 Exact solution of Circular Interior Layer example 5.1 49
5.2 Exact solution of Outflow Layer example (5.2) 50
5.3 The indicator function for the Circular Interior Layer 5.1.1. 52
5.4 The indicator function of the Outflow Layer 5.1.2. 53
5.5 Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.1 after training over 1,000 epochs. . . 56
5.6 Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.1 after training over 1,000 epochs. . . 57

5.7 Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.1 after training over 1,000 epochs. . . 58
5.8 History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam optimizers over 1,000 epochs to Example 5.1.1. 59
5.9 History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam optimizers over 10,000 epochs to Example5.1.1. 60
5.10 Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.1 after training over 10,000 epochs. . . 61
5.11 Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.1 after training over 10,000 epochs. . . 61
5.12 Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.1 after training over 10,000 epochs. . 62
5.13 Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.2 after training over 1,000 epochs. . . . 63
5.14 Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.2 after training over 1,000 epochs. . . 64
5.15 Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.2 after training over 1,000 epochs. . . 65
5.16 History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam over 1,000 epochs to the Example (5.2). 66
5.17 History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam over 10,000 epochs to the Example (5.2). . . . 67
5.18 Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.2 after training over 10.000 epochs. . . 68
5.19 Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.2 after training over 10.000 epochs. . . 68
5.20 Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.2 after training over 10.000 epochs. . 69

List of Tables

3.1 Computational complexity of SGD optimizer [RZS20]. 20

5.1 Values of hyperparameters used in training hard-constrained PINN. 50
5.2 Values of L^2 error, loss functional, and training time over 1,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Circular Interior Layer problem 5.1.1. 59
5.3 Values of L^2 error, loss functional, and training time over 10,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Circular Interior Layer problem 5.1.1. 60
5.4 Values of L^2 error, loss functional, and training time over 1,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Outflow Layer problem 5.1.2. 65
5.5 Values of L^2 error, loss functional, and training time over 10,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Outflow Layer problem 5.1.2. 66

Chapter 1

Introduction

1.1 Motivation

Solving partial differential equations (PDEs) in high-dimensional domains using classical approaches, such as finite elements and finite difference methods, is not always the best solution, due to complexity issues [KKL+20]. In recent years, solving these complex PDEs using advanced computational intelligence models has been among the most interesting topics for many researchers. Many studies focused on applying the artificial neural network (ANN) as an alternative approach to solve these complex problems. However, the effectiveness of ANN requires a large amount of data, in order to guarantee an optimal solution during the training. Collecting large size of data is expensive and prohibitive. On the other hand, having a limited size of data becomes challenging. It causes lack of robustness and the model becomes sensitive to any small changes made to the data [RPK17].

To address this challenge, a new method has been developed that combines the neural networks (NN) with physics-informed methodologies. The most popular paper in this field was published by Maziar Raissi, Paris Perdikaris and George Em Karniadakis [RPK17]. The approach is based on training NN model by adding all the additional information that comes from the physics side by general nonlinear PDE and it is called physics-informed neural network (PINN). PINNs represent a significant advancement, as they are capable of integrating data with inherent noise into the NN, discovering underlying physical principles and effectively tackling problems with a high number of dimensions [KKL+20].

This master thesis explores the training of PINN using different widely-used optimizers. Selecting the right optimizer for training a PINN model is a crucial decision, as it directly impacts the model's speed, performance and accuracy. Additionally, the selected optimizer's sensitivity with respect to the hyperparameters must not be overlooked. [CSN+20]. The optimizers, that are chosen for this study, are the stochastic gradient descent method (SGD), Adaptive Moment Estimation (Adam), Nadam and Broyden-Fletcher-Goldfarb-Shanno (BFGS).

Many recent research, that focus on the comparisons of optimization algorithms, overlook

the relevance of the inclusion relationships between the optimizers or the choice of the hyperparameters in ways that break their inclusions [CSN+20]. Currently, there is no theoretical guidance that practitioners follow to choose the right optimizer for them. Instead, they rely on empirical studies and benchmarking. The comparison is actually conducted across variety of tasks and then concentrated on the optimizer's effectiveness. However, finding the optimal hyperparameters for the optimizers to match their performance together, is extremely difficult and sometimes even impossible [CSN+20].

SGD was initially chosen for its simplicity and straightforward implementation in training PINN [KB17]. However, leveraging the inclusion relationship among optimizers, further Adam is employed. This inclusion relationship indicates that one optimizer can approximate or encompass the behavior of another. For instance, Adam can simulate the performance of SGD by tuning its hyperparameters. This characteristic of inclusion ensures that as hyperparameter tuning approaches an optimal configuration, the more expressive optimizer, Adam, will not underperform relative to any of its simpler specializations [CSN+20]. Nadam is a recent alternative variation of the Adam optimizer. It is a combination between Adam algorithm and Noestrov acceleration gradient. Additionally, BFGS algorithm is selected as last optimizer, due to its strength in handling second-order information. BFGS, being a quasi-Newton method, efficiently approximates the Hessian matrix, thus accelerating convergence in complex, high-dimensional landscapes often encountered in PINN [NW06]. This combination of optimizers aims to balance simplicity, expressiveness, and rapid convergence for optimal performance in training the PINN model. Those optimizers will be discussed more in details in the third chapter of this master thesis.

This study focuses on the convection-diffusion-reaction problem, a complex mathematical representation critical for modeling various natural and engineered systems, making it an ideal choice for evaluating the effectiveness of PINN. The training process of the PINN looks to find an optimal solution to the loss function to determine the best parameters. In chapter four, a special PINN architecture will be explored, Hard-Constrained, highlighting how their differences in loss function setup and structural design influence the quality and efficiency of solutions. This analysis will clarify the role of these variations in improving PINN performance for complex PDE such as the convection-diffusion-reaction problem.

1.2 Outline

The main focus of this master's thesis is comparing optimizers for training PINN, particularly in solving convection-diffusion-reaction problems. The second chapter gives an introduction to machine learning and detailed explanation of the ANN, exploring its architecture, hyperparameters, loss functional. At the end of this chapter, the backpropagation equations are proved and gathered in an algorithm.

The third chapter explores four different optimizers in depth, which are SGD, Adam, Nadam and the BFGS algorithm. This part focuses on the optimizers mechanisms, strengths, and challenges in NN training.

The fourth chapter provides the PINN framework tailored for the convection-diffusion-reaction problem. Hard-constrained PINNs are described to showcase special approaches in embedding physics directly into the network, in order to incorporate the inlet boundary conditions into the training process rather than being learned.

Th final chapter investigates a comparative analysis of the optimizers mentioned in the previous chapter. The analysis is conducted using hard-constrained PINN, assessing each optimizer's effectiveness based on convergence speed, accuracy, computational efficiency and the approximation of the solutions.

Chapter 2

The Artificial neural network

A brief overview of machine learning is presented, highlighting its key principles and significance. The second and third section delves into the components and workflow of the NN model, providing a detailed explanation of its structure and functioning.

2.1 Machine learning

A machine learning is a subfield of artificial intelligence that develops algorithms, capable of learning patterns from data. It improves a computer's performance at some task by learning via experience based on data-driven insights. Machine learning tasks are usually described in terms how the machine learning system should process a sample. A sample is a collection of features that have been quantitatively measured from some object or event, represented as a vector $x \in \mathbf{R}^n$, with x_i as a distinct feature [GBC16].

The most common machine learning tasks are:

- Classification: it splits the input data into k categories. Then, the algorithm produces a function $f : \mathbf{R}^n \to \{1, 2, ..., k\}$ that maps a vector x to a category.
- Regression: it predicts a continuous numerical value based on a given input. The algorithm produces a function $f : \mathbf{R}^n \to \mathbf{R}$ that estimates a target value associated with the vector x.

Using the experience, we can categorize the machine learning into supervised and unsupervised learning algorithms.

- Unsupervised learning algorithm: experience a dataset containing hidden features. So, the algorithm extracts the properties and learns the entire probability distribution that generated this dataset. In other words, the algorithm is observing several samples of a random vector x and attempts to implicitly and explicitly to learn the probability distribution $P(x)$.

- Supervised learning algorithm: experience a dataset incorporating features. At the same time, each sample is associated with an existing output. The algorithm observes several examples of a vector x and an associated value y, which is the existing output then learning to predict $\hat{y}$ from x by estimating $P(y/x)$.

Machine learning algorithm requires always a training dataset, which provides labeled samples. The goal of this dataset is to build a relationship between the target output and the input features, such as in the classification case. Another separate set, called the test dataset, is essential in evaluating the performance of the learning algorithm with a new, unseen data [GBC16].

To evaluate the capabilities of a machine learning algorithm, performance quantitative metrics are designed. A common metric is accuracy, which is the proportion of samples for which the model produces a correct output. An equivalent metric is the error rate, which is the proportion of samples for which the model produces an incorrect output. This is referred as 0-1 loss, where 0 if it correctly classified and 1 if it is not [GBC16]. In some other tasks, it does not make sense to measure the accuracy. For example in regression, a different performance metric is used. It gives the model a continuous valued score for each sample [GBC16].

In some cases, a learning algorithm performs perfectly well on the training dataset, but struggles on the test dataset. This situation is known as Overfitting. Overfitting occurs when the model memorizes the training data, including its noises, rather than identifying the underlying patterns. As a result, the model loses accuracy when applied to new, unseen data [Bab].

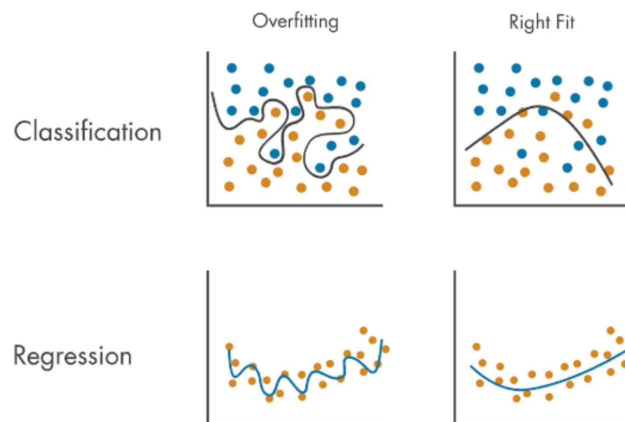


Figure 2.1: Overfitting in Machine Learning: Insights from Classification and Regression [Bab].

While choosing the appropriate performance metric is crucial for the evaluation of any machine learning algorithm, it is also important focusing on the model's underlying architecture. Artificial NN have become a powerful and flexible approach for handling complex

datasets and achieving high accuracy in both classification and regression tasks. By leveraging interconnected layers of neurons, NN enable advanced learning capabilities, making them a fitting choice for tackling diverse and intricate problems.

2.2 Artificial neural network

In this section, the key terminologies related to the NN model are clarified, including the loss functional, the activation function. Each term is explained and presented by their respective roles. Furthermore, the inner workings of the layers and neurons are examinated to provide a deeper understanding of how the network operates. Finally, the backpropagation process is explained in detail, including its role and its mathematical formulation.

The foundational model of NN was introduced by Warren McCulloch and Walter Pitts in 1943. Their model was based on building binary switches, which are components that can turn on or off ,analogous to bilogical neurons. They demonstrated that even simple networks made from these switches are capable of solving a range of almost any logical or arithmetic functions [Kri07]. Another model was introduced in 1960 by Bernard Widrow and Marcian E.Hoff, called Adaline-Adaptive Linear Neuron. This model was very simple, consisting of a one single layer model, and it focuses on minimizing the loss function to update its parameter. However, this model is considered as very simple NN that cannot solve complex, nonlinear problems [NW06].

NN experienced a surge of interest from their inception until 1969, a period often known as Golden age and following this period the field saw a slower silence of development [Kri07].

2.2.1 Neural network architecture

This subsection outlines the computation process inside the units of NN, the computation inside the artificial neuron and the propagation from the input layer to the last output layer. The NN's architecture plays an important role, since it directly impacts the model's performance. The discussed NN model is a feedforward neural network (FNN). In this model, the input flows strictly forward from the input layer through each hidden layer, without any cycles or feedback loops, ensuring a clear path from inputs to outputs [HH18].

The fundamental building block of the FNN is the artificial neuron, which performs critical computations that drive the model's functionality. The computation process inside the artificial neurons is modeled after the idea of biological neurons inside the human brain. In biological neurons, signals are received through dendrites, which pass an electrical impulse to the cell body. If this impulse reaches a certain threshold, it triggers an action potential that goes down the axon and releases neurotransmitters at the synapses, signaling other neurons. The same artificial process is explained based on the Figure 2.2 step by step.

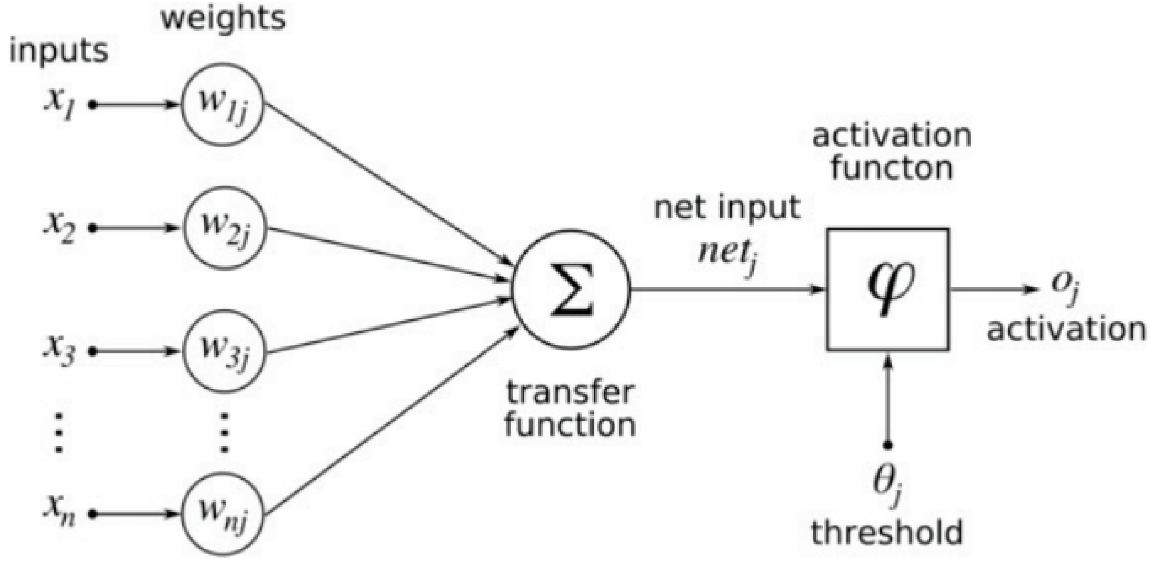


Figure 2.2: Representation of an artificial neuron's components, including the weights, the net input, the activation function and the activation output [CdCV18].

As shown in Figure 2.2, each neuron receives an input $x \in \mathbf{R}^n$ with $n \in \mathbf{N}$, where $x = (x_1, x_2, ..., x_n)$. The input may be either from a previous neuron or an external source. The input is first multiplied by a corresponding *weight* $w \in \mathbf{R}$ and next the multiplied weighted inputs get summed all together. Then, a *bias* $b \in \mathbf{R}$ is added to the result, forming what is called *the net input*. Afterward, the net input is further processed by *the activation function* σ [Kri07], which is a nonlinear function, that determines the neuron's output, known as *the activation* and denoted by a.

The computation of the output of an arbitrary k neuron is represented as:

$$a^k = \sigma(W^k a^{k-1} + b^k), \tag{2.1}$$

where W is the matrix of the weights, with the columns are the indices of the neurons in the previous layer $k-1$, and the rows are the indices of the neuron in the layer k. The entries will be denoted by w^k_{jl}, it is the weight from the l-th node in the $k-1$ layer connected to the node of index j-th in the layer k. The weights control the strength of the connections between neurons, while the bias is an additional input to the neuron that helps to adjust the output of the activation function [Sal23].

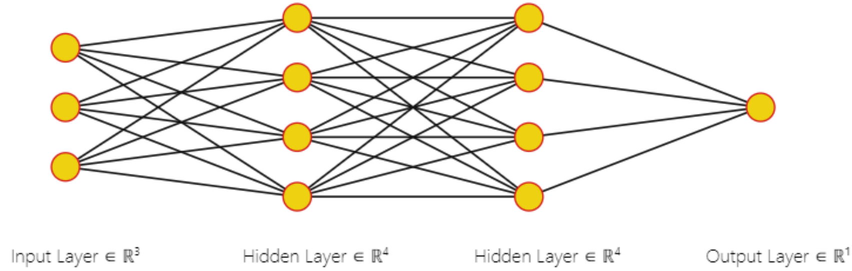


Figure 2.3: Representation of simple neural network architecture with two hidden layers with 4 neurons per layer, an input layer with 3 nodes and output layer with single node.

The neurons are arranged into blocks, known as *layers*. Each layer can contain at least one neuron. The layers in the FNN model are typically three types. The first type is *the input layer*, where no computation is performed, but it simply receives the raw inputs and passes them to the next layer. The second type is *the output layer*, which produces the output of the neural network. In between there are *the hidden layers*. Those intermediate layers perform the core computations. The model can have one or more hidden layers. The number of neurons and layers in the model determines its capacity and ability to capture complex relationships in the data.

In summary, forward propagation is the process by which input data is passed through a FNN to generate an output. It involves computing the output of each neuron in each layer of the network by applying the weights and biases to the inputs and passing the results through an activation function [Sal23].

2.2.2 Activation function

As shown in the Figure 2.2, the net input passes through a threshold nonlinear function called activation function. Similar to the biological neurons, the output of the activation function determines whether the neuron will get activated or not. The activation depends actually on the previous activation state of the neuron and the current input [Kri07]. If the input is large enough then the neuron is activated, otherwise it will remain inactive [HH18]. Each neuron has a unique threshold value, which marks the position where the gradient of the activation function reaches its maximum [Kri07].

The choice of the activation function should not be arbitrary. The following are the most common used activation functions:

1. **The Swish function**

$$\sigma(x) = \frac{1}{+e^{\beta x}}. \tag{2.2}$$

2. **The Sigmoid function**

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.3}$$

3. **The Tanh function**

$$\sigma(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.4}$$

4. **The GeLU function**

$$\sigma(x) = 0,5x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0,044175x^3)\right)\right) \tag{2.5}$$

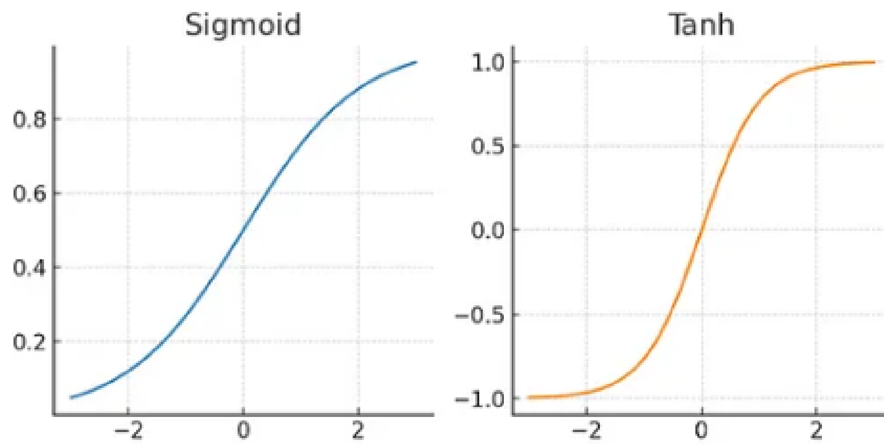


Figure 2.4: Sigmoid (2.3) and Tanh (2.4) activation functions [Ajm23].

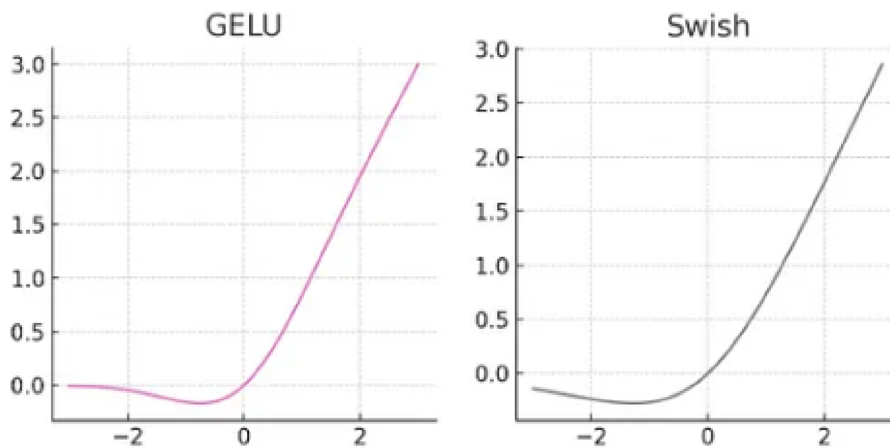


Figure 2.5: GeLU (2.5) and Swish (2.2) activation functions [Ajm23].

The plots of these activation functions are presented in Figures 2.4 and 2.5. Over the years, mathematicians found out that simpler the activation function tend to yield better performance [Cas20]. While a variety of activation functions exist, such as the Sigmoid and Tanh functions, they share a common limitation: they are prone to the vanishing gradient problem. This issue arises during backpropagation, the process where gradients are propagated backward through the network to update the weights and biases. If the gradients are less than one, their repeated multiplication leads to progressively smaller values. Consequently, these gradients can diminish to the point where they effectively approach zero, resulting in minimal or no updates to the weights and biases during training [NIGM18].

The outputs of the tanh and Sigmoid functions are confined to a limited range, which results in smaller gradient values. These gradients, when used in backpropagation, can contribute to the vanishing gradient problem. In addition, the activation function should possess more properties, which are non-linearity, because it helps detecting the complex relationships between the inputs and the outputs, and the differentiability, which will be used in the backpropagation [Cas20].

2.2.3 Tuning hyperparameters

Besides model parameters, hyperparameters play a crucial role in FNN by controlling the convergence, accuracy, and robustness of the model. The hyperparameters include the number of layers or neurons, batch size, number of epochs, learning rate, etc. Despite extensive research, there is still no agreement on a single FNN architecture that guarantees optimal convergence and accuracy.

