FINDING OPTIMAL HYPERPARAMETERS OF FEEDFORWARD NEURAL NETWORKS FOR SOLVING 2D LAPLACE'S EQUATION USING A GENETIC ALGORITHM

CHALOEMRAT BOONTHANAWAT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF MASTER OF SCIENCE (PHYSICS)
FACULTY OF GRADUATE STUDIES
MAHIDOL UNIVERSITY
2021

Thesis
entitled

FINDING OPTIMAL HYPERPARAMETERS OF FEEDFORWARD NEURAL NETWORKS FOR SOLVING 2D LAPLACE'S EQUATION USING A GENETIC ALGORITHM

.....
Mr. Chaloemrat Boonthanawat
Candidate

.....
Chaiwoot Boonyasiriwat,
Ph.D. (Scientific Computing)
Major advisor

.....
Tanapon Tantisripreecha,
Ph.D. (Computer Science)
Co-advisor

.....
Prof. Patcharee Lertrit,
M.D., Ph.D. (Biochemistry)
Dean
Faculty of Graduate Studies
Mahidol University

.....
Assoc. Prof. Kittiwit Matan,
Ph.D. (Physics)
Program Director
Master of Science Program in Physics
(International Program)
Faculty of Science
Mahidol University

Thesis
entitled

FINDING OPTIMAL HYPERPARAMETERS OF FEEDFORWARD NEURAL NETWORKS FOR SOLVING 2D LAPLACE'S EQUATION USING A GENETIC ALGORITHM

was submitted to the Faculty of Graduate Studies, Mahidol University
for the degree of Master of Science (Physics)
on
March 23, 2021

.....
Mr. Chaloemrat Boonthanawat
Candidate

.....
Asst. Prof. Kitiporn Plaimas,
Ph.D. (Applied Mathematics)
Chair

.....
Chaiwoot Boonyasiriwat,
Ph.D. (Scientific Computing)
Member

.....
Tanapon Tantisripreecha,
Ph.D. (Computer Science)
Member

.....
Prof. Patcharee Lertrit,
M.D., Ph.D. (Biochemistry)
Dean
Faculty of Graduate Studies
Mahidol University

.....
Assoc. Prof. Palangpon Kongsaeree,
Ph.D. (Chemistry)
Dean
Faculty of Science
Mahidol University

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my research advisor, Dr. Chaiwoot Boonyasiriwat, for accepting me into his group and for his constant scientific support during my study. I get to carry out research in the field of neural networks in the environment which encourages the development of my research ideas. Moreover, I would like to thank him for insightful discussion and giving me many opportunities.

I would like to thank MAHIDOL UNIVERSITY CENTER FOR SCIENTIFIC COMPUTING (MCSC) for support. I also would like to thank all the staff members of the Department of Physics, Faculty of Science, Mahidol University, for all kindly support.

A special thank goes to my friends, in and out of the lab for their support when I was in trouble and had a bad time. It has been a pleasure to know them and work with them. I also enjoyed sharing feelings and moments with them all. Thank you so much.

Finally, I would like to express my gratitude to my father and my mother for their unconditional love and continued support through my experience. I also appreciate their abundant patience. I could not have completed my studies without them.

Chaloemrat Boonthanawat

FINDING OPTIMAL HYPERPARAMETERS OF FEEDFORWARD NEURAL NETWORKS FOR SOLVING 2D LAPLACE'S EQUATION USING A GENETIC ALGORITHM

CHALOEMRAT BOONTHANAWAT 5836047 SCPY/M

M.Sc. (PHYSICS)

THESIS ADVISORY COMMITTEE : CHAIWOOT BOONYASIRIWAT, Ph.D., TANAPON TANTISRIPREECHA, Ph.D.

ABSTRACT

In this work, feedforward neural networks were used to solve 2D Laplace's equation on rectangular and irregular domains. Optimal values of weights and biases of a network were recursively computed during the network training by minimizing a cost function using data at collocation points to approximate the true solution. The performance of the network largely depends on network architecture and model capability. In this work, an optimal set of hyperparameters was searched on various measures of relative errors. The genetic algorithm was used to find the optimal activation function, optimization algorithm, and weight initialization. In addition, it also searched for the optimal number of hidden layers when the total number of parameters is fixed. The optimal total number of parameters for a specific number of hidden layers was also searched. Numerical results showed that an optimal set of hyperparameters that are consistent across many values of relative errors could be successfully obtained.

KEY WORDS : NEURAL NETWORK/ GENETIC ALGORITHM/ OPTIMAL HYPERPARAMETER/ DIFFERENTIAL EQUATION

121 pages

CONTENTS

CONTENTS (cont.)

CONTENTS (cont.)

LIST OF TABLES

LIST OF TABLES (cont.)

LIST OF TABLES (cont.)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES (cont.)

LIST OF FIGURES (cont.)

Figure **Page**

LIST OF FIGURES (cont.)

CHAPTER I
INTRODUCTION

Many real-world problems in sciences, engineering, and finance are modeled using ordinary differential equations (ODEs) and/or partial differential equations (PDEs). Acquiring the solution to the governing differential equations is important for the understanding of the problem. In real-world problems, most of the ODEs and PDEs cannot be solved analytically, and various numerical techniques are needed to approximate the solutions. Traditional numerical methods for solving differential equations include, but not limited to, finite difference (FDM), finite element (FEM), finite volume (FVM), and spectral method. These methods are mesh-based methods which mean they approximate the solution on discrete grid points inside the computational domain [2, 3]. The solution outside of the grid points must be interpolated from the nearby points. Different numerical techniques would have different strengths and weaknesses. One of the advantages of traditional methods is that they are highly efficient for low dimensional problems. On the contrary, one of the disadvantages is that to increase the accuracy of the approximation, we must use a finer grid requiring a greater computational resource.

Neural network method is one of the approaches that can be used to approximate the solution of differential equations. It is a meshfree method which means the value of the approximation can be found in the entire computation domain. The advantage of neural network method is that it can approximate the solution in a close analytical form, and infinitely differentiable. The method also requires a small number of parameters which help reduce the computational resource when compared to the traditional methods. Neural network method is also suitable for high dimensional problems (e.g. pricing of financial derivatives with dimension ≈ 100) or the problem with complex

domain [1].

The computational model of neural networks was first proposed by Warren McCulloch and Walter Pitts [4]. In the paper, they proposed a theoretical model of the nervous system using a collection of neurons, and each neuron is connected to its neighbors. With the model, the network will be able to accept input signals from the outside and can also send out the signals if the value of the signal exceeds a specific threshold value. Following this work, in 1958, Rosenblatt [5] proposed an idea of perceptron which can be used for pattern recognition. Perceptron is the simplest neural network architecture with only a single layer, and it uses step function as activation function. Perceptron can only distinguish problems with linearly separable classes. After Rosenblatt's work, the research of neural networks became stagnant due to the inability of perceptrons to distinguish a simple Boolean function (XOR), and the learning algorithm created by Rosenblatt cannot be extended to a neural network with multiple layers.

In 1986, Rumelhart *et al.* [6] proposed a backpropagation algorithm which enables the training of a multilayer neural network. In the 1980s and 1990s, different kinds of neural networks were proposed (e.g. autoencoder, recurrent neural networks (RNNs), long short-term memory recurrent neural networks (LSTMs), and Boltzmann machines). In 1989, LeCun *et al.* [7] proposed convolutional neural network (CNN) which is a type of deep neural network that works well for image recognition problems. Despite the success, the training of a deep neural network was still widely considered impossible. The main reason is due to the vanishing and exploding gradient problem.

In the last decade, a renewed interest of a deep neural network came from a published paper by Geoffrey Hinton in 2006 [8]. The paper showed a deep neural network that can recognize handwritten digits with state-of-the-art precision. There are a few reasons why training a deep neural network becomes successful. The first reason is the new activation function such as a rectified linear unit activation (ReLU) which helps reduce the vanishing and exploding gradient problem. The second reason is the new way of initializing the weight parameters that outperform the simple random weight

initialization. The third reason is the greater amount of training data due to the rise of the internet. The fourth reason is the advancement of computing resources, especially GPUs, that made it cheaper and faster for researchers to train deep neural networks.

For the application of neural network methods on solving differential equations, different network architectures were proposed such as cellular neural network [9, 10], finite element neural network (FENN) [11], wavelet neural network [12], radial basis function networks (RBFNs) [13, 14, 15], Hopfield neural network [16, 17, 18], and feedforward neural network [19, 20]. A comprehensive overview of the neural network method for differential equations can be found in [21]. The ability of a neural network to solve differential equations rely upon the function approximation capability of a neural network. A neural network has been known to be a universal approximator since the 1990s [22, 23]. The theorem [24] states that "a feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous functions, under mild assumptions on the activation function". This means that a simple neural network can represent a wide range of functions when given appropriate parameters.

For the feedforward neural network approach, an early method was to use a feedforward neural network with B_1-splines as basis functions [19, 20]. The approximation can be obtained by solving a system of linear or nonlinear equations in order to find the coefficients of splines.

In 1998, Lagaris *et al.* proposed the feedforward neural network to approximate the solution of both ODEs and PDEs on rectangular domains [25]. In their approach, a trial solution is explicitly constructed to satisfy the boundary conditions. Using the trial function, the problem has been reduced to an unconstrained optimization problem. Lagaris *et al.* adopted a collocation method which assumes a discretization of the domain and its boundary into a set of points. This set of points will be used as training data for neural networks, and the cost function is defined using the differential equation. In 2000, Lagaris *et al.* [26] used a combination of feedforward and radial basis function neural networks to solve ODEs and PDEs on irregular domains, but require the

solving system of linear equations. In 2009, McFall and Mahan [27] removed the radial basis function neural network by replacing it with length factors that are computed using thin plate splines. The disadvantage of the method is that it also involves the solution of many linear systems which add extra complexity to the problem.

In 2018, Berg and Nyström [1] using the same trial solution approach, proposed a method for solving stationary PDEs on irregular domains using deep feedforward neural networks without the need to explicitly satisfy the boundary condition. They used pre-computed low capacity neural networks to satisfy inexact boundary conditions. Alternatively, Sirignano and Spiliopoulos [28] proposed to solve high-dimensional time dependent PDEs with a deep neural network which is called the deep Galerkin method. In their approach, the initial and boundary conditions are enforced on the cost function instead on the trial solution. They used long short-term memory neural networks (LSTMs) instead of feedforward neural networks, but the method still can be applied. In 2019, Liyao *et al.* [29] proposed a method based on a feedforward neural network called mixed residual method. The method rewritten a given PDE into a first-order system. A neural network is then used to approximate the solution and its high-order derivatives. The cost function is constructed from the first-order system. The initial and boundary condition can be satisfied by using the approach from Lagaris *et al.* or Sirignano and Spiliopoulos.

In the work of Lagaris *et al.*, they use 1 hidden layer and 10 neurons per hidden layer. The activation function is a sigmoid function, and the optimization algorithm is the BFGS method. There is no information on the weight initialization algorithm. For McFall and Mahan, they use 1 hidden layer and $5 - 50$ neurons per hidden layer. The activation function is a sigmoid function, and the optimization algorithm is Levenberg–Marquardt gradient descent method. There is no information on weight initialization. For Berg and Nyström, they use 1 hidden layer and 20 neurons per hidden layer for low capacity neural networks, and 5 hidden layers and 10 neurons per hidden layer for the main neural network. The activation function is a sigmoid function, and the optimiza-

Table 1.1 Summary of the set of hyperparameters in the former works.

	Lagaris	McFall	Berg	Sirignano	Liyao
Number of hidden layers	1	1	1-5	4	4
Number of neurons per layer	10	5-50	10-20	50	5-20
Activation function	Sigmoid	Sigmoid	Sigmoid	Tanh	ReLU
Optimization algorithm	BFGS	LM	BFGS and SGD	Adam	Adam
Weight initialization	-	-	-	Glorot	-

tion algorithms are the BFGS method and SGD. There is also no information on weight initialization either. For Sirignano and Spiliopoulos, they use 4 hidden layers and 50 neurons per hidden layer. The activation function is a hyperbolic tangent function, and the optimization algorithm is the adaptive moment estimation (Adam). The weight initialization algorithm is Glorot initialization. For Liyao, they use 4 hidden layers and 5-20 neurons per hidden layer. The activation function is ReLU, and the optimization algorithm is the adaptive moment estimation (Adam). There is no information on weight initialization. The summarize of the set of hyperparameters of former works is shown in Table 1.1. In the former works, they only suggest a set of hyperparameters through observation or by using a trial and error approach, but none of them has ever presented a means to find a set of hyperparameters systematically. Different sets of hyperparameters would greatly affect the performance of the neural network

In this work, we use a genetic algorithm which is a searching technique based on the principle of natural selection. A genetic algorithm has been introduced into the work of neural networks at roughly three different levels [30]. The first level is to directly find the values of the weights and biases of neural networks. The second one is to find the optimal neural network architecture. The third one is to find the learning rules. An early research of finding an optimal neural network architecture starts with constructive and destructive algorithms [31, 32, 33]. Construction algorithms will start with a simple network, and then build up to a complex one. On the contrary, destructive algorithms

are the opposite. This approach was later pointed out by Angeline *et al.* [34] that both algorithms can easily be trapped at the local optimum.

A genetic algorithm is a better alternative to constructive and destructive algorithms due to many characteristics of the surface of the search space [30]. The first characteristic is that the surface is large due to the number of possible neurons and their connections. The second one is that the surface is non-differentiable due to the discrete nature of the parameters of the neural network. The third one is that there are many local optimums on the surface. Considerable research on finding neural network architectures using a genetics or evolutionary algorithm has been done in the 1990s focusing on topological structures of neural networks [35, 36, 37]. Recently, with the advance of neural networks, the use of genetic algorithms has been applied to deep learning [38]. In [39], an evolutionary algorithm is applied to convolutional neural networks for cancer diagnosis application.

In this work, the problem is divided into three parts. The first part is to compare different neural network methods both for exact and inexact boundary conditions. The methods we compared in this work are the Lagaris, deep Galerkin, and mix residual methods. The second part is to find the optimal activation function, optimization algorithm, and weight initialization using a genetic algorithm. The third part is to find the optimal value of the number of hidden layers when the total number of parameters of the network is fixed. We will also find the optimal total number of parameters when the number of hidden layers is fixed. The total number of parameters is the total number of learnable weights and biases of the neural network. We cannot directly optimize the total number of parameters using a genetic algorithm because model capability is generally dependent on the total number of parameters.

This thesis is organized as follows. Chapter 2 explains the details of a neural network and its mathematical description. Later, we explain the learning algorithm used for deep neural networks, and the set of hyperparameters of a feedforward neural network. In Chapter 3, different neural network methods for solving differential equa-

tions, the stopping criteria, and performance measurement are explained in detail. A genetic algorithm in the context of the neural network is also explained. In Chapter 4, the problem setup for our work and numerical results of this work are presented. Finally in Chapter 5, a conclusion of our work is presented.

CHAPTER II

FEEDFORWARD NEURAL NETWORK

A fully connected feedforward neural network is one of the simplest types of neural network. All neurons in a layer are connected to all neurons in the previous layer except at the input layer. One of the characteristics of a feedforward neural network is that the connections between neurons do not form a cycle. The information from the input layer can only travel forward in the neural network. Figure 2.1 shows a representation of a fully connected feedforward neural network.

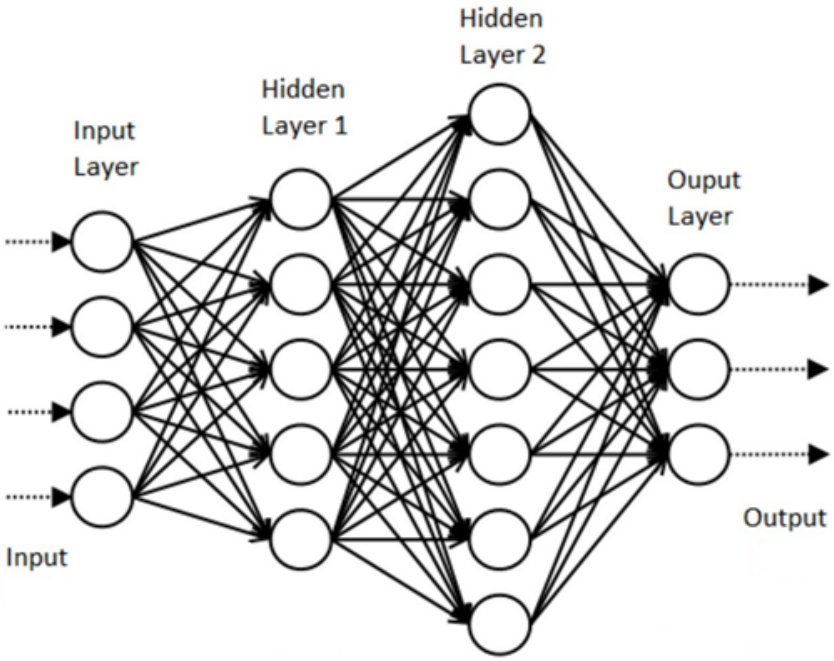

Figure 2.1 Representation of a fully connected feedforward neural network.

2.1 Components of a Feedforward Neural Network

A feedforward neural network is composed of many important components. The list of components is,

- **Neuron**. It is a basic unit of computation of a neural network which is represented by a circle in Figure 2.1. Neuron receives input from the previous neuron and computes an output.
- **Input layer**. It is the first layer of a network. Input layer receives data from the outside to a network. No computation is performed in any of the input neuron. Input layer only passes on the data to the hidden neurons on the next layer.
- **Hidden layer**. It has no direct connection with the outside. It performs computations and transfers data to the next layer. The depth of a neural network is determined by the number of hidden layers.
- **Output layer**. It is the last layer of a neural network. Output layer is responsible for computations and transferring data to the outside. Normally, the output layer will be passed through the cost function for training of the neural network.
- **Weights and biases**. They represent the parameters of a neural network, and can be learned to make the network perform better. Weight is represented by an arrow between layers shown in Figure 2.1. Collectively, weights on each layer are represented by matrices and the size will depend on the number of neurons. Bias is associated with each neuron (except for neurons at the input layer). Collectively on each layer, biases are represented by column vector.
- **Activation function**. It introduces nonlinearity into the output of a neuron because most real world data is nonlinear and we want the neural network to learn nonlinear representation. Activation function will follow immediately after the output of each neuron (except for neurons at the input layer).

2.2 Mathematical Descriptions

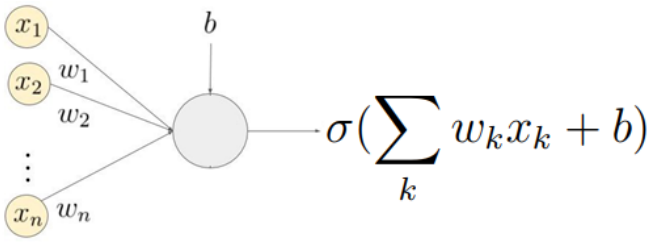

Figure 2.2 Basic computation of neuron.

The basic computation of a neuron is shown in Figure 2.2. First, the value of outputs from the previous layer $(x_1, x_2, \ldots, x_n)$ are multiplied by associated weight $(w_1, w_2, \ldots, w_n)$. Each value is then combined with the bias of the neuron. Then, the total output value will be passed through the activation function,

$$y = \sigma(\sum_k w_k x_k + b) \tag{2.1}$$

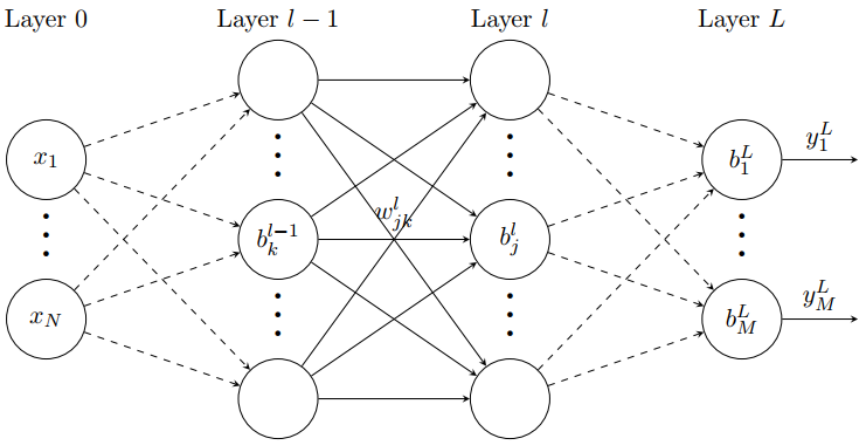

Figure 2.3 Fully connected feedforward neural network [1].

In this section, we use the notation from the work of Berg [1]. A fully connected feedforward neural network consisting of $L-1$ hidden layers is shown in Figure 2.3. We denote weight and bias as,

$$w_{jk}^l, b_j^l \tag{2.2}$$

when w^l_{jk} denote the weight between neuron j in layer l and neuron k in layer $l-1$, and b^l_j denote the bias of neuron j in layer l. The output of neuron j in layer l will be denoted by y^l_j. With above definitions, the output of neuron j in layer l can be written as,

$$y^l_j = \sigma(\sum_k w^l_{jk} y^{l-1}_k + b^l_j) \tag{2.3}$$

$$y^l_j = \sigma(z^l_j) \tag{2.4}$$

where $z^l_j = \sum_k w^l_{jk} y^{l-1}_k + b^l_j$ is the input of a neuron. By definition, a neuron j of the input layer is,

$$y^0_j = x_j \tag{2.5}$$

By collecting every neurons in the layer, we can rewrite the equation in matrix form,

$$y^l = \sigma(W^l y^{l-1} + b^l) \tag{2.6}$$

$$y^l = \sigma(z^l) \tag{2.7}$$

With all the definitions, we can write the full equation of the feedforward neural network for all layers in the network. The feedforward neural network with output y^L, given the

input x, is given by,

$$y^L = \sigma(W^L y^{L-1} + b^L) \tag{2.8}$$

$$y^{L-1} = \sigma(W^{L-1} y^{L-2} + b^{L-1}) \tag{2.9}$$

$$\vdots$$

$$y^2 = \sigma(W^2 y^1 + b^2) \tag{2.10}$$

$$y^1 = \sigma(W^1 y^0 + b^1) \tag{2.11}$$

$$y^0 = x \tag{2.12}$$

2.3 Neural Network Training

We train a neural network by finding the appropriate values of weights and biases such that the output from the network will closely approximate the desired solution. Finding weights and biases is a type of optimization problem, and we need to define what the cost function is. In order to train the network, we also need training datasets,

$$s = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \tag{2.13}$$

where x is the input data, y is desired or correct output, and the total number of data is N. Mathematically, the optimization problem can be written as,

$$w^*, b^* = \underset{\vec{w},\vec{b}}{\operatorname{argmin}} \frac{1}{N} \sum_i^N C(y_i, y^L(x_i; \vec{w}, \vec{b})) \tag{2.14}$$

when y^L is the output of the neural network with $L-1$ hidden layers.

By using a neural network to solve differential equations, we will not have the correct output (the solution of the equation). Therefore, the training datasets will

only compose of the input,

$$s = \{x_1, x_2, \ldots, x_N\} \tag{2.15}$$

Then, the optimization problem will be written as,

$$w^*, b^* = \underset{\vec{w},\vec{b}}{\text{argmin}} \frac{1}{N} \sum_i^N C(y^L(x_i; \vec{w}, \vec{b})) \tag{2.16}$$

On the optimization problem is set. We can use gradient based or gradient free methods for optimization. In this thesis, we focus on gradient based methods. Gradient descent method is one of the most commonly used optimization algorithms for the neural network. Gradient descent is an iterative optimization algorithm for finding a local minimum of a differentiable function. The standard gradient descent method use all the training data to optimize the cost function,

$$w = w - \eta \frac{\partial}{\partial w} \frac{1}{N} \sum_i^N C_i \tag{2.17}$$

$$b = b - \eta \frac{\partial}{\partial b} \frac{1}{N} \sum_i^N C_i \tag{2.18}$$

when η is learning rate, C_i is the cost function of the i$^{\text{th}}$ training data.

The problem with standard gradient descent methods is that there are a lot of training data. We need to use all of the training data to complete one iteration of gradient descent, and it has to be done for every iteration until the local minimum is reached. Therefore, it becomes computationally very expensive to perform. Alternatively, the mini-batch gradient descent can be used instead [40]. The method estimates the gradient of cost function using randomly selected small samples of data. The total number of samples is called batch size. Since only a small samples is chosen at random for each iteration, the gradient of cost function will become noisier comparing to the traditional gradient descent,

$$\frac{1}{N}\sum_{i}^{N} C_i \approx \frac{1}{n}\sum_{i}^{n} C_i \tag{2.19}$$

when n is the batch size which is usually less than the full training datasets ($n << N$).

2.4 Backpropagation

In order to calculate the gradient of cost function with respect to weights and biases, the backpropagation algorithm is commonly used. Backpropagation algorithms are a family of methods used to efficiently calculate the partial derivative of gradient descent using the chain rule. It was originally introduced in the 1970s but its importance isn't fully appreciated until the work by Rumelhart *et al.* in 1986 [6]. Today, the backpropagation algorithm is the core algorithm of the neural network. There are two assumptions about the cost function required for the backpropagation.

1. The cost function can be written as an average,

$$C = \frac{1}{N}\sum_{i}^{N} C_i \tag{2.20}$$

   The requirement for this assumption is needed because what backpropagation actually lets us do is to compute the partial derivative for single training data.

2. The cost function is a function of the output of the neural network,

$$C = C(y^L(x; \vec{w}, \vec{b})) \tag{2.21}$$

The main idea of backpropagation is about understanding how the change of weight and bias affect the change of cost function, $\frac{\partial C_i}{\partial w^l_{jk}}$, $\frac{\partial C_i}{\partial b^l_j}$. To explain the backpropa-

gation, we need to introduce an intermediate quantity,

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{2.22}$$

The quantity is called the error in the j neuron in the l layer. The backpropagation algorithm will give us a procedure to compute the error δ_j^l. Then, the gradient of the cost function with respect to weights and biases will be calculated using the error value. There are four fundamental equations behind backpropagation,

1. An equation for the error in the output layer.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{2.23}$$

$$= \sum_k \frac{\partial C}{\partial y_k^L} \frac{\partial y_k^L}{\partial z_j^L} \tag{2.24}$$

$$= \frac{\partial C}{\partial y_j^L} \frac{\partial y_j^L}{\partial z_j^L} \tag{2.25}$$

$$= \frac{\partial C}{\partial \sigma_j^L} \sigma^{'}(z_j^L) \tag{2.26}$$

when $\sigma_j^L = \sigma(z_j^L)$. The first term of Equation 2.26 measures how fast the cost function is changing as a function of the output activation. The second term of Equation 2.26 measures how fast the activation function is changing at z_j^L. It is easy to rewrite the equation in a matrix-based form,

$$\delta^L = \nabla_{\sigma^L} C \odot \sigma^{'}(z^L) \tag{2.27}$$

2. An equation for the error δ^l in terms of the error in the next layer δ^{l+1}.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{2.28}$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{2.29}$$

$$= \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{2.30}$$

Note that $z_k^{l+1} = \sum_m w_{km}^{l+1} y_m^l(z_m^l) + b_k^{l+1}$, when differentiating we obtained,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \tag{2.31}$$

Substituting back, we then obtain,

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{2.32}$$

We can think intuitively about the equation as moving the error backward through the network. It is easy to rewrite the equation in a matrix-based form,

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.33}$$

3. An equation for the rate of change of the cost function with respect to any bias in the network.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} \tag{2.34}$$

$$= \delta_j^l \frac{\partial}{\partial b_j^l} (\sum_m w_{jm}^l y_m^{l-1} + b_j^l) \tag{2.35}$$

$$= \delta_j^l \tag{2.36}$$

The result shows that δ_j^l is exactly equal to the gradient of bias.

4. An equation for the rate of change of the cost function with respect to any weight in the network.

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} \tag{2.37}$$

$$= \delta^l_j \frac{\partial}{\partial w^l_{jk}}(\sum_m w^l_{jm} y^{l-1}_m + b^l_j) \tag{2.38}$$

$$= \delta^l_j y^{l-1}_k \tag{2.39}$$

2.4.1 Backpropagation Algorithm

With the four fundamental equations, there are five steps for the backpropagation algorithm,

1. Set the corresponding input data for the input layer.

$$y^0 = x \tag{2.40}$$

2. For each layer, $l = 1, 2, 3, \ldots, L$, we computed,

$$z^l = W^l y^{l-1} + b^l \tag{2.41}$$

$$y^l = \sigma(z^l) \tag{2.42}$$

3. Compute the error vector in the output layer.

$$\delta^L = \nabla_{\sigma^L} C \odot \sigma^{'}(z^L) \tag{2.43}$$

4. For each layer, $l = L-1, L-2, \ldots, 1$, we computed,

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(z^l) \tag{2.44}$$

5. Calculate the gradient of cost function with respect to weight and bias.

$$\frac{\partial C}{\partial w_{jk}^{l}} = y_k^{l-1}\delta_j^l \tag{2.45}$$

$$\frac{\partial C}{\partial b_{j}^{l}} = \delta_j^l \tag{2.46}$$

2.5 The Set of Hyperparameters of Feedforward Neural Network

Hyperparameters are variables in which values are set before training a neural network. Choosing hyperparameters can be seen as a type of model selection. The set of hyperparameters of interest in this work are activation function, optimization algorithm, weight initialization algorithm, learning rate, number of hidden layers, and number of neurons per layer. Number of hidden layers together with number of neurons per layer will determine the total number of parameters of a neural network which is roughly corresponding to the capability of the network.

2.5.1 Activation Function

In this section, we will look into different types of activation function, their mathematical description and their graphs. Activation function plays an important role in the performance of the neural network. The performance can significantly be improved by choosing the right function.

Linear Function

A linear function and its derivative is given by,

$$\sigma(z) = z \tag{2.47}$$

$$\sigma^{'}(z) = 1 \tag{2.48}$$

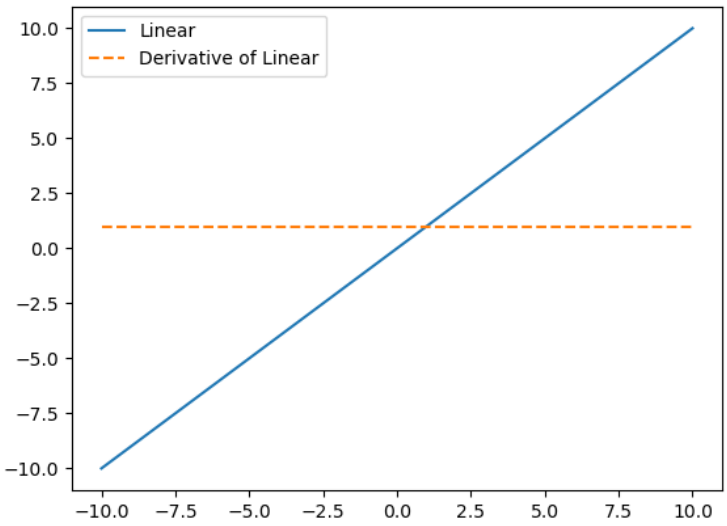

Figure 2.4 Linear function and its derivative

Figure 2.4 demonstrates the graph of a linear function and its derivative. A linear function is not normally used in the hidden layer. If it is used, the neural network model will become linear. The linear model will perform poorly when the data is non-linear. In this work, a linear function is used on the output layer in order to make the model be able to approximate the real function.

Sigmoid Function

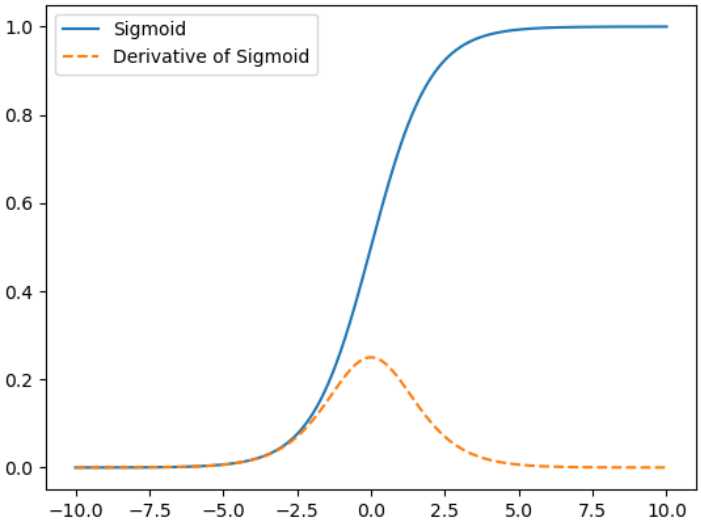

Figure 2.5 Sigmoid function and its derivative

A sigmoid function and its derivative is given by,

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.49}$$

$$\sigma^{'}(z) = \sigma(z)(1 - \sigma(z)) \tag{2.50}$$

Figure 2.5 demonstrates the graph of a sigmoid function and its derivative. In the literature, the function is sometimes referred to as a logistic function. A sigmoid function is a nonlinear function that maps any real value to $[0, 1]$. In particular, a large number will become 0 and a large positive number will become 1. The gradient value is peak at around -3 to 3, and becomes flatter at the tails. The main advantage of a sigmoid function is that it is easy to understand mathematically. It also has a nice probability interpretation. For the disadvantage, the gradient will tend to approach zero when using backpropagation. Moreover, a sigmoid function is sensitive to the weight initialization algorithm.

Hyperbolic Tangent Function (Tanh)

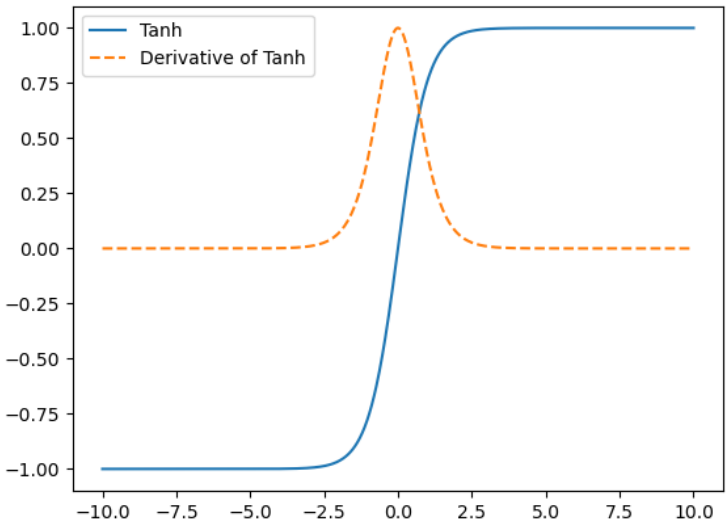

Figure 2.6 Hyperbolic tangent function and its derivative

A hyperbolic tangent function and its derivative is given by,

$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.51}$$

$$\sigma^{'}(z) = 1 - \sigma^2(z) \tag{2.52}$$

Figure 2.6 demonstrates the graph of a hyperbolic tangent function and its derivative. A hyperbolic tangent function is very similar to a sigmoid function. The difference is that the function is zero centered around the origin, and it maps any real value to $[-1, 1]$. The gradient of a hyperbolic tangent function is steeper when compared to a sigmoid function, and becomes the preferred activation function. Despite the advantage, a hyperbolic tangent function still suffers from vanishing gradient problems which cause the gradient to become zero as the depth of a neural network becomes larger.

ReLU Function

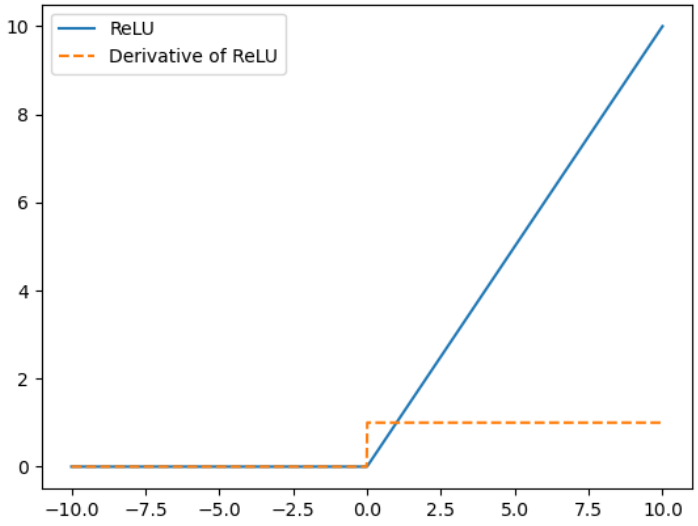

Figure 2.7 ReLU function and its derivative

A ReLU function and its derivative is given by,

$$\sigma(z) = \max(0, z) \tag{2.53}$$

$$\sigma^{'}(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases} \tag{2.54}$$

Figure 2.7 demonstrates the graph of a ReLu function and its derivative. A ReLu function has become a widely used activation function for deep learning applications especially convolutional neural networks (CNN) in the last few years. The main advantage is that it activates the neuron only when $z > 0$ which helps prevent vanishing gradient problems. A ReLu function is also computationally very efficient because it is implemented using a simple threshold. One of the disadvantages is that the gradient becomes 0 at the negative side which can hinder the learning of the neural network. To address the issue, a leaky ReLU function is proposed to fix the problem. A ReLu function doesn't work well with ODE's and PDE's problem because the backpropagation algorithm requires higher order derivatives in which ReLU function becomes zero.

Softplus Function

A softplus function and its derivative is given by,

$$\sigma(z) = \ln(1 + e^{z}) \tag{2.55}$$

$$\sigma^{'}(z) = \frac{1}{1 + e^{-z}} \tag{2.56}$$

Figure 2.8 demonstrates the graph of a softplus function and its derivative. A softplus function was proposed in 2000 by Dugas *et al.* [41]. A softplus function is a smooth approximation of a ReLU function. The derivative of a softplus function is equal to a sigmoid function.

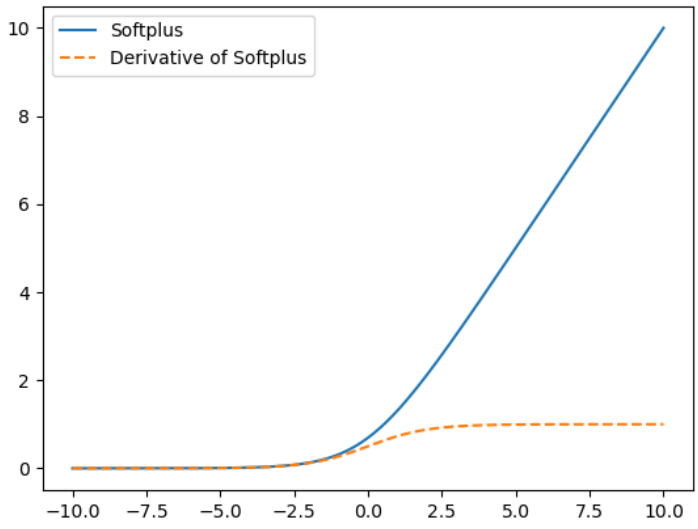

Figure 2.8 Softplus function and its derivative

Swish Function

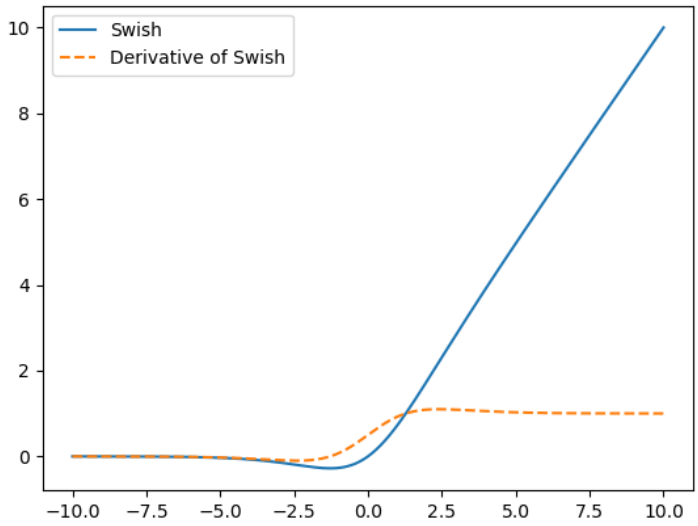

Figure 2.9 Swish function and its derivative

A swish function and its derivative is given by,

$$\sigma(z) = \frac{z}{1+e^{-z}} \tag{2.57}$$

$$\sigma'(z) = \frac{1}{1+e^{-z}} + \sigma(z)(z-\sigma(z)) \tag{2.58}$$

Figure 2.9 demonstrates the graph of a swish function and its derivative. A swish function is also known as a self-gated activation function. A swish function is proposed in 2017 by Ramachandran *et al.* [42]. The function multiplies the input of the neuron with sigmoid function creating hybrid activation function. A swish function is as computationally efficient as a ReLU function. A swish function is smooth and is differentiable at all points, and was reported to outperform ReLU function on some deep learning classification tasks.

Mish function

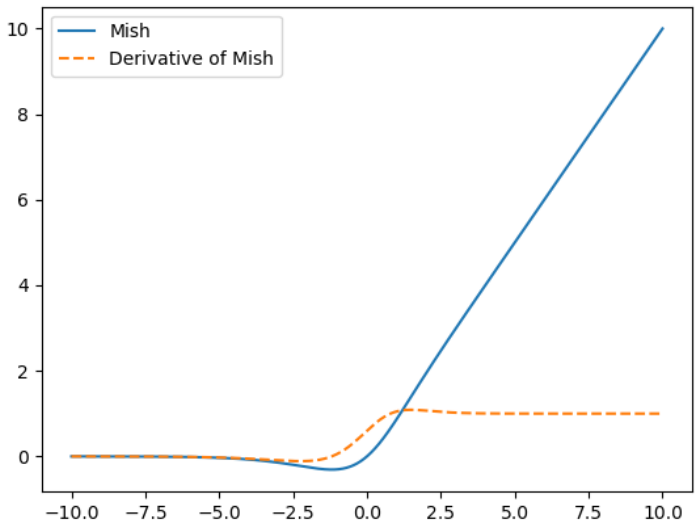

Figure 2.10 Mish function and its derivative

A mish function and its derivative is given by,

$$\sigma(z) = z\tanh(\ln(1+e^z)) \tag{2.59}$$

$$\sigma'(z) = \frac{e^z(4xe^x + 4x + 6e^x + 4e^{2x} + e^{3x} + 4)}{(2e^x + e^{2x} + 2)^2} \tag{2.60}$$

Figure 2.10 demonstrates the graph of a mish function and its derivative. A mish function is a new activation function which was proposed in 2019 by Misra [43]. A mish function is inspired by a swish function. The function is continuously differentiable

with infinite order, and unbounded above and bounded below.