The learning rate is the size of the steps the model takes toward the convergence. If the learning rate value is high, the model can overshoot the optimal solution. Conversely, if the learning rate value is too low, the convergence may take too long and make the training process inefficient. Batch size is another important factor. A large batch size processes more data at once before updating the parameters, which can be memory intensive. However, a small batch size may result in higher variance in parameter updates. Similarly to the number of epochs, if the number is too high, it can lead to overfitting, where the model performs well on training data but poorly on unseen data [Sod22].

To avoid these problems, hyperparameter tuning is used. Many techniques exist to identify the optimal combination of hyperparameters for a given model. A commonly used approach is manual search, where different hyperparameter combinations are tested to find the best-performing one [Sod22].

2.2.4 Loss functional

Training a FNN is a process adjusting the model's parameters, weights and biases, to improve the model's performance on a given task. During the training phase, the FNN uses a loss functional. The model updates iteratively its parameters to minimize this loss functional, in order to improve its ability to make accurate predictions. A key goal is to achieve a smaller

value of the loss functional. There are various types of loss functionals, each is related to a different task.

1. **Cross entropy**
$$\mathcal{L}(p) = -\sum \hat{y}(x) \log(\sigma(x; p)). \tag{2.6}$$
with x is the input feature, the $y(x)$ is the target output, the $\hat{y}(x)$ is the predicted output, the $\mathcal{L}$ is the loss functional and the p is the parameter vector gathering the weights and the biases together.

2. **Mean squared error (MSE)**
At the last step of the forward propagation phase, MSE computes the difference between the predicted outputs of the model and the actual target values in the training data
$$\mathcal{L}(p) = \frac{1}{N} \sum_i \|y(x) - \hat{y}(x)\|^2 . \tag{2.7}$$
with x is the input feature, the $y(x)$ is the target output, the $\hat{y}(x)$ is the predicted output, the $\mathcal{L}$ is the loss functional and the p is the parameter vector gathering the weights and the biases together.

In addition, the loss functional is typically defined as the average of the loss functionals for individual loss functionals for each training samples:

$$\mathcal{L}(p) = \sum_{i=1}^{N} Loss_i(p). \tag{2.8}$$

where N is the number of samples, $Loss_i$ is the loss functional of the i-th sample and p is the vector of the parameters. This property is needed in the backpropagation process, since it facilitates the computation of the gradient of the loss functional with respect to the parameters, weights and biases.

2.3 Backpropagation process

The primary objective when training the FNN model is to minimize the loss functional, in order to obtain optimal parameter values and better performance. This training process can be modeled as minimization problem as following

$$p^* = \min_p \mathcal{L}(p). \tag{2.9}$$

where p is the parameter of the model, weights and biases and p^* represents the optimal parameter. Finding a solution of this minimization problem requires a computation of the loss functional's the gradient with respect to the parameters, weights and biases [HH18].

In 1974, Paul Werbos developed a learning procedure called *Backpropagation*, but it was not until one decade later grabbed attention of other researchers [Kri07]. In 1986, David Rumelhart, Geoffrey Hinton and Ronald Williams published a study [DRW86] that describes several NNs where backpropagation works faster than earlier approaches to training, to leverage NNs to tackle problems that were previously unsolvable. Today, the backpropagation algorithm is the workhorse of training in FNNs [Nie19].

In order to find a solution for (2.9), the backpropagation concept determines the loss functional's sensitivity to each parameter iteratively starting from the last layer backward until the input layer. Firstly, the intermediate quantity known as error is needed, denoted by σ_j^l:

$$\gamma_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l}. \tag{2.10}$$

where z_j^l is the net input of the neuron, the linear combination of the input, weights and bias, the j is the index of the neuron, the l represents the index of the layer and $\mathcal{L}$ is the loss functional.

The process is based on four mathematical equations, which they represent the way of computing the error, the gradients and the connection between them [Nie19]. For the next part, the Hadamard product is used in the equations, also known as the element-wise product. It is a binary operation that takes two matrices of the same dimensions and produces a new matrix, where each element is the product of the corresponding elements in the original matrices. Mathematically, for matrices A and B of size $m \times n$, the Hadamard product $C = A \circ B$ is defined as $C_{ij} = A_{ij} \odot B_{ij}$.

The error of the output layer

The backpropagation starts by computing the error of the last layer, its index is L [HH18].

$$\begin{aligned} \gamma_j^L &= \frac{\partial \mathcal{L}}{\partial z_j^L} && (2.11)\\ &= \frac{\partial \mathcal{L}}{\partial a_j^L} \odot \frac{\partial a_j^L}{\partial z_j^L} && (2.12)\\ &= \frac{\partial \mathcal{L}}{\partial a_j^L} \odot \sigma'(z_j^L). && (2.13) \end{aligned}$$

where $\mathcal{L}$ is the loss functional, the j is the index of the neuron, z_j^L is the net input of the last layer of the j-th neuron and a_j^L is the activation of the last layer of the $j-th$ neuron. This equation linked the gradient of the loss functional with respect to the net input to the activation function, which helps in the adjustment of the parameters [Nie19]. The equation is written in matrix form:

$$\gamma^L = \nabla_a \mathcal{L} \odot \sigma'(z^L). \tag{2.14}$$

The calculation of error in terms of the next layer

The next step is computing the error in the hidden layers using the connection between them.

$$\gamma_j^l = \frac{\partial \mathcal{L}}{\partial z_j^l} \tag{2.15}$$

$$= \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{2.16}$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \gamma_k^{l+1}. \tag{2.17}$$

where l is the index of the layer, with $l \in \{1, 2, ..., L-1\}$, $\mathcal{L}$ is the loss functional and z_j^l is the net input of $j-th$ neuron and $l-th$ layer. To simplify (2.15), the formula of net input incorporating the terms of the output of $l-th$ layer $z_k^{l+1} = \sum_j \sigma(w_{kj}^{l+1}\sigma(z_j^l) + b_k^{l+1})$ is used [Nie19].

$$\gamma_j^l = \sum_k w_{kj}^{l+1} \sigma'(z_j^l) \gamma_k^{l+1}. \tag{2.18}$$

where the w_{kj}^{l+1} is the weight, $\sigma'(z_j^l)$ is the derivative of the activation function with respect to the net input. The equation can be written in matrix form:

$$\gamma^l = \sigma'(z^l) W^{l+1} \odot \gamma^{l+1}. \tag{2.19}$$

where the W^{l+1} is the weight matrix of the l+1 layer.

The gradient of the loss functional with respect to the parameter

Using the previous equations (2.18) and (2.11), now it becomes easy to compute the gradient of the loss functional. Firstly, the gradient of the loss functional with respect to any bias is computed:

$$\frac{\partial \mathcal{L}}{\partial b_j^l} = \frac{\partial \mathcal{L}}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l}$$
$$= \frac{\partial \mathcal{L}}{\partial z_j^l}$$
$$= \gamma_j^l.$$

where $\frac{\partial z_j^l}{\partial b_j^l} = 1$, b_j^l and z_j^l is the bias and the net input of the $j-th$ neuron and $l-th$ layer. In general, the gradient of the loss functional with respect to the bias is written as follows:

$$\nabla_b \mathcal{L} = \gamma. \tag{2.20}$$

The gradient of the loss functional with respect to any weight in the network proceeds as

follows:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w^l_{jk}} &= \frac{\partial \mathcal{L}}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} \\ &= \frac{\partial \mathcal{L}}{\partial z^l_j} a^{l-1}_k \\ &= \gamma^l_j a^{l-1}_k .\end{aligned}$$

where $\frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k \quad \text{if} \quad k = j$, $\mathcal{L}$ is the loss functional, a^{l-1}_k is the activation output, z^l_j is the net input and w^l_{jk} represents the weight. Rewriting this equation in general, then the gradient of loss functional with respect to weight is written as follows:

$$\nabla_W \mathcal{L} = \gamma a. \tag{2.21}$$

The four equations are completed and gathered in one algorithm

Algorithm 1 Backpropagation Algorithm

Require: Input x feature data, σ activation function, $\mathcal{L}$ loss functional and p parameters

Start the feedforward step

for each layer $l = 1$ to N (number of layers in the network) **do**

$z^l = W^l a^{l-1} + b^l$

$a^l = \sigma(z^l)$

end for

Compute the error of the output layer

$\gamma^L_j = \frac{\partial \mathcal{L}}{\partial a^L_j} \odot \sigma'(z^L_j)$

Backward pass:

for each layer $l = L - 1$ to 1 **do**

$\gamma^l_j = \sigma'(z^l_j) W^{l+1} \odot \gamma^{l+1}$

end for

The gradient of loss functional with respect to the parameters

$\frac{\partial \mathcal{L}}{\partial w_j k^l} = \gamma^l_j a^{l-1}_k$

$\frac{\partial L}{\partial b^l_j} = \gamma^l_j$

In summary, backpropagation is a key step in the training process, as it computes the gradients of the loss function with respect to the network's parameters. These gradients are used further in the optimizer's algorithm for determining the direction of updates to reduce the loss and improve the model's performance.

Chapter 3

Optimizers in training the neural network

Training a FNN model is an iterative process in which it runs until it reaches the minimum value of the loss functional. In other words, it adjusts the parameters of the model until the loss functional no longer decreases. To reach this goal, the FNN model requires an optimizer. At each iteration, the forward pass calculates the predictions and computes the loss functional. After, the process starts to run in the backward direction computing the gradient of the loss functional using the backpropagation algorithm 2.3. Then, the optimizer uses the gradients of the loss functional to determine the direction and magnitude of parameter updates. The gradients indicate how much each parameter contributes to the loss and the optimizer adjusts the weights and biases accordingly to reduce the loss. Consequently, the FNN can be formed as an optimization problem [TWBB22]. Thus, the choice of the optimizers is a crucial step, as it has a huge influence on the FNN model performance.

The paper [CSN+20], published in 2020, found out that the adaptive optimizers perform better than simpler optimizer. Also, the inclusion relationships between optimizers influence the model performance. Another empirical work [ZLN+19] shows that increasing the batch size can do gaps between training times for different optimizers. However, sometimes the optimizers used in the training do not guarantee the convergence to a global minimum of the loss functional. For those reasons, this Chapter focuses on explaining the optimizers Stochastic Gradient Descent (SGD), Adaptive Moment Estimation (Adam), Nadam and Broyden-Fletcher-Goldfarb-Shanno (BFGS) in detail, incorporating their algorithm and strengths.

3.1 Stochastic gradient descent

SGD is one of the widely-used optimizers in the machine learning and one of the variants of the traditional gradient descent algorithm [Rud17]. It is well known due to its efficiency and adaptability, particularly in handling large datasets. In this section, the gradient descent algorithm is first introduced, which lays the groundwork for understanding the fundamental

principles of gradient-based optimization. Next, the mechanics of SGD is introduced, exploring how it builds upon and differs from the standard approach.

3.1.1 Gradient descent method

The gradient descent is an iterative optimization algorithm for finding a local minimum of a real valued, differentiable loss functional $\mathcal{L} : \mathbf{R}^n \to \mathbf{R}$. The gradient of the loss functional acts in the direction of the maximum rate of increase of the loss functional. The purpose of this part is to determine that, for a given small displacement in the parameter, the loss functional decreases more steeply in the opposite direction of the gradient than in any other direction [CZ04].

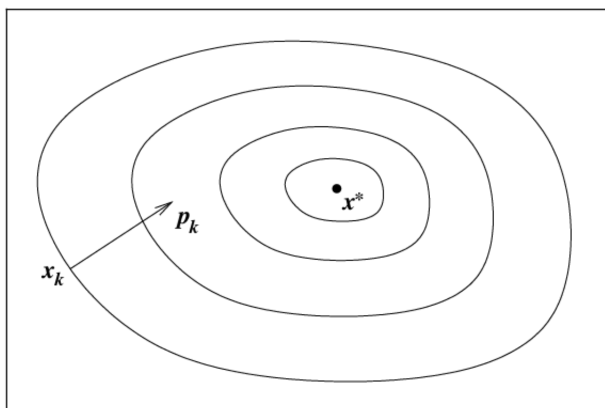


Figure 3.1: Representation of steepest descent direction for a function of two variables [NW06].

The level set of the loss functional consists of points satisfying $\mathcal{L}(p) = c$ for some constant c. If the gradient of $\mathcal{L}$, denoted by $\nabla\mathcal{L}$, is not a zero vector, it is orthogonal to the tangent vector to any smooth curve passing through a point on the level set. According to the Figure 3.1, the gradient direction is perpendicular to the contours of $\mathcal{L}$ [CZ04].

Consider the Taylor's approximation theorem applied to the loss functional $\mathcal{L}$:

$$\mathcal{L}(p + \alpha\Delta p) \approx \mathcal{L}(p) + \alpha(\Delta p)^T \nabla\mathcal{L}(p) + \frac{\alpha^2}{2}(\Delta p)^T \nabla^2\mathcal{L}(p)\Delta p. \tag{3.1}$$

where the $\mathcal{L}$ is the loss functional, p is the parameter vector, α is the step size and the Δp is a small adjustment that helps in updating the parameters in the next iteration. The rate of change of the loss functional $\mathcal{L}$ at p along the direction Δp is given by $(\Delta p)^T \nabla\mathcal{L}(p)$. The aim here is to determine the direction in which the loss functional decreases most rapidly. The change rate can be expressed as follows:

$$(\Delta p)^T \nabla\mathcal{L} = \|(\Delta p)^T\| \|\nabla\mathcal{L}\| \cos(\theta). \tag{3.2}$$

where the θ is the angle between the Δp and $\nabla \mathcal{L}$. Minimizing this expression with respect to the Δp leads to a minimization problem [NW06]:

$$\min(\Delta p)^T \nabla \mathcal{L}. \tag{3.3}$$

Applying the Cauchy-Schwarz inequality, which states that for any vectors f, g, $|f^T g| \leq \|f^T\| \|g\|$, the minimum value that $f^T g$ can reach is $-\|f^T\| \|g\|$ occurs when $f = -g$ [HH18]. In this case, this minimum is reached only when $cos\theta = -1$. Therefore, the solution of the minimization problem 3.3 is

$$\Delta p = -\frac{\nabla \mathcal{L}(p)}{\|\nabla \mathcal{L}(p)\|}. \tag{3.4}$$

Given $p^{(0)}$ be the initial parameter and consider the next parameter is defined by $(p^{(0)} - \alpha \nabla \mathcal{L}(p^{(0)}))$. The value of the loss functional at this updated parameter is approximated as:

$$\mathcal{L}(p^{(0)} - \alpha \nabla \mathcal{L}(p^{(0)})) = \mathcal{L}(p^{(0)}) - \alpha \|\nabla \mathcal{L}(p)\|^2 + o(\alpha). \tag{3.5}$$

where $o(\alpha)$ denotes higher-order terms that becomes negligible as $\alpha \to 0$. If $\nabla \mathcal{L}(p^{(0)}) \neq 0$ and α is sufficiently small:

$$\mathcal{L}(p^{(0)} - \alpha \nabla \mathcal{L}(p^{(0)})) < \mathcal{L}(p^{(0)}). \tag{3.6}$$

indicating the parameter $p^{(0)} - \alpha \nabla \mathcal{L}(p^{(0)})$ is an improvement over the parameter $p^{(0)}$. In general, the iteration leads to

$$p^{(k+1)} \to p^{(k)} - \alpha \nabla \mathcal{L}(p^{(k)}). \tag{3.7}$$

where α is small step size, known as *Learning rate* [HH18].

Algorithm 2 Gradient descent optimization algorithm

Require: $\mathcal{L}$ loss functional; $\alpha > 0$ learning rate

1: Choose an initial point $p^{(0)}$
2: Set iteration counter $k = 0$
3: **while** no convergence **do**
4: Update the parameter vector: $p^{(k+1)} = p^{(k)} - \alpha \nabla \mathcal{L}(p^{(k)})$
5: Increment iteration counter: $k = k + 1$
6: repeat until the convergence
7: **end while**
8: **return** p

However, a notable limitation of gradient descent, particularly for large datasets, is the requirement to compute gradients over the entire dataset for each update, which can be computationally expensive and slow [HH18]. For dataset that does not fit in memory, this approach becomes impractical. Here comes the SGD, a better version of the gradient descent, in play.

3.1.2 SGD algorithm

The gradient descent algorithm faces challenges with dealing with large dataset, as it requires time for repeatedly calculating the gradient for similar examples before updating the model [HH18]. To address this inefficiency, SGD is the best alternative. The idea of this optimizer is based on avoiding this redundant computations by performing one update of one example at a time, which means for each example sequentially, making it faster and allowing it to learn in real-time [Rud17].

The loss functional is often of stochastic type. It is a sum of individual loss functionals evaluated over subsets of the data. In this example, updating the model incrementally with SGD by considering only a single loss functional or a small batch at each step can be more efficient than using the full dataset. This stochastic approach aligns with the nature of the loss functional and can lead to more effective optimization [KB17]. The SGD algorithm follows the procedure outlined below:

Algorithm 3 Stochastic gradient descent optimization algorithm

Require: $\mathcal{L}$ the loss functional; α the learning rate
1: Initialize an initial point $p^{(0)}$
2: **for** $i = 1$ to N **do**
3: Sample an observation i uniformly at random
4: Update the parameter vector $p^{(i+1)} = p^{(i)} - \alpha \nabla \mathcal{L}(p^{(i})$
5: **end for**
6: **return** the optimal parameter p

In the algorithm, the number N represents the cardinality of the available samples. Each update of the parameters requires $\mathcal{O}(d)$ computations, where d is the dimensionality of the data. This computational efficiency makes SGD feasible for large-scale datasets. For strongly convex functions, the results are presented in the following Table 3.1, where ϵ is the tolerance of convergence [RZS20]:

Table 3.1: Computational complexity of SGD optimizer [RZS20].

Method	Number of iterations	Cost per iteration	Total cost
SGD	$\mathcal{O}(log(\frac{1}{\epsilon}))$	$\mathcal{O}(d)$	$\mathcal{O}(\frac{d}{\epsilon})$

However, SGD performs frequent updates with a high variance that leads the loss functional to fluctuate heavily. these fluctuations enable the optimizer to jump around a lot. Thus jumping can help it to explore new and possibly better solutions, but also makes it harder to settle on the exact minimum, as it tends to overshoot [Rud17]. In addition, the learning rate is a critical factor in managing this trade-off, as it influences the magnitude of the updates and consequently the extent of the fluctuations.

3.2 Adaptive momentum estimation

In FNN training, the learning rate α is an important hyperparameters that influences the update of the parameters, weights and biases. It is responsible for the speed of SGD optimizer, until the loss functional reaches its minimizer, as it dictates the step size at each iteration. A large value of learning rate may cause problems in reaching the convergence and let the loss functional fluctuate around the minimum. if it has a low value, this may lead to a very slow convergence and the model gets to stuck in the local minima [NW06].

Many algorithms have been proposed as solution for this challenge, for example, the adaptive approaches. These algorithms learn the overall scale of the problem, avoid slow progress and converge to the minimizer of the loss functional [HH18]. This section focuses on explaining Adam and Nadam in detail.

3.2.1 Adam Algorithm

The Adam optimizer is the most popular adaptive approacher optimizer. It was developed in 2015 by Diederik Kingma and Jimmmy Lei Ba and published as *Adam: a method for stochastic optimization* [KB17]. This part is based on the Kingma and Ba paper.

Adam uses the first and the second order moments of the loss functional gradients, the mean and the variance. It is designed to adapt to the characteristic of the data and update the learning rate for each parameter based on these characteristic. The algorithm stores the previous gradients and the new gradients, which are computed use the previous ones. This helps the optimizer to apply adjustments to the parameters. In areas where the slope of the loss functional changes rapidly, Adam takes a small learning rate value. This helps to avoid the overshooting of the minimum. If the slop changes slowly, then it permits a large learning rate value, enabling faster traversal of the error surface. This adaptability is the key of the Adam efficiency [KB17]. The following are the steps of the Adam algorithm. The index t in the exponential decay rates for the moment estimates represents the power of β_1 and β_2, raised to the number of time steps.

Algorithm 4 Adam algorithm [KB17]

Require: α the learning rate, $\mathcal{L}$ the loss functional and β_1, β_2 the exponential decay rates for the moment estimates
Ensure: $m_0 \leftarrow 0$
$v_0 \leftarrow 0$
$t \leftarrow 0$
$p^{(0)}$ initial parameter vector
while p_t does not converge **do**
$t \leftarrow t + 1$
$m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1)\nabla\mathcal{L}(p_{t-1})$
$v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)(\nabla\mathcal{L}(p_{t-1}))^2$
$\hat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}$
$\hat{v}_t \leftarrow \frac{v_t}{(1-\beta_2^t)}$
$p_t \leftarrow p_{t-1} - \alpha\frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
end while
return the optimal parameter p

Adam algorithm 4 seamlessly combines two optimization methods. The first the alternative version of SGD, known as momentum, which accelerates the optimization process by incorporating the direction and velocity of previous gradients.The second is another adaptive moment method, called RMSprop, which adjusts the learning rate for each parameter in a manner that reflects the recent magnitudes of their gradients.

Using these optimization techniques, Adam looks for better convergence speed and stability for the FNN model. Adam uses two distinct exponential moving averages during the algorithm, one for tracking the gradient, denoted by m_t, and one for the squared gradient, denoted by v_t. These two vectors provide different information about the gradient's behavior, the mean direction of the gradient and the uncentered variance of the gradient. Hyperparameters β_1 and β_2, each ranging between 0 and 1, control the exponential decay rates of these moving averages, thus influencing the memory and responsiveness of the algorithm to new gradient information [KB17].

A moving average is a method to smooth fluctuations in data, thereby facilitating the analysis of patterns by reducing noise. The concept is about a calculation to analyze data points by creating a series of averages of different selection of the full dataset. The primary type of moving average is exponential moving average. This is a weighted average by applying exponentially decreasing weights to older data. In adaptive optimization methods like Adam, the exponential moving average is preferred over other type of moving average, because it reduces lag in simple moving average by applying more weight to recent data points, providing a more accurate reflection of recent gradient behavior. This characteristic is crucial in non-stationary environments, such as training FNN.