Gaussian Error Linear Unit (GELU)

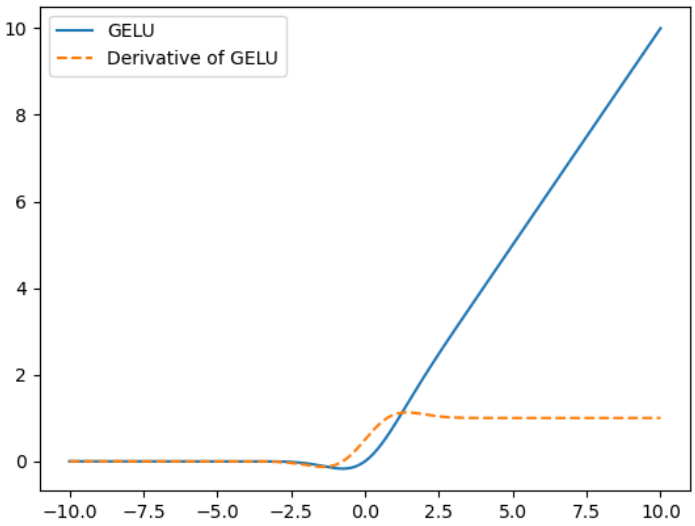

Figure 2.11 GELU function and its derivative

A GELU function is given by,

$$\sigma(z) = \frac{z}{2}(1 + \text{erf}(\frac{z}{\sqrt{2}})) \tag{2.61}$$

$$\sigma(z) \approx 0.5z(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(z + 0.044715z^3)\right)) \tag{2.62}$$

The derivative of GELU function is given by,

$$\sigma^{'}(z) = \frac{1}{2}(1 + \text{erf}(\frac{z}{\sqrt{2}})) + \frac{ze^{-\frac{z^2}{2}}}{\sqrt{2\pi}} \tag{2.63}$$

$$\begin{aligned}\sigma^{'}(z) \approx\ & 0.5 + 0.5\tanh\left(0.0356774z^3 + 0.797885z\right) \\ & + (0.0535161z^3 + 0.398942z)\,\text{sech}^2(0.0356774z^3 + 0.797885z)\end{aligned} \tag{2.64}$$

Figure 2.11 demonstrates the graph of a GELU function and its derivative. A GELU function was proposed in 2016 by Hendrycks and Gimpel [44]. The function uses the cumulative Gaussian distribution function as the activation function. Results

of the various experiments show a GELU function has better performance than a ReLU function for nonlinear problems.

Non-Parametric Linearly Scaled HyperbolicTangent Activation Function (LiSHT)

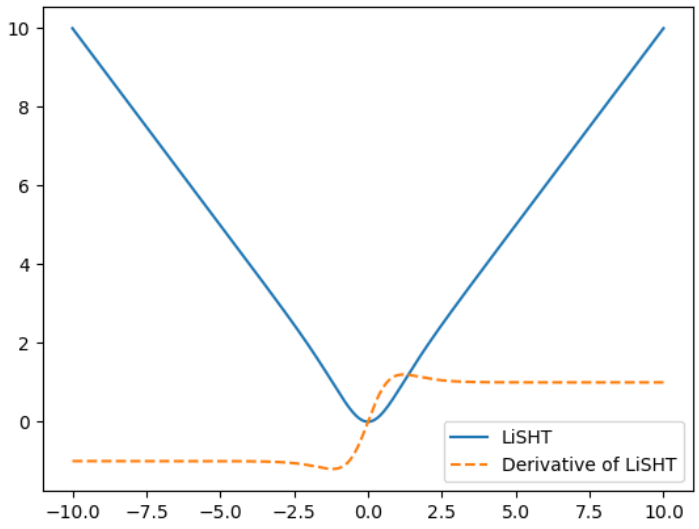

Figure 2.12 LiSHT function and its derivative

A LiSHT function and its derivative is given by,

$$\sigma(z) = z\tanh(z) \tag{2.65}$$

$$\sigma'(z) = \tanh(z) + x\,\mathrm{sech}^2(z) \tag{2.66}$$

Figure 2.12 demonstrates the graph of a LiSHT function and its derivative. A LiSHT function was proposed in 2019 by Roy *et al.* [45]. A LiSHT function is an attempt to scale the non-linear hyperbolic tangent function (Tanh) by the input of the neuron to tackle the vanishing gradient problem.

2.5.2 Optimization Algorithm

An overview of gradient descent optimization algorithms can be found in the work of Ruder [46]. In this work, we denote weight weights (w) and biases (b) of the

neural network by θ to simplify the mathematics. The list of optimization algorithms in this work are,

Mini-batch gradient descent

Mini-batch gradient descent computes the gradient of the cost function with respect to the parameters of neural network using mini-batch of n training data,

$$\theta_{t+1} = \theta_t - \eta \frac{\partial}{\partial \theta_t} \frac{1}{n} \sum_i^n C_i \tag{2.67}$$

By using mini-batch, the variance of the gradient will be lower when compared to the algorithm that uses only one training data. Another advantage is that the algorithm can make use of highly optimized matrix optimization of the recent libraries that make computing the gradient very fast. Mini-batch gradient descent is sometimes referred to as Stochastic gradient descent (SGD) in the literature. We can also implement the momentum

$$v_t = \gamma v_{t-1} - \eta \frac{\partial}{\partial \theta_t} \frac{1}{n} \sum_i^n C_i \tag{2.68}$$

$$\theta_{t+1} = \theta_t + v_t \tag{2.69}$$

when γ is momentum parameter. We can also use Nesterov term

$$\theta' = \theta_t + \gamma v_{t-1} \tag{2.70}$$

$$v_t = \gamma v_{t-1} - \eta \frac{\partial}{\partial \theta'} \frac{1}{n} \sum_i^n C_i \tag{2.71}$$

$$\theta_{t+1} = \theta_t + v_t \tag{2.72}$$

Adaptive Gradient Algorithm (Adagrad)

In 2011, Adagrad was proposed by Duchi *et al.* [47]. Adagrad is well-suited for dealing with sparse data. Adagrad uses a different learning rate for every parameter at every time step. The update method are,

$$G_t^\theta = \sum_{\tau=0}^{t} g_{\theta_\tau} \tag{2.73}$$

$$\theta_{t+1} = \theta_t - (\frac{\eta}{\sqrt{\delta I + G_t^\theta + \epsilon}}) \odot g_{\theta_t} \tag{2.74}$$

when $g_{\theta_t} = \frac{\partial}{\partial \theta_t} \frac{1}{n} \sum_i^n C_i$ and δ is the initial accumulator.

In this method, the learning rate was modified by the sum of the square of the gradient up to the last time step. ϵ is used to avoid division by zero (usually on the order of $1e-8$). The advantage is that it eliminates the need to manually set the learning rate. The disadvantage is that as the time step increases, the learning rate will decrease to zero making the algorithm unable to learn any longer.

Adadelta

Adadelta was first proposed by Zeiler in 2012 [48]. Adadelta is an extension of Adagrad to solve a monotonically decreasing learning rate issue. Instead of summing all squared gradients, we restricted the window of past gradients that are accumulated to be some fixed size w where $w < t$ (i.e. the number of epoch). Since storing w previous squared gradients is inefficient, the method implements this accumulation as an exponentially decaying average of the squared gradients. Initially, we set,

$$E[g^2]_{\theta_0} = 0 \tag{2.75}$$

$$E[\Delta^2]_{\theta_0} = 0 \tag{2.76}$$

We define the decay constant as ρ which usually is around 0.9. At time step

t, the update rules is given by,

$$E[g^2]_{\theta_t} = \rho E[g^2]_{\theta_{t-1}} + (1-\rho) g^2_{\theta_\tau} \tag{2.77}$$

$$E[\Delta^2]_{\theta_t} = \rho E[\Delta^2]_{\theta_{t-1}} + (1-\rho) \Delta^2_{\theta_t} \tag{2.78}$$

when g_{θ_t} and Δ_{θ_t} are defined by,

$$g_{\theta_t} = \frac{\partial}{\partial \theta_t} \frac{1}{n} \sum_i^n C_i \tag{2.79}$$

$$\Delta_{\theta_t} = \frac{\sqrt{E[\Delta^2]_{\theta_{t-1}} + \epsilon}}{\sqrt{E[g^2]_{\theta_t} + \epsilon}} g_{\theta_t} \tag{2.80}$$

Finally, we update the parameters by,

$$\theta_{t+1} = \theta_t - \Delta\theta_{\theta_t} \tag{2.81}$$

For Adadelta, we no longer need to set the learning rate at the start of the training of the neural network.

Root Mean Square Propagation (RMSprop)

RMSprop was proposed by Geoff Hinton in one of his lectures. RMSprop has been developed to also resolve the issue of Adagrad. RMSprop use the same update as Adadelta,

$$E[g^2]_{\theta_t} = \rho E[g^2]_{\theta_{t-1}} + (1-\rho) g^2_{\theta_t} \tag{2.82}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_{\theta_t} + \epsilon}} g_{\theta_t} \tag{2.83}$$

when $E[g^2]_{\theta_0} = 0$ and $g_{\theta_t} = \frac{\partial}{\partial \theta_t} \frac{1}{n} \sum_i^n C_i$

We can also implement plain momentum in the algorithm

$$\theta_{t+1} = \theta_t + \gamma(-\frac{\eta}{\sqrt{E[g^2]_{\theta_{t-1}} + \epsilon}} g_{\theta_{t-1}}) - \frac{\eta}{\sqrt{E[g^2]_{\theta_t} + \epsilon}} g_{\theta_t} \tag{2.84}$$

when γ is momentum parameter.

Adaptive Moment Estimation (Adam)

In 2015, Adam was proposed by Kingma and Ba [49]. Adam keeps an exponentially decaying average of past gradients m_{θ_t} and past squared gradients υ_{θ_t}. At the first time step, we initialize

$$m_{\theta_0} = 0 \tag{2.85}$$

$$\upsilon_{\theta_0} = 0 \tag{2.86}$$

At time step t, the update rule are given by

$$m_{\theta_t} = \beta_1 m_{\theta_{t-1}} + (1 - \beta_1) g_{\theta_t} \tag{2.87}$$

$$\upsilon_{\theta_t} = \beta_2 \upsilon_{\theta_{t-1}} + (1 - \beta_2) g^2_{\theta_t} \tag{2.88}$$

when β_1, β_2 are decaying constants which are usually close to 1. The original authors observe that the value of m_{θ_0} and υ_{θ_0} are biased towards zero in the beginning of the training. So they used the correction of m_{θ_t} and υ_{θ_t} instead,

$$\hat{m}_{\theta_t} = \frac{m_{\theta_t}}{1 - (\beta_1)^t} \tag{2.89}$$

$$\hat{\upsilon}_{\theta_t} = \frac{\upsilon_{\theta_t}}{1 - (\beta_2)^t} \tag{2.90}$$

Finally, we update the parameters by,

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_{\theta_t}}{\sqrt{\hat{\upsilon}_{\theta_t}} + \epsilon} \tag{2.91}$$

Adamax

Adamax is a variant of Adam proposed in the same paper as Adam. Adamax is a generalisation of Adam from the l_2 norm to the l_∞ norm,

$$\upsilon_{\theta_t} = \beta_2^\infty \upsilon_{\theta_{t-1}} + (1-\beta_2^\infty) g_{\theta_t}^\infty \tag{2.92}$$

$$= \max(\beta_2 \upsilon_{\theta_{t-1}}, |g_{\theta_t}|) \tag{2.93}$$

The exponential average of past gradients is the same as Adam,

$$m_{\theta_t} = \beta_1 m_{\theta_{t-1}} + (1-\beta_1) g_{\theta_t} \tag{2.94}$$

$$\hat{m}_{\theta_t} = \frac{m_{\theta_t}}{1-(\beta_1)^t} \tag{2.95}$$

The parameters are updated by,

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_{\theta_t}}{\upsilon_{\theta_t}} \tag{2.96}$$

Nesterov-accelerated Adaptive Moment Estimation (Nadam)

Nadam was proposed by Dozat in 2015 [50]. Nadam incorporates Adam and Nesterov Accelerated Gradient (NAG). We initialize

$$m_{\theta_0} = 0 \tag{2.97}$$

$$\upsilon_{\theta_0} = 0 \tag{2.98}$$

The update rule are the same as Adam algorithm

$$m_{\theta_t} = \beta_1 m_{\theta_{t-1}} + (1-\beta_1) g_{\theta_t} \tag{2.99}$$

$$\upsilon_{\theta_t} = \beta_2 \upsilon_{\theta_{t-1}} + (1-\beta_2) g_{\theta_t}^2 \tag{2.100}$$

when β_1, β_2 are decaying constants. Nadam uses Nesterov momentum to look ahead by adding the momentum term in the corrected m_{θ_t},

$$\hat{m}_{\theta_t} = \frac{1}{1-(\beta_1)^t}(\beta_1 m_{\theta_t} + (1-\beta_1)g_{\theta_t}) \tag{2.101}$$

$$\hat{\upsilon}_{\theta_t} = \frac{\upsilon_{\theta_t}}{1-(\beta_2)^t} \tag{2.102}$$

Finally, we update the parameters by

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_{\theta_t}}{\sqrt{\hat{\upsilon}_{\theta_t}} + \epsilon} \tag{2.103}$$

2.5.3 Weight Initialization Algorithm

Weight initialization algorithm is responsible for the initial values of the parameters of the neural network just before the network begins training. The goal of weight initialization is to help prevent exploding or vanishing gradient problems during training. This list of initializations used in this work are the following.

Identity initialization

In this initialization, weights are initialized to be an identity matrix in every layer of the neural network. Identity initialization was used in the identity RNN architecture which was proved to work well in MNIST digits dataset.

Orthogonal initialization

The initialization creates an orthogonal matrix obtained from the QR decomposition of a random matrix drawn from normal distribution. The initialization was suggested by McClelland and Ganguli [51] by studying the learning dynamics of deep linear neural networks.

Random uniform and normal distribution initialization

In this technique, we initialize all the weights randomly from univariate normal distribution or uniform distribution, and biases are set to zero.

$$W^l = \mathcal{N}(\mu = 0, \sigma^2 = 0.05) \tag{2.104}$$

$$W^l = \mathcal{U}(-0.05, 0.05) \tag{2.105}$$

$$b^l = 0 \tag{2.106}$$

Random initialization is used to perform symmetry breaking, which is the way of preventing all of the weights in the model from being the same. It prevents neurons from learning the same features of its inputs. However, if values of weight are too large or too small, it can cause exploding and vanish gradient problems respectively.

Lecun uniform and normal distribution initialization

The initialization was proposed in 1998 by Lecun *et al.* [52] to help prevent exploding and vanishing problems. By only taking forward propagation into account, the method was initially derived for a hyperbolic tangent function (Tanh), but can be extended to sigmoid function. LeCun method only works for the activation functions that are differentiable at $z = 0$.

In this technique, we initialize all the weights randomly from univariate normal distribution or uniform distribution with specific variance, and biases are set to zero.

$$W^l = \mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{n_{in}^l}) \tag{2.107}$$

$$W^l = \mathcal{U}(-\sqrt{\frac{3}{n_{in}^l}}, \sqrt{\frac{3}{n_{in}^l}}) \tag{2.108}$$

$$b^l = 0 \tag{2.109}$$

when n_{in}^l is the number of neurons of the previous layer $(l-1)$.

Glorot uniform and normal distribution initialization

Glorot initialization which also known as Xavier initialization was proposed in 2010 by Glorot and Bengio [53]. The method takes forward and backward propagation into account. The method was derived using a hyperbolic tangent function (Tanh). In this technique, we initialize all the weights randomly from univariate normal distribution or uniform distribution with specific variance, and biases are set to zero.

$$W^l = \mathcal{N}(\mu = 0, \sigma^2 = \frac{2}{n_{in}^l + n_{out}^l}) \tag{2.110}$$

$$W^l = \mathcal{U}(-\sqrt{\frac{6}{n_{in}^l + n_{out}^l}}, \sqrt{\frac{6}{n_{in}^l + n_{out}^l}}) \tag{2.111}$$

$$b^l = 0 \tag{2.112}$$

when n_{in}^l is the number of neurons of the previous layer $(l-1)$, and n_{out}^l is the number of neurons of the next layer $l+1$.

He uniform and normal distribution initialization

The method was first proposed in 2015 by He *et al.* [54]. The initialization was derived using a ReLU function which is not differentiable at $z = 0$. He initialization method is suitable for a non-differentiable activation function. In this technique, we initialize all the weights randomly from univariate normal distribution or uniform distribution with specific variance, and biases are set to zero.

$$W^l = \mathcal{N}(\mu = 0, \sigma^2 = \frac{2}{n_{in}^l}) \tag{2.113}$$

$$W^l = \mathcal{U}(-\sqrt{\frac{6}{n_{in}^l}}, \sqrt{\frac{6}{n_{in}^l}}) \tag{2.114}$$

$$b^l = 0 \tag{2.115}$$

2.5.4 Learning Rate

The learning rate is a hyperparameter that controls how fast or slow the weights and biases are updated based on the gradient. In this work, we use a cyclical learning rate schedule with exponential decay policy. The method was proposed by Smith in 2017 [55]. It is used to help reduce the issue of fixed learning rate. If the value of fixed learning rate is too large, the gradient will tend to overshoot the global optimum. For a small fixed learning rate, the gradient will tend to have slow convergence.

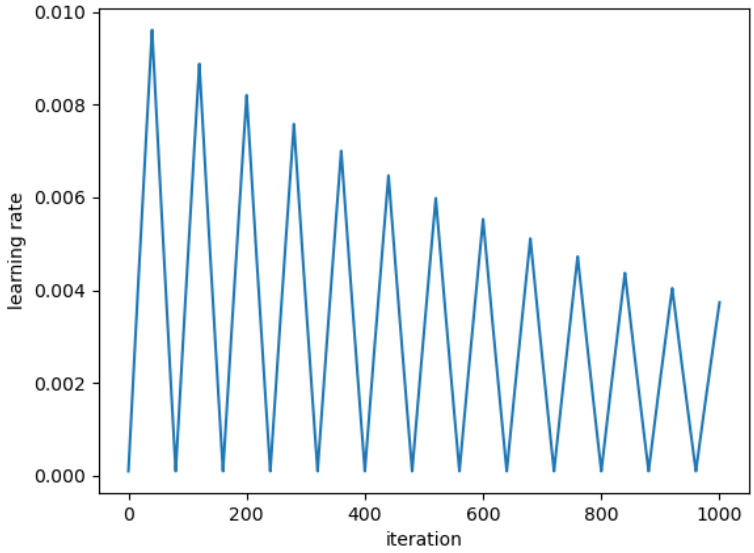

Figure 2.13 Cyclical learning rate.

The cyclical learning rate will change the learning rate back and forth between upper bounds (lr_m) and lower bounds (lr_b) in a zig-zag way. Figure 2.13 shows an example of a cyclical learning rate schedule with exponential decay policy. To explain the details, we must first define some terms as follows :

- Iteration (i) : Number of times that learning rate has been updated. The learning rate will be updated at the same time as the weights and biases get updated by one training batch.

- Step size (s) : Number of iterations it takes for the learning rate to go from the lower bound value to the upper bound value or vice versa (half cycle). In practice,

the step size is recommended to be 2 to 8 times of

$$\frac{\text{total number of data}}{\text{batch size}} \tag{2.116}$$

With the term defined, the learning rate of exponential decay policy is found by

$$cycle = \text{floor}(1 + \frac{i}{2s}) \tag{2.117}$$

$$x = |\frac{i}{s} - 2\text{floor}(1 + \frac{i}{2s}) + 1| \tag{2.118}$$

$$lr = lr_b + (lr_m - lr_b)\ \text{max}(0, 1 - x)\gamma^i \tag{2.119}$$

when lr is the learning rate, γ is decay rate, and floor function gives the greatest integer that is less than or equal to the input.

The advantage of a cyclical learning rate schedule is that it combines the benefit of both large and small learning rates. By setting a large learning rate, the gradients can overshoot a local optimum and help speed up the search. The small learning rate will prevent overshooting if the network is closer to global optimum.

2.5.5 Number of Hidden Layers and Total Number of Parameters

The goal of the neural network is to learn a mapping function. The capability of a neural network will determine the complexity of the function that can be approximated by the model. A model with greater capability will be able to learn a wide range of function or learn it much faster.

The most basic way to estimate the capability of a neural network is to count the total number of parameters. Usually, the more parameters, the higher the capability of a network will be. But even this measure is not perfect. In this work, we use the total number of parameters to measure model capability due to its simplicity. The total number of parameters of a feedforward neural network with linear activation function

on the output layer with no bias is,

$$p = n(n_i + n_o + 1) + n(n + 1)(l - 1) \tag{2.120}$$

when n is the number of neurons per hidden layer, n_i is the number of neurons in the input layer, n_o is the number of neurons in the output layer, and l is the number of hidden layers.

Normally, the number of neurons in the input and output layer are fixed by the problem. Therefore, the total number of parameters of a neural network will usually depend on two aspects, number of hidden layers and number of neurons per layer. By fixing the total number of parameters, many researches show that the deep neural network will usually perform better than the shallow neural network. But if the neural network becomes too deep, the network will start to experience the vanishing gradient problem, and the convergence will start to deteriorate.

CHAPTER III
METHODOLOGY AND BACKGROUND

In this chapter, we present methods of the neural network for solving differential equations. Later, the stopping criterion which is used to determine when to stop the neural network training is explained with the corresponding performance measurement. Lastly, a genetic algorithm in the context of the neural network is explained. A genetic algorithm is used to find the optimal activation function, optimization algorithm, and weight initialization. In this work, we are interested in a 2D Laplace's equation of the form,

$$\nabla^2 u(x,y) = 0, \qquad (x,y) \in \Omega \tag{3.1}$$

$$u(x,y) = g(x,y), \quad (x,y) \in \partial\Omega \tag{3.2}$$

when $g(x,y)$ is a Dirichlet boundary condition, $u(x,y)$ is the solution, Ω is the domain of interest, and $\partial\Omega$ is the boundary of the domain.

3.1 The Lagaris Method

Lagaris *et al.* [25] proposed a method for solving ODE's, systems of coupled ODE's, and PDE's on rectangular domains by relying on the function approximation capability of the feedforward neural network. They proposed to solve the following general differential equation,

$$G(\vec{x}, u(\vec{x}), \nabla u(\vec{x}), \nabla^2 u(\vec{x})) = 0, \quad \vec{x} \in \Omega \tag{3.3}$$

where Ω is the domain of the problem and $u(\vec{x})$ is the solution of the differential equation. The differential equation is also subjected to certain boundary conditions which can be Dirichlet and/or Neumann boundary conditions.

Lagaris *et al.* adopted the collocation method which enforces that the approximate solution satisfies the differential equation at a given set of points in the domain. The locations of the collocation points will become the training data for the neural network. The problem is then transformed into the following system of equations subject to the constraints imposed by the boundary conditions

$$G(\vec{x}_i, u(\vec{x}_i), \nabla u(\vec{x}_i), \nabla^2 u(\vec{x}_i)) = 0, \quad \forall \vec{x}_i \in \Omega \tag{3.4}$$

In the proposed approach, the trial solution $\hat{u}(\vec{x})$ employs a feedforward neural network with parameters corresponding to weights and biases of the neural network. The form of trial function is chosen such that by construction it satisfies the boundary condition exactly. This can be achieved by writing it as a sum of two terms as follows :

$$\hat{u}(\vec{x}) = A(\vec{x}) + F(\vec{x}, y^L(\vec{x}; \vec{w}, \vec{b})) \tag{3.5}$$

where $y^L(\vec{x}; \vec{w}, \vec{b})$ is the output of the neural network.

The term $A(\vec{x})$ contains no adjustable parameters and satisfies the boundary condition. The second term $F(\vec{x}, y^L(\vec{x}; \vec{w}, \vec{b}))$ is constructed such that it will become zero at the boundary. With the form of the trial function in Equation 3.5, the problem is reduced from constrained to unconstrained optimization problem. In order to train the neural network to find the parameters $\vec{w}$ and $\vec{b}$, we define the cost function as

$$J(\vec{w}, \vec{b}) = \sum_{\forall \vec{x}_i \in \Omega} G(\vec{x}_i, \hat{u}(\vec{x}_i), \nabla \hat{u}(\vec{x}_i), \nabla^2 \hat{u}(\vec{x}_i))^2 \tag{3.6}$$

and the optimal parameters $\vec{w}^*$ and $\vec{b}^*$ are given by

$$(\vec{w}^*, \vec{b}^*) = \underset{\vec{w},\vec{b}}{\operatorname{argmin}} J. \tag{3.7}$$

To illustrate the method, we consider a 2D Laplace's equation on a rectangular domain. We assume the domain of a rectangular to be $[a, b] \times [c, d]$. The Dirichlet boundary conditions are $u(a, y) = f_0(y)$, $u(b, y) = f_1(y)$, $u(x, c) = g_0(x)$, $u(x, d) = g_1(x)$. Then, the trial solution can be written as

$$\hat{u}(x, y) = A(x, y) + D(x, y)y^L(x, y; w, b), \tag{3.8}$$

where $A(x, y)$ is chosen so as to satisfy the boundary condition on the rectangular domain

$$\begin{aligned} A(x,y) =& (b-x)f_0(y) + (x-a)f_1(y) + \\ & (d-y)\Big(g_0(x) - \Big((d-x)g_0(a) + (x-a)g_0(b)\Big)\Big) + \\ & (y-c)\Big(g_1(x) - \Big((d-x)g_1(a) + (x-a)g_1(b)\Big)\Big) \end{aligned} \tag{3.9}$$

and $D(x, y)$ is a distance function which is zero at the boundary

$$D(x, y) = (x - a)(b - x)(y - c)(d - y). \tag{3.10}$$

With the form of the trial function, we can directly optimize weights and biases of the neural network. The cost function then can be written as

$$C(x, y) = \sum_{(x_n,y_n)} \left| \frac{\partial^2 \hat{u}(x_n, y_n)}{\partial x^2} + \frac{\partial^2 \hat{u}(x_n, y_n)}{\partial y^2} \right|^2, \tag{3.11}$$

when $(x_n, y_n) \in [a, b] \times [c, d]$, $n = 1, 2, \ldots, N$, and N is the total number of training data.

3.2 The Deep Galerkin Method

Sirignano and Spiliopoulos [28] proposed a method for solving a class of time dependent quasilinear parabolic PDEs on irregular domains. The method was designed for solving PDEs of the form of the following equations.

$$\frac{\partial u(t,\vec{x})}{\partial t} + \mathcal{L}u(t,\vec{x}) = 0, \qquad (t,\vec{x}) \in [0,T] \times \Omega \tag{3.12}$$

$$u(t,\vec{x}) = g(t,\vec{x}), \quad (t,\vec{x}) \in [0,T] \times \partial\Omega \tag{3.13}$$

$$u(0,\vec{x}) = u_0(\vec{x}), \qquad \vec{x} \in \Omega \tag{3.14}$$

where $\partial\Omega$ is the boundary of the domain (Ω), $\mathcal{L}$ is a differential operator, $g(t,\vec{x})$ is the Dirichlet boundary condition, and $u_0(\vec{x})$ is the initial conditions.

In their proposed approach, the trial solution $\hat{u}(t,\vec{x})$ directly employs a neural network without any additional function to satisfy the initial or boundary conditions. Therefore, the solution of the neural network will not exactly satisfy the boundary and initial condition.

$$\hat{u}(t,\vec{x}) = y^L(t,\vec{x};\vec{w},\vec{b}) \tag{3.15}$$

The weights and biases of the neural network are trained by minimizing a least-squares cost function using data at collocation points $s_n = \{(t_n,\vec{x}_n),(\tau_n,\vec{z}_n),\vec{q}_n\}$,

$$(t_n,\vec{x}_n) \in [0,T] \times \Omega \tag{3.16}$$

$$(\tau_n,\vec{z}_n) \in [0,T] \times \partial\Omega \tag{3.17}$$

$$\vec{q}_n \in \Omega \tag{3.18}$$

Equation 3.16 specifies the points inside the domain of the problem, Equation 3.17 specifies only the points on the boundary. Equation 3.18 specifies the points at which the initial condition is enforced. The three sets of data are used to train different compo-

nents of the cost function.

The cost function is constructed from Equations 3.12, 3.13, and 3.14,

$$J(\vec{w},\vec{b}) = \sum_{(t_n,\vec{x}_n)} \left| \frac{\partial \hat{u}(t_n,\vec{x}_n)}{\partial t} + \mathcal{L}\hat{u}(t_n,\vec{x}_n) \right|^2 + \sum_{(\tau_n,\vec{z}_n)} \left| \hat{u}(\tau_n,\vec{z}_n) - g(\tau_n,\vec{z}_n) \right|^2 + \sum_{\vec{q}_n} \left| \hat{u}(0,\vec{w}_n) - u_0(\vec{w}_n) \right|^2 \tag{3.19}$$

The first term of the cost function measures how well the approximated solution satisfies the differential equation. The second and third terms enforce the Dirichlet boundary condition and initial condition to the approximated solution. Contrary to the Lagaris method which strictly enforces the boundary condition on the trial solution, the solution from the deep Galerkin method will not exactly satisfy the initial and boundary conditions.

3.3 The Mixed Residual Method

Liyao *et al.* [29] proposed the method called mixed residual network. The method focuses on the same equation as Sirignano and Spiliopoulos (Equations 3.12 - 3.14). They also consider the case without temporal derivatives,

$$\mathcal{L}u(\vec{x}) = 0, \qquad \vec{x} \in \Omega \tag{3.20}$$

$$u(\vec{x}) = g(\vec{x}), \quad \vec{x} \in \partial\Omega \tag{3.21}$$

The method first rewrites high-order derivatives into low-order ones using

auxiliary variables, and then rewrites a given PDE into a first-order system,

$$\vec{p} = \nabla u \tag{3.22}$$

$$q = \nabla \cdot \vec{p} = \Delta u \tag{3.23}$$

$$\vec{w} = \nabla q = \nabla(\Delta u) \tag{3.24}$$

To illustrate the method, we consider a 2D Laplace's equation. Using auxiliary variables, the first-order system is,

$$\vec{p}(x, y) = \nabla u(x, y), \quad (x, y) \in \Omega \tag{3.25}$$

$$\nabla \cdot \vec{p}(x, y) = 0, \quad (x, y) \in \Omega \tag{3.26}$$

$$u(x, y) = g(x, y), \quad (x, y) \in \partial\Omega \tag{3.27}$$

In their paper, they addressed both the exact and inexact boundary conditions. To handle time independent differential equations with exact boundary conditions on rectangular domains, they use the approach from Lagaris *et al.*,

$$\hat{u}(x, y) = A(x, y) + D(x, y) y_1^L(x, y; \vec{w}, \vec{b}) \tag{3.28}$$

$$\hat{p}_1(x, y) = y_2^L(x, y; \vec{w}, \vec{b}) \tag{3.29}$$

$$\hat{p}_2(x, y) = y_3^L(x, y; \vec{w}, \vec{b}) \tag{3.30}$$

when y_1^L, y_2^L, y_3^L are the output of the same neural network, $A(x, y)$ is chosen so as to satisfy the boundary condition, and $D(x, y)$ is a distance function which is zero at the boundary. For a 2D Laplace's equation, the cost function is defined using the first-order form,

$$J(\vec{w}, \vec{b}) = \sum_{(x_n, y_n)} \left| \hat{p}(x_n, y_n) - \nabla \hat{u}(x_n, y_n) \right|^2 + \sum_{(x_n, y_n)} \left| \nabla \cdot \hat{p}(x_n, y_n) \right|^2 \tag{3.31}$$

when $(x_n, y_n) \in \Omega$.

For the case of inexact boundary conditions which are needed when dealing with irregular domains, and/or when solving time dependent differential equations. They used the approach from Sirignano and Spiliopoulos. The approximated solution employs a neural network directly,

$$\hat{u}(x, y) = y_1^L(x, y; \vec{w}, \vec{b}) \tag{3.32}$$

$$\hat{p}_1(x, y) = y_2^L(x, y; \vec{w}, \vec{b}) \tag{3.33}$$

$$\hat{p}_2(x, y) = y_3^L(x, y; \vec{w}, \vec{b}) \tag{3.34}$$

when y_1^L, y_2^L, y_3^L are the output of the same neural network.

For a 2D Laplace's equation, the collocation points used for training is $s_n = \{(t_n, x_n), (\tau_n, z_n)\}$,

$$(t_n, x_n) \in \Omega \tag{3.35}$$

$$(\tau_n, z_n) \in \partial\Omega \tag{3.36}$$

The cost function is then used the first-order form of a 2D Laplace's equation,

$$\begin{aligned} J(\vec{w}, \vec{b}) = &\sum_{(x_n, y_n)} \left| \hat{p}(x_n, y_n) - \nabla \hat{u}(x_n, y_n) \right|^2 + \sum_{(x_n, y_n)} \left| \nabla \cdot \hat{p}(x_n, y_n) \right|^2 \\ &+ \sum_{(\tau_n, z_n)} \left| \hat{u}(\tau_n, z_n) - g(\tau_n, z_n) \right|^2 \end{aligned} \tag{3.37}$$

3.4 Stopping Criterion and Performance Measurement

In this work, the stopping criterion we used is the relative error norm from Mcfall [27].

$$E_{norm} = \frac{\sqrt{\sum_{\vec{x}\in\Omega}\left|u(\vec{x}) - \hat{u}(\vec{x};\vec{w},\vec{b})\right|^2}}{\sqrt{\sum_{\vec{x}\in\Omega} u(\vec{x})^2}} \tag{3.38}$$

where $\vec{x}$ is the grid point in the domain (Ω), $u(\vec{x})$ is the analytical solution of the differential equation, and $\hat{u}(\vec{x};\vec{w},\vec{b})$ is the approximated solution from the neural network.

The relative error produces an aggregate error that accurately reflects relative errors of the analytical solution for regions of the domain with both large and small values. When we do not have the analytical solution, we can directly use the cost function as one of such stopping criteria.

We measure the performance of the neural network with the "number of epochs". An epoch refers to one cycle of training through the full training dataset. One epoch means that each data in the training dataset has had an opportunity to update weights and biases. The number of epochs is recorded once the specific relative error norm is reached. The relative error can be calculated using the testing dataset. The testing will be done at the end of every epoch. The lower the number of epochs is, the faster the network is trained to reach the specified relative error. We can also measure the performance of the network in terms of computational time using the average time per epoch (seconds) which will depend on a set of hyperparameters of the neural network, and the physical computing resource such as GPU.

3.5 Genetic Algorithm

There are many types of genetic algorithms varying in their structure and purpose. Nonetheless, all of them share a few common components. The components

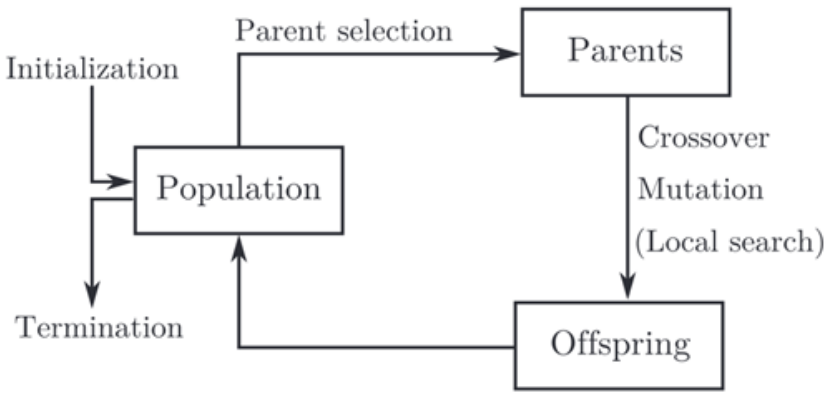

Figure 3.1 The evolutionary cycle of genetic algorithm.

are shown in Figure 3.1. The algorithm begins by initializing a population of individuals randomly. Then, it runs each member of the population through a fitness function. The fitness function will determine how "fit" or how "good" the member is. Members will get selected randomly based on their fitness value to be the parents to reproduce the offsprings. Normally, the reproduction methods are crossover and mutation. The offsprings are then used as a new generation of population. The algorithm is repeated until a desired number of iterations have passed. At the termination, the algorithm presents all members of the last generation according to the fitness function.

3.5.1 Chromosome

To use the neural network in the genetic algorithm, we must first encode the neural network into an appropriate form called chromosome. Chromosomes will store information of the neural network that can be used to create the offsprings. There are two types of encoding schemes used for the neural network [30].

The first type is the direct encoding scheme. In this scheme, each connection of neurons is represented by the binary value, and collectively is represented by the matrix. The disadvantages of direct encoding schemes are scalability and permutation problems. In the permutation problem, many chromosomes can represent the same neural network.

The second type is the indirect encoding scheme. One type of the indirect encoding schemes is the parametric representation [56]. The neural network is specified by a set of parameters such as activation function, weight initialization, and etc. This type of representation is most suitable when we know what type of architecture we are trying to find, e.g. a feedforward neural network.

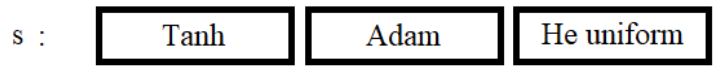

Figure 3.2 Example of chromosome.

In this work, we use parametric representation [56]. Each chromosome is composed of vectors of strings, and each string is the hyperparameter of the neural network. The network is specified by a set of parameters shown in Figure 3.2, e.g. activation function, optimization algorithm, and weight initialization algorithm.

3.5.2 Fitness Function

Once the network is trained, and reach a specific value of relative error defined in Equation 3.38. The fitness value is given by,

$$f = \frac{1}{\text{epochs}} \tag{3.39}$$

For a specific value of relative error, the lower the number of epoch is, the better the performance of the neural network will be. The maximum fitness value is equal to 1.

3.5.3 Population Initialization and Selection

In the first part of a genetic algorithm, we need to initialize the population. First, we initialize N random neural networks to create a population we called P. The first population is generated randomly to allow the entire range of possible sets of hyperparameters, and to avoid a condition known as premature convergence which make

the algorithm fall into a local minimum [57],

$$P = \{s_1, s_2, \ldots, s_N\} \tag{3.40}$$

Then, we sort the members in the population according to its fitness value using Equation 3.39 from the best to the worst member.

In this work, the members are then selected through elitism. The process of elitism will keep a specific percentage of the best members in the population from the previous generation to the next generation. Next, the members are selected through the roulette wheel selection method in which fitter members have a proportionally higher chance for selection to be reproduced. The probability for each member to be selected is

$$q_i = \frac{f(s_i)}{\sum_{k=1}^{N} f(s_k)}, \quad i = 1, 2, \ldots, N, \tag{3.41}$$

where $f(s_i)$ is the fitness of the i_{th} member. The cumulative probability $\hat{q}_i$ for member s_i is defined as

$$\hat{q}_i = \sum_{k=1}^{i} q_k, \quad i = 1, 2, \ldots, N. \tag{3.42}$$

To choose a member, we will generate a random number $p \in [0, 1]$. The member s_i will be chosen if $\hat{q}_{i-1} < p \leq \hat{q}_i$, when $\hat{q}_0 = 0$.

3.5.4 Crossover and Mutation Operator

The crossover and mutation operator is one of the simplest reproduction methods. The reproduction will determine how to create a new population based on the existing members of the population. In this work, we use a single point crossover. A point between both parents' chromosomes is picked randomly, and it will be designated as a crossover point. The process is shown in Figure 3.3. All parameters beyond the crossover point will be swapped between the two parents creating two children.

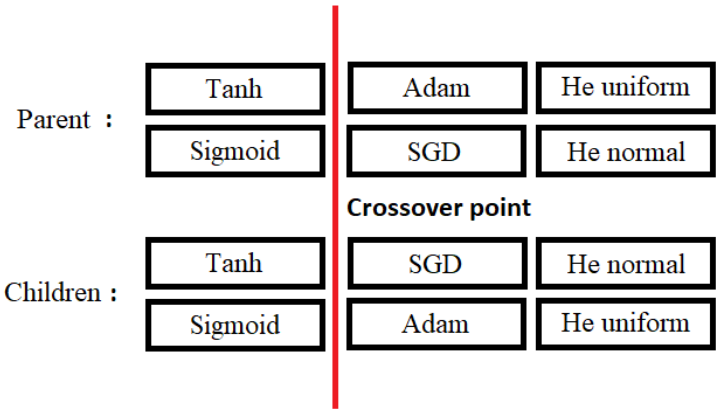

Figure 3.3 Single point crossover.

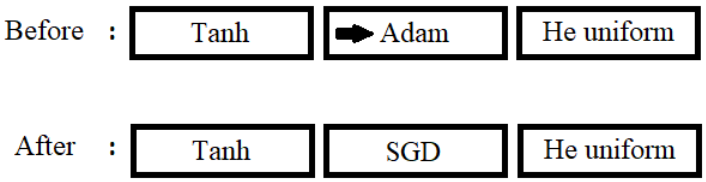

Figure 3.4 Mutation.

Mutation is also used to maintain, and introduce diversity in the population. A mutation chance is represented by percentage. Once the mutation occurs, we will randomly choose one of the parameters of the child. Then, its values will be randomly picked from the search space. The process of the mutation is shown in Figure 3.4.

CHAPTER IV
PROBLEM SETUP AND NUMERICAL RESULTS

In this chapter, we report our results of the investigation on the hyperparameters of the feedforward neural network for solving a 2D Laplace's equation on rectangular domains. We divide this chapter into four sections. The first one contains the setup of the problem used in this work which includes the domain of the problem, batch size, number of training and testing points, parameters of learning rate schedule, and etc. In the second section, we compare the performance of different neural network methods both for exact and inexact boundary conditions for different relative errors. We also consider the additional problem of 2D Laplace's equation on circular domains, and 2D Poisson's equation on a shar shaped domain. In the third section, we show the results of a genetic algorithm by considering the best members in the population and the proportion of hyperparameters in the last generation of the algorithm for different boundary conditions and relative errors. In the last section, also for different boundary conditions and relative errors, we compare the performance of different values of hidden layers while the total number of parameters is fixed. Lastly, we compare the performance of different total numbers of parameters while the number of hidden layers is fixed.

4.1 Problem Setup

In this work, we consider a 2D Laplace's equation on a rectangular domain,

$$\nabla^2 u(x, y) = 0, \quad (x, y) \in [0, 1] \times [0, 1] \tag{4.1}$$

We choose the boundary condition such that the analytical solution is,

$$u(x, y) = \sin(\omega y)e^{-\omega x} \tag{4.2}$$

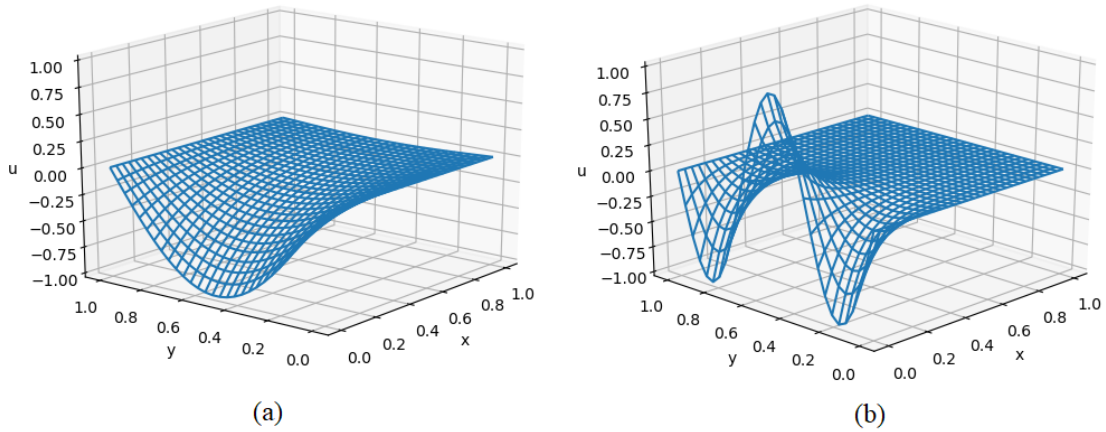

Figure 4.1 The graph of Equation 4.2, (a) Solution with $\omega = \pi$. (b) Solution with $\omega = 3\pi$.