Adam uses two statistical concepts the first-order moment (the mean) and the second-order moment (the variance). The first-order moment is represented as the exponentially

weighted moving average of the gradient, capturing the overall direction of parameter updates. The second-order moment, on the other hand, is the exponentially weighted moving average of the squared gradient, which provides information about the variance of the gradient's magnitude. Together, these two moments allow Adam to modulate the learning rate dynamically based on both the average gradient direction and its variability [KB17].

The hyperparameters β_1 and β_2 control the importance given to previous values of the gradient and squared gradient in forming the moving averages. Higher values of β_1 and β_2 imply slower decay rates. In other words, the algorithm stores the past values for many iteration, yielding a smoother but potentially slower average. Conversely, lower values result in quicker decay, allowing Adam to adapt rapidly to changes by prioritizing more recent gradients.

The definition of decay can be understood as a progressive reduction in the weight of past data points over iterations. The exponential decay introduces an adjustment term at each time step in Adam. During an iteration t, the effective first-order moment m_t is computed as:

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\nabla\mathcal{L}(p_{t-1}). \tag{3.8}$$

This formulation weights recent gradients more heavily, with the influence of older data decaying exponentially at each step. When β_1 is close to 1, decay is low, and historical gradients have a prolonged influence. When β_1 is smaller, the decay is faster, leading to an update process dominated by more recent gradients.

Since in the beginning m and v are zeros, they are biased toward zero, which leads to a bias toward zero. Adam corrects this bias using the decay rate, which $\hat{m}_t$ for the first-moment decay rate m, and $\hat{v}_t$ for the second-moment decay rate v. This correction is important as it ensures that the moving averages are more representative, particularly in the early stages of training. The final step is the update of the model parameters [KB17].

Adam's update rule

The final step of Adam's algorithm 4 is the update rule. It takes advantage of both the average direction (first moment) and the variability (second moment) of the gradient, dynamically adjusting the learning rate for each parameter. This characteristic makes Adam robust to noisy gradients and effective in high-dimensional parameter spaces, which are typical in deep learning models.

Assume that $\epsilon = 0$, then the effective step size is written as:

$$\Delta_t = \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}}. \tag{3.9}$$

After replacing the terms $\hat{m}_t$, $\hat{v}_t$ by their formula from the algorithm (5) and (6), the effective

step size becomes

$$\Delta_t = \alpha \frac{m_t}{\sqrt{v_t}} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}. \tag{3.10}$$

In case of sparsity, which means that the gradient of the loss functional has been zero at all time step except the current one t, the equation (3.10) is written as follows

$$\Delta_t = \alpha \frac{(1-\beta_1)\nabla\mathcal{L}}{\sqrt{1-\beta_2}\nabla\mathcal{L}} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \tag{3.11}$$

$$= \alpha \frac{(1-\beta_1)}{\sqrt{1-\beta_2}} \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t}. \tag{3.12}$$

$$\tag{3.13}$$

The effective step has two upper bounds:

$$\Delta_t = \begin{cases} \alpha \frac{1-\beta_1}{\sqrt{1-\beta_2}} & \text{if } (1-\beta_1) > \sqrt{1-\beta_2} \\ \alpha & \text{otherwise} \end{cases} \tag{3.14}$$

The first upper bound occurs only in cases of highly sparse or less sparse. In these cases, the effective time steps take smaller values that leads to $\beta_1^t \approx 0$ and $\beta_2^t \approx 0$. If $(1-\beta_1) = \sqrt{1-\beta_2}$, then $|\frac{1-\beta_1}{\sqrt{1-\beta_2}}| < 1$ and therefore $|\Delta_t| < \alpha$.

In case $|\frac{\mathbb{E}[g]}{\sqrt{\mathbb{E}[g^2]}}| \leqslant 1$, then $\frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \approx \pm 1$ and the effective step is bounded by α, $|\Delta_t| \lessapprox \alpha$. It is easy to choose a suitable learning rate value, since it controls the maximum size of the steps taken when adjusting the parameters. A good range of alpha can be figured out, so the algorithm can reach the best solution from the starting point within a certain number of steps.

Let $\frac{\hat{m}_t}{\sqrt{\hat{v}_t}}$ be called SNR, *Signal to noise ratio.* If the SNR value is small, it gives the information if the direction of $\hat{m}_t$ corresponds the direction of the true gradient, then the effective step size is closer to zero. In the case of getting closer to the optimal parameter, SNR becomes closer to zero and this behavior is similar to annealing.

Another property of the effective step is that the effective step does not change if the gradients are scaled, which means when the gradient is multiplied by a constant, Δ_t remains the same [KKL+20].

Adam convergence

The Adam's paper [KB17] proposed evaluating the optimizer using its regret. In other words, it deals with a series of unknown loss functionals over time and aims to predict the best parameter values. Regret, in this case, is the total difference between the loss when using the predicted values and the optimal possible fixed parameter. Adam achieves a low regret bound of $O(\sqrt{S})$, with S is the total number of steps, meaning that it performs comparably to the best available methods for this type of learning problem. For sparse types of data, Adam performs even better than non-adaptive methods. A key result, the study shows

that Adam's effectiveness is partly due to how its settings change over time: specifically, the learning rate gradually decreases, and the first-moment averaging factor decreases towards zero. This latter adjustment has been found to help the model converge better, especially at the end of training. Finally, it shows that as the total number of steps increases, Adam's average regret approaches zero, meaning its predictions get closer and closer to optimal one.

3.2.2 Nadam optimizer

Nadam, an abbreviation of *Nesterov-accelerated Adaptive Moment estimation*, is a new optimization technique with limited literature exploring its potential applications and advantages. It was mentioned first in academic literature appeared in a 2016 by Dozat [Doz16], which provided a detailed explanation and its application in image recognition.

This optimizer is a combination of the Adam optimizer, discussed in the previous section 3.2 and Nesterov acceleration gradient (NAG). Adam is an optimization method that computes adaptive learning rates for each parameter, by combining the adaptive learning rate scaling of RMSProp with classical momentum. With maintaining moving averages of the gradients and their squared values, it achieves efficient convergence, especially in non-convex optimization problems. However, traditional Adam uses classical momentum, which NAG has been shown to surpass in many scenarios. The process of Nadam optimizer is presented in the following algorithm 5

Algorithm 5 Nadam algorithm [Doz16]

Require: α the learning rate, $\mathcal{L}$ the loss functional
Require: β_1, β_2 the exponential decay rates for the moment estimates
Ensure: $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  $t \leftarrow 0$
  $p^{(0)}$ Initial parameter vector
  while p_t does not converge **do**
    $t \leftarrow t + 1$
    $g_t = \nabla_{p_{t-1}} \mathcal{L}(p_{t-1})$
    $\hat{g}_t = \frac{g_t}{1-\beta_1}$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2)(g_t)^2$
    $\hat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}$
    $\hat{v}_t \leftarrow \frac{v_t}{(1-\beta_2^t)}$
    $\bar{m}_t \leftarrow (1 - \beta_1)\hat{g}_t + \beta_1 \hat{m}_t$
    $p_t \leftarrow p_{t-1} - \alpha \frac{\bar{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$
  end while
  return the optimal parameter p

On the other hand, NAG is a method which helps in increasing the step size in the in-

crease direction of the loss functional, which leads to a faster convergence. NAG has been shown to achieve better convergence rates for convex, non stochastic objectives compared to classical gradient descent [Doz16]. It improves the traditional calculation of the gradient within momentum methods after applying the momentum step. This allows for a more informed update direction, often resulting in faster convergence and higher-quality solutions.

The algorithm begins in the same way as Adam by computing the gradient of the loss functional. After the moving average is updated and before applying the update rule, the Noestrov trick is applied. The momentum term (9) is pushed to the current gradient, which called Noestrov trick. Instead of relying solely on classical momentum, Nadam predicts the next updating parameters based on both the current gradient and the anticipated momentum. As with Adam, Nadam includes initialization bias corrections for both momentum and gradient estimates to ensure robust performance during the early stages of optimization.

By integrating Nesterov's acceleration into the Adam algorithm, Nadam optimizer has got the adaptive learning rate benefits of Adam and at the same time the superior convergence properties of NAG.

3.3 Broyden–Fletcher–Goldfarb–Shanno

The second-order optimization techniques have gained popularity due to their ability to converge rapidly. Unlike first-order methods, second-order optimization techniques use the second derivative, which can provide significant advantages, such as the elimination of the learning rate. Also, this method uses the information about the curvature of the loss functional.

The second-order optimization uses the Hessian matrix, a square matrix of second-order partial derivatives that describes the local curvature of a function. The Hessian matrix allows second-order methods to navigate the optimization landscape more effectively. The Newton method is a well-known example of a second-order optimization approach, which directly uses the Hessian to update the parameter estimates. However, computing the exact Hessian can be computationally expensive, especially in high-dimensional problems.

In order to solve the limitations of computing the full Hessian matrix, Quasi-Newton methods have been developed to approximate the Hessian matrix instead of calculating it explicitly. This method reduces the computational cost while retaining the advantages of second-order optimization. The BFGS, named for its discoverers Broyden, Fletcher, Goldfarb, and Shanno, method is one of the most popular Quasi-Newton algorithms. This iterative method approximates the Hessian matrix, denoted as B_k, and uses it to find the optimal solution to an optimization problem. The key distinction between BFGS and the Newton method lies in this approximation. BFGS updates an estimate of the Hessian iteratively rather than recalculating it from scratch. The following Section focus on exploring the BFGS optimizer in detail based on the book [NW06] written by Jorge Nocedal and Stephan H.Wright.

3.3.1 Approximation of the Hessian matrix

In each iteration, the BFGS optimizer approximates the loss functional $\mathcal{L}$ using a quadratic model m_k:

$$m_k(h) = \mathcal{L} + \nabla\mathcal{L}h + \frac{1}{2}h^T B_k h. \tag{3.15}$$

where B_k is $n \times n$ symmetric positive definite matrix, h is the direction. To find the search direction, the quadratic model has to be minimized, by setting its gradient with respect to h to zero:

$$\nabla m_k(h) = \nabla\mathcal{L} + B_k h = 0. \tag{3.16}$$

This equation reaches its minimizer at p_k:

$$h_k = -B_k^{-1}\nabla\mathcal{L}. \tag{3.17}$$

Then, the new iterate is

$$p_{k+1} = p_k + \alpha_k h_k. \tag{3.18}$$

where the p is the parameter and the step α_k should satisfy the Wolfe conditions.

The Wolfe condition is actually composed of two important inequalities. The first condition states that α_k should decrease the loss functional $\mathcal{L}$ such that

$$\mathcal{L}(p_k + \alpha h_k) \leq \mathcal{L}(p_k) + c_1 \alpha \nabla\mathcal{L}_k^T h_k. \tag{3.19}$$

for some constant $c_1 \in (0, 1)$. This inequality is known as *Armijo condition* or sufficient decrease condition. In other words, the reduction of the loss functional should be proportional to the step length and the directional derivative. The second inequality rules out the unacceptable short steps, known as *curvature condition*. It states:

$$\nabla\mathcal{L}(p_k + \alpha_k h_k)h_k \geq c_2 \nabla\mathcal{L}^T h_k. \tag{3.20}$$

for some constant $c_2 \in (c_1, 1)$. This inequality describes that the slope of left-hand side is grater than c_2 times its initial slope.

Davidon proposed a simple idea that avoids the computation of B_k from scratch at every iteration [NW06]. In order to construct a new iterate p_{k+1}, the new quadratic model needs to satisfy that its gradient should match the gradient of the loss functional at the latest two iterates p_k and p_{k+1}, this can be written as:

$$m_{k+1}(h) = \mathcal{L}_{k+1} + \nabla\mathcal{L}_{k+1}^T h + \frac{1}{2}h^T B_{k+1} h \tag{3.21}$$

Its gradient at $h = -\alpha_k h_k$ becomes :

$$\nabla m_{k+1}(-\alpha_k h_k) = \nabla\mathcal{L}_{k+1} - \alpha_k B_{k+1} h_k. \tag{3.22}$$

Using the following Taylor's approximation

$$\nabla\mathcal{L}_{k+1} = \nabla\mathcal{L}_k + \alpha_k B_{k+1} h_k. \tag{3.23}$$

Therefore, simplifying the previous equation

$$\nabla\mathcal{L}_{k+1} - \alpha_{k+1} B_{k+1} h_k = \nabla\mathcal{L}_k. \tag{3.24}$$

By rearranging this equation, the secant equation is constructed

$$B_{k+1} s_k = y_k. \tag{3.25}$$

where $s_k = p_{k+1} - p_k = \alpha_k h_k$ and $y_k = \nabla\mathcal{L}_{k+1} - \nabla\mathcal{L}_k$. In this equation, the update matrix should be positive definite. This condition is important for the approximation of the curvature of the loss functional. This is guaranteed if the vector s_k and y_k satisfy *the curvature condition*

$$s_k^T y_k > 0. \tag{3.26}$$

This condition indicates that the functional has sufficient curvature along the direction s_k to ensure a positive definite B_{k+1}, which is essential for stability and convergence in optimization.

In cases where the function is strongly convex, the curvature condition is satisfied for any steps. However, with the case of non-convex functions, there's no guarantee that the condition is satisfied, then it leads to an indefinite or negative definite Hessian matrix, destabilizing the optimization. To prevent this, the BFGS method imposes constraints on the step length, often by enforcing the Wolfe condition. These conditions ensure that the step length is chosen to satisfy both a sufficient decrease in the function and a controlled gradient slope, thereby guaranteeing the curvature condition.

The curvature condition provides the information that there exists a solution that maps s_k to y_k, which means that the secant equation has an infinite number of solutions. In n dimensional space, a symmetric positive definite matrix B_{k+1} has $\frac{n(n+2)}{2}$ degrees of freedom. On the other side, the secant equation imposes n conditions on the matrix B_{k+1}, but the degrees of freedom exceed the n conditions imposed by the secant equation. To ensure that the approximate Hessian matrix is positive definite, additional constraints are imposed all determinants of the submatrices must be positive. In n dimensional space, there are n such conditions, one for each principal minor. These conditions reduce the degrees of freedom somewhat but still do not use up all of them [NW06].

3.3.2 BFGS

The next step now is to determine the updated matrix B_{k+1} uniquely, which satisfies the secant equation and the curvature condition. The process is to select the matrix that is close to the current B_k. In this way, one ensures that the update is smooth and does not deviate away from the previous ones. To find the B_{k+1} , consider the following minimization problem

$$\min_B \|B - B_k\| \quad \text{with } B = B^T \text{and } B_{k+1} s_k = y_k. \tag{3.27}$$

The norm used in this part is the weighted Frobenius norm which is written as following

$$\|A\| = \left\| W^{\frac{1}{2}} A W \right\|_F . \tag{3.28}$$

where A is the matrix and W is the weighted matrix with the norm of Frobenius is written as $\|C\| = \sum_i \sum_j c_{ij}^2$. The choice of the weight matrix should ensure that it is aligned with the curvature information from the secant equation.

For example, the specific choice of the weight matrix can be $W = \hat{G}_k^{-1}$, where $\hat{G}_k$ is the average Hessian defined by

$$\hat{G}_k = \int_0^1 \nabla^2 \mathcal{L}(p_k + \tau \alpha_k h_k) d\tau. \tag{3.29}$$

Using Taylor's theorem, another property can be extracted from it, $y_k = \hat{G}_k \alpha_k h_k = \hat{G}_k s_k$. This choice helps the Frobenius norm to be non-dimensional, since the solution of the minimization problem should depend on the units of the problem.

Solving the minimization problem from scratch is complex. For practical purposes, it is sufficient to know that the approximate Hessian can be updated at each iteration by adding two symmetric, rank-one matrices U and V. These matrices are defined as $U = auu^T$ and $V = bvv^T$ with $B_{k+1} = B_k + U_k + V_k$, where u and v are linearly independent vectors, ensuring the symmetry of the Hessian with each update. Then the secant equation (3.25) can be written as

$$B_{k+1} s_k = B_k s_k + auu^T s_k + bvv^T s_k = y_k. \tag{3.30}$$

replacing by $u = y_k$ and $v = B_k s_k$ and rearranging gives

$$B_k s_k (1 + b s_k^T B_k^T s_k) = y_k (1 - a y_k^T s_k). \tag{3.31}$$

To get the wanted result, the two constant in the equation can be replaced by $b = -\frac{1}{1 + b s_k^T B_k^T s_k}$ and $a = \frac{1}{y_k^T s_k}$. At the end, the BFGS formula, the solution of the minimization problem, is defined by the following:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}. \tag{3.32}$$

It is easy to check that this formula satisfies the secant equation (3.25), where the fact that B_k is symmetric has been exploited. To ensure that B_{k+1} is a positive definite matrix, we start by assuming that B_k is positive definite and that f is strictly convex. This strict convexity guarantees the monotonic gradient property of convex functions, which implies that $y_k^T s_k > 0$. This condition is essential because, if it fails, the update rule could result in an indefinite matrix. For twice differentiable functions with a strictly convex structure, the Hessian remains invertible, and we can proceed with confidence that the updated matrix B_{k+1} will retain positive definiteness [NW06].

The construction of B_{k+1} relies on adding terms that are positive semi-definite, as shown by breaking down the expression into a positive semi-definite form based on Cauchy-Schwarz

inequality. This assures that each update step builds upon a matrix that does not introduce any negative eigenvalues. Moreover, by examining the nullspace of the update term, we can see that only one eigenvalue could potentially be zero, but this is prevented by the condition $y_k^T s_k > 0$, confirming that B_{k+1} is strictly positive definite.
In order to calculate the inverse of the approximate Hessian matrix, the Sherman Morrison–Woodbury formula is used :

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}. \tag{3.33}$$

Now, rewrite the BFGS update in suitable form and plug it into the Woodbury formula, the unique solution H_{k+1} is given by

$$H_{k+1} = (I - \rho_k s_k y_k^T)H_k(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T. \tag{3.34}$$

where $\rho_k = \frac{1}{y_k^T s_k}$. There is not a single best way to choose the initial approximation H_0 in the BFGS algorithm. It's often set to the identity matrix, a scaled version of it (to match the scale of the variables), or, if possible, the inverse of an approximate Hessian calculated at p_0.

Algorithm 6 BFGS algorithm [NW06]

Require: p_0 the initial parameter, $\nabla\mathcal{L}(p_0)$ the loss functional and the initial inverse Hessian matrix H_0

$k \leftarrow 0$
while not converged **do**
  $h_k = -H_k \nabla\mathcal{L}(p_k)$
  select α_k using a line search
  $p_{k+1} = p_k + \alpha_k h_k$
  $s_k = p_{k+1} - p_k$
  $y_k = \nabla\mathcal{L}(p_{k+1}) - \nabla\mathcal{L}(p_k)$
  $H_{k+1} = (I - \rho_k s_k y_k^T)H_k(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$
  $k \leftarrow k + 1$
end while

Each iteration of the BFGS algorithm 6 requires only $\mathcal{O}(n^2)$ arithmetic operations, aside from function and gradient evaluations, and avoids more costly $\mathcal{O}(n^3)$ operations like solving linear systems or performing matrix-matrix operations. The algorithm is robust, with a superlinear convergence rate that is generally fast enough for practical use [NW06].

Chapter 4

PINNs for convection-diffusion-reaction problem

Convection-diffusion-reaction problems have always been an essential part of computational fluid dynamics, as they describe the distribution of a scalar quantity like temperature, energy or concentration within a moving medium [DFMb]. Several studies have employed PINN for solving the convection-diffusion-reaction problems, focusing on many aspects. For instance, the papers [DFMb], [DFMa] and [VJ24] focus on the choice of the loss functional, collocation points distribution and bound-preserving techniques. Based on these studies, the equidistant collocation points, hard constrained PINN approach, vanilla PINN and standard loss functional are implemented in this master thesis, to solve the convection-diffusion- reaction problems. The four optimizers, SGD, Adam, Nadam and BFGS are used to train hard-constrained PINN using standard loss functional on equidistant collocation points.

This chapter starts by providing a detailed introduction to convection-diffusion-reaction problems, by mentioning the strong form of their equations. Then, it introduces PINNs by explaining the standard loss function and the concept of collocation points. The chapter ends with introducing a variation of hard-constrained PINNs, which avoids training of the boundary conditions to save time.

4.1 Strong form of the convection-diffusion-reaction problem

The convection-diffusion-reaction equation refer to the advection-diffusion equation. It is a cornerstone of mathematical modeling in the physical sciences and provides a framework for describing processes, such as heat transfer, the behavior of fluids and gases, and pollutant dispersion in natural systems [Buc23].

Since the mid-20th century, numerical methods for solving this equation have evolved significantly, with finite difference techniques that ensure that the solutions are both stable and accurate. Recent research now explore innovative approaches instead of classical ones, including machine learning techniques, for example PINNs. PINNs provide a flexible and

efficient way to solve convection-diffusion-reaction problems by involving the deep learning to approximate solutions while respecting the underlying physical laws. This thesis focuses on applying PINNs to convection-diffusion-reaction problems with concentrating on their potential to enhance the accuracy and efficiency of numerical computations.

In this context, the word convection refers to the transport of a quantity due to the fluid motion of the medium itself. If the flow is absent, the diffusion phenomenon appears and causes the spread of the quantities. Diffusion facilitates the movement of a substance from regions of higher concentration to regions of lower concentration until a uniform concentration is achieved across the domain. The third word is reaction, which indicates the potential of the scalar quantity to undergo chemical reactions, forming new compounds [Mat24]. This subsection introduces the strong form of the convection-diffusion-reaction equation, beginning with the governing PDE and relevant boundary conditions, as outlined in Professor Volker John's lecture at WIAS [Joh].

Consider a fixed domain $\Omega \in \mathbf{R}^3$. The domain Ω is assumed to be a Lipschitz domain, meaning that every point on the boundary has a neighborhood that can be represented as the graph of a Lipschitz continuous function. The points within the domain are denoted by x with $x = (x_1, x_2, x_3)$ and the time $t \in \mathbf{R}_+$ progress as the system evolves.

Our aim is to derive the energy conservation model over a subset $V \subseteq \Omega$ for a time interval $(t, t+\Delta t)$, where V is the volume occupied by the fluid. According to the principle of energy conservation, the total quantity of energy in an isolated system remains constant, though it may change form over time [emp]. This conservation is described by three main components:

Energy source or sink contributions:

The energy contribution from internal heat sources or sinks in the domain over an interval time $(t, t + \Delta t)$ is given by

$$Q_1 = \int_t^{t+\Delta t} \int_V F(t, x) \; dx \; dt. \tag{4.1}$$

where F represents the intensity of the internal heat sources or sinks distributed throughout the volume. In this case F is the rate of the generated or absorbed heat in the volume and its unit is Joule, denoted by [J].

Energy transfer by molecular and convective motion:

Energy may also enter or leave the volume V through its boundary ∂V both due to molecular motion and convective transport. The Fourier's law states that the rate of heat transfer is proportional to the area perpendicular to the direction of heat flow and to the temperature gradient, with heat moving from warmer to cooler regions. Then using Fourier's law, to describe molecular heat transfer and accounting for the convective motion of fluid with velocity v, the heat flux across the boundary can be written as:

$$Q_2 = \int_t^{t+\Delta t} \int_{\partial V} (k\frac{\partial u}{\partial \vec{n}} - c\rho(v\cdot\vec{n})u)(t,s)\, dS\ dt. \tag{4.2}$$

where k is the thermal conductivity, c is the specific heat capacity, ρ is the fluid density, v is the velocity, u is the temperature and $\vec{n}$ is the outward unit normal vector on the boundary. The unit of this equation is also joule [J]. This equation can be further simplified using the divergence theorem, also known as Gauss theorem, which is simplified in the following equation:

$$\int_\Omega \nabla\cdot u\, dx = \int_\Gamma u\cdot\vec{n}\, ds. \tag{4.3}$$

where Ω is Lipschitz domain, Γ is its boundary, u defined on Ω and $\vec{n}$ is the outward unit normal vector to Ω. This theorem leads to a simpler formulation of the previous integral (4.2):

$$Q_2 = \int_t^{t+\Delta t} \int_V \nabla\cdot(k\nabla u) - c\rho(v\cdot\nabla)u)(t,x)\, dx\, dt. \tag{4.4}$$

The unit of this equation is also joule [J].

Rate of change of energy in the volume:

The third component addresses the rate of temperature change within the volume V. Assuming u the temperature, is sufficiently smooth, we apply a Taylor expansion in time interval $(t, t+\Delta t)$ with neglecting the higher-order terms to obtain:

$$u(t+\Delta t, x) - u(t,x) = \frac{\partial u}{\partial t}\Delta t. \tag{4.5}$$

Thus, the corresponding energy variation in V due to the temperature change is written as:

$$Q_3 = \int_t^{t+\Delta t} \int_V c\rho\frac{\partial u}{\partial t}(t,x)\, dx\, dt. \tag{4.6}$$

where t is the time. The unit of this equation is also joule [J].
By the principle of energy conservation, the rate of change of energy within the volume V is the sum of the energy that is either entering and leaving the volume V, with the energy that comes from internal sources. In mathematical expression, the conservation law Q_3 is the sum of (4.1) and (4.4). Then the final form of the integral is represented by:

$$\int_t^{t+\Delta t} \int_V (c\rho\frac{\partial u}{\partial t} - \nabla\cdot(k\nabla u) - c\rho uv - F)(t,x)\, dx\, dt = 0. \tag{4.7}$$

Since the volume V and time interval Δt are arbitrary, the integral vanishes. Then the internal of the integral is equal to zero, by dividing through $c\rho$, we obtain the standard form of the convection-diffusion equation:

$$\frac{\partial u}{\partial t} - \frac{k}{c\rho}\Delta u - (v\cdot\nabla)u = \frac{F}{c\rho}, \tag{4.8}$$

Considering the steady-state conditions in this case, the temperature does not vary with time, then $\frac{\partial u}{\partial t} = 0$. The standard form becomes

$$-\mu \Delta u + v \cdot \nabla u = f' \quad in \quad \Omega. \tag{4.9}$$

where $\mu = \frac{k}{c\rho}$ is the thermal diffusivity and $f' = \frac{F}{c\rho}$ represents the normalized heat source.

The next step is focusing on simplifying and generalizing the problem, to be perfectly used for mathematical purposes. Then the non-dimensional variables are the perfect tool. Let L represent the characteristic length scale of the domain and U the characteristic temperature scale. Changing the $x = \frac{x'}{L}$ and $u = \frac{u'}{U}$ in the original equation and after some simplification, the convection diffusion equation becomes

$$-\epsilon \Delta u + \beta \cdot \nabla u = f \quad in \quad \Omega. \tag{4.10}$$

where $\epsilon = \frac{\mu}{LV}$ is dimensionless diffusion coefficient, $\beta = \frac{v'}{V}$ is the dimensionless velocity field and $f = \frac{Lf'}{UV}$ represents the dimensionless heat source.

The problem is not finished yet, there is still prior information that must be known, that it evolves the well possessness of the problem. So, the governing equation must be supplemented with appropriate boundary conditions on the boundary $\partial\Omega$ of Ω. The boundary of the domain is decomposed, depending on the problem, into parts, where the solution receives a specific, exact value. Those parts typically are:

- Dirichlet boundary condition specifies the temperature's value on the Dirichlet boundary part: $u = g_1$ on $\partial\Omega_D$ where g_1 is the temperature distribution.
- Neumann boundary condition describes the heat flux through the Neumann boundary part: $k\frac{\partial u}{\partial n} = g_2$ on $\partial\Omega_N$ where g_2 is the heat flux and n is the outward unit normal vector on the boundary.

where both $\partial\Omega_D \cup \partial\Omega_N = \Omega$ and $\partial\Omega_D \cap \partial\Omega_N = \emptyset$

Only the the fluid flow and the spread of particles due to molecular motion are considered in the convection diffusion equation. But in case the units of the fluid can react chemically, then their equations become interconnected. The law of mass action models the reaction in the equations by adding a coupling term to it. This term depends on the concentrations of the substances, but not on their derivatives, and includes a coefficient, which depends on the concentrations of the other substances. This coupling term, r , is called the reaction term.

Now, the convection-diffusion-reaction problem is ready. In case of steady-state concentration field, the convection-diffusion-reaction problem is written as following:

$$-\epsilon \Delta u + \beta \cdot \nabla u + ru = f \quad in \quad \Omega, \tag{4.11}$$

$$u = g_D \quad on \quad \Omega_D, \tag{4.12}$$

$$\epsilon \nabla u \cdot \vec{n} = g_N \quad on \quad \Omega_N \tag{4.13}$$

4.2 Physics-informed neural networks

Over decades, numerical methods have been developed for solving numerically PDEs, such as the incompressible and compressible Navier Stokes equations, using finite elements or meshless methods. Unfortunately, these methods have a significant drawback. They do not involve seamlessly data into existing algorithms [CMW+21]. Also, since those methods are mesh-based, they are time-consuming and introduce computational complexity [LMMK21].

On the other hand, the FNN could be a better alternative, since it is mesh-free and considered as universal function approximators, capable of solving nonlinear problems without the need for previous assumptions [LMMK21]. But, unfortunately, it is time-consuming and not less accurate than traditional methods. In other words, it is computationally expensive and does not necessarily offer a better accuracy than the traditional networks.

The alternative to approximate the solution to the boundary value problem is an advanced version of FNN first introduced by Raissi et al. [RPK17], known as physics-informed neural network (PINN). The concept of PINN is to seamlessly integrate data with physical laws, enabling effective modeling even in cases of high-dimensional spaces. It makes it easier to find the solution, and to reduce the amount of data required to train a model or solve the problem [RPK17]. It combines the information from both the measurements and PDE by embedding it into the loss function [KKL+20].

Consider first the following parametrized PDE system:

$$
\begin{aligned}
f(x, t, u, \nabla u, \Delta u, .., \lambda) = 0 \quad x \in \Omega \quad \textit{and} \quad t \in (0, T], \\
u(x, 0) = g_0(x) \quad x \in \Omega, \\
u(x, t) = g_\Gamma(x) \quad x \in \partial\Omega \quad \textit{and} \quad t \in (0, T]
\end{aligned}
$$

$x \in \mathbb{R}^d$ is the spatial coordinate, t is the time, λ is the PDE parameter and f denotes the residual of the PDE containing the differential operator by taking x and t as the inputs. Also $u(x, t)$ is considered as the solution of the PDE with the initial condition $g_0(x)$ and the boundary condition $g_b(x)$. The boundary condition can be Dirichlet or Neumann. The goal is to solve this boundary value problem using a PINN, which approximates the solution by minimizing the residual across the spatial-temporal domain.

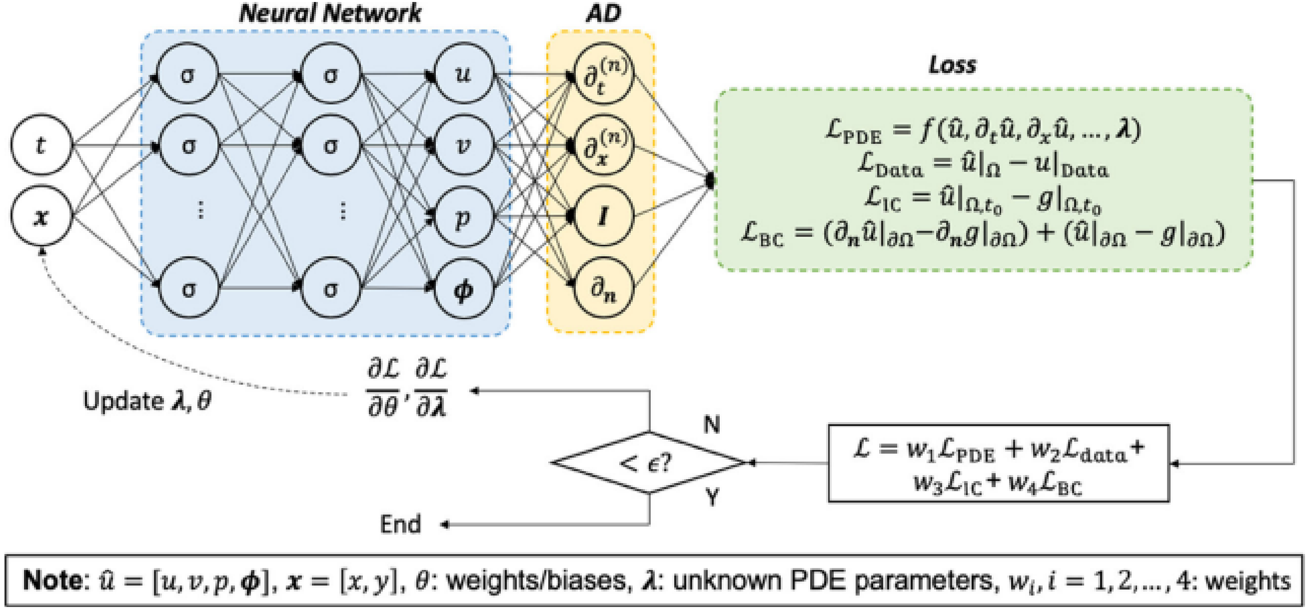


Figure 4.1: Schematic of a physics-informed neural network (PINN) [CMW+21]

The PINN contains three main phases as has been depicted in the Figure 4.1, the FNN, automatic differentiation (AD) and the calculation of the loss functional. The FNN accepts (x, t) as inputs and computes the output in the same way described in the Chapter 2.2. The model contains $L \in \mathbf{N}$ layers which have $n_i \in \mathbf{N}$ nodes, with $i = 1, ..., L$, where $n_1 = \dim(\Omega)$ and $n_L = \dim(\mathrm{Im(u)})$. The network can be transformed mathematically into the following equations:

$$\begin{aligned} z^0 &= (x, t) \\ z^k &= \sigma(W^k z^{k-1} + b^k) \quad for \quad 1 \leqslant k \leqslant L \\ z^L &= W^L z^{L-1} + b^L. \end{aligned}$$

where z^k is the hidden variable of the $k-th$ hidden layer, σ is the activation function and $W^k$and b^k are the weight matrix and the bias vector of the $k-th$ layer and the last term z^L is the last output of the model, which will be denoted for the rest of the thesis by u_N [CMW+21].

For the choice of the activation functional in the case of PINNs, the RELU activation function will not be a good option because it is not smooth and not consistent for PINNs, which means it fails to converge to the exact solution even within the limit of an infinite training dataset. But smooth functions as Tanh and Sigmoid have shown to be a good choices to approximate the exact solution [Mar21].

The next phase of PINN is the AD. AD works on calculating the derivatives of the outputs with respect to the inputs directly in computational graph in order to construct the loss [CMW+21]. This is similar to the backpropagation process, which is explained in details in

Section 2.3. But the difference is that AD is considered as a collection of techniques that are more general than backpropagation [BPRS 4]. The last step in PINNs is solving the PDE system by converting it into an optimization problem by iteratively updating the parameters p, with the goal is to minimize the loss functional.

The last step can be written as the following minimization problem

$$p \in \arg\min_{p} \mathcal{L}(u_N). \tag{4.14}$$

where u_N is the network output and $\mathcal{L}$ is the loss functional of the PINN model. If the exact solution u was known, then the standard loss functional is the difference between the real output and the model output. But PINN is actually approximating this solution, which is not known [FM23]. Raissi et al. presented the loss functional as the sum of the strong form of the residual of the governing equations, the initial and the boundary conditions of the problem [DFMa].

The standard loss functional $\mathcal{L}^{st}$ is defined as:

$$\mathcal{L}^{st} = w_I \mathcal{L}_I^{st} + w_B \mathcal{L}_B^{st} + w_{IC} \mathcal{L}_{IC}^{st} \tag{4.15}$$

where the w_I, w_B and w_{IC} are weights and $\mathcal{L}_I^{st}$, $\mathcal{L}_B^{st}$ and $\mathcal{L}_{IC}^{st}$ are the residual loss functional, the boundary condition loss functional and the initial condition loss functional . The weights' role is to balance the interplay between the loss functionals. They can be chosen by an user or tuned automatically. Some terms can appear and some other terms can disappear from the equation. It depends on the problem [CMW+21].

The loss functionals mentioned above are written in the following form:

$$\mathcal{L}_I^{st} = \frac{1}{N_I} \sum_{i=1}^{N_I} |f(x_I^i, t_I^i, u_N(x_I^i, t_I^i), \nabla u_N(x_I^i, t_I^i), ..., \lambda)|^2, \tag{4.16}$$

$$\mathcal{L}_B^{st} = \frac{1}{N_B} \sum_{i=1}^{N_B} |u_N(x_B^i, t_B^i) - g_\Gamma|^2, \tag{4.17}$$

$$\mathcal{L}_{IC}^{st} = \frac{1}{N_{IC}} \sum_{i=1}^{N_{IC}} |u_N(x_{IC}^i, t_{IC}^i) - g_0|^2, \tag{4.18}$$

$$\tag{4.19}$$

where (x_I^i, t_I^i), (x_B^i, t_B^i), (x_{IC}^i, t_{IC}^i) are the internal, boundary and initial collocation points and N_I, N_B, N_{IC} are the number of the collocation points.

4.2.1 Collocation points

Collocation points play a crucial role in influencing the quality of PINN approximations for boundary value problems. The primary objective of PINNs is to minimize the loss functional,

which is evaluated at collocation points strategically distributed within the problem domain and on the space time boundary [DFMa]. In reference [DFMb], three collocation point distributions were compared in the context of convection-dominated convection-diffusion-reaction problems, where solutions often exhibit layers localized regions with steep solution gradients.

The distribution of collocation points in these cases has a significant impact on the network's ability to capture such complex features. The three strategies assessed include layer adapted collocation points inspired by Shishkin meshes, equispaced points, and uniformly random point distributions, each evaluated for their effectiveness in representing convection-dominated solutions. For the remainder of this work, equidistant collocation points will be employed to ensure simplicity in implementation while maintaining a baseline accuracy level appropriate for our purposes.

4.2.2 Standard loss functional

In the context of the convection-diffusion-reaction problem, the PINN model aims to approximate the solution while it minimizes the loss functional. The loss functional, used in approximating the convection-diffusion-reaction problem (4.11), is the standard PINN loss functional (4.15).

The strong residual of the convection-diffusion-reaction equation is written as follows:

$$Res(u) = -\epsilon \Delta u + \beta \cdot \nabla u + ru - f. \tag{4.20}$$

The standard loss functional is written as the sum of squared residuals over the whole domain Ω and its boundary, defined by:

$$\mathcal{L} = \int_{\Omega} |Res(u_N)|^2 d\Omega + \int_{\Omega_D} |u_N - g_D|^2 d\Omega_D + \int_{\Omega_N} |\epsilon \nabla u \cdot n - g_N|^2 d\Omega_N + \mathcal{L}_p. \tag{4.21}$$

where the $\mathcal{L}_p$ is the loss of the penalty [FM23], u_N is the output of the PINN model, g_D is the Dirichlet boundary condition and g_N is the Neumann boundary condition.

To compute the loss functional, integral involve high-order derivatives of the neural network output need to be evaluated. The integrals in this case cause problem, since their computation using traditional quadrature methods is very complicated, due to their high-dimensional. A solution for this challenge is the Monte Carlo (MC) method. This method plays a crucial role in the loss functional of PINNs for several reasons. Its primary advantage lies in its simplicity and flexibility. It does not require knowledge of the functional form of the integrand in high dimensions or adaptions for complex geometries. Instead, MC methods leverage random sampling to estimate the integral, making it a better choice for a variety of problems, including those faced in PINNs [FM23].

The strong residual form transformed in the following form:

$$\int_{\Omega} Res(u_N) d\Omega \approx \frac{1}{N_I} \sum_{i=1}^{N_I} |Res(u_N)|^2. \tag{4.22}$$

For the Dirichlet boundary condition, The Dirichlet residual is

$$Res_D(u_N) = u_N - g_D. \tag{4.23}$$

The approximation of the Monte Carlo gives the following equation :

$$\int_{\Omega_D} |Res(u_N)|^2 d\Omega_D \approx \frac{1}{N_D} \sum_{i=1}^{N_D} |Res(u_N)|^2. \tag{4.24}$$

For the Neumann Dirichlet boundary condition, the Neumann residual is

$$Res_N(u_N) = \epsilon \nabla u_N \cdot \vec{n} - g_N. \tag{4.25}$$

MC gives the following approximation :

$$\int_{\Omega_N} |Res(u_N)|^2 d\Omega_N \approx \frac{1}{N_N} \sum_{i=1}^{N_N} |Res(u_N)|^2. \tag{4.26}$$

After changing the terms of (4.21) with the equations (4.22), (4.24) and (4.26), the form of the standard loss functional becomes as following

$$\mathcal{L}^{st} = w_I^{st} \frac{1}{N_I} \sum_{i=1}^{N_I} |Res(u_N)|^2 \tag{4.27}$$

$$+ w_D^{st} \frac{1}{N_D} \sum_{i=1}^{N_D} |Res_D(u_N)|^2 \tag{4.28}$$

$$+ w_N^{st} \frac{1}{N_N} \sum_{i=1}^{N_N} |Res_N(u_N)|^2 \tag{4.29}$$

$$+ w_p ||\mathcal{L}_p||_{L_2}^2. \tag{4.30}$$

Numerical experiment with a smooth known solution

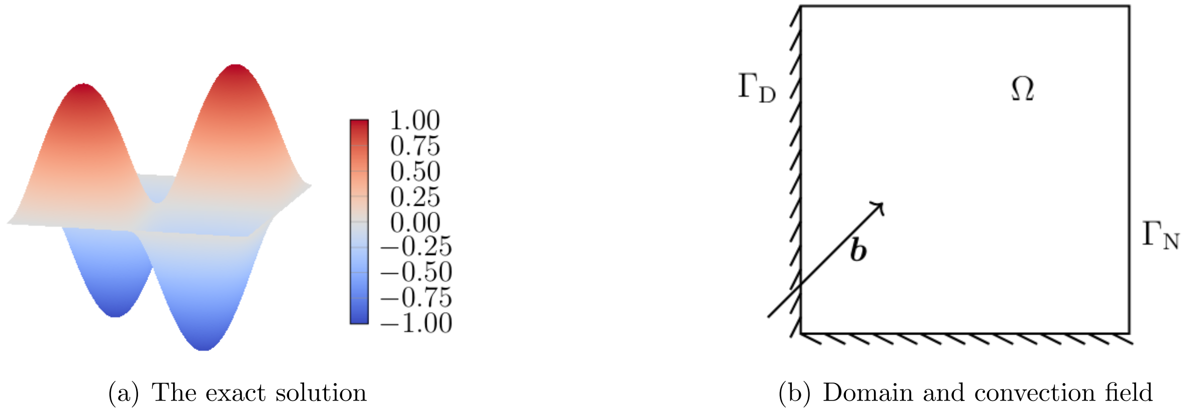
This numerical experiment is based on the example given in [FM23], p.96. It aims in understanding the effectiveness of the PINN in solving the convection-diffusion-reaction problems by comparing the approximated solution with the exact solution. The selected problem is called *Adapted Sine Laplace problem* in two dimensions. This problem uses a known exact solution that the PINN will try to approximate in computing L^2 error.

The formulation of Adapted Sine Laplace PDE is of the form

$$-\epsilon \Delta u(x, y) + \beta \nabla u + r u(x, y) = f(x, y). \tag{4.31}$$

where $\epsilon = 10^{-8}, \beta = (1, 1)^T$ and $r = 1$ and the source term is defined as

$$f(x, y) = 8\pi^2 \sin(2\pi x) \sin(2\pi y). \tag{4.32}$$



(a) The exact solution

(b) Domain and convection field

Figure 4.2: Domain, direction of convection field and exact solution of 4.2.2. Lines with ticks indicate Dirichlet boundary Ω_D and lines without ticks indicate Neumann boundary Ω_N [FM23].

Moreover, the boundary conditions are homogeneous Dirichlet conditions

$$\begin{aligned} u(0,y) =& 0 \quad for \quad 0 \leq y \leq 1 \\ u(1,y) =& 0 \quad for \quad 0 \leq y \leq 1 \\ u(x,0) =& 0 \quad for \quad 0 \leq x \leq 1 \\ u(x,1) =& 0 \quad for \quad 0 \leq x \leq 1. \end{aligned}$$

The exact solution for this problem is given by the following

$$u(x,y) = \sin(2\pi x)\sin(2\pi y). \tag{4.33}$$

where the domain $\Omega = (0,1)^2$. The exact solution is depicted in the Figure 4.2. The right and the top edge of the unit square are set as Neumann boundary, while on the left and the bottom edge Dirichlet boundary conditions are prescribed.