In this work, the frequency is chosen to be $\omega = \pi$ and 3π. The figure of the solution is shown in Figure 4.1. For the case of exact boundary condition, the boundary function is defined as

$$A(x, y) = (1 - x)\sin(-\omega y) + x\sin(-\omega y)e^{-\omega}. \tag{4.3}$$

For distance function, it is defined as,

$$D(x, y) = x(1 - x)y(1 - y). \tag{4.4}$$

For the training data, we use 100×100 collocation points uniformly distributed on the rectangular domain, and 40000 collocation points uniformly distributed on the boundary. For the testing data used for calculating the relative error, the numbers of collocation points are 1000×1000 uniformly distributed across the domain including the boundary. The batch size is 1000, and we use the cyclical learning rate schedule with

exponential decay policy to adjust the learning rate during the network training. For the parameters of the learning rate schedule, the upper bound is $1e^{-2}$, the lower bound is $1e^{-4}$, the step size is 40, and the decay parameter is 0.9998. In this work, we use Tensorflow version 2.3.0. to perform the computations in parallel.

The workflow in this work is shown in Figure 4.2. First, we will choose the neural network method for the problem. Then, we will sample the training points and testing points from the domain, and define the cost function for the neural network. We will also choose a set of hyperparameters of the network. Finally, we will set the value of relative error as the stopping criterion. The average number of epochs is then recorded.

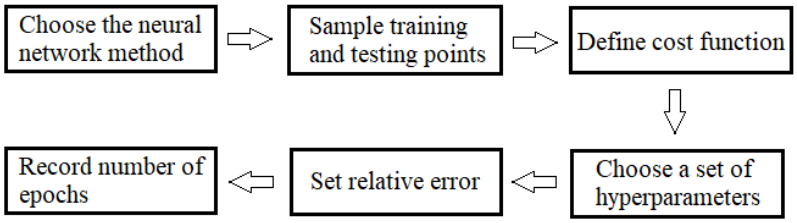

Figure 4.2 Workflow for solving 2D Laplace's equation using the neural network method.

4.2 Performance of Different Neural Network Methods

In this section, we will compare the performance of different neural network methods for exact and inexact boundary conditions. For the exact boundary condition on a rectangular domain, we compare the Lagaris method with the mixed residual method on a 2D Laplace's equation. For the inexact boundary condition on a rectangular and circular domain, we compare the deep Galerkin method with the mixed residual method also on a 2D Laplace's equation. Finally, we compare the deep Galerkin method with the mixed residual method on a 2D Poisson's equation on a star shape domain for the

case of inexact boundary condition.

Table 4.1 The hyperparameters for comparing different neural network methods.

Hyperparameters	
Number of hidden layers	4
Number of neurons per layer	30
Activation function	Swish
Output activation function	Linear
Optimization algorithm	Adam
Weight initialization	Glorot uniform
Total number of parameters	2911

The hyperparameters of the neural network are shown in Table 4.1. Using 50 randomly initialized neural networks, we calculate the average number of epochs and the standard deviation. The results are compared across different relative errors. Lastly, we also present the average time per epoch for different neural networks methods. The average time per epoch is calculated using 1000 epochs,

$$\text{average time per epoch} = \frac{\text{total time}}{\text{number of epochs}} \tag{4.5}$$

4.2.1 Exact Boundary Condition : Rectangular Domain

For the exact boundary condition on a rectangular domain, we compare the Lagaris method with the mixed residual method on a 2D Laplace's equation using boundary function (Equations 4.3) and distance function (Equation 4.4). The relative error norm that we choose are,

$$E_{\omega=\pi} = \{1\%, 0.5\%, 0.1\%, 0.05\%, 0.01\%\} \tag{4.6}$$

$$E_{\omega=3\pi} = \{1\%, 0.5\%, 0.1\%, 0.05\%\} \tag{4.7}$$

We compare two different values of frequency ($\omega = \pi, 3\pi$) of the solution

shown in Equations 4.2. The higher frequency is expected to be more challenging for the neural network to learn because the shape of the surface is more complex than the lower frequency. The results for $\omega = \pi$ and 3π are shown in Table 4.2 and 4.3, respectively.

Table 4.2 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of the Lagaris and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = \pi$)										
Method	Lagaris method					Mixed residual method				
Error (%)	1	0.5	0.1	0.05	0.01	1	0.5	0.1	0.05	0.01
Average epoch	7.9	10.3	31.3	78.0	394.8	24.8	43.8	281.0	661.5	7959.7
Standard deviation	1.8	2.4	12.1	31.2	118.7	4.9	16.6	91.6	231.0	3391.6
Avg. time per epoch (s)	0.0884					0.0714				

Table 4.3 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Comparing the performance of the Lagaris and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = 3\pi$)								
Method	Lagaris method				Mixed residual method			
Error (%)	1	0.5	0.1	0.05	1	0.5	0.1	0.05
Average epoch	460.3	525.6	1136.1	5138.7	319.6	873.0	15787.0	42606.7
Standard deviation	708.3	780.1	1127.0	6136.2	103.3	226.5	5698.0	13922.0
Avg. time per epoch (s)	0.0870				0.0713			

For the average number of epochs, the results show that the values are lower for the Lagaris method across different frequencies ($\omega = \pi, 3\pi$) and relative errors except only when $E = 1\%$ with $\omega = 3\pi$. As relative error decreases, the results show that the performance of the Lagaris method improves substantially. For example, when $\omega = \pi$, the average number of epochs of the Lagaris method is almost 20 times greater when $E = 0.01\%$, but only 3 times greater when $E = 1\%$.

For the standard deviation, the results also show that values are lower for the Lagaris method across different frequencies ($\omega = \pi, 3\pi$) and relative errors except only

when $E = 1\%, 0.5\%$ with $\omega = 3\pi$. A higher standard deviation is undesirable. With higher spread of the average number of epochs, the number of epochs from only one result becomes less reliable as an indicator for the performance of a neural network, and more results are needed.

The average time per epoch when $\omega = \pi$ are 0.0884 and 0.0714 seconds, and when $\omega = 3\pi$ are 0.0870 and 0.0713 seconds for the Lagaris method and the mixed residual method, respectively. We consider the time to be comparable, and the average number of epochs can be used directly to measure the performance.

Combining the lower value of average number of epochs and standard deviation across different relative errors, the results show that the Lagaris method is more efficient than the mixed residual method in the case of exact boundary condition on a rectangular domain. We believe that the reason for the mixed residual method to perform worse than the Lagaris method is due to the fact that the term $\hat{p}(x, y)$ (Equation 3.31) must approximate the derivative of boundary and distance function directly (Equation 4.3 and 4.4). The shape of the derivative of boundary and distance function are much more complicated compared to the analytical solution. This difficulty probably is the reason for the inferior performance.

4.2.2 Inexact Boundary Condition : Rectangular Domain

For the inexact boundary condition on a rectangular domain, we compare the deep Galerkin method with the mixed residual method on a 2D Laplace's equation. The relative error norm that we choose are,

$$E_{\omega=\pi} = \{1\%, 0.5\%, 0.1\%, 0.05\%\} \tag{4.8}$$

$$E_{\omega=3\pi} = \{5\%, 1\%, 0.5\%, 0.2\%\} \tag{4.9}$$

We compare two different values of frequency ($\omega = \pi, 3\pi$) of the solution shown in Equations 4.2. The higher frequency is also expected to be more challenging

for the neural network to learn. The results for $\omega = \pi$ and 3π are shown in Table 4.4 and 4.5, respectively.

Table 4.4 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of the deep Galerkin and mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = \pi$)								
Method	Deep Galerkin method				Mixed residual method			
Error (%)	1	0.5	0.1	0.05	1	0.5	0.1	0.05
Average epoch	228.8	375.3	2045.4	4671.5	111.5	200.2	1273.6	3988.2
Standard deviation	72.5	107.5	950.7	2180.0	23.4	58.9	437.2	2095.4
Avg. time per epoch (s)	0.1024				0.0947			

Table 4.5 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Comparing the performance of the deep Galerkin and mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = 3\pi$)								
Method	Deep Galerkin method				Mixed residual method			
Error (%)	5	1	0.5	0.2	5	1	0.5	0.2
Average epoch	1024.4	7782.2	18938.2	69117.7	607.2	3867.1	8594.3	25681.7
Standard deviation	423.3	3878.5	10151.1	36660.6	86.2	1011.3	2205.6	9534.4
Avg. time per epoch (s)	0.1024				0.0910			

When $\omega = \pi$, the results for the average number of epochs and standard deviation show that the mixed residual method performs slightly better than the deep Galerkin method across different relative errors. On the other hand, when $\omega = 3\pi$, the mixed residual method outperforms the deep Galerkin method both on the average number of epochs and standard deviation. Moreover, the mixed residual method seems to perform better as the value of relative error gets lower (2.7 times when $E = 0.2\%$ compared to 1.7 times when $E = 5\%$).

The average time per epoch when $\omega = \pi$ are 0.1024 and 0.0947 seconds, and when $\omega = 3\pi$ are 0.1024 and 0.0910 seconds for the deep Galerkin method and the mixed residual method, respectively. We consider the time to be comparable.

We believe the reason for the superior performance of the mixed residual method is due to the derivative of the cost function. Due to the first-order systems of the PDE, we only need to include the first derivatives with respect to inputs in the cost function. The gradients of cost function with respect to weights and biases only involve second order derivatives of activation function which help reduce vanishing gradient problems coming from higher order derivatives of activation function.

4.2.3 Inexact Boundary Condition : Circular Domain

To investigate the deep Galerkin and mixed residual method on an irregular domain, we consider a 2D Laplace's equation on a circular domain with radius equal to 1. We choose the boundary condition such that the analytical solution is,

$$u(x, y) = (x^2 + y^2)^{m/2} \sin(m\theta) \tag{4.10}$$

when $\theta = \tan^{-1}(y/x)$.

We choose $m = 2, 3$ for the solution. The figures of the solution are shown in Figure 4.3 and 4.4 respectively. We use the same number of training points, testing points, batch size, and cyclical learning rate schedule with exponential decay policy as described in the problem setup section. The relative error norm that we choose are,

$$E_{m=2} = \{1\%, 0.5\%, 0.1\%, 0.05\%\} \tag{4.11}$$

$$E_{m=3} = \{5\%, 1\%, 0.5\%, 0.2\%\} \tag{4.12}$$

The results for $m = 2$ and $m = 3$ are shown in Table 4.6 and 4.7, respectively. When $m = 3$, the solution has more change on the surface, and is expected to be harder to learn.

When $m = 2$, the results for the average number of epochs show that the deep Galerkin method performs slightly better than the mixed residual method. Interest-

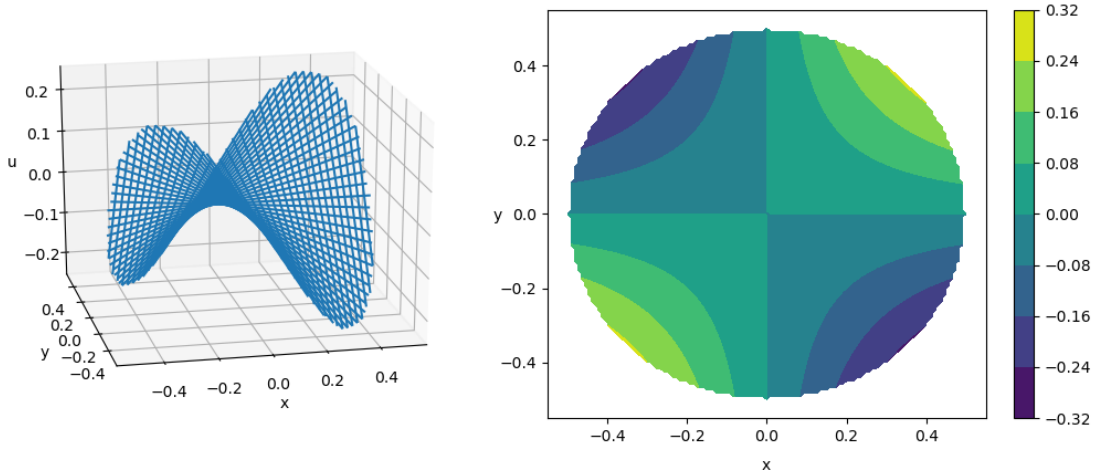

Figure 4.3 The graph of Equation 4.10 when $m = 2$.

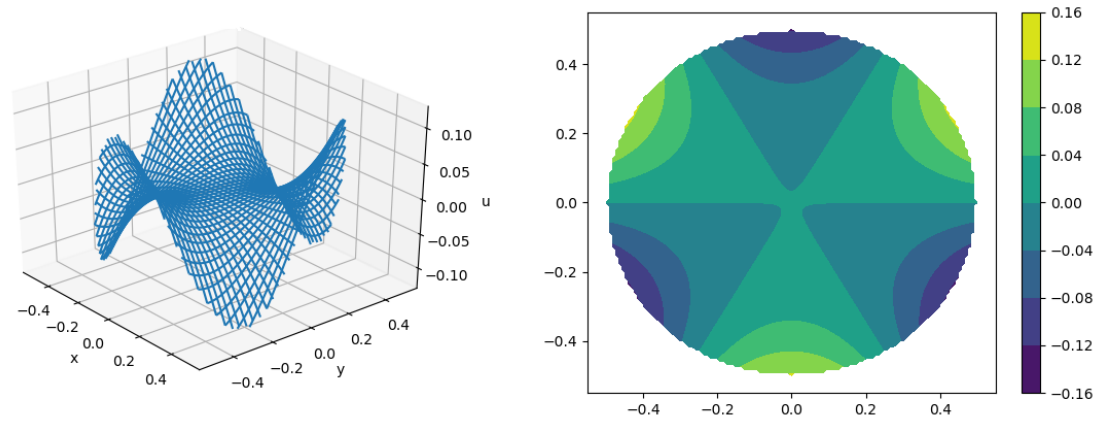

Figure 4.4 The graph of Equation 4.10 when $m = 3$.

ingly, the performance of the mixed residual method starts to catch up when the value of relative error gets lower. Surprisingly, the results for standard deviation show that the mixed residual method perform better than the deep Galerkin method especially at lower relative errors (e.g. $E = 0.05\%$).

On the contrary, when $m = 3$, the results for the average number of epochs and standard deviation show that the mixed residual method outperforms the deep Galerkin method across all different errors, and also performs better the lower value of relative error is.

The average time per epoch when $m = 2$ are 0.0910 and 0.0795 seconds,

Table 4.6 Inexact boundary condition (Circular domain, $m = 2$) : Comparing the performance of the deep Galerkin and mixed residual method.

Performance of the neural network (Circular domain, $m = 2$)								
Method	Deep Galerkin method				Mixed residual method			
Error (%)	1	0.5	0.1	0.05	1	0.5	0.1	0.05
Average epoch	48.5	138.9	1197.2	5399.4	121.1	222.5	2549.2	5721.1
Standard deviation	47.7	133.4	2357.3	8779.6	37.5	126.8	1762.5	2968.7
Avg. time per epoch (s)	0.0910				0.0795			

Table 4.7 Inexact boundary condition (Circular domain, $m = 3$) : Comparing the performance of the deep Galerkin and mixed residual method.

Performance of the neural network (Circular domain, $m = 3$)								
Method	Deep Galerkin method				Mixed residual method			
Error (%)	5	1	0.5	0.2	5	1	0.5	0.2
Average epoch	465.0	1365.9	2881.3	15395.2	449.8	1094.2	1849.3	4593.5
Standard deviation	140.7	470.4	2023.4	30900.4	122.1	346.2	642.1	2028.7
Avg. time per epoch (s)	0.0939				0.0886			

and when $m = 3$ are 0.0939 and 0.0886 seconds for the deep Galerkin method and the mixed residual method, respectively. We consider the time to be comparable.

On a circular domain, the results suggest that the performance of the mixed residual method depends on the shape of the solution. For the solution with simpler shape, the mixed residual method seems to perform slightly worse. But when the shape of the solution gets more complex, the mixed residual method outperforms the deep Galerkin method. For the standard deviation, the mixed residual method performs better than the deep Galerkin method in all cases.

4.2.4 Inexact Boundary Condition : Star Shape Domain

To further investigate the deep Galerkin and mixed residual method on an irregular domain, we consider a 2D Poisson's equation in a star shape domain,

$$\nabla^2 u(x, y) = f \tag{4.13}$$

We choose the boundary condition and function f such that the analytical solution is,

$$u(x, y) = e^{-(2x^2+4y^2)} + 0.5 \tag{4.14}$$

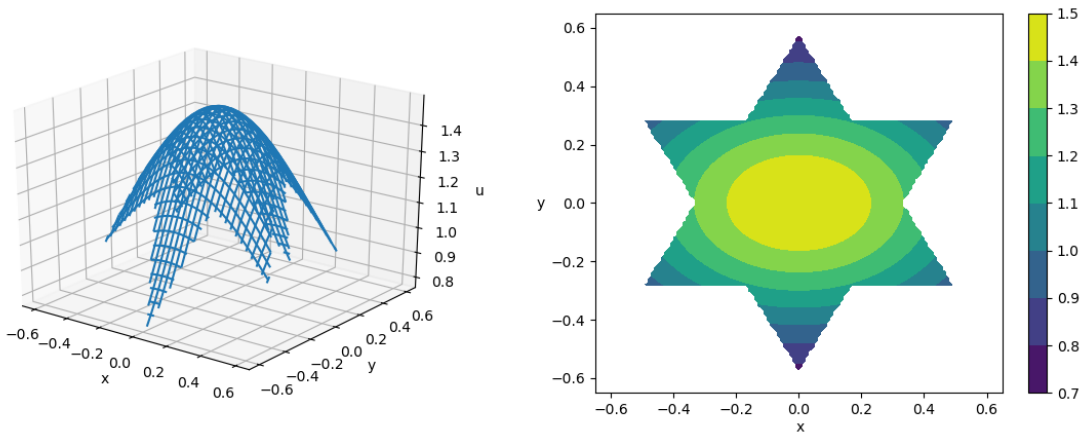

Figure 4.5 The graph of an analytical solution used on the star shape domain.

The figures of the solution are shown in Figure 4.5. We use the same number of training points, testing points, batch size, and cyclical learning rate schedule with exponential decay policy as described in the problem setup section. The result for both neural network methods is shown in Table 4.8. The relative error norm that we choose are,

$$E = \{1\%, 0.5\%, 0.1\%, 0.05\%\} \tag{4.15}$$

Table 4.8 Inexact boundary condition (Star shape domain) : Comparing the performance of the deep Galerkin and mixed residual method.

Performance of the neural network (Star shape domain)								
Method	Deep Galerkin method				Mixed residual method			
Error (%)	1	0.5	0.1	0.05	1	0.5	0.1	0.05
Average epoch	53.3	97.7	485.5	1227.9	47.6	117.1	567.3	1332.1
Standard deviation	31.9	45.4	281.3	696.0	20.0	60.6	220.4	639.3
Avg. time per epoch (s)	0.0966				0.0800			

The results for the average number of epochs show that the deep Galerkin method performs slightly better than the mixed residual method except when $E = 1\%$. On the contrary, the results for standard deviation show that the mixed residual method performs better than the deep Galerkin method except when $E = 0.5\%$.

The average time per epoch are 0.0966 and 0.0800 seconds for the deep Galerkin method and the mixed residual method, respectively. We consider the time to be comparable.

On a star shape domain, the results seem to be consistent with the results of the circular domain. When the shape of the solution is simple, the deep Galerkin method will perform slightly better than the mixed residual method. But we expect the mixed residual method to outperform when the shape of the solution becomes more complex.

4.2.5 Relative Error Norm and Cost function

In this work, we use the relative error norm as the stopping criterion which requires the analytical solution for the calculation. But in many cases, we won't always have the analytical solution, and would need a new stopping criterion. One choice of stopping criteria is cost function.

In this section, we present the average cost values of each relative error norm. For the exact boundary condition in a rectangular domain, the results for $\omega = \pi$ and 3π

are shown in Table 4.9 and 4.10. For the inexact boundary condition on a rectangular domain, the results for $\omega = \pi$ and 3π are shown in Table 4.11 and 4.12.

The results show that lower value of relative error corresponds to lower average cost value. Therefore, by using cost function instead of relative error as a stopping criterion, we expect the results of optimal hyperparameters would be the same. One of the problems when using cost function is how to determine the appropriate cost value to get the desirable value of relative error. Different neural network methods or analytical solutions would have different values of cost function for specific values of relative error.

Table 4.9 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Average cost values of the Lagaris method and the mixed residual method.

Average cost (Rectangular domain, $\omega = \pi$)					
Error (%)	1	0.5	0.1	0.05	0.01
Average cost (Lagaris method)	6.7×10^{-2}	2.6×10^{-2}	2.1×10^{-3}	6.6×10^{-4}	4.2×10^{-5}
Average cost (Mixed residual method)	7.9×10^{-3}	2.3×10^{-3}	1.9×10^{-4}	6.2×10^{-5}	2.7×10^{-6}

Table 4.10 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Average cost values of the Lagaris method and the mixed residual method.