The implementation of the PINN model uses Python, along with the open-source libraries TensorFlow and Numpy [Gé19]. For the optimization step and after trying 60 combinations, the architecture and hyperparameters used here closely follow those in [FM23]. First, Adam algorithm is the selected optimizer with TensorFlow's default values, except for the learning rate that is defined specific for this numerical example.

For this experiment, the FNN architecture consists of an input layer with two nodes, followed by eight hidden layers, each containing 20 nodes, and a single node output layer. The hidden layers use the GeLU activation function, while the output layer employs a linear activation function. The model weights are initialized using Glorot initialization with a random seed of 41 and biases are initially set to zero.

The PINN model minimizes a standard loss function of the formula (4.27) with uniform weights of 1. The training proceeds over 7500 epochs and 32 batch size with a learning rate

of $10^{-2} \cdot 3^{-3}$. This learning rate has been selected, because it provides the lowest value of standard loss functional in the search of the best architecture. Integral approximations are computed using 320 evenly spaced boundary points and 512 evenly spaced interior points, half of the boundary points fall on the Dirichlet boundary and the remaining half on the Neumann boundary, as illustrated in the Figure 4.4. After the training, the model returns the smallest loss functional value during the optimization, the time taken for the training process and the L^2 error. Note that this could be not the last optimization step.

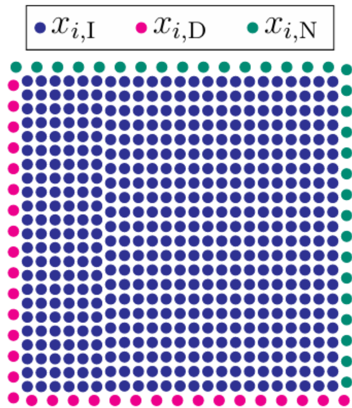
Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 20)	60
dense_1 (Dense)	(None, 20)	420
dense_2 (Dense)	(None, 20)	420
dense_3 (Dense)	(None, 20)	420
dense_4 (Dense)	(None, 20)	420
dense_5 (Dense)	(None, 20)	420
dense_6 (Dense)	(None, 20)	420
dense_7 (Dense)	(None, 20)	420
dense_8 (Dense)	(None, 1)	21

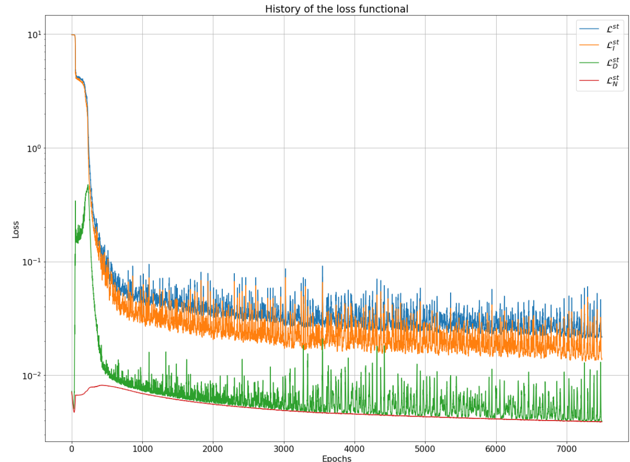
Total params: 3,021
Trainable params: 3,021
Non-trainable params: 0

Figure 4.3: Representation of the FNN model's layers, nodes, number of parameters and output shape used in training PINN to approximate Example 4.2.2.

Training the model took approximately $534s$ updating 3,021 parameters used in the model. `print(pinn.model.summary())`,in the code, provides a table that shows in details the composition of the model: the name of the layers, the output shape and the number of the parameter of each layer. It adds also the total parameters and the total of trainable and non-trainable parameters. Those parameters are distributed over the layers in the following way 60, 420 and 21.



(a) Training points

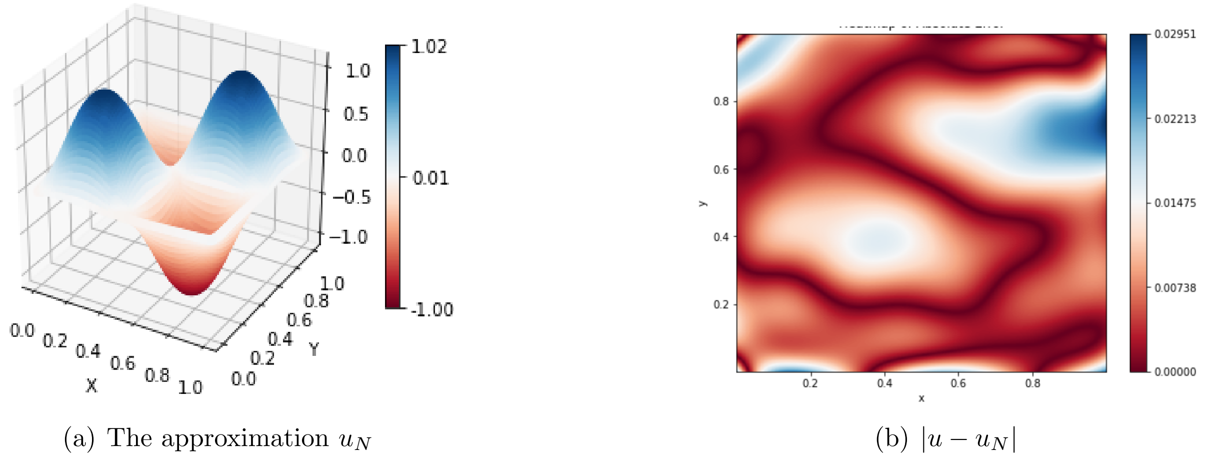


(b) training history

Figure 4.4: Training points [FM23] and history of loss functional values for PINN approximation to Example 4.2.2.

The approximated solution and the value of the standard loss functional and its components are depicted in the Figures 4.2 and 4.4. It can be observed that the loss functional decreases in oscillation over 7500 epochs. There a tendency that the values of the loss functional decrease when the number epochs increases. On the other hand, the Neumann boundary loss functional increases slightly at the beginning and after 1000 epochs, it starts to decrease significantly without oscillation.

The model produces a reasonable approximation to the exact solution, even though the maximum is not perfectly met and the Dirichlet boundary conditions are not precisely satisfied. This is due to their inclusion in the loss function, which requires them to be learned during training unlike continuous Galerkin finite element methods, where Dirichlet conditions are enforced explicitly. The PINN approximation achieves a L^2 error of about $||u - u_{\mathcal{N}}||_{L^2} \approx 0,0092504$ and a minimum loss functional of about $\approx 0,021239$. However, the approximation probably becomes more accurate if more training points are chosen, or the model is trained for more epochs since after 7500 epochs, the loss is still decreasing.



(a) The approximation u_N

(b) $|u - u_N|$

Figure 4.5: PINN approximated solution u_N and point-wise error for Example 4.2.2.

In summary, the implementation appears to be correct, demonstrating the ability of PINNs to approximate solutions to convection-diffusion-reaction problems. However, it is noted in the literature that PINNs may struggle significantly when it comes to solutions that possess layers [DFMa].

4.3 Hard-constrained PINNs

Training the boundary conditions has been a challenge, since they are important for the proper possession of the problem and the shape of the solution in the interior. Any incorrect or missing information about the boundary conditions can cause a problem with the inaccuracy of the approximation of the solution [DFMa]. Learning the boundary condition in the training phase is expensive in time, because it can be taken as a-prior known information [DFMa].

A new variation of PINNs has been proposed and detailed in the paper [LPY+21], called *hard-constrained PINN*. It ensures accurate representation of inlet boundary conditions. The idea of this approach prescribes the Dirichlet boundary condition the same way as in the standard finite element methods and it avoids the need to select user-chosen constants, since $w_D^{st} = 0$. The other constant just scales the functional, which does not change the problem and let $w_I^{st} = 1$ [DFMa].

Consider the following Dirichlet boundary condition of the convection-diffusion equation

$$u(x,t) = g_D(x) \quad x \in \Omega_D \tag{4.34}$$

where Ω_D is the Dirichlet boundary set. The main work is to construct a function $\tilde{g} : \overline{\Omega} \to \mathbf{R}$ as a continuous extension of g_D from Ω_D to Ω. Next, the solution is written as:

$$\tilde{u}_N = \tilde{g}_N + \mathrm{l}u_N \tag{4.35}$$

where u_N is the network output and the l is an indicator function satisfying the following two conditions:

$$\begin{aligned} \mathrm{l}(x) &= 0, \quad \text{if } x \in \Omega_D \\ \mathrm{l}(x) &> 0, \quad \text{if } x \in \Omega_D^c \end{aligned} \tag{4.36}$$

where Ω_D is the Dirichlet boundary domain. By construction, $u_N|_{\Omega_D} = g_D$, which means that the Dirichlet conditions are satisfied exactly and the terms $\mathcal{L}_D^{st}$ are zero and neglected during the training.

Numerical experiment with a smooth known solution

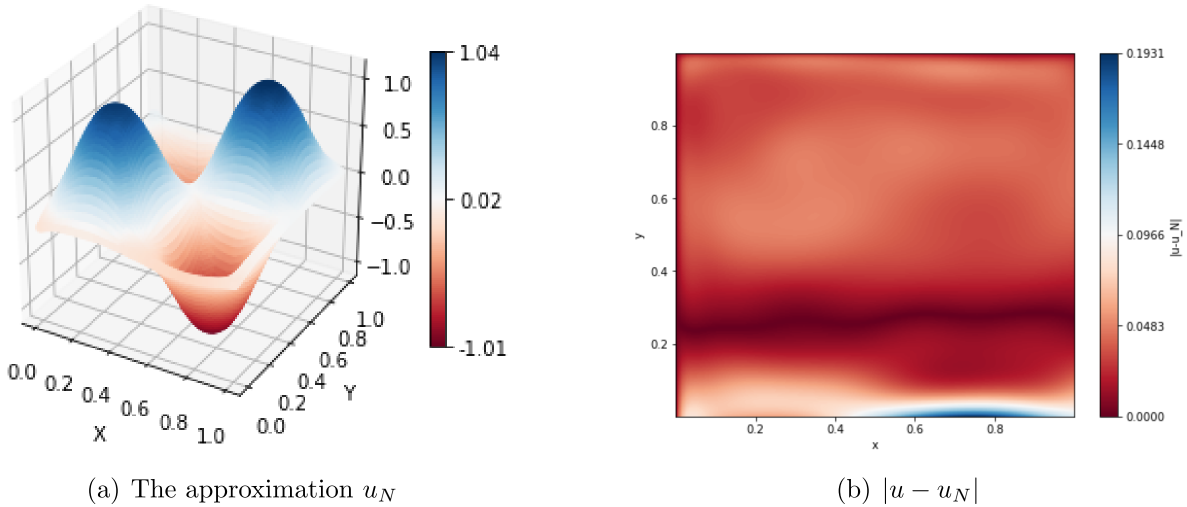
The same example 4.2.2, which is utilized in proving the effectiveness of the PINN in the subsection 4.2.2, is also utilized in this part to assess the performance of the hard-constrained PINN. The exact solution for this problem is defined as

$$u(x, y) = \sin(2\pi x) \sin(2\pi y), \tag{4.37}$$

where the domain $\Omega = (0, 1)^2$. In order to enforce the boundary conditions, the indicator function l, adapted from [VJ24], was applied with $\kappa = 30$ as

$$\mathrm{l}(x, y) = (1 - e^{-\kappa x})(1 - e^{-\kappa y})(1 - e^{-\kappa(1-x)})(1 - e^{-\kappa(1-y)}), \tag{4.38}$$

The same FNN architecture, that is used in the Example 4.2.2, is trained over 7500 epochs. The hard-constrained PINN demonstrated a good ability to approximate the exact solution in the given example, achieving an error of approximately $||u - u_{\mathcal{N}}||_{L^2} \approx 0.040316$ and a loss functional value of $\approx 0,021136$ after 7500 epochs. These results were obtained within 487 seconds. Using the Figure 4.6, the hard-constrained PINN manages to produce an approximation that closely similar to the exact solution, with minimal deviations that exceed the maximum value by 0.04 and the minimum by -0.01. This underscores the efficacy of the hard-constrained approach in maintaining boundary precision while minimizing computational overhead.



(a) The approximation u_N (b) $|u - u_N|$

Figure 4.6: Hard-constrained PINN approximated solution u_N and point-wise error for Example 4.3.

Compared to the vanilla PINN, the hard-constrained PINN not only achieved a lower loss functional value but also it took a shorter computational time. In vanilla PINN, the Dirichlet boundary conditions were included as part of the standard loss functional but were not strictly enforced. This reflected in higher computational costs and slightly less precise adherence to the boundary conditions. On the other side, the hard-constrained PINN enforces these constraints directly, eliminating the Dirichlet loss functional and ensuring superior boundary precision. In the Figure 4.7, the Dirichlet loss is reflected in a zero value.

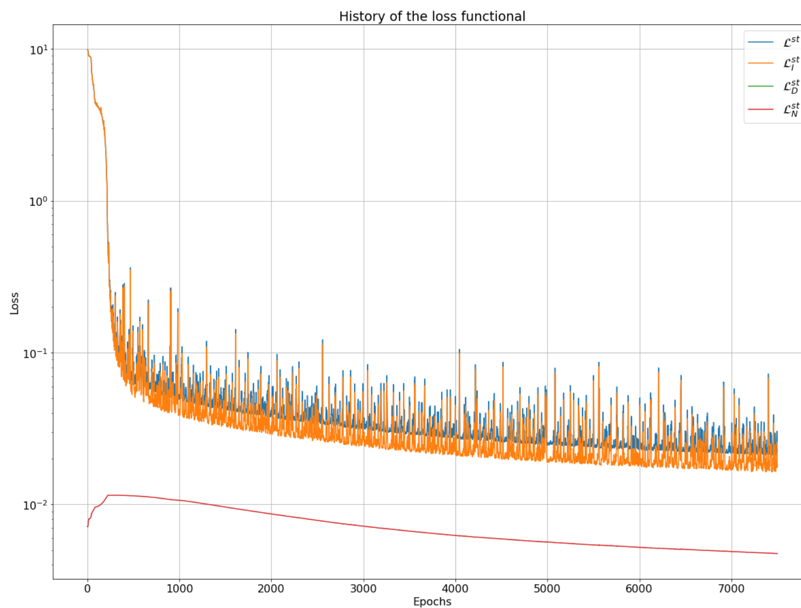


Figure 4.7: History of loss functional values for hard-constrained PINN approximation to Example 4.3.

The findings show the significant advantages of hard-constrained PINNs over vanilla PINNs, particularly in scenarios where precision and computational efficiency are important. By enforcing boundary conditions directly rather than relying on their inclusion in the loss functional, hard-constrained PINNs achieve better approximation accuracy and reduced computational time. The hard-constrained PINN is used in the numerical studies in the next Chapter 5 of this master thesis.

Chapter 5

Numerical Studies

Adam, an adaptive learning rate optimizer, adjusts the learning rate for each parameter based on the magnitude and history of gradients. This feature allows Adam to exploit the local geometry of the loss function effectively. Compared to SGD, Adam often provides faster convergence due to its ability to adaptively modify the learning rates during optimization. Despite its advantages, there are scenarios, where Adam performed poorly in compared to SGD. For instance, in certain image classification tasks, Adam's generalization capabilities can be significantly inferior to SGD [AGK21]. On the other hand, for applications such as training transformers, Adam tends to outperform SGD, demonstrating its versatility across various tasks. Recent research [PL23] investigates deeper into this performance gap between the two optimizers, revealing that SGD exhibits worse directional sharpness in update steps compared to Adam, contributing to its slower convergence.

Nadam combines the Adam algorithm with Nesterov momentum. This optimizer predicts the direction of future updates based on the current gradient. According to the study [Doz16], Adam and Nadam were evaluated across various scenarios. Nadam showed superior performance in tasks such as word2vec and LSTM-LM, particularly when dropout was applied. However, in simpler tasks like MNIST, Adam sometimes performed equally well or better, depending on factors such as model size, task complexity and the potential for overfitting.

BFGS, a quasi-Newton method, aims to combine the benefits of Newton's method without incurring the heavy computational cost associated with calculating the Hessian matrix. Similar to the conjugate gradient method, BFGS utilizes second-order derivative to enhance the optimization. However, it is less reliant on line search accuracy, enabling faster refinement at each iteration [GBC16]. This makes BFGS a compelling choice for problems where the computational overhead of Newton's method is prohibitive. However, its implementation within TensorFlow presents significant challenges, which are explained in details in the subsection (5.1.5). Due to those challenges, the analysis are continued using three optimizers, SGD, Adam and Nadam.

This chapter focuses on comparing three optimizers on solving convection-diffusion-reaction problems using PINN. The primary goal is to evaluate and compare the performance

of three used optimizers: SGD, Adam, and Nadam based on the low value of loss functional, the time of training the model and also the accuracy. The chosen examples are the circular Interior Layer and Outflow Layer. The comparison will include a table of hyperparameters based on [VJ24] with a bit of change, ensuring consistency in the experimental setup.

5.1 Benchmark Problem Overview

The Outflow Layer and the Circular Interior Layer have been selected as examples of steady-state convection-diffusion-reaction problems for evaluating the hard-constrained PINN, because they possess analytical solutions. This allows a direct comparison with the exact solution, which facilitates the assessment of the accuracy of the model.

This section introduces both examples, followed by a detailed discussion of the hyperparameters and evaluation metrics used during training. Finally, the challenges encountered while implementing the BFGS optimizer are addressed.

5.1.1 Circular interior layer

The circular Interior Layer problem is introduced in the following example, as detailed in [JVMJT97]. The exact solution is given by :

$$u(x,y) = 16x(1-x)y(1-y)(\frac{1}{2} + \frac{\arctan(200(r_0^2 - (x-x_0)^2 - (y-y_0)))}{\pi}), \tag{5.1}$$

where $r_0 = 0,25$ and $x_0 = y_0 = 0,5$. For the convection-diffusion-reaction equation, the parameters have the following values $\beta = (2,3)^T$, $r = 2$ and $\epsilon = 10^{-8}$ within the domain $\Omega = (0,1)^2$. The boundary $\Gamma_D = \partial\Omega$ is the subject to Dirichlet conditions. Dirichlet boundary conditions are defined along Γ_D to match the exact solution $u(x,y)$ at the boundaries. The exact solution of the Circular Interior Layer is visualized in Figure 5.1.

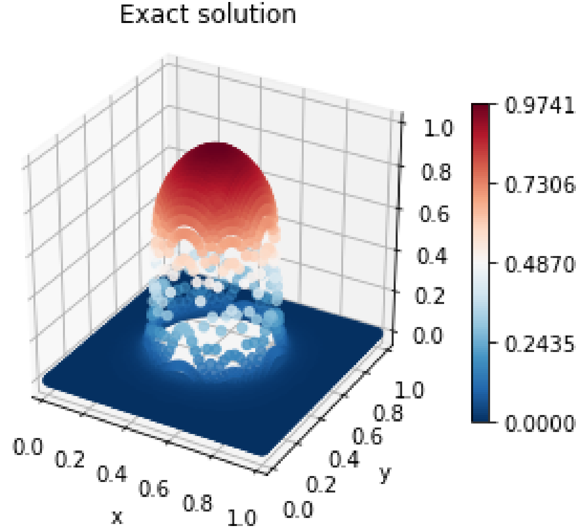


Figure 5.1: Exact solution of Circular Interior Layer example 5.1

The solution consists of an interior layer in the circle. The radius of the circle is 0.25. This example involves studying diffusion and convection effects in a bounded domain with a particular focus on the behavior near the interior layer defined by a circular region around (x_0, y_0). The derived source term $f(x, y)$ and the boundary conditions are chosen such that they are consistent with the exact solution, ensuring that the solution satisfies the specified convection diffusion reaction equation within Ω.

5.1.2 Outflow layer

This subsection introduces the Outflow Layer problem, a key example derived for the steady state convection-diffusion-reaction equation as detailed in example 3 provided by [JVMJT97]. This equation has the values of $\beta = (2, 3)^T$, $r = 1$ and $\epsilon = 10^{-8}$. The exact solution is written as:

$$u(x, y) = xy^2 - y^2 \exp\Big(\frac{2(x-1)}{\epsilon}\Big) - x \exp\Big(\frac{3(y-1)}{\epsilon}\Big) + \exp\Big(\frac{2(x-1) + 3(y-1)}{\epsilon}\Big) \tag{5.2}$$

which visualized in Figure 5.2. The domain Ω is $(0, 1)^2$, the unit square and the boundary condition is $\Gamma_D = \partial\Omega$.

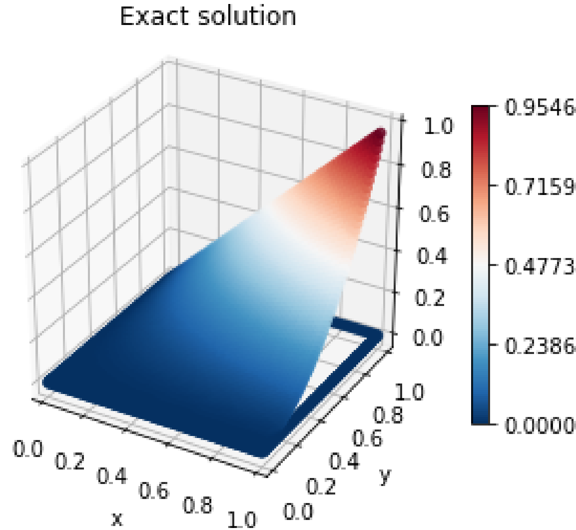


Figure 5.2: Exact solution of Outflow Layer example (5.2)

The Outflow Layer problem models the transport of a quantity, such as temperature, through the domain where the flow is primarily in the direction of the convection field. The small diffusion parameter and strong convection field create unique challenges in accurately capturing the solution's behavior, especially the layers near the boundaries.

5.1.3 Hyperparameters for the training process

In the Chapter (2.2.3), the crucial role of the hyperparameters in the FNN model is explained in details and how it affects its performance. The optimal combination of the hyperparameters can affect directly the accuracy of the model. In this case, the hyperparameters are chosen in correspondence with the practical recommendations from [FM23] and [VJ24]. To this end, in each experiment different configurations of PINN are optimized whose hyperparameters are given in Table 5.1.

Table 5.1: Values of hyperparameters used in training hard-constrained PINN.

Hyperperparameters	Values
Number of epochs	1,000, 10,000
Learning rate	$0.01.3^k$ for k in $\{-6, .., 0\}$
Number of hidden layers	5, 8, 10, 15
Activation function	GeLU, Tanh, Sigmoid, Swish
Batch size	32
Nodes per hidden layer	20
Initialization seed	41
Optimitzers	SGD, Adam, Nadam

The choice of the epochs number is very important, since for a very low value the model may not train long enough to minimize the loss functional. Also, the high value of the epochs number can cause memorizing the training data, where the model loses the ability to be accurate when it comes to new data that never seen before [GBC16]. Based on [GBC16] and [Ben12], the ideal number of training epochs is reached when the model's performance on the training dataset continues to improve, but its performance on the test dataset starts to degrade. This marks the onset of overfitting, where the model begins to lose its ability to generalize to a new, unseen data. To mitigate this, the goal is to adopt strategies that prevent overfitting. In this thesis, overfitting is controlled using early stopping a technique that stops training as soon as the model's accuracy on the test dataset begins to decline, indicating a shift away from optimal generalization. This early stoppage ensures that there is no overfitting.

According to [Ben12], the good default value of batch size is 32. Batch size's impact is mostly computational. In other words, it effects the training time. Its larger value yields to a faster computation but requires visiting more examples, in order to reach the same error, since there are less updates per epoch.

The next crucial hyperparameter is the learning rate. A value that is too high prevents convergence, while values that are too low cause the model to become stuck in local minima. [Ben12] and [GBC16] recommend in choosing a high learning rate and then gradually reduce it. Starting with a learning rate of $0.01 \cdot 3^0$ and scaling it down by a factor of three in each adjustment phase, this strategy is applied until the learning rate reaches a minimal threshold of $0.01 \cdot 3^{-6}$.

The main architecture of the FNN model in hard-constrained PINN is described as the following, the input layer consists of two nodes and the output layer of a single node with linear activation function. The selected numbers of hidden layers are 5, 8, 10, 15. This choice is based on [VJ24]. This paper is based on paper that aims on understanding the impact of the depth of the network on the ability of the model to learn the complex patterns and improve prediction accuracy in contexts informed by physics [PSN23]. Additionally, each hidden layer within the neural network will uniformly contain 20 nodes, establishing a consistent structure across the network to facilitate a controlled evaluation of the model's performance and the impact of other varied hyperparameters.

Finally, the selection of activation functions for the hidden layers will be evaluated, drawing on insights from specialized studies in PINNs and their applications to complex problems. The activation functions considered include GeLU, Tanh, Sigmoid and Swish. These choices are inspired by their effectiveness invarious PINN contexts, aiming to optimize the model's ability to capture and represent the underlying physics of the problem being solved [PSN23]. As mentioned in the previous Chapter (4.1), Sigmoid and Tanh have shown to be a good choice for PINN [Mar21] and GeLU was identified as the most effective with Adam for the example of circular Interior Layer in [VJ24]. The values of the weights are initialized at the beginning based on Glorot initialization [GB10] with the seeds given in Table 5.1, the initial biases are set to zero.

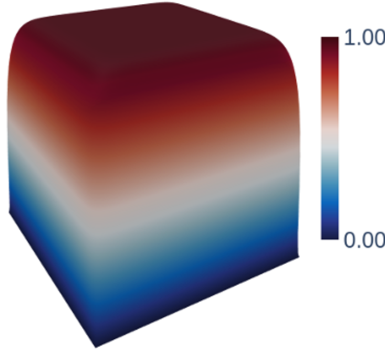


Figure 5.3: The indicator function for the Circular Interior Layer 5.1.1.

Since the focus is on hard-constrained PINN as discussed in the previous Section (4.3), a smooth extension of $\tilde{g}_D$ for the boundary condition is established for any point $(x, y) \in \bar{\Omega}$ where $\tilde{g}_D(x, y) = 0$. For the Circular Interior Layer, the indicator function l as shown in figure, has the following form:

$$l(x, y) = (1 - e^{-\kappa x})(1 - e^{-\kappa y})(1 - e^{-\kappa(1-x)})(1 - e^{-\kappa(1-y)}), \tag{5.3}$$

where $\kappa = 30$ is the chosen scaling factor.

In the Outflow Layer Example 5.1.2, the indicator function, as depicted in Figure 5.4, follows the same structure as Equation (5.3), but employs a different approach to scaling. When examining boundary conditions within a specific domain, the inclusion of a scaling parameter, denoted κ, is crucial to effectively manage the impact of these conditions on the solution's behavior. This parameter is determined based on the convection field's orientation with respect to the boundary and the domain's location.

At boundaries where the fluid exhibits a transition indicative of a boundary layer, the scaling factor is defined as $\kappa_0 = \frac{1}{\epsilon}$, enhancing the model's sensitivity to steep gradients and rapid changes in properties. Conversely, at boundaries where the solution demonstrates minimal changes in flow characteristics—effectively a 'no boundary layer' scenario $\kappa_i = 30$ is set to a significantly different value to reflect the lesser impact of boundary conditions on the flow behavior. The choice of κ in such cases is guided by the need to ensure that the boundary conditions do not artificially constrain the natural development of the solution within the domain.

5.1.4 Metrics for Model Implementation

For problems with analytical solutions, the accuracy of hard-constrained PINNs is measured using the L^2 error. This metric checks how close the model's predictions are to the known

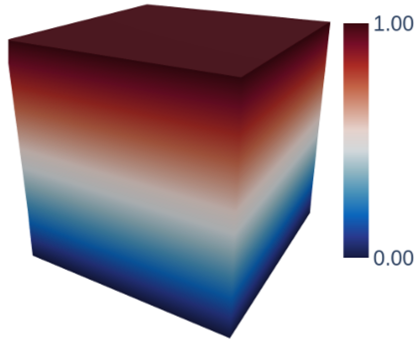


Figure 5.4: The indicator function of the Outflow Layer 5.1.2.

analytical solutions across the problem's domain. The L^2 error measures the difference between two elements u and u_N over a domain Ω using the formula:

$$||u - u_N||_{L^2} = \left(\int_{\Omega} |u - u_N|^2 \, d\Omega \right)^{1/2} \tag{5.4}$$

where u is the exact solution and u_N is the approximation of the PINN. This metric helps to evaluate how well the model approximates the solution. The evaluation also involves checking whether the predictions exceed the known minimum or maximum values of the problem. Two methods are used: tracking oscillations during training and spotting non-physical values in the final output. Together, these methods ensure that the final solution is accurate and physically meaningful while also examining the model's behavior during training.

5.1.5 Challenges of implementing BFGS

```
if loss_functional == "standard" or loss_functional == "crosswind":
res_squared = tf.math.square(batch_elem[:,2] - val_pred)
loss = tf.math.reduce_sum(res_squared * batch_elem[:,3])

train_loss= tf.keras.metrics.Sum()  #sum of interior and boundary losses
interior_loss = tf.keras.metrics.Sum()  #sum of batch losses
boundary_D_loss = tf.keras.metrics.Sum()  #sum of batch losses
boundary_N_loss = tf.keras.metrics.Sum()  #sum of batch losses
```

The code, utilized in this master thesis, is the same used in [FM23]. It relies not only on old versions of the Python's library TensorFlow, but also it introduces difficulties when trying to implement the BFGS optimizer from libraries such as `scipy.optimize.minimize` or TensorFlow Probability's `bfgs_minimize`.

In the code, the loss measures the square difference between the predicted and actual values, denoted by `val_pred` and `batch_elem` [:,2] and then multiplies them by a weight

batch_elem[:,3]. This weighted sum of squared differences gives the loss for a single batch of data. Additionally, the code tracks all components of the loss functional separately.

Upgrading Python version and its library Tensorflow to implement TensorFlow Probability's `bfgs_minimize` unfortunately leads to crashes and incompatibility issues of the code when is runned in Spyder.

`scipy` or TensorFlow Probability's BFGS optimizer expect the loss functional to be a simple function, explicitly defined, that takes `numpy` arrays as inputs and returns a number. However, the loss functional is built using TensorFlow operations like `tf.math.square` and `tf.math.reduce_sum`, which rely on TensorFlow's internal system for automatic differentiation and computational graphs. These operations do not work directly with the libraries mentioned.

Also, BFGS requires that the gradient of the loss function with respect to the model parameters are explicitly provided. TensorFlow automatically calculates gradients for its built-in optimizers like Adam or SGD during training, but this does not directly help with BFGS. To make it work, we would need to manually compute these gradients using TensorFlow tools like `tf.GradientTape` and then ensure that they are in the right format for BFGS.

The loss functional is based on batches. It is calculated separately for each batch of data and then integrateted into the training process. However, `scipy.optimize.minimize` expect a single, global loss function, not one that depends on batch processing. The term `batch_elem[:,3]` is a weight used in the loss computation. In order to use it with `scipy.optimize.minimize`, the optimizer needs to interpret these batch weights during the updating of the parameters. Unfortunately, this is not directly supported by `scipy` or `TensorFlow Probability`. The challenges faced by implementing BFGS prevented a direct comparison of the three optimizers.

5.2 Results

The evaluation of the various optimizers introduced in Chapter 3 is based on training Examples 5.1.1 and 5.1.2 using a hard-constrained PINN. The optimal optimizer is determined by minimizing the L^2 error and maximizing the effectiveness. The following subsection introduces the analysis method, followed by the evaluation of the Circular Interior Layer and the Outflow Layer.

5.2.1 Analysis method

This section focuses on the procedure followed to analyze the results after training hard-constrained PINN approximating Examples 5.1.1 and 5.1.2. The performance of the three optimizers, SGD, Adam and Nadam, is analyzed based on their effectiveness, accuracy and

speed. First, each optimizer, SGD, Adam, and Nadam, is evaluated with different combinations of architectures, as outlined in Table 5.1. These combinations vary in terms of activation functions, learning rates and the number of hidden layers.

Each combination is trained for 1,000 epochs to allow the initial convergence. The optimal architecture of each optimizer was selected based on the lowest L^2 error. These optimal architectures, where each optimizer performed well, allow the optimizers to be fairly compared. Then, these selected configurations were trained again for an additional 10,000 epochs. Comparing the performance of the optimizers after both short-term (1,000 epochs) and long-term (10,000 epochs) training allows for a deeper analysis of the optimizers' performance. The efficiency is related to the values of the loss functional, while accuracy is related to the values of the errors and speed to the time spent by each optimizer to achieve convergence.

Based on each architecture proposed in Table 5.1, the combinations of optimizers with various hyperparameters are analyzed, to assess the influence of the activation function and the number of layers on the optimizer's behavior. After training the optimal models again for short and long-term, the point-wise error is tracked, providing a direct measure of accuracy at individual collocation points. In addition, the training history of the loss functional is presented in graph to gain insight into the overall progress and the rate of change of the loss functional with respect to the number of epochs. Additionally, a comparison between the approximated solution, u_N, and the exact solution presents whether the optimal architecture successfully produced an approximation that is close to the exact solution. The results are summarized in a table that presents the lowest values of L^2 error, loss functional, and training time for the three optimizers. This comparison table serves as a basis for evaluating the efficiency, accuracy and speed of the optimizers.

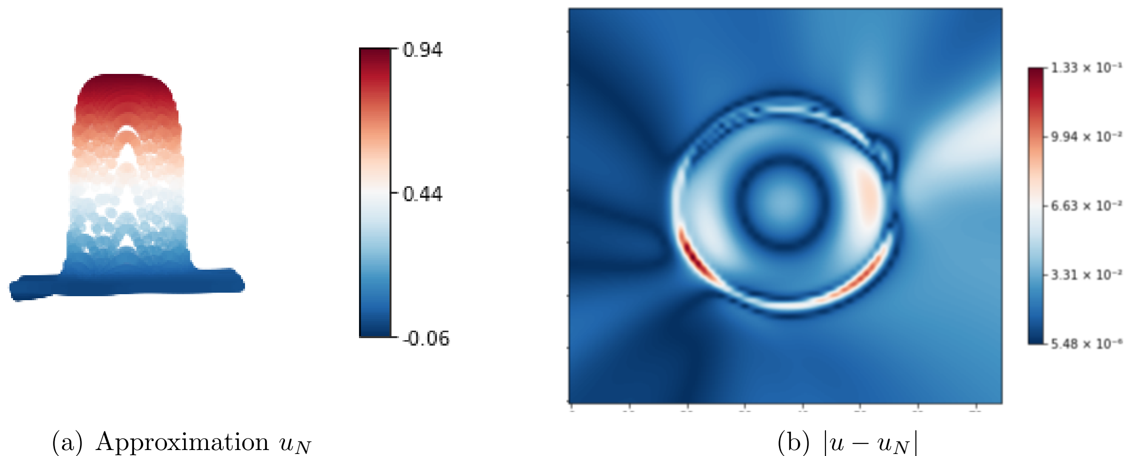
It is important to note that the training time varied slightly each time the code was executed. However, for the purpose of this analysis, only one training time per optimizer was considered. Although the exact training time fluctuates, the speed of the optimizers remained consistent, meaning that the differences in training time did not reflect any significant variation in the optimizers' performance or efficiency.

5.2.2 Circular interior layer

The evaluation of the results achieved by solving the Example 5.1.1 using hard-constrained PINN follows the analysis method mentioned in 5.2.1.

The first used optimizer for approximating (5.1) is SGD. After training the combinations between SGD and the values in Table 5.1 over 1,000 epochs, the results indicated that SGD with various configuration performed well by achieving low L^2 error, but they struggled while minimizing the loss functional. The Figure 5.8 depicted the values of the loss functional of the optimal architectures, which shows that SGD had the highest loss functional values among the other optimizers during training the model over 1,000 epochs. The best performing architecture of SGD composes of 10 hidden layers, a learning rate of value $10^{-2} \cdot 3^{-2}$ and Tanh activation function. This architecture yields a L^2 error of value ≈ 0.023498 and a

corresponding loss functional of value ≈ 7.835987.

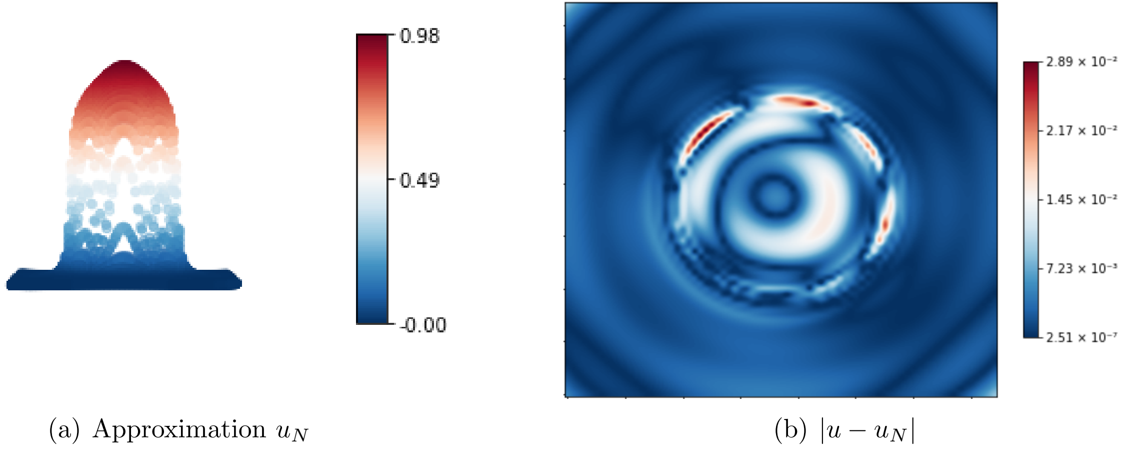


(a) Approximation u_N (b) $|u - u_N|$

Figure 5.5: Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.1 after training over 1,000 epochs.

Figure 5.5 illustrated the approximated solution u_N and the point-wise error produced by SGD optimal architecture. The findings indicated that SGD, after training for 1,000 epochs, successfully produced an approximation that closely similar to the exact solution. However, the optimal architecture did not achieve the maximum value and the minimum value of exact solution. The maximum value was underestimated by 0.02, while the minimum value exceeded the exact solution by -0.06. The pointwise error, depicted in the same Figure, revealed inaccuracies concentrated in the interior regions, particularly around the circular domain. These regions exhibited notable deviations, underscoring areas where the SGD optimizer struggled to achieve high accuracy within the 1,000 epochs training period.

The results, obtained from training hard-constrained PINN using Adam for 1,000 epochs, show that the activation functions Tanh and GeLU outperformed Sigmoid and Swish, achieving lower values of L^2 error and loss functional. In contrast, Sigmoid and Swish resulted in higher values for both metrics. As shown in the Figure 5.8, the loss functional produced by Adam exhibited a significant decrease with respect to the number of epochs, showing superior performance compared to SGD. The optimal architecture of Adam optimizer is decomposed of 8 hidden layers, a learning rate of value $10^{-2} \cdot 3^{-4}$ and the GeLU activation function. This architecture achieved an L^2 error of value ≈ 0.004156 and a significantly lower loss functional of ≈ 0.553932.

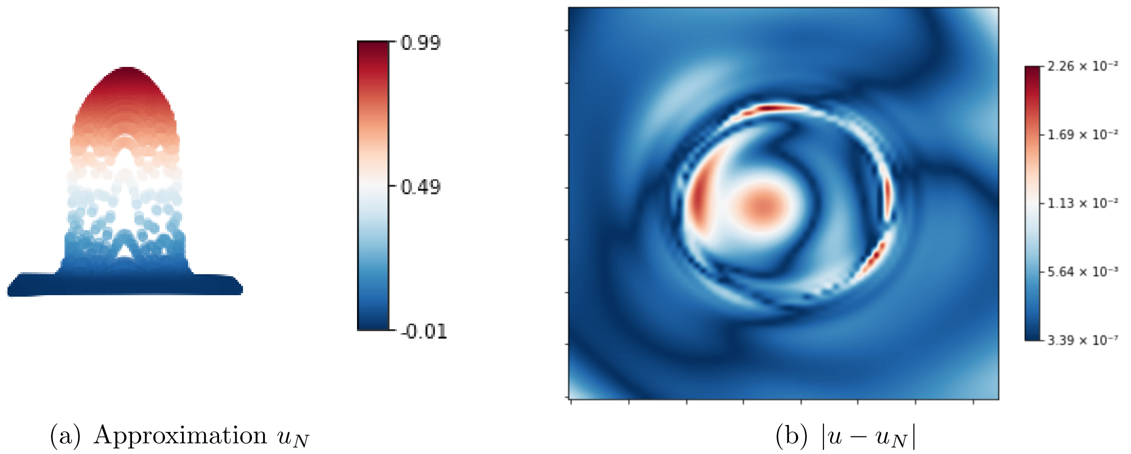


(a) Approximation u_N (b) $|u - u_N|$

Figure 5.6: Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.1 after training over 1,000 epochs.

The Figure 5.6 illustrated the approximated solution u_N and the point-wise error by the optimal architecture of the optimizer Adam. The findings indicated that Adam, after training for 1,000 epochs, successfully managed to produce an approximation that closely mirrors the exact solution in compare to SGD5.1. The maximum value of the approximated exceed the maximum value of the exact solution by 0.01. Analysis of the point-wise error revealed where certain areas within the interior are not accurately approximated, similarly to the observed scenario of the optimizer SGD.

After training the model using Nadam optimizer, Nadam's best architecture had 5 hidden layers, with the same optimal learning rate as SGD $10^{-2} \cdot 3^{-2}$ and GeLU activation function, resulting an L^2 error of value ≈ 0.004197 and a loss functional of value ≈ 0.471659. Compared to Adam, Nadam seemed to favor shallower architectures, likely due to its momentum-based update mechanism, that stabilizes training in smaller networks. The activation functions Tanh, GeLU and Swish outperformed Sigmoid, by achieving lower loss functional and L^2 error values. As shown the Figure 5.8, the loss functional produced by Nadam exhibited a significant decrease with respect to the number of epochs, showing superior performance compared to SGD and Adam.



(a) Approximation u_N

(b) $|u - u_N|$

Figure 5.7: Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.1 after training over 1,000 epochs.

The Figure 5.7 illustrated the approximated solution u_N and the pointwise error after 1,000 epochs by the optimal architecture of the optimizer Nadam for the Example 5.1.1. The findings indicated that Nadam managed to produce an approximation that closely similar to the exact solution 5.1. The maximum value exceed the maximum value of the exact solution by 0.02 and the minimum value by $-0, 01$. Analysis of the point-wise error in the Figure 5.7 reveals that Nadam failed to accurately represent the peak region of the problem and the surrounding area of the cycle is also poorly approximated.

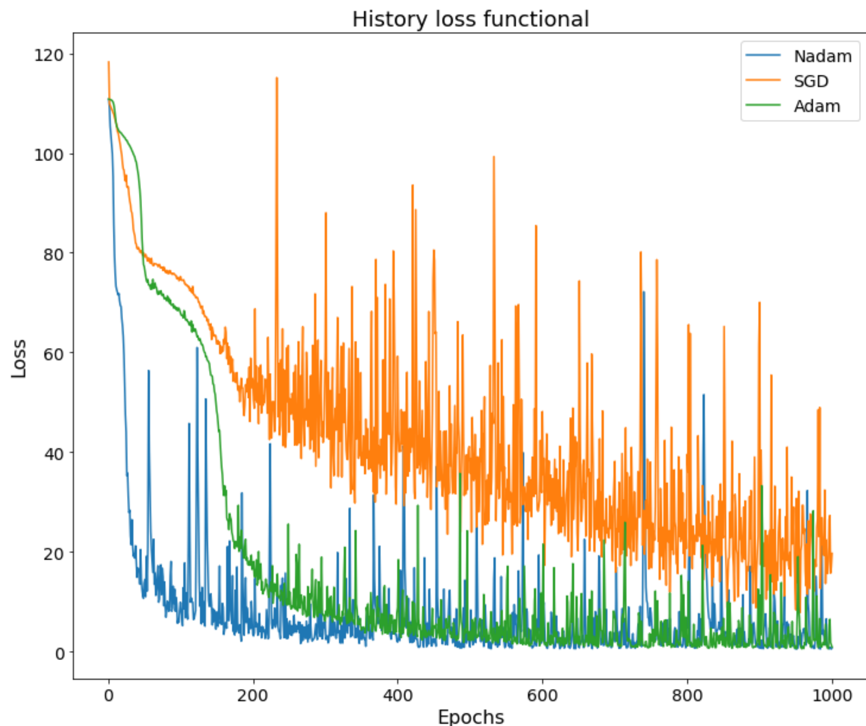


Figure 5.8: History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam optimizers over 1,000 epochs to Example 5.1.1.