Average cost (Rectangular domain, $\omega = 3\pi$)				
Error (%)	1	0.5	0.1	0.05
Average cost (Lagaris method)	3.9×10^{-1}	1.6×10^{-1}	9.2×10^{-3}	1.8×10^{-3}
Average cost (Mixed residual method)	1.4×10^{-2}	3.7×10^{-3}	1.4×10^{-4}	4.8×10^{-5}

For the exact boundary condition, the results from Table 4.9 and 4.10 show that the average cost of the Lagaris method is roughly 10 times lower than the mixed

Table 4.11 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Average cost values of the deep Galerkin method and the mixed residual method.

Average cost (Rectangular domain, $\omega = \pi$)				
Error (%)	1	0.5	0.1	0.05
Average cost (Deep Galerkin method)	3.9×10^{-4}	1.4×10^{-4}	1.1×10^{-5}	3.5×10^{-6}
Average cost (Mixed residual method)	7.7×10^{-4}	3.2×10^{-4}	2.6×10^{-5}	7.5×10^{-6}

Table 4.12 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Average cost values of the deep Galerkin method and the mixed residual method.

Average cost (Rectangular domain, $\omega = 3\pi$)				
Error (%)	5	1	0.5	0.2
Average cost (Deep Galerkin method)	5.8×10^{-3}	3.8×10^{-4}	1.4×10^{-4}	4.8×10^{-5}
Average cost (Mixed residual method)	7.6×10^{-3}	6.5×10^{-4}	2.5×10^{-4}	7.0×10^{-5}

residual method across different relative errors. For the inexact boundary condition, the results from TTable 4.11 and 4.12 show that the deep Galerkin method has slightly lower average cost than the mixed residual method across different relative errors.

4.3 Optimal Hyperparameters Using Genetic Algorithm

In this section, we will find the optimal activation function, optimization algorithm, and weight initialization using the genetic algorithm for a 2D Laplace's equation on a rectangular domain both for exact and inexact boundary conditions. For the exact boundary condition, we use the Lagaris method. For the inexact boundary condition, we use the mixed residual method. We use the same relative error as the previous section.

We still use the neural network which has 4 hidden layers, and 30 neurons per layer.

The hyperparameters of the genetic algorithm used in this section are shown in Table 4.13. The search space for activation function, optimization algorithm, and weight initialization is shown in Table 4.14.

Table 4.13 Hyperparameters of a genetic algorithm.

Hyperparameters	
Total generations	10
Total population	50
Fitness value	$1/\text{epochs}$
Elitism	10%
Selection	Roulette wheel selection method
Mutation chance	20%

Table 4.14 Search space for the hyperparameters of the neural network.

Hyperparameters	
Activation function	Tanh, Sigmoid, Swish, Softplus, Mish, GELU, LiSHT
Optimization algorithm	Adamax, Adam, Nadam, SGD, Rmsprop, Adadelta, Adagrad
Weight initialization	He normal, He uniform, Glorot normal, Glorot uniform, Lecun normal, Lecun uniform, Random normal, Random uniform, Identity, Orthogonal

In the previous section on the performance of different neural network methods, it shows that as relative error gets lower, the larger the value of standard deviation. This means that the number of epochs from only one result is less reliable as an indicator for the performance of its hyperparameters. This makes brute-force search for the optimal hyperparameters highly inefficient. A genetic algorithm has an advantage such that on average the optimal hyperparameters would have a higher chance to be selected and

pass on to the next generation. As the number of generations increases, the proportion of the optimal hyperparameters in the population will also increase as well. Other hyperparameters that on average don't perform as well will die off. In the last generation, the results show both the hyperparameters with the greatest proportion in the population and the hyperparameters with the lowest number of epochs across all generations.

In addition, we also calculate the average time per epoch for different combinations of activation functions and optimization algorithms, as shown in Appendix A. The results show that the average time per epoch for different combinations of activation function and optimization algorithm is comparable. Therefore, we can use the number of epochs directly as an indicator for performance.

4.3.1 Average and Best Fitness Values

In this section, we show the average and best fitness values of each generation in the algorithm for different values of relative error. For the exact boundary condition, the results are shown in Figure 4.6 and 4.7 for $\omega = \pi$ and 3π, respectively. For the inexact boundary conditions, the results are shown in Figure 4.8 and 4.9 for $\omega = \pi$ and 3π, respectively. In the graph, the vertical axis is the logarithm with base 10 of the fitness values, and the horizontal axis is the number of generations.

The results show that for exact and inexact boundary conditions with different frequencies ($\omega = \pi, 3\pi$), the average fitness value increases steadily as the number of generations increases. This indicates that the population starts to converse toward the optimal hyperparameters. Interestingly, as the relative error gets lower, the best fitness values are acquired relatively late in the generations. It suggests that for the optimal hyperparameters to converse for low value of relative error, we may need more number of generations in the algorithm.

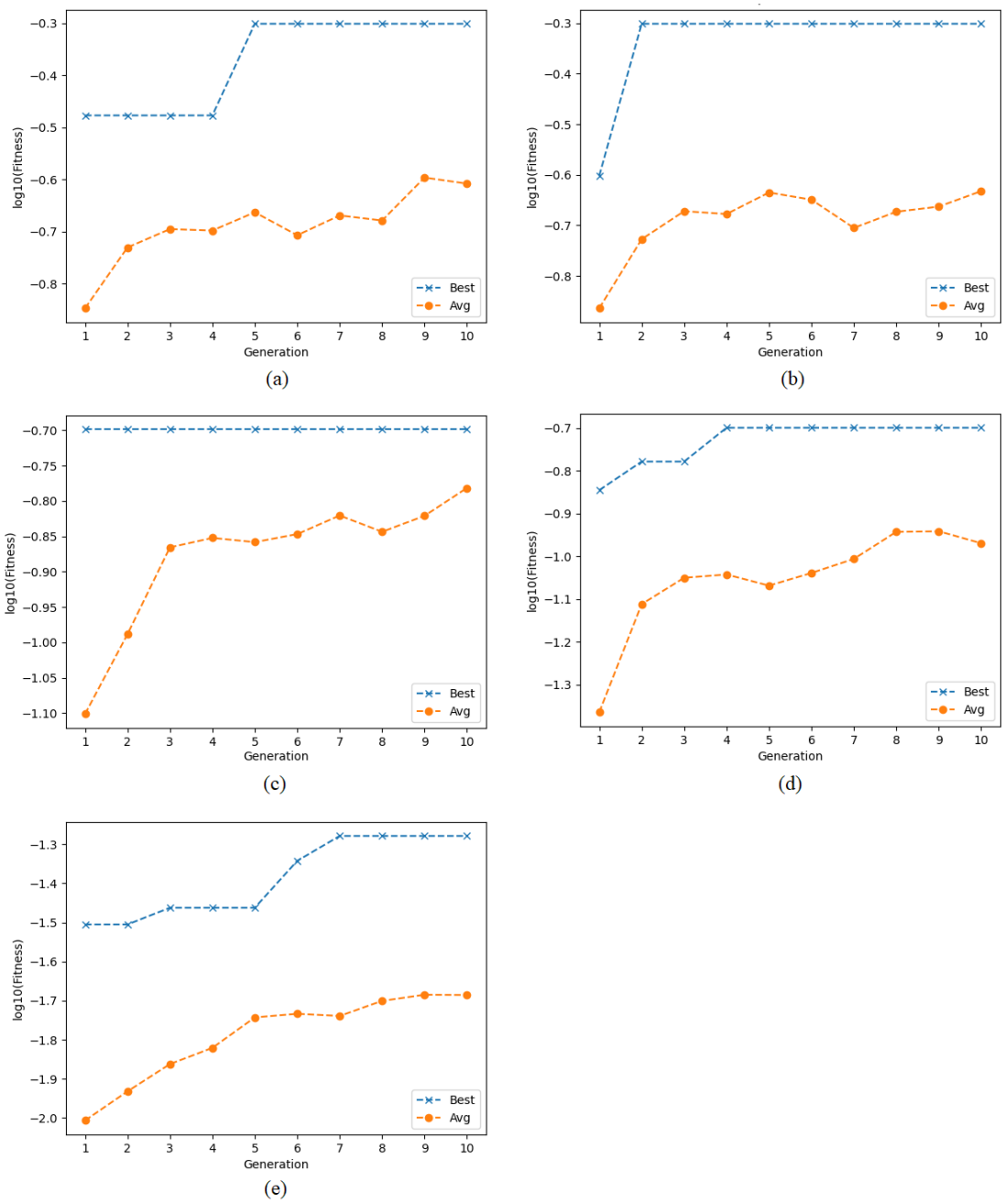

Figure 4.6 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between fitness value and number of generations when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%, 0.01\%$ for (a), (b), (c), (d), and (e), respectively.

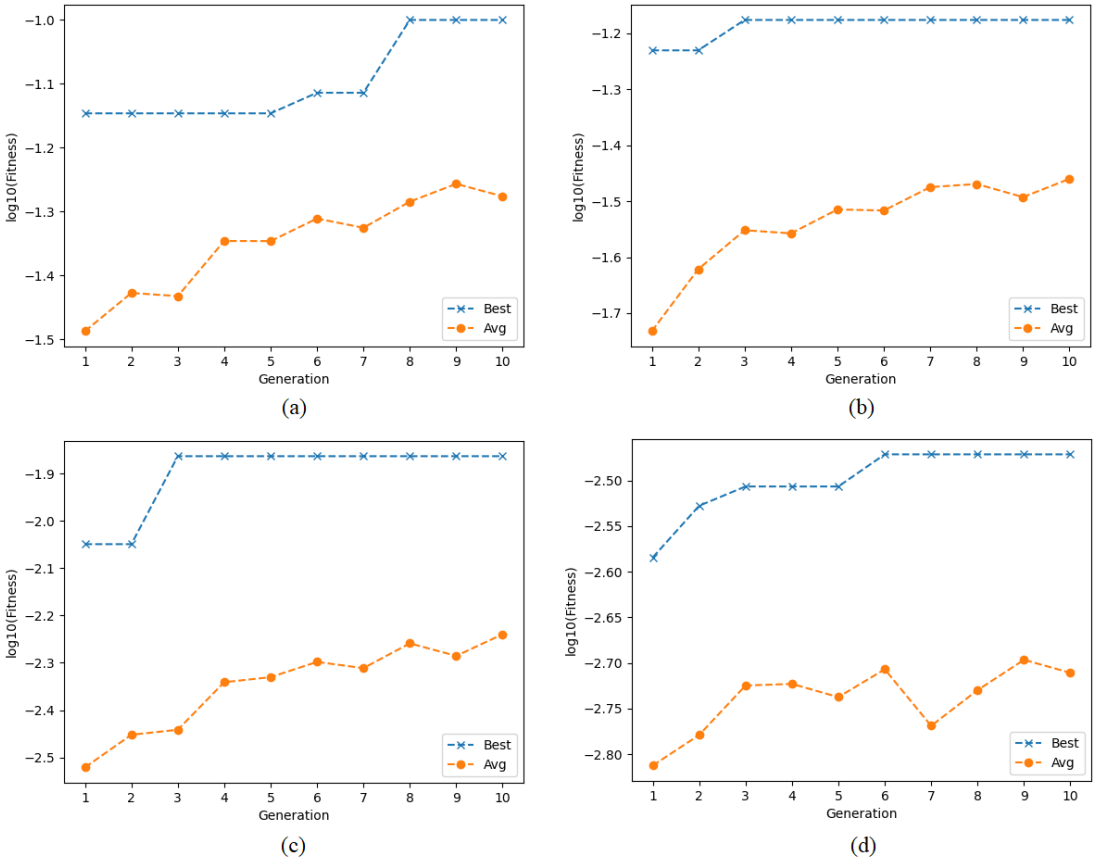

Figure 4.7 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between fitness value and number of generations when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

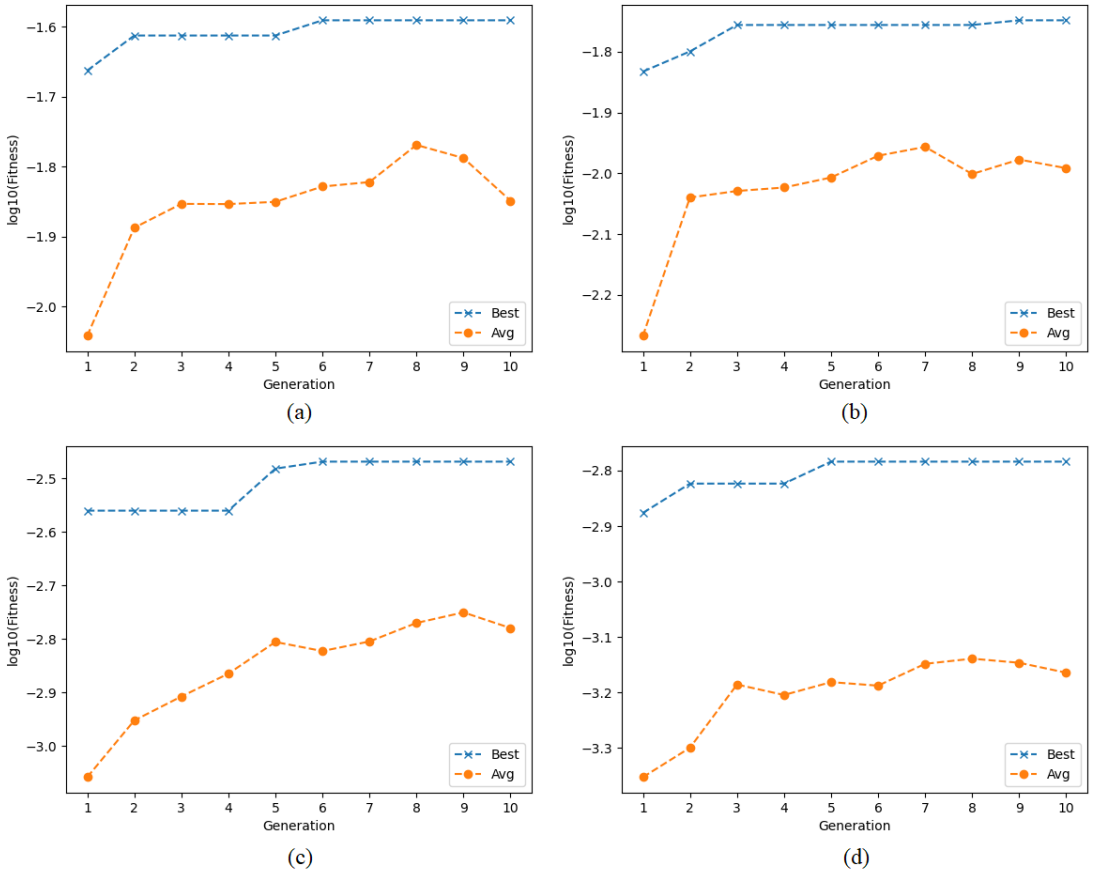

Figure 4.8 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between fitness value and number of generations when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

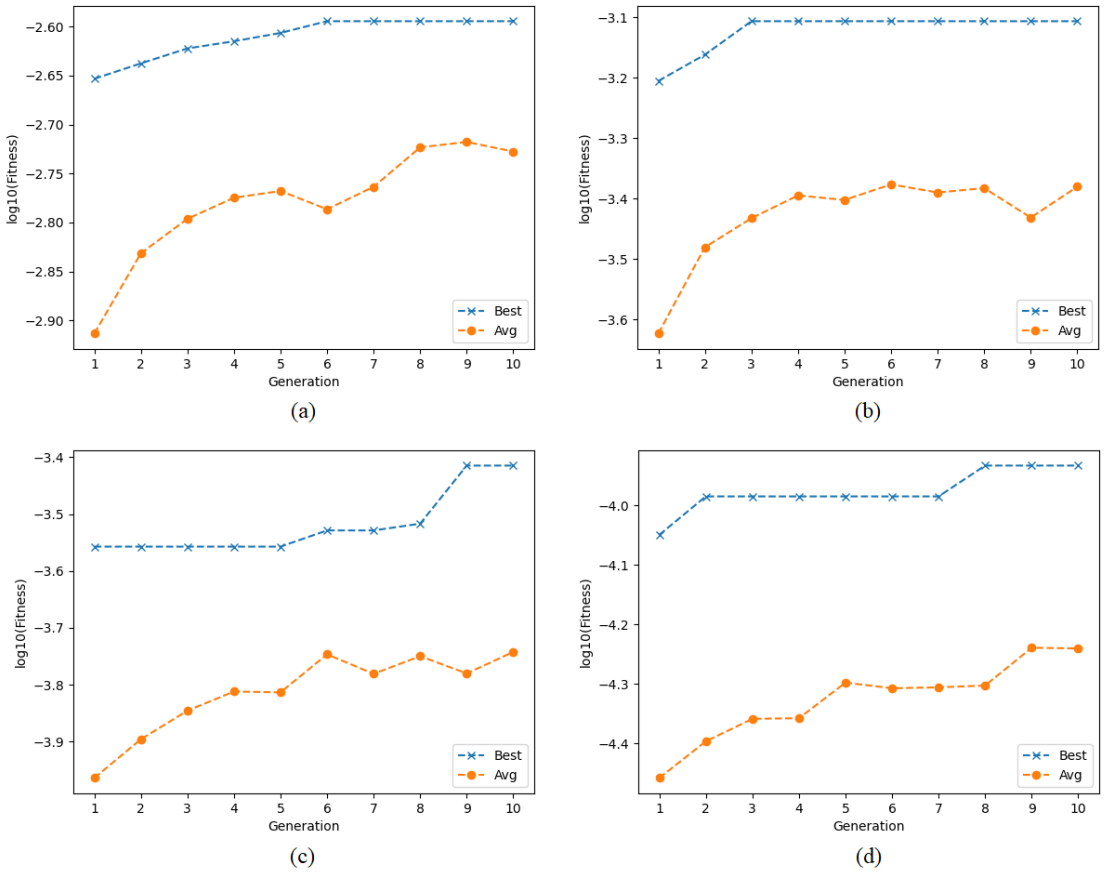

Figure 4.9 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between fitness value and number of generations when relative error equal to $5\%, 1\%, 0.5\%, 0.2\%$ for (a), (b), (c), and (d), respectively.

4.3.2 Exact Boundary Condition

In this section, we report the hyperparameters of the 1st, 2nd and 3rd best members, and the proportion of the hyperparameters of the last generation for the case of exact boundary condition with different relative errors. For $\omega = \pi$, the results are shown in Table 4.15 and 4.16. For $\omega = 3\pi$, the results are shown in Table 4.17 and 4.18.

For $\omega = \pi$, the hyperparameters of the best member and the ones with the greatest proportion are a GELU function, Nadam optimization algorithm, and He uniform initialization. The results are consistent across all relative errors. Interestingly, He normal initialization also performs quite well, and even has the greatest proportion when $E = 0.05\%$.

For $\omega = 3\pi$, the activation function of the best member and the ones with the greatest proportion is a Mish function. For optimization algorithms, Nadam and Adam optimization algorithms are the two that perform the best. Nadam performs better when relative error gets lower. For weight initialization, He uniform initialization is the hyperparameter of the best member and the ones with the greatest proportion except when $E = 0.05\%$ in which Lecun normal initialization performs better.

Table 4.15 Exact boundary condition (Rectangular domain, $\omega = \pi$) : The hyperparameters of the 1st, 2nd and 3rd best member of the last generation.

Hyperparameters of the 1st, 2nd and 3rd best member ($\omega = \pi$)						
		Error (%)				
		1	0.5	0.1	0.05	0.01
Activation function	1st	GELU	GELU	GELU	GELU	GELU
	2nd	Mish	GELU	GELU	GELU	GELU
	3rd	GELU	Swish	GELU	GELU	GELU
Optimization algorithm	1st	Nadam	Nadam	Nadam	Nadam	Nadam
	2nd	Nadam	Nadam	Nadam	Nadam	Nadam
	3rd	Nadam	Nadam	Nadam	Nadam	Nadam
Weight initialization	1st	He uniform	He normal	He uniform	He uniform	He uniform
	2nd	He uniform	He uniform	He normal	He normal	He normal
	3rd	He uniform	He uniform	Lecun normal	He uniform	He uniform
Epochs	1st	2	2	5	5	19
	2nd	2	3	5	6	21
	3rd	3	3	5	6	22

Table 4.16 Exact boundary condition (Rectangular domain, $\omega = \pi$) : The proportion of the hyperparameters of the last generation.

Proportion of the hyperparameters ($\omega = \pi$)					
	Error (%)				
	1	0.5	0.1	0.05	0.01
Activation function	GELU (48%) Swish (28%)	GELU (64%) Swish (28%)	GELU (94%) Sigmoid (4%)	GELU (86%) Swish (4%)	GELU (90%) LiSHT (6%)
Optimization algorithm	Nadam (82%) Adam (12%)	Nadam (96%) Adadelta (2%)	Nadam (94%) Adam (4%)	Nadam (100%) -	Nadam (96%) Adam (4%)
Weight initialization	He uniform (58%) Lecun uniform (20%)	He uniform (52%) He normal (26%)	He uniform (66%) He normal (32%)	He normal (66%) He uniform (20%)	He uniform (70%) He normal (18%)

Table 4.17 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : The hyperparameters of the 1st, 2nd and 3rd best member of the last generation.