In summary, the Sigmoid activation function consistently performed poorly in every optimizer. Its tendency to saturate at extreme values results in vanishing gradients, severely hampering optimization. Similarly, architectures with 15 hidden layers led to high L^2 errors and loss functional values for all three optimizers. Additionally, Adam was the only optimizer that successfully produced an approximated solution u_N that is very close to the exact solution in comparison to the other optimizers.

Table 5.2: Values of L^2 error, loss functional, and training time over 1,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Circular Interior Layer problem 5.1.1.

Optimizer	Loss functional	L^2 error	Training time (sec)
SGD	7,835987	0,023498	114
Adam	0,553932	0,004156	177
Nadam	0,471659	0,004197	229

The Table 5.2 presents the values of the loss functionals, L^2 error and the training time of the best architecture of the three used optimizers after training over 1,000 epochs solving the example 5.1.1. It shows that Adam achieved the lowest L^2error. Nadam followed closely a slightly higher L^2 error, while SGD lagged behind the highest L^2error. This observation was a bit similar to the values of the loss functional, where Nadam achieved the lowest

loss functional, marginally outperforming Adam. In contrast, SGD produced a significantly higher loss functional. In the Figure 5.8, It is evident that Nadam outperformed other optimizers with minimizing the loss functional values. However, in terms of training speed, SGD required a less time compared to Adam and Nadam. This indicates that while SGD is computationally faster, it struggles to achieve comparable efficiency and accuracy to the other two optimizers within the same training duration.

Table 5.3: Values of L^2 error, loss functional, and training time over 10,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Circular Interior Layer problem 5.1.1.

Optimizer	Loss functional	L^2 error	Training time (sec)
SGD	0,21427	0,004441	1466
Adam	0,194481	0,001364	2669
Nadam	0,209872	0,002784	2191

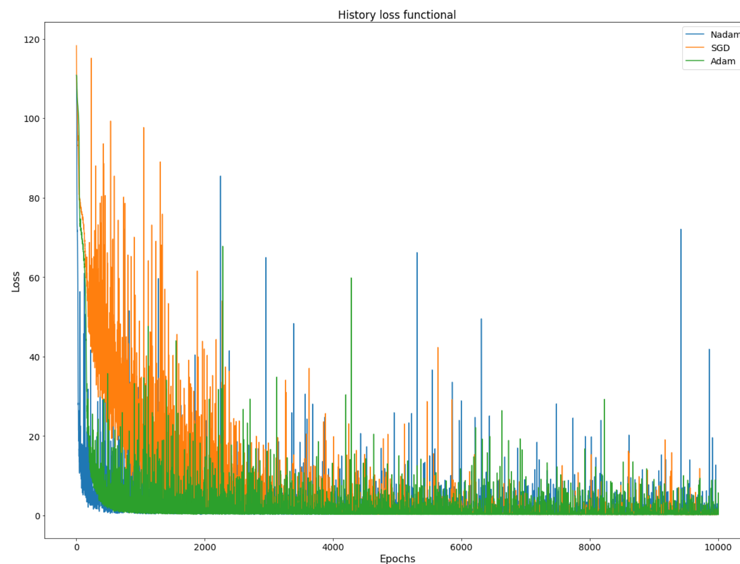
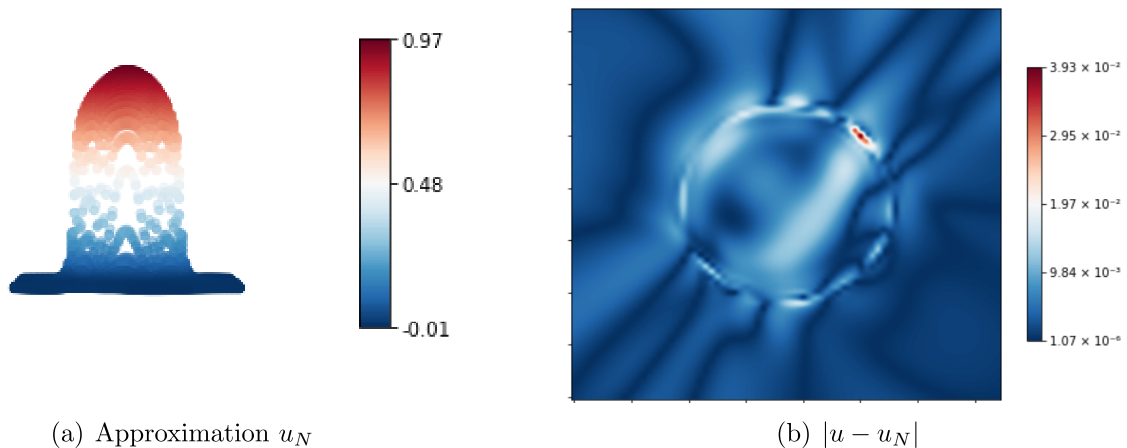


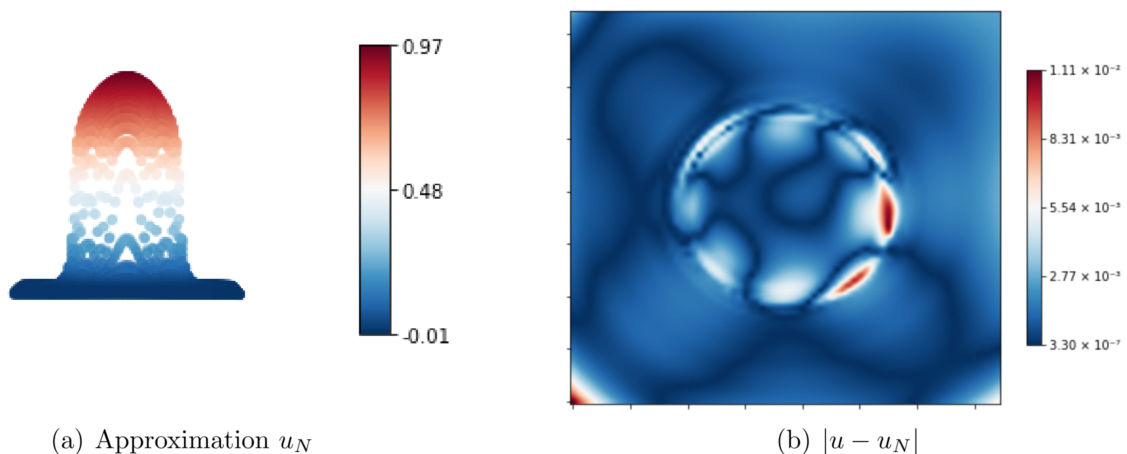
Figure 5.9: History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam optimizers over 10,000 epochs to Example5.1.1.

Extending the training epochs to 10,000 provides a significant reduction in both loss functional and L^2 error across all optimizers. Adam continued outperforming the other optimizers by achieving the lowest values of the error and loss functional. But, the values of Adam and Nadam are still closed to each other as shown in the Figure 5.9. SGD, though improved, remained the least accurate. It achieved approximately close $L2$ error, that was achieved by Adam after 1,000 epochs. The training time analysis reveals that longer training durations amplify the computational cost differences among the optimizers. Adam required the longest time at 2669 seconds in compared to Nadam and SGD.



(a) Approximation u_N (b) $|u - u_N|$

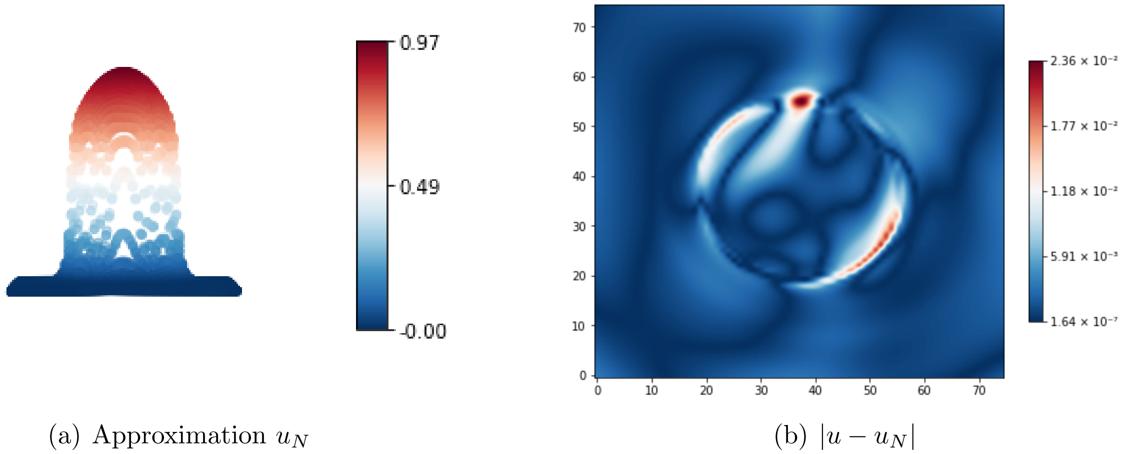
Figure 5.10: Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.1 after training over 10,000 epochs.



(a) Approximation u_N (b) $|u - u_N|$

Figure 5.11: Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.1 after training over 10,000 epochs.

The Figures 5.10, 5.11 and 5.12 illustrate the approximated solution and point-wise error after training over 10,000 epochs for the optimizers SGD, Adam and Nadam. Compared to the Figures 5.5, 5.6 and 5.7, the optimizers achieved notable improvements related to the approximated solution and the point-wise error. The findings indicate that the optimizers, successfully managed to produce a better approximated solution than the ones obtained after 1,000 epochs. Nadam is the only optimizer, which produced exactly the same exact solution. Analysis of the point-wise error revealed significant correction to the inaccuracies concentrated in the interior, the peak regions and the surrounding area of the cycle. The most remarkable one belongs to the SGD optimizer.



(a) Approximation u_N (b) $|u - u_N|$

Figure 5.12: Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.1 after training over 10,000 epochs.

In summary, based on this analysis, during short-term training, Adam demonstrated superior accuracy, while SGD emerged as the fastest optimizer. However, Nadam outperformed in efficiency, by achieving the lowest loss functional value. In long-term training, Adam proved to be the best optimizer, achieving the highest accuracy and efficiency. Meanwhile, SGD maintained its position as the fastest optimizer, though it lagged behind in accuracy and efficiency compared to Adam and Nadam. Nadam offers a good middle ground, achieving results close to Adam but with slightly less training time.

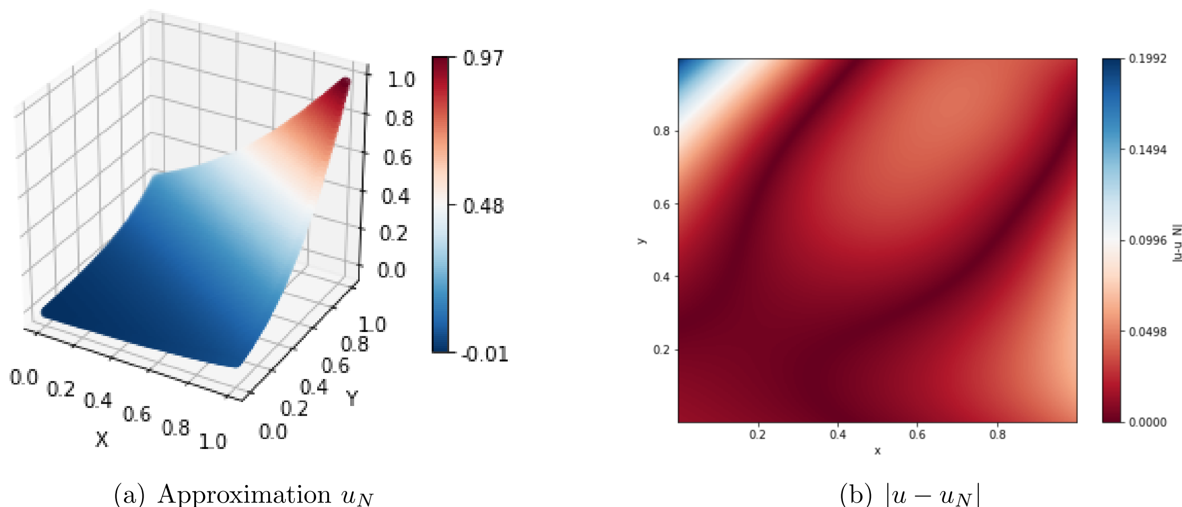
5.2.3 Outflow layer

The same analysis method 5.2.1 is followed, while analyzing the results of the Outflow Layer 5.1.2. In order to have a fair comparison between different optimizers in training hard-constrained PINN solving the example 5.1.2, the optimizers SGD, Adam, and Nadam are trained for 1,000 epochs to find the optimal architecture. Firstly, the interplay between optimizers and hyperparameters, including the learning rate, number of hidden layers, and activation functions are analyzed. Then, these best architecture are trained again for 10,000 epochs, to examine how the optimizers performed over extended training durations. It allowed for a detailed comparison of the optimizers' accuracy, efficiency and speed, contrasting the performance after short-term and long-term training.

According to [Mat24], the approximation of the Outflow Layer appears to be more challenging. One possible explanation for the challenges faced by hard-constrained PINNs in modeling the Outflow Layer problem could be the complexity of the solution near the boundaries. This complexity makes it difficult for hard-constrained models to adjust adequately, emphasizing the necessity for models that provide greater flexibility at the boundaries to effectively capture steep gradients.

After training hard-constrained PINN using SGD for 1,000 epochs to approximate Ex-

ample 5.1.2, the results show that the SGD optimizer generally does not perform well with the Tanh activation function. In contrast, the GeLU activation function achieves lower L^2 error values compared to Tanh. However, both Swish and Sigmoid show poor performance. The optimal architecture for SGD consists of GeLU as the activation function, 15 hidden layers and a learning rate of value 10^{-2}. This architecture achieves L^2 error of ≈ 0.032732 and loss functional of value $\approx 0,230783$.



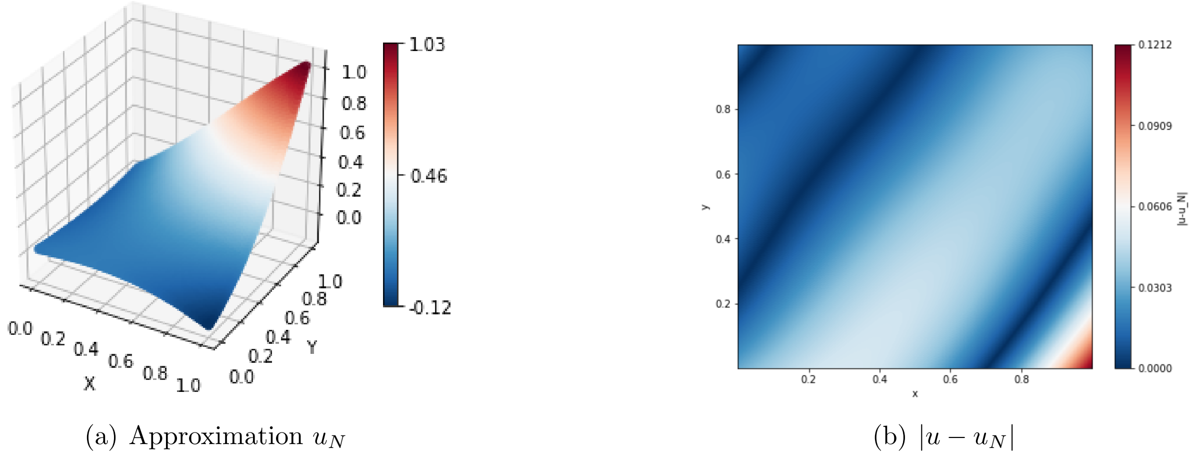
(a) Approximation u_N

(b) $|u - u_N|$

Figure 5.13: Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.2 after training over 1,000 epochs.

The Figure 5.13 illustrates the hard-constrained PINN approximation and the point-wise L^2 error of the optimizer SGD over 1,000 epochs. The findings indicate that SGD optimal architecture successfully produces an approximation similar to the exact solution 5.2. However, the approximated solution shows overshoots in the maximum value by $0,97$ and also in the minimum value by -0.01. Additionally, the approximation on the corner (1,1) appears to be inaccurate, as indicated by the pointwise error.

After training the Adam optimizer combinations for 1,000 epochs, the activation functions Sigmoid and Tanh perform very poorly by achieving consistently high L^2 error values. In contrast, GeLU and Swish perform poorly only with high learning rate value but deliver good results otherwise. The optimal configuration of the Adam optimizer consists of 5 hidden layers, a learning rate of $10^{-2} \cdot 3^{-5}$, and GeLU as the activation function, resulting in an error of approximately 0.030243 and loss functional of value $\approx 0,019591$.

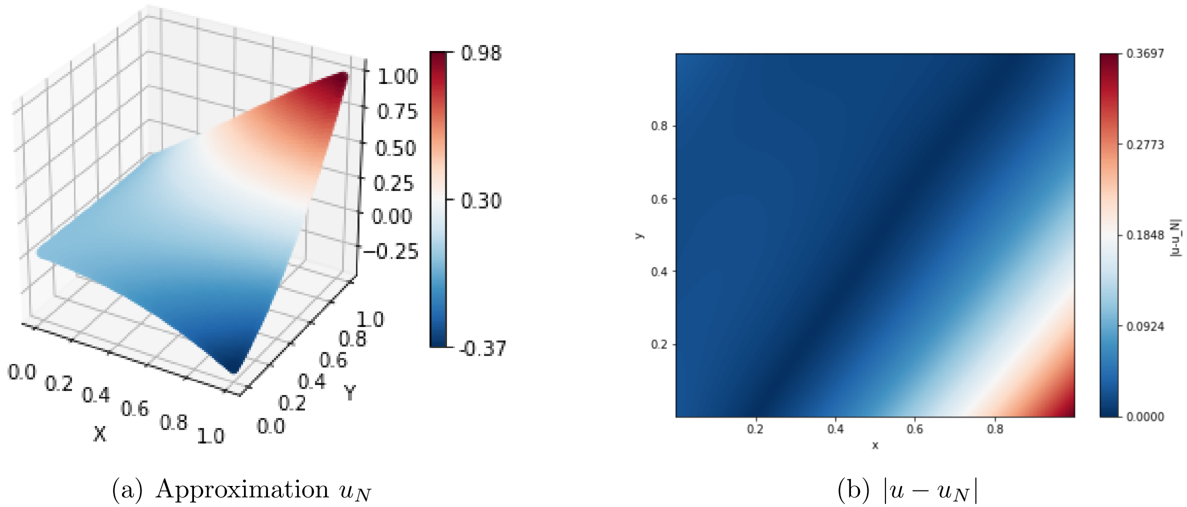


(a) Approximation u_N (b) $|u - u_N|$

Figure 5.14: Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.2 after training over 1,000 epochs.

The Figure 5.14 illustrates the hard-constrained PINN approximation and the point-wise L^2 error of the optimizer Adam over 1,000 epochs. The findings show the model exceed the maximum and the minimum values of the exact solution 5.2 by $0,08$ and $-0,12$. Also, the error in the corner (1,1) appears to be of the same magnitude as the error in the rest of the solution.

For 1,000 epochs, Nadam optimizer displays a strong performance across various configurations, only with the activation function Sigmoid and the highest learning rate value. The Swish activation function achieves excellent results, providing the optimal architecture, incorporating 5 hidden layers and a learning rate of $10^{-2} \cdot 3^{-2}$. This architecture achieves L^2 error of approximately ≈ 0.074862 and a loss functional value of ≈ 0.01236.



(a) Approximation u_N (b) $|u - u_N|$

Figure 5.15: Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.2 after training over 1,000 epochs.

The Figure 5.15 illustrates the hard-constrained PINN approximation and the point-wise L^2 error of the optimizer Nadam for 1,000 epochs. Similar to Adam's case, the findings show that the model exceeds the maximum and the minimum values of the exact solution 5.2 by $0,03$ and $-0,37$. Also, the error in the corner (1,1) appears to be of the same magnitude as the error in the rest of the solution.

In summary, the Sigmoid activation function is the only optimizer that performs poorly. For all other activation functions, the three optimizers demonstrate good performance and architectures with 5 hidden layers consistently outperform other configurations. Additionally, the results show that only SGD optimizer produces approximated solution similar to the exact solution 5.2, while Adam and Nadam fail, because of the high differences between the maximum and minimum values of the models.

Table 5.4: Values of L^2 error, loss functional, and training time over 1,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Outflow Layer problem 5.1.2.

Optimizer	Loss functional	L^2 error	Training time (sec)
SGD	0,230783	0,032732	270
Adam	0,019591	0,030243	188
Nadam	0,01236	0,074862	181

After training for 1,000 epochs, the results for accuracy, loss functional values, and training time are presented in Table 5.4. Nadam achieves the lowest loss functional value of 0.01236 in the shortest time. However, SGD produces the highest L^2 error and loss functional values while requiring the longest training time. In contrast, Adam demonstrates the

best accuracy, despite having a relatively high loss functional value and a longer training time.

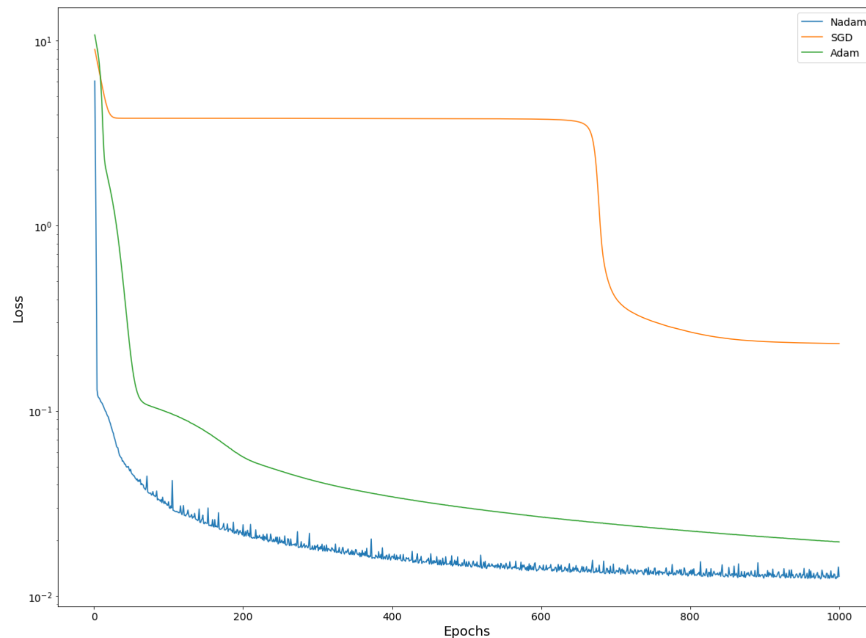


Figure 5.16: History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam over 1,000 epochs to the Example (5.2).

The Figure 5.16 illustrates the history of the loss functional for the three optimizers during the training of a hard-constrained PINN to approximate the solution of the Outflow Layer problem 5.2 over 1,000 epochs. Nadam was the fastest convergence, minimizing the loss functional since the initial epochs. In comparison, Adam required more time to achieve similar values. Conversely, SGD demonstrated significantly slower convergence, with its loss functional only approaching those of Adam and Nadam after approximately 700 epochs. This confirms the results found in the Table 5.4.

Table 5.5: Values of L^2 error, loss functional, and training time over 10,000 epochs for training hard-constrained PINN using SGD, Adam and Nadam to approximate the Outflow Layer problem 5.1.2.

Optimizer	Loss functional	L^2 error	Training time (sec)
SGD	0,15064	0,327266	3356
Adam	0,009638	0,077798	1110
Nadam	0,008739	1,682296	1145

Here's the corrected and refined version of your paragraph:

The optimal architectures identified previously are trained again for 10,000 epochs. The findings are summarized in Table 5.5. The values for the Adam and Nadam optimizers change significantly. Adam achieves the lowest L^2 error in the shortest time among the

three optimizers. On the other hand, SGD produces the highest loss functional while requiring the longest training time of 3356 seconds, and Nadam results in the highest L^2 error. Compared to the values at 1,000 epochs, the optimizers continue minimizing the loss functional but struggle to maintain accuracy. In other words, the L^2 error values are higher than they were previously.

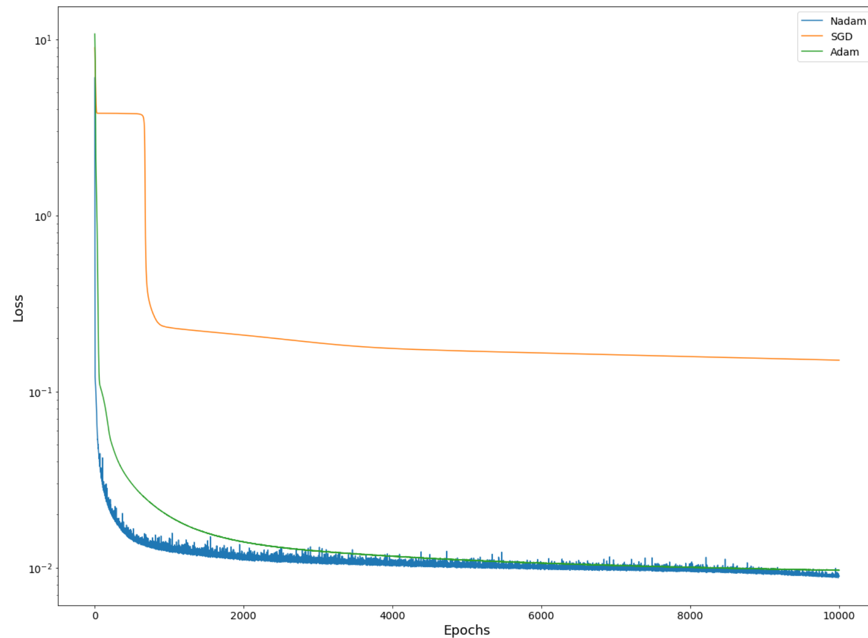
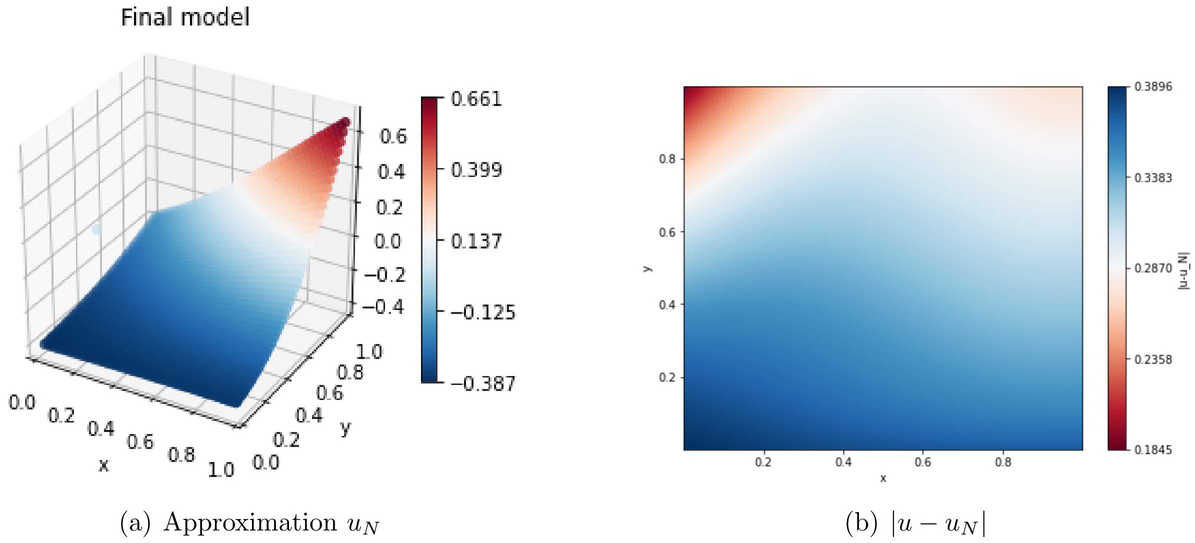


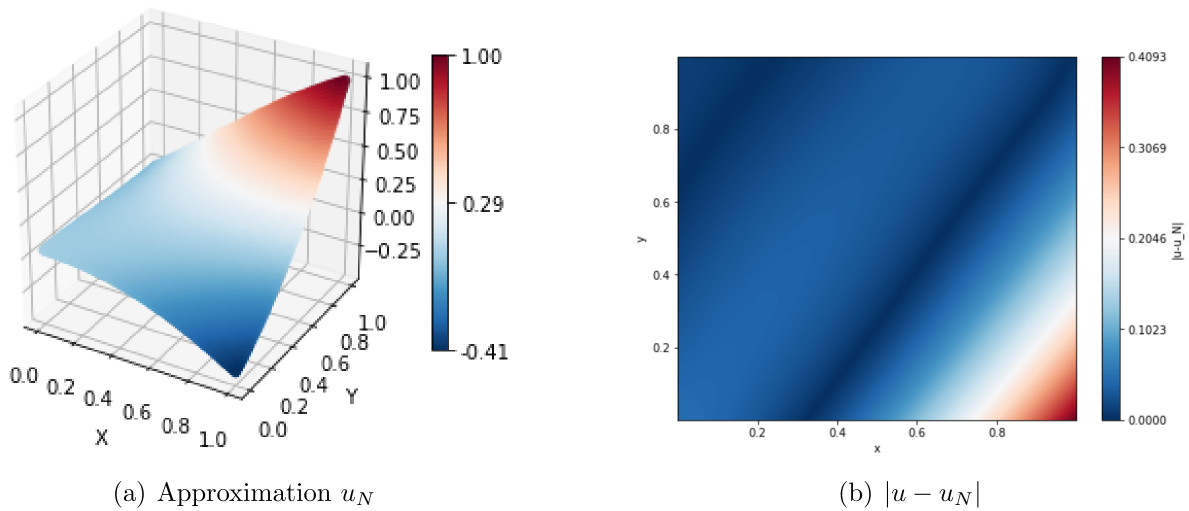
Figure 5.17: History of loss functional values for hard-constrained PINN approximation with SGD, Adam and Nadam over 10,000 epochs to the Example (5.2).

The Figure 5.17 depicted the decrease of the loss functional through the 10,000 epochs during the training of hard-constrained PINN using the three optimizers. The optimizers Adam and Nadam have approximately the same convergence, minimizing the loss functional starting from the initial epochs. In contrast, SGD showed significantly slower convergence. When the number of the epochs extended, the three optimizers continued minimizing the loss functional but they lost their accuracy significantly. This can be related to overfitting. However, Adam and Nadam proved that it is the most efficient optimizer in compare to SGD, which was the slowest optimizer.



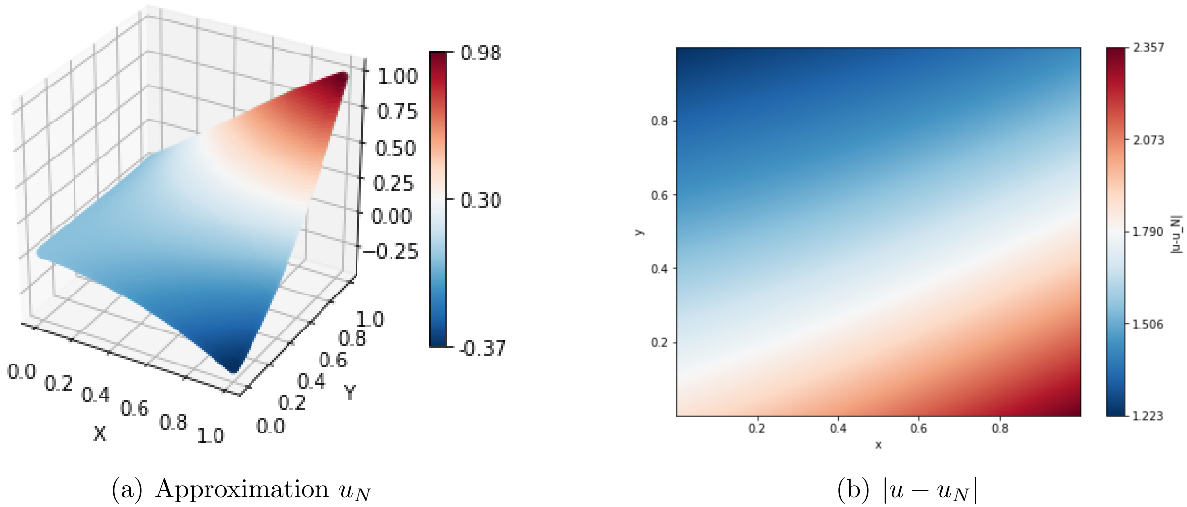
(a) Approximation u_N (b) $|u - u_N|$

Figure 5.18: Hard-constrained PINN approximated solution u_N and the point-wise error using SGD optimizer to Example 5.1.2 after training over 10.000 epochs.



(a) Approximation u_N (b) $|u - u_N|$

Figure 5.19: Hard-constrained PINN approximated solution u_N and the point-wise error using Adam optimizer to Example 5.1.2 after training over 10.000 epochs.



(a) Approximation u_N

(b) $|u - u_N|$

Figure 5.20: Hard-constrained PINN approximated solution u_N and the point-wise error using Nadam optimizer to Example 5.1.2 after training over 10.000 epochs.

The Figures5.18, 5.19 and 5.20 illustrate the approximation results and point-wise error using the three optimizers after training the hard-constrained PINN for 10.000 epochs. The findings show that none of the optimizers produce a similar approximation to the exact solution 5.2, based on the exceed values of the maximum and the minimum in the figures.

The comparison between short-term and long-term training for approximating the Outflow Layer example 5.1.2 reveals distinct strengths of the optimizers. Adam maintains its accuracy across both timeframes, while Nadam retains its efficiency. For short-term training, Nadam proves to be the fastest optimizer. However, over the longer term, Adam surpasses Nadam in speed, becoming the fastest optimizer while still delivering the best accuracy. These findings highlight the adaptability and reliability of Adam for extended training durations.

Chapter 6

Conclusion

This Chapter aims to summarize all the findings and the contribution of this thesis of comparing the optimizers stochastic gradient descent, Adam, Nadam and Broyden–Fletcher–Goldfarb –Shanno optimizer effectiveness and accuracy when solving convection-diffusion-reaction problems using physics-informed neural network.

6.1 Summary

This thesis investigates the influence and critical role of optimizers in PINNs by comparing their performance, accuracy, effectiveness and speed. The thesis is structured into four main chapters to build a comprehensive understanding of the problem and methodology. Chapter 2.2 offers a basic introduction of the fundamental concepts of machine learning, NN and its components and backpropagation. Chapter 3 delves into the optimizers SGD, Adam, Nadam and BFGS, providing detailed explanations of their algorithms and features. Chapter 4 presents the mathematical formulation of the convection-diffusion-reaction problems, along with a detailed discussion about PINN, its components and its variation hard-constrained PINNs.

The optimizers are tested against one type of convection-diffusion-reaction problem, benchmark problems with analytical solution: the Circular Interior Layer and the Outflow Layer problems. However, due to challenges encountered during the implementation of BFGS in the PINN framework, the analysis continues with only three optimizers, SGD, Adam and Nadam. Throughout the experimental analysis, significant observations are made about the behavior of the selected optimizers.

The results from approximating the Circular Interior Layer 5.2.2 demonstrate clear strengths and weaknesses of the optimizers. SGD proves to be the fastest optimizer across all epoch values. However, its accuracy and effectiveness lag with high L^2 error and loss functional values. On the other hand, Adam consistently achieves the best accuracy, regardless of the number of epochs. For short-term training, Nadam shows higher efficiency. In contrast, Adam proves , over a longer training period, to be the most efficient.

For the Outflow Layer example 5.1.2, extending the number of epochs leads to a loss of accuracy for all optimizers. However, for a lower number of epochs, Nadam shows superior efficiency and speed. When it comes to accuracy, Adam proves to be the best optimizer. As the number of epochs increases, Adam continues to excel in both accuracy and speed, while Nadam maintains its efficiency throughout the training process.

In conclusion, the Adam optimizer proved to be the best choice for the problems studied in this thesis. Adam provided a good balance between speed and accuracy, outperforming SGD in accuracy and closely matching Nadam's results. When SGD was the fastest, it struggled with accuracy and Nadam performed similarly to Adam, but didn't consistently do better. In general, Adam showed the most consistent and reliable performance among different scenarios, making it the preferred optimizer for PINNs.

6.2 Implications for future work

This study demonstrates the importance of selecting the appropriate optimizer for training PINNs, especially when the problem involves highly complex physical systems that require precise solutions. The results suggest that Adam is the optimal choice for applications demanding high accuracy.

For future work, it could extend this analysis by incorporating other optimizers, to compare their performance in hard-constrained PINN applications. As another area for exploration, one could explore ways to improve the speed of Adam or investigate hybrid approaches that combine the strengths of multiple optimizers for enhanced PINN training efficiency.

Bibliography

[AGK21] Jun Shi Aman Gupta, Rohan Ramanath and Sathiya Keerthi, *Adam vs. sgd: Closing the generalization gap on image classification*, 2021, pp. 1–7.

[Ajm23] Girish Ajmera, *Decoding activation functions: Navigating the landscape for image segmentation*, Dec 27,2023.

[Bab] Tushar Babbar, *Are your machine learning models making these common mistakes? learn how to avoid overfitting and underfitting.*

[Ben12] Yoshua Bengio, *Practical recommendations for gradient-based training of deep architectures*, CoRR **abs/1206.5533** (2012), 437–478.

[BPRS 4] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind, *Automatic differentiation in machine learning: a survey*, 2018, 1-4, pp. 1–20.

[Buc23] Noam M. Buckman, *The linear convection-diffusion equation in two dimensions*, 2023, pp. 1–6.

[Cas20] Emma Juliana Gachancipá Castelblanco, *How to choose an activation function?*, 2020.

[CdCV18] Rosangela Cintra and Haroldo Fraga de Campos Velho, *Data assimilation by artificial neural networks for an atmospheric general circulation model*, February 2018, p. 269.

[CMW+21] Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis, *Physics-informed neural networks (pinns) for fluid mechanics: A review*, 1–10.

[CSN+20] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl, *On empirical comparisons of optimizers for deep learning*, 2020, pp. 1–27.

[CZ04] E.K.P. Chong and S.H. Zak, *An introduction to optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimi, Wiley, 2004.

[DFMa] Volker John Derk Frerichs-Mihov, Linus Henning, *On loss functionals for physics-informed neural networks for steady-state convection-dominated convection-diffusion problems*, pp. 1–22.

[DFMb] Volker John Derk Frerichs-Mihov, Marwa Zainelabdeen, *On collocation points for physics-informed neural networks applied to convection-dominated convection-diffusion problems*, pp. 1–10.

[Doz16] Timothy Dozat, *Incorporating Nesterov Momentum into Adam*, Proceedings of the 4th International Conference on Learning Representations, 2016, pp. 1–4.

[DRW86] Geoffrey Hintont David Rumelhart and Ronald Williams, *Learning representations by back-propagating errors*, 1986, pp. 533–536.

[emp] empty, *Conservation law.*

[FM23] Derk Frerichs-Mihov, *On slope limiting and deep learning techniques for the numerical solution to convection-dominated convection-diffusion problems dissertation zur erlangung des grades eines doktors der naturwissenschaften (dr. rer. nat.) am fachbereich mathematik und informatik*, Master's thesis, FU Berlin, 2023.

[GB10] Xavier Glorot and Yoshua Bengio, *Understanding the difficulty of training deep feedforward neural networks*, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Yee Whye Teh and Mike Titterington, eds.), Proceedings of Machine Learning Research, vol. 9, PMLR, 13–15 May 2010, pp. 249–256.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, `http://www.deeplearningbook.org`, 96-114.

[Gé19] Aurélien Géron, *Hands-on machine learning with scikit-learn, keras, and tensorflow, 2nd edition*, O'Reilly Media, Incorporated, 2019.

[HH18] Catherine F. Higham and Desmond J. Higham, *Deep learning: An introduction for applied mathematicians*, 2018, pp. 1–15.

[Joh] Volker John, *Chapter 1: Convection-diffusion-reaction equations and maximum principles*, pp. 1–7.

[JVMJT97] L. John V. Maubach J.M. Tobiska, *Nonconforming streamline-diffusion-finite-element-methods for convection-diffusion problems. numerische mathematik, 78(2)*, 1997, pp. 165–188.

[KB17] Diederik P. Kingma and Jimmy Ba, *Adam: A method for stochastic optimization*, 2017, pp. 1–15.

[KKL+20] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang, *Physics-informed machine learning*, Nature Reviews Physics **3** (2021, 1-20), no. 6, 422–440.

[Kri07] David Kriesel, *A brief introduction to neural networks*, 2007, pp. 8–12.

[LMMK21] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis, *Deepxde: A deep learning library for solving differential equations*, SIAM Review **63** (2021), no. 1, 208–228.

[LPY+21] Lu Lu, Raphael Pestourie, Wenjie Yao, Zhicheng Wang, Francesc Verdugo, and Steven G. Johnson, *Physics-informed neural networks with hard constraints for inverse design*, 2021, pp. 1–6.

[Mar21] Stefano Markidis, *The old and the new: Can physics-informed deep-learning replace traditional linear solvers?*, Frontiers in Big Data **4** (2021), 1–6.

[Mat24] Marina Matthaiou, *Solutions to elliptic boundary value problems based on machine learning techniques that satisfy the discrete maximum principle*, Master's thesis, FU Berlin, 2024.

[Nie19] Michael Nielsen, *How the backpropagation algorithm works, chapter 2*, 2019.

[NIGM18] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall, *Activation functions: Comparison of trends in practice and research for deep learning*, 2018, pp. 1–14.

[NW06] Jorge Nocedal and Stephen J. Wright, *Numerical optimization*, 2e ed., Springer, New York, NY, USA, 2006.

[PL23] Yan Pan and Yuanzhi Li, *Toward understanding why adam converges faster than sgd for transformers*, 2023, pp. 1–37.

[PSN23] Michelle Tindall Prakhar Sharma, Llion Evans and Perumal Nithiarasu, *Hyperparameter selection for physics-informed neural networks (pinns) – application to discontinuous heat conduction problems*, Numerical Heat Transfer, Part B: Fundamentals (2023), 1–15.

[RPK17] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis, *Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations*, 2017, pp. 1–7.

[Rud17] Sebastian Ruder, *An overview of gradient descent optimization algorithms*, 2017, pp. 1–12.

[RZS20] Matroid Reza Zadeh and Stanford, *Introduction to optimization for machine learning*, 2020, p. 6.

[Sal23] Sadaf Saleem, *Neural networks in 10mins. simply explained!*, 2023.

[Sod22] Pontus Soderstrom, *Physics-informed neural networks for liquid chromatography*, Master's thesis, Umea University, 2022.

[TWBB22] John Taylor, Wenyi Wang, Biswajit Bala, and Tomasz Bednarz, *Optimizing the optimizer for data driven deep neural networks and physics informed neural networks*, 2022, pp. 1–23.

[VJ24] Marwa Zainelabdeen Volker John, Marina Matthaiou, *Bound-preserving pinns for steady-state convection-diffusion-reaction problems*, 2024, pp. 1–19.

[ZLN+19] Guodong Zhang, Lala Li, Zachary Nado, James Martens, Sushant Sachdeva, George E. Dahl, Christopher J. Shallue, and Roger Grosse, *Which algorithmic choices matter at which batch sizes? insights from a noisy quadratic model*, 2019, pp. 1–21.