Hyperparameters of the 1st, 2nd and 3rd best member ($\omega = 3\pi$)					
		Error (%)			
		1	0.5	0.1	0.05
Activation function	1st	Mish	Mish	Mish	Mish
	2nd	Swish	Swish	Mish	Mish
	3rd	Mish	Mish	Mish	Mish
Optimization algorithm	1st	Adam	Adam	Nadam	Nadam
	2nd	Adam	Nadam	Nadam	Nadam
	3rd	Adam	Nadam	Nadam	Nadam
Weight initialization	1st	He uniform	He uniform	He uniform	Lecun normal
	2nd	He normal	He uniform	He uniform	Lecun normal
	3rd	Lecun normal	He uniform	Lecun uniform	He uniform
Epochs	1st	10	15	73	296
	2nd	13	15	81	297
	3rd	14	16	97	321

Table 4.18 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : The proportion of the hyperparameters of the last generation.

Proportion of the hyperparameters ($\omega = 3\pi$)				
	Error (%)			
	1	0.5	0.1	0.05
Activation function	Mish (74%) Swish (14%)	Mish (60%) Swish (32%)	Mish (70%) GELU (24%)	Mish (92%) GELU (6%)
Optimization algorithm	Adam (96%) Nadam (2%)	Nadam (52%) Adam (44%)	Nadam (98%) Adamax (2%)	Nadam (92%) Adamax (4%)
Weight initialization	He uniform (38%) He normal (30%)	He uniform (90%) Glorot normal (6%)	He uniform (52%) Lecun uniform (28%)	Lecun normal (54%) He uniform (16%)

4.3.3 Inexact Boundary Condition

In this section, we report the hyperparameters of the 1st, 2nd and 3rd best members, and the proportion of the hyperparameters of the last generation for the case of inexact boundary condition with different relative errors. For $\omega = \pi$, the results are shown in Table 4.19 and 4.20, respectively. For $\omega = 3\pi$, the results are shown in Table 4.21 and 4.22, respectively.

For $\omega = \pi$, the hyperparameters of the best member and the ones with the greatest proportion are a GELU function, Adam optimization algorithm, and Lecun uniform initialization. The results are consistent across all relative errors.

For $\omega = 3\pi$, the activation function of the best member is a GELU function. But for the parameter with the greatest proportion, the results are inconclusive between a GELU function and a Mish function. For optimization algorithms, Adam optimization algorithm is the hyperparameter of the best member and the ones with the greatest proportion. For weight initialization, Lecun uniform initialization performs slightly better than Lecun normal initialization.

Interestingly, for the case of exact boundary conditions, the optimal weight initialization is He uniform initialization. For the case of inexact boundary condition, Lecun uniform is the optimal hyperparameter followed by Lecun normal. He and Lecun initialization use the same parameter which is the size of a neuron of the previous layer. The difference between the two is the constant factor (Equation $2.113 - 2.115$ and $2.107 - 2.109$). This reason could explain the similar performance of the two initializations. Moreover, Nadam and Adam optimization algorithms are the optimal hyperparameter for exact and inexact boundary conditions, respectively. Nadam is the extension of Adam by combining it with Nesterov accelerated gradient.

Table 4.19 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : The hyperparameters of the 1st, 2nd and 3rd best member of the last generation.

Hyperparameters of the 1st, 2nd and 3rd best member ($\omega = \pi$)					
		Error (%)			
		1	0.5	0.1	0.05
Activation function	1st	GELU	GELU	GELU	GELU
	2nd	GELU	GELU	LiSHT	LiSHT
	3rd	GELU	GELU	GELU	GELU
Optimization algorithm	1st	Adam	Adam	Adam	Adam
	2nd	Adam	Adam	Adam	Adam
	3rd	Adam	Adam	Adam	Adam
Weight initialization	1st	Lecun uniform	Lecun uniform	Lecun uniform	Lecun normal
	2nd	Lecun uniform	Lecun uniform	Lecun uniform	Glorot uniform
	3rd	Lecun uniform	Lecun uniform	Lecun uniform	Lecun uniform
Epochs	1st	39	56	294	608
	2nd	40	57	303	666
	3rd	41	59	305	687

Table 4.20 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : The proportion of the hyperparameters of the last generation.

Proportion of the hyperparameters ($\omega = \pi$)				
	Error (%)			
	1	0.5	0.1	0.05
Activation function	GELU (84%) LiSHT (6%)	GELU (84%) Mish (6%)	GELU (68%) LiSHT (18%)	GELU (64%) LiSHT (30%)
Optimization algorithm	Adam (90%) Nadam (6%)	Adam (96%) Rmsprop (4%)	Adam (96%) Nadam (2%)	Adam (98%) Adamax (2%)
Weight initialization	Lecun uniform (80%) Lecun normal (10%)	Lecun uniform (72%) Lecun normal (22%)	Lecun uniform (70%) Glorot uniform (8%)	Lecun uniform (38%) Lecun normal (30%)

Table 4.21 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : The hyperparameters of the 1st, 2nd and 3rd best member of the last generation.

Hyperparameters of the 1st, 2nd and 3rd best member ($\omega = 3\pi$)					
		Error (%)			
		5	1	0.5	0.2
Activation function	1st	LiSHT	GELU	GELU	GELU
	2nd	GELU	LiSHT	GELU	GELU
	3rd	GELU	GELU	GELU	Mish
Optimization algorithm	1st	Adam	Adam	Adam	Adam
	2nd	Adam	Adam	Adam	Adam
	3rd	Adam	Adam	Adam	Adam
Weight initialization	1st	Lecun normal	Lecun uniform	Lecun uniform	Lecun uniform
	2nd	Lecun normal	Glorot normal	Lecun uniform	Lecun normal
	3rd	Lecun normal	Lecun uniform	Lecun uniform	Lecun normal
Epochs	1st	393	1278	2599	8575
	2nd	395	1318	3175	9358
	3rd	396	1407	3288	9527

Table 4.22 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : The proportion of the hyperparameters of the last generation.

Proportion of the hyperparameters ($\omega = 3\pi$)				
	Error (%)			
	5	1	0.5	0.2
Activation function	Swish (36%) Mish (28%)	LiSHT (52%) GELU (40%)	GELU (46%) Mish (30%)	Mish (52%) GELU (40%)
Optimization algorithm	Adam (100%) -	Adam (94%) Nadam (6%)	Adam (94%) Nadam (4%)	Adam (96%) Nadam (2%)
Weight initialization	Lecun normal (56%) Lecun uniform (14%)	Lecun uniform (52%) Glorot normal (28%)	Lecun uniform (66%) Lecun normal (14%)	Lecun normal (50%) Lecun uniform (30%)

4.3.4 Performance of the Results : Exact Boundary Condition

In this section, we use the optimal activation function, optimization algorithm, and weight initialization from the results of a genetic algorithm. These hyperparameters will be used to find the average epochs and standard deviation to compare how well they actually perform. For the exact boundary condition, we use Nadam optimization algorithm, and He uniform initialization. But for activation function, we use a GELU function when $\omega = \pi$ and a Mish function when $\omega = 3\pi$. The results for $\omega = \pi$ and 3π are shown in Table 4.23 and 4.24, respectively.

Table 4.23 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of the Lagaris method and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = \pi$)					
Method	Lagaris method				
Error (%)	1	0.5	0.1	0.05	0.01
Average epoch	3.8	4.3	5.9	9.6	53.3
Standard deviation	1.0	0.9	1.3	4.9	19.4
Avg. time per epoch (s)	0.1119				

Table 4.24 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Comparing the performance of the Lagaris method and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = 3\pi$)				
Method	Lagaris method			
Error (%)	1	0.5	0.1	0.05
Average epoch	17.8	30.3	229.9	581.3
Standard deviation	5.3	11.6	76.8	139.1
Avg. time per epoch (s)	0.1072			

When comparing to Table 4.2 and Table 4.3 which use a Swish function, Adam optimization algorithm, and Glorot uniform initialization, the results show that

the optimal hyperparameters help reduce the average number of epochs and standard deviation significantly. The results are consistent across different frequencies ($\omega = \pi$, 3π) and relative errors. In conclusion, the optimal hyperparameters help with the performance and stability (lower standard deviation) of the training of the neural network.

4.3.5 Performance of the Results : Inexact Boundary Condition

In this section, we also use the optimal activation function, optimization algorithm, and weight initialization from the results of a genetic algorithm. For the inexact boundary condition, we use Adam optimization algorithm, and Lecun uniform initialization both for $\omega = \pi$ and 3π. For activation function, we use a GELU function when $\omega = \pi$. When $\omega = 3\pi$, due to inconclusiveness of the results, we will test both GELU function and Mish function to see how they compare. The results for $\omega = \pi$ is shown in Table 4.25. The results for $\omega = 3\pi$ using a GELU function and a Mish function are shown in Table 4.26 and 4.27, respectively.

Table 4.25 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of the deep Galerkin method and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = \pi$)				
Method	Mixed residual method			
Error (%)	1	0.5	0.1	0.05
Average epoch	61.0	94.5	589.3	1542.3
Standard deviation	9.7	17.9	164.4	753.2
Avg. time per epoch (s)	0.1113			

For $\omega = \pi$, when comparing to Table 4.4, the results show that the optimal hyperparameters have lower value of the average number of epochs and standard deviation. But the effect is not as great as the case of the exact boundary condition.

For $\omega = 3\pi$, when comparing to Table 4.5, the results show that for the average number of epochs a GELU function and a Mish function perform better. Moreover,

Table 4.26 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$, GELU) : Comparing the performance of the deep Galerkin method and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = 3\pi$, GELU)				
Method	Mixed residual method			
Error (%)	5	1	0.5	0.2
Average epoch	479.0	2346.8	5570.0	17252.8
Standard deviation	138.6	1174.7	2484.2	8023.8
Avg. time per epoch (s)			0.1159	

Table 4.27 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$, Mish) : Comparing the performance of the deep Galerkin method and the mixed residual method.

Performance of the neural network (Rectangular domain, $\omega = 3\pi$, Mish)				
Method	Mixed residual method			
Error (%)	5	1	0.5	0.2
Average epoch	548.4	2918.4	6580.1	20046.3
Standard deviation	90.8	786.7	1703.4	6068.5
Avg. time per epoch (s)			0.1065	

a GELU function has a substantially lower value of the average number of epochs than a Mish function. On the contrary, only a Mish function has a lower value of standard deviation across different relative errors when compared to Table 4.5.

4.4 Differents Learning Rate Approaches

In this section, we will compare the performance of different approaches used to set the learning rate. We will solve 2D Laplace's equation on a rectangular domain when $\omega = \pi$, and use the Lagaris method. The results are calculated using 50 randomly initialized neural networks. We use the neural network with a GELU activation function, Nadam optimization algorithm, and He uniform initialization. In this

section we compared three separate approaches which are cyclical learning rate with exponential decay, exponential decay, and fixed learning rate value. For the first two, we set the maximum learning rate to be 10^{-2}, and the minimum to be 10^{-4}. The results are shown in Table 4.28. The results show that cyclical learning rate with exponential decay performs the best, and followed by exponential decay. The results confirm that the cyclical learning rate helps improve the performance of the neural network.

Table 4.28 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different approaches used to set the learning rate.

Average number of epochs (Rectangular domain, $\omega = \pi$)					
Method	Lagaris method				
Error (%)	1	0.5	0.1	0.05	0.01
Cyclical learning rate with exponential decay	3.8	4.3	5.9	9.6	53.3
Exponential decay	2.4	2.8	16.0	31.5	227.0
Fixed value : 10^{-2}	5.1	8.6	36.0	71.1	455.5
Fixed value : 5×10^{-3}	2.8	3.6	12.6	19.1	444.6
Fixed value : 10^{-3}	6.9	10.5	143.8	211.1	548.4
Fixed value : 5×10^{-4}	7.1	9.5	96.2	375.5	1166.7
Fixed value : 10^{-4}	60.0	82.6	194.0	288.8	2203.2
Fixed value : 5×10^{-5}	171.8	221.8	407.3	530.6	3766.0
Fixed value : 10^{-5}	983.8	1109.0	1432.9	1600.2	2387.2
Fixed value : 5×10^{-6}	2089.5	2288.2	2774.4	3032.3	4197.1

4.5 Hyperparameters of Optimization Algorithm

In this section, we compare the performance of different hyperparameters of optimization algorithms. We will solve 2D Laplace's equation on a rectangular domain when $\omega = \pi$, and use the Lagaris method. The results are calculated using 50 randomly initialized neural networks. We use the neural network with a GELU activation function, and He uniform initialization. We will vary the hyperparameters of different optimization

algorithms to see how well they perform compared to default value.

4.5.1 Nesterov accelerated Adaptive Moment Estimation (Nadam)

There are two hyperparameters for the Nadam optimization algorithm which are β_1 and β_2. β_1 is the exponential decay rate for the 1st moment estimates, β_2 is the exponential decay rate for the exponentially weighted infinity norm, and the default values are 0.9 and 0.999, respectively. The results are shown in Table 4.29. The results show that changing the values of hyperparameters does not significantly help improve the performance.

Table 4.29 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of Nadam optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)						
Method		Lagaris method				
Error (%)		1	0.5	0.1	0.05	0.01
$\beta_1 = 0.9$,	$\beta_2 = 0.999$	3.8	4.3	5.9	9.6	53.3
$\beta_1 = 0.995$,	$\beta_2 = 0.999$	10.6	16.0	62.2	111.6	455.1
$\beta_1 = 0.98$,	$\beta_2 = 0.999$	5.1	6.1	14.7	21.9	70.5
$\beta_1 = 0.85$,	$\beta_2 = 0.999$	3.2	3.9	6.0	9.1	60.9
$\beta_1 = 0.8$,	$\beta_2 = 0.999$	3.3	4.5	6.1	11.6	72.2
$\beta_1 = 0.75$,	$\beta_2 = 0.999$	3.8	4.5	6.1	11.0	74.5
$\beta_1 = 0.9$,	$\beta_2 = 0.9$	5.7	6.4	12.1	15.7	48.9
$\beta_1 = 0.9$,	$\beta_2 = 0.8$	6.8	7.4	11.2	15.8	43.0
$\beta_1 = 0.9$,	$\beta_2 = 0.7$	7.1	7.6	10.7	16.5	43.7
$\beta_1 = 0.9$,	$\beta_2 = 0.6$	6.9	7.6	12.2	15.8	55.4
$\beta_1 = 0.9$,	$\beta_2 = 0.5$	7.0	7.6	13.0	16.3	58.4

4.5.2 Adaptive Moment Estimation (Adam)

There are three hyperparameters for the Nadam optimization algorithm which are β_1, β_2, and amsgrad. β_1 is the exponential decay rate for the 1st moment estimates, β_2 is the exponential decay rate for the 2nd moment estimates, and amsgrad will deter-

mine whether to apply the AMSGrad variant of the algorithm. The default values are 0.9,0.999, and amsgrad = False, respectively. The results are shown in Table 4.30. The results show that changing the value of $beta_2$ makes the performance worsen. Overall, the performance of Adam is still worse than Nadam with default hyperparameters.

Table 4.30 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of Adam optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)						
Method		Lagaris method				
Error (%)		1	0.5	0.1	0.05	0.01
$\beta_1 = 0.9$,	$\beta_2 = 0.999$	5.6	6.6	10.6	14.7	84.4
$\beta_1 = 0.995$,	$\beta_2 = 0.999$	33.7	80.0	268.3	405.7	1400.0
$\beta_1 = 0.98$,	$\beta_2 = 0.999$	12.3	19.6	41.9	55.5	167.8
$\beta_1 = 0.85$,	$\beta_2 = 0.999$	4.3	5.2	7.2	9.8	68.2
$\beta_1 = 0.8$,	$\beta_2 = 0.999$	3.8	4.7	6.6	9.1	71.3
$\beta_1 = 0.9$,	$\beta_2 = 0.9$	6.9	8.8	16.7	25.2	231.2
$\beta_1 = 0.9$,	$\beta_2 = 0.8$	7.4	10.0	21.4	35.7	352.5
$\beta_1 = 0.9$,	$\beta_2 = 0.7$	8.5	11.9	22.1	57.8	508.8
$\beta_1 = 0.9$, amsgrad = True	$\beta_2 = 0.999$	5.2	6.3	10.4	13.6	95.6

4.5.3 Root Mean Square Propagation (RMSprop)

There are two hyperparameters for the RMSprop optimization algorithm which are ρ and γ. ρ is the discounting factor for the coming gradient and γ is the momentum, and the default values are 0.9 and 0, respectively. The results are shown in Table 4.31. The results show that the network performs the best when $\rho = 0.9$ and $\gamma = 0.5 - 0.7$.

Table 4.31 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of RMSprop optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)						
Method		Lagaris method				
Error (%)		1	0.5	0.1	0.05	0.01
$\rho = 0.9$,	$\gamma = 0$	7.0	7.2	16.1	26.4	125.4
$\rho = 0.995$,	$\gamma = 0$	5.5	6.0	13.3	34.6	257.7
$\rho = 0.98$,	$\gamma = 0$	5.8	6.0	16.6	38.1	216.3
$\rho = 0.85$,	$\gamma = 0$	7.7	8.0	16.0	25.1	123.8
$\rho = 0.8$,	$\gamma = 0$	8.0	8.0	16.0	24.0	121.4
$\rho = 0.9$,	$\gamma = 0.9$	5.9	7.8	58.6	163.0	695.9
$\rho = 0.9$,	$\gamma = 0.8$	5.5	6.5	11.6	26.6	249.6
$\rho = 0.9$,	$\gamma = 0.7$	4.8	6.0	7.5	14.2	79.8
$\rho = 0.9$,	$\gamma = 0.6$	5.5	6.2	8.1	14.0	78.7
$\rho = 0.9$,	$\gamma = 0.5$	5.4	6.1	7.9	15.6	69.9
$\rho = 0.9$,	$\gamma = 0.4$	5.3	6.0	11.6	17.8	85.8

4.5.4 Adamax

There are two hyperparameters for the Nadam optimization algorithm which are β_1 and β_2. β_1 is the exponential decay rate for the 1st moment estimates, and β_2 is the exponential decay rate for the exponentially weighted infinity norm, and the default values are 0.9 and 0.999, respectively. The results are shown in Table 4.32. The results show that overall the Adamax performs worse than Nadam with default hyperparameters.

Table 4.32 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of Adamax optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)						
Method		Lagaris method				
Error (%)		1	0.5	0.1	0.05	0.01
$\beta_1 = 0.9,$	$\beta_2 = 0.999$	7.44	10.1	25.2	50.04	212.9
$\beta_1 = 0.995,$	$\beta_2 = 0.999$	102.2	174.8	722.1	987.3	1811.5
$\beta_1 = 0.98,$	$\beta_2 = 0.999$	24.6	34.3	64.1	85.4	254.9
$\beta_1 = 0.85,$	$\beta_2 = 0.999$	6.1	8.1	24.1	48.92	205.8
$\beta_1 = 0.8,$	$\beta_2 = 0.999$	5.4	6.7	23.4	46.6	207.5
$\beta_1 = 0.75,$	$\beta_2 = 0.999$	5.3	6.9	25.0	49.0	216.0
$\beta_1 = 0.9,$	$\beta_2 = 0.9$	8.82	11.4	20.5	41.0	395.8
$\beta_1 = 0.9,$	$\beta_2 = 0.8$	12.5	16.7	38.6	111.5	776.7
$\beta_1 = 0.9,$	$\beta_2 = 0.7$	19.6	32.44	277.8	573.9	1558.9

4.5.5 Stochastic gradient descent (SGD)

There are two hyperparameters for the SGD optimization algorithm which are γ, and Nesterov. γ is the momentum parameter, and Nesterov will determine whether to apply Nesterov momentum or not. The default value is 0, and Nesterov = False, respectively. SGD optimization algorithm has problem learning when relative error is equal to 0.01%. The results are shown in Table 4.33. The results show that using high values of momentum seem to help with the performance of the neural network.

Table 4.33 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of SGD optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)						
Method		Lagaris method				
Error (%)		1	0.5	0.1	0.05	0.01
$\gamma = 0$,	Nesterov = False	314.1	24.2	145.1	704.9	9177.1
$\gamma = 0.9$,	Nesterov = False	7.7	9.8	20.3	43.8	3864.3
$\gamma = 0.85$,	Nesterov = False	6.9	8.3	25.6	62.4	3802.7
$\gamma = 0.8$,	Nesterov = False	6.0	7.1	31.9	80.9	31514.7
$\gamma = 0.75$,	Nesterov = False	5.9	7.9	37.2	95.34	54405.6
$\gamma = 0.7$,	Nesterov = False	6.3	9.4	47.7	124.3	>100000
$\gamma = 0.9$,	Nesterov = True	5.3	6.8	20.6	45.4	1918.4
$\gamma = 0.85$,	Nesterov = True	5.7	7.1	26.5	65.1	13431.5
$\gamma = 0.8$,	Nesterov = True	6.2	8.1	36.7	95.1	31153.3
$\gamma = 0.75$,	Nesterov = True	5.8	9.2	45.9	120.4	45108.3
$\gamma = 0.7$,	Nesterov = True	6.4	9.7	51.5	140.6	>100000

4.5.6 Adaptive Gradient Algorithm (Adagrad)

There is one hyperparameter for the Adadelta optimization algorithm which is δ. δ is the starting value for the accumulators, and the default value is 0.1. The results are shown in Table 4.34. The results show that Adagrad performs substantially worse at relative error equal to 0.05%, and will perform worse when the value of δ increases.

Table 4.34 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of Adagrad optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)					
Method	Lagaris method				
Error (%)	1	0.5	0.1	0.05	0.01
$\delta = 0.1$	34.52	51.52	216.64	1276.12	>100000
$\delta = 0.2$	36.5	54.0	222.4	1774.8	>100000
$\delta = 0.3$	39.02	59.3	264.1	2491.1	
$\delta = 0.4$	43.4	65.7	319.6	4331.7	
$\delta = 0.5$	44.7	66.0	352.8	5537.6	

4.5.7 Adadelta

There is one hyperparameter for the Adadelta optimization algorithm which is ρ. ρ is the decay rate, and the default value is 0.95. The results are shown in Table 4.35. Adadelta performs the worst when compared to all other optimization algorithms in this work.

Table 4.35 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Comparing the performance of different hyperparameters of Adadelta optimization algorithm.

Average number of epochs (Rectangular domain, $\omega = \pi$)					
Method	Lagaris method				
Error (%)	1	0.5	0.1	0.05	0.01
$\rho = 0.95$	522.6	789.9	8206.3	24016.5	>100000
$\rho = 0.9$	1072.0	2417.8	16776.2	33415.3	>100000

4.6 Optimal Number of Hidden Layers and Total Number of Parameters

In this section, we will find the optimal number of hidden layers and total number of parameters for a 2D Laplace's equation on a rectangular domain both for exact and inexact boundary conditions. We use the Lagaris method and mixed residual method for exact and inexact boundary condition respectively. For the exact boundary condition, we use Nadam optimization algorithm, and He uniform initialization. But for activation function, we use a GELU function when $\omega = \pi$ and a Mish function when $\omega = 3\pi$. For the inexact boundary condition, we use GELU activation function, Adam optimization algorithm, and Lecun uniform initialization both for $\omega = \pi$ and 3π. We use the same relative error as the two previous sections. But for exact boundary condition whem $\omega = \pi$, we change the relative error to,

$$E_\pi = \{0.01\%, 0.005\%, 0.001\%\} \tag{4.16}$$

We change to lower relative errors because when using previous relative errors (Equation 4.6), the neural network will train too fast and the value of the average number of epochs will be too low to get meaningful results. The results are calculated using 50 randomly initialized neural networks.

4.6.1 Number of Hidden Layers : Average Time per Epoch

In the section of finding optimal number of hidden layers, a neural network is trained by varying the number of hidden layers while fixing the total number of parameters. We train two sets of neural networks with $2 - 9$ hidden layers. The first one has a total number of parameters roughly equal to 2900 parameters, and has 51, 36, 29, 25, 22, 20, 18, 16 neurons per layer respectively. The second one has a total number of parameters roughly equal to 14500 parameters which is 5 times greater, and has 117, 83, 68, 58, 52, 48, 44, 41 neurons per layer respectively. The average number of epochs are

calculated using 50 randomly initialized neural networks. The average time per epoch is calculated using 1000 epochs.

First, we would like to show the results of the average time per epoch, and discuss its importance. For the exact boundary condition when $\omega = \pi$ and 3π, the average time per epoch is shown in Figure 4.10. For the inexact boundary condition when $\omega = \pi, 3\pi$, the average time per epoch is shown in Figure 4.11.

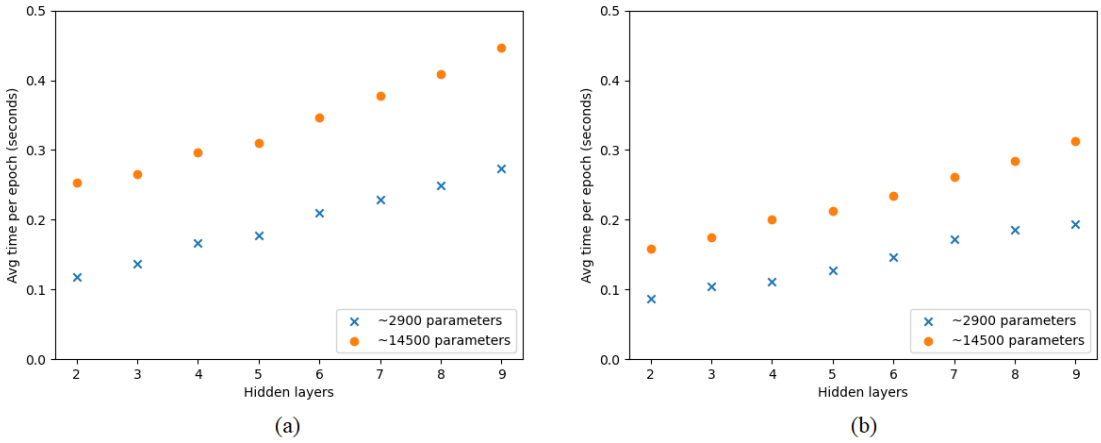

Figure 4.10 Exact boundary condition (Rectangular domain) : Average time per epoch for different hidden layers, (a) $\omega = \pi$. (b) $\omega = 3\pi$.

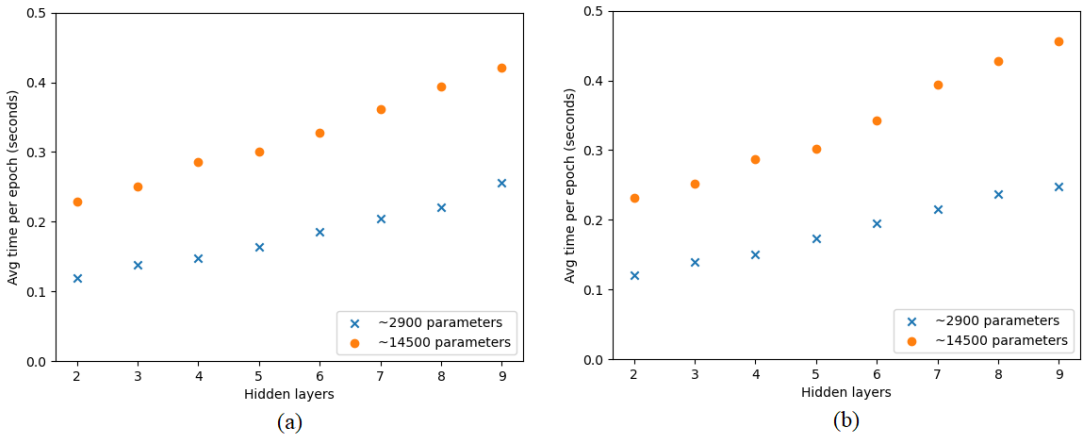

Figure 4.11 Inexact boundary condition (Rectangular domain) : Average time per epoch for different hidden layers, (a) $\omega = \pi$. (b) $\omega = 3\pi$.

In the cases of exact and inexact boundary conditions, the results show that, when the total number of parameters is fixed, the average time per epoch increases lin-

early with the number of hidden layers. Also at the same time when the number of hidden layers is fixed, the average time per epoch increases with the total number of parameters. The results suggest that we should take the average time per epoch into account when considering the best number of hidden layers. Therefore, in the next section, we will show the results with the average epoch and the average times (seconds),

$$\text{Average time} = \text{Average epoch*Average time per epoch} \tag{4.17}$$

4.6.2 Number of Hidden Layers : Exact Boundary Condition

In this section, we show the results of the average number of epochs and average time for the case of exact boundary condition. For $\omega = \pi$, the average number of epochs and the average time for different hidden layers are shown in Figure 4.12 and 4.13, respectively. For $\omega = 3\pi$, the average number of epochs and the average time for different hidden layers is shown in Figure 4.14 and 4.15, respectively.

For $\omega = \pi$, the results from a neural network with 2900 parameters show that the hidden layer with the lowest number of epochs are 6, 6, and 6 layers, and with the lowest average time are 5, 4, and 3 layers for $E = \{0.01\%, 0.005\%, 0.001\%\}$, respectively. For 14500 parameters, the best hidden layer with the lowest number of epochs are 4, 4, and 4 layers, and with the lowest average time are 4, 4, and 3 layers.

For $\omega = 3\pi$, the results from a neural network with 2900 parameters show that the hidden layer with the lowest number of epochs are 4, 4, 4, and 4 layers, and for the hidden layer with the lowest average time are 4, 4, 4, and 4 layers for $E = \{1\%, 0.5\%, 0.1\%, 0.05\%\}$, respectively. For 14500 parameters, the best hidden layer with the lowest number of epochs are 4, 3, 4, and 3 layers, and with the lowest average time are 3, 3, 3, and 3 layers.

At 2 hidden layers, the network is too shallow and cannot learn the data as well as the deep network. On the other hand, the vanishing gradient problem starts to occur at around 5 hidden layers and greater, making the performance of the network

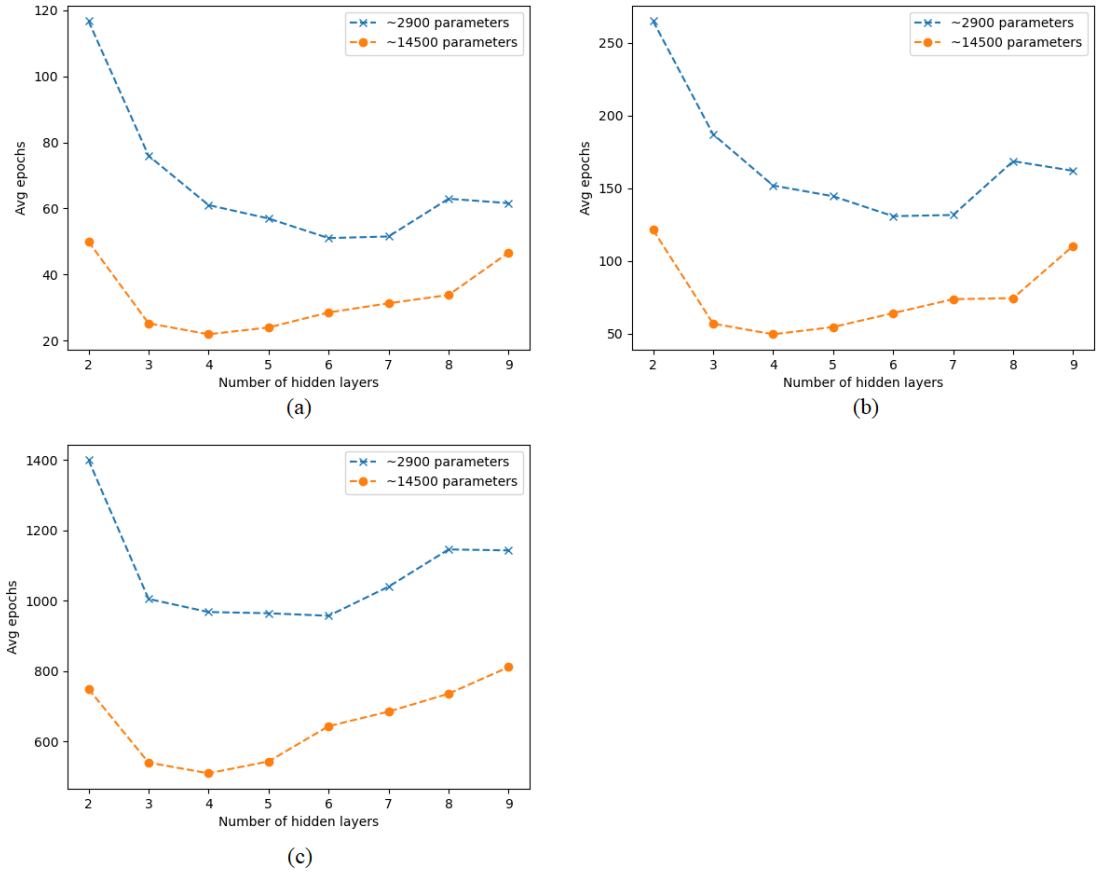

Figure 4.12 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average number of epochs and number of hidden layers when relative error equal to $0.01\%, 0.005\%, 0.001\%$ for (a), (b), and (c), respectively.

decrease substantially. When we look at the performance of a neural network with 14500 parameters, the average number of epochs is lower when compared to 2900 parameters across all hidden layers.

In summary, for both $\omega = \pi$ and 3π, the best hidden layers are in the range of $3 - 6$ layers. When taking the average time per epoch into consideration, the best hidden layer will shift to $3 - 4$ layers.

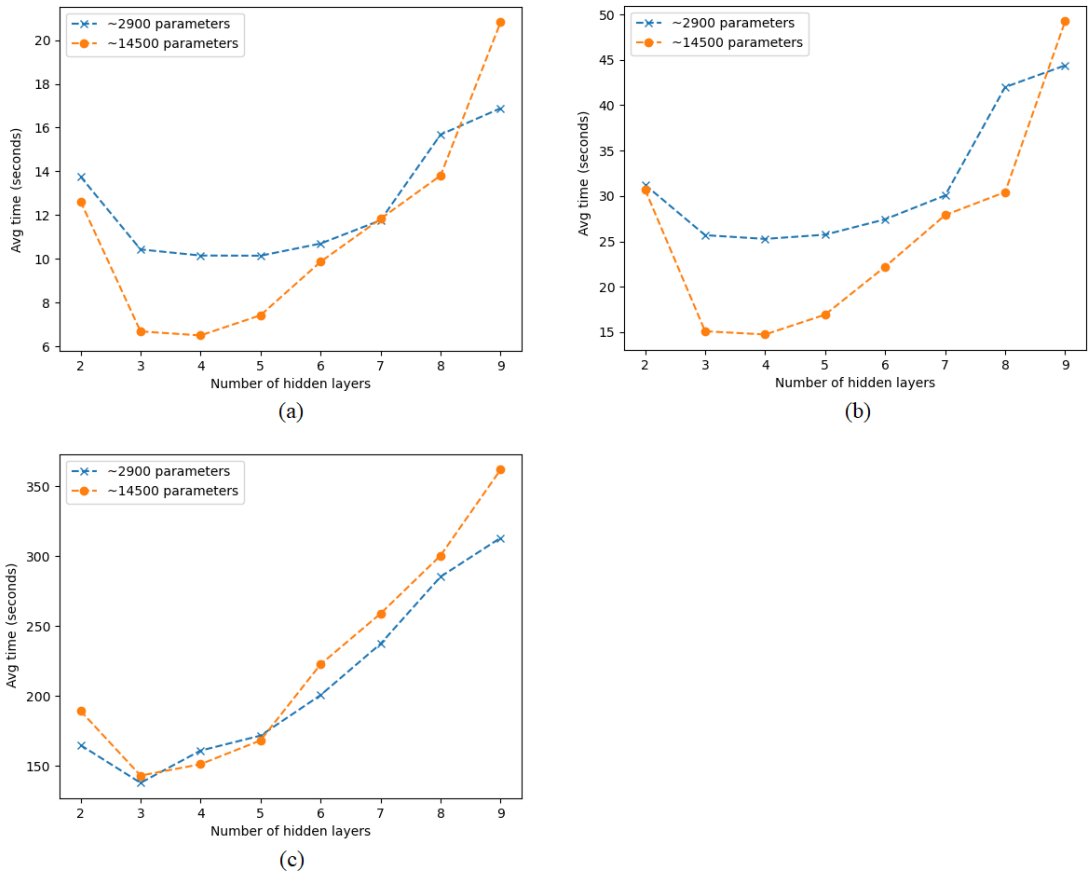

Figure 4.13 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average time and number of hidden layers when relative error equal to $0.01\%, 0.005\%, 0.001\%$ for (a), (b), and (c), respectively.

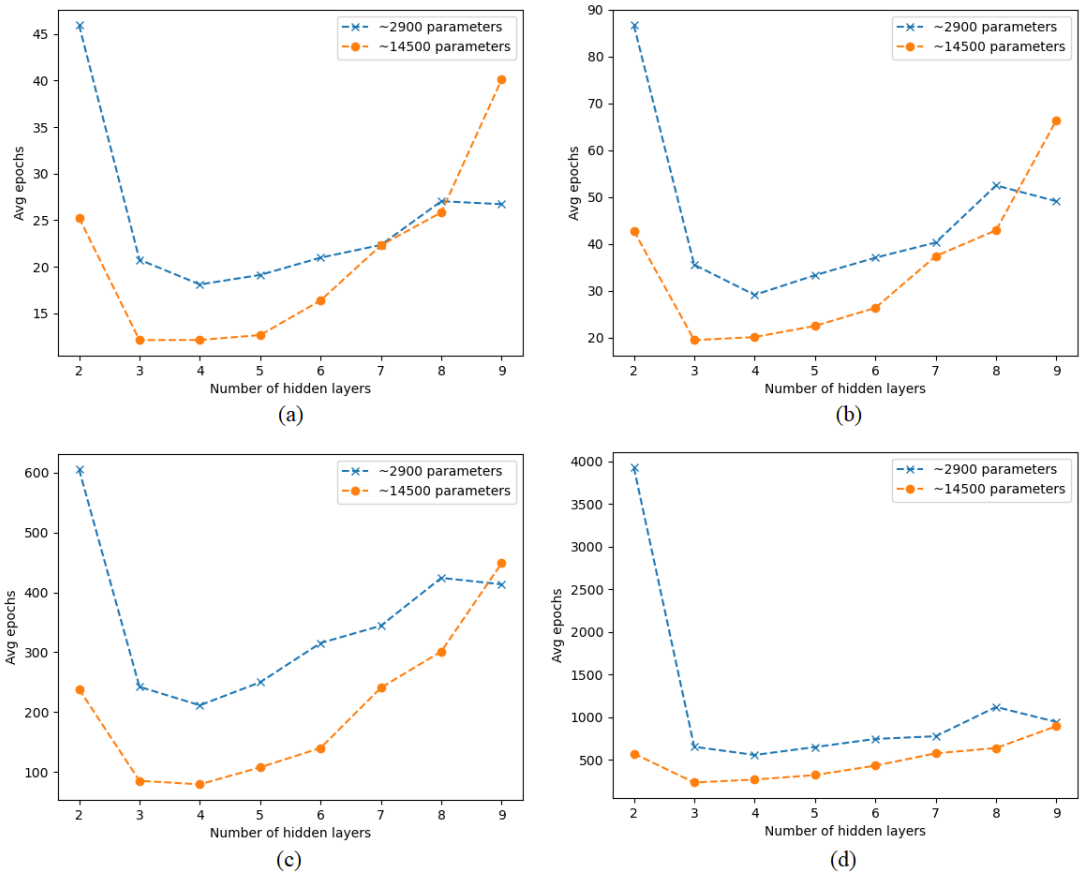

Figure 4.14 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average number of epochs and number of hidden layers when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

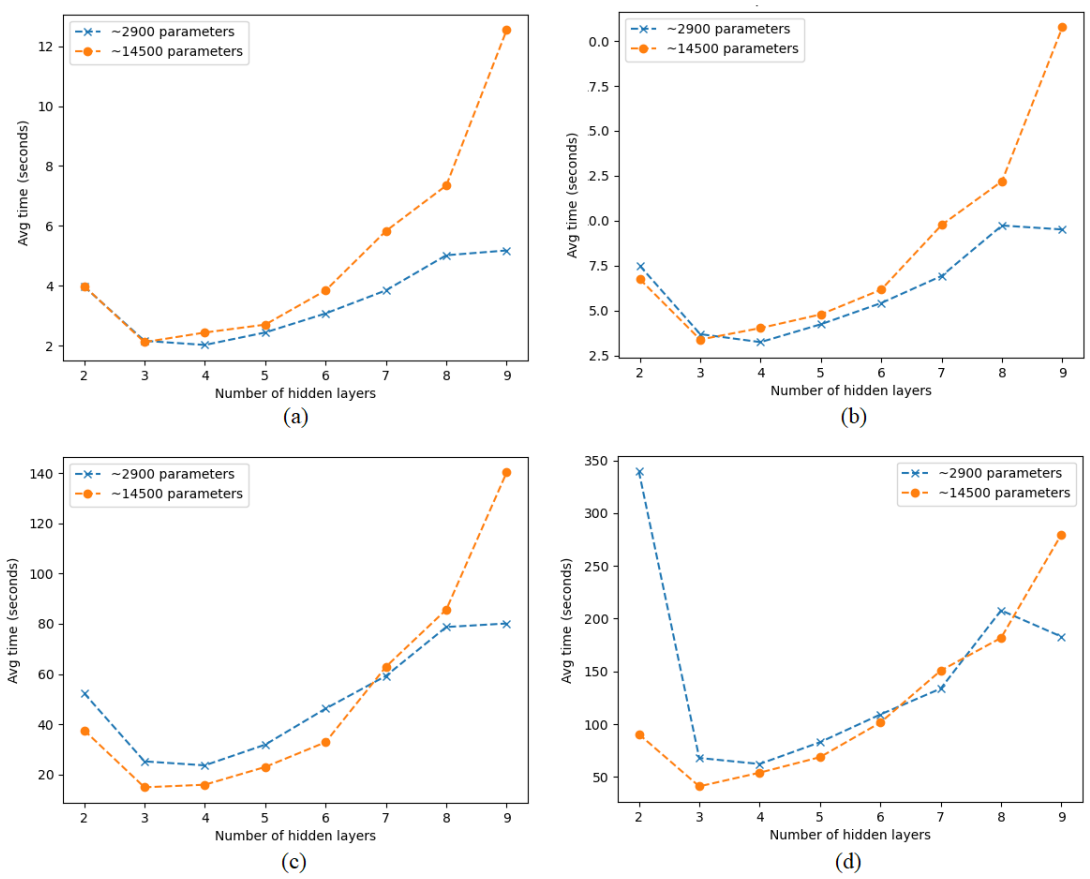

Figure 4.15 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average time and number of hidden layers when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

4.6.3 Number of Hidden Layers : Inexact Boundary Condition

In this section, we show the results of the average number of epochs and average time for the case of inexact boundary condition. For $\omega = \pi$, the average number of epochs and the average time for different hidden layers are shown in Figure 4.16 and 4.17, respectively. For $\omega = 3\pi$, the average number of epochs and the average time for different hidden layers is shown in Figure 4.18 and 4.19, respectively.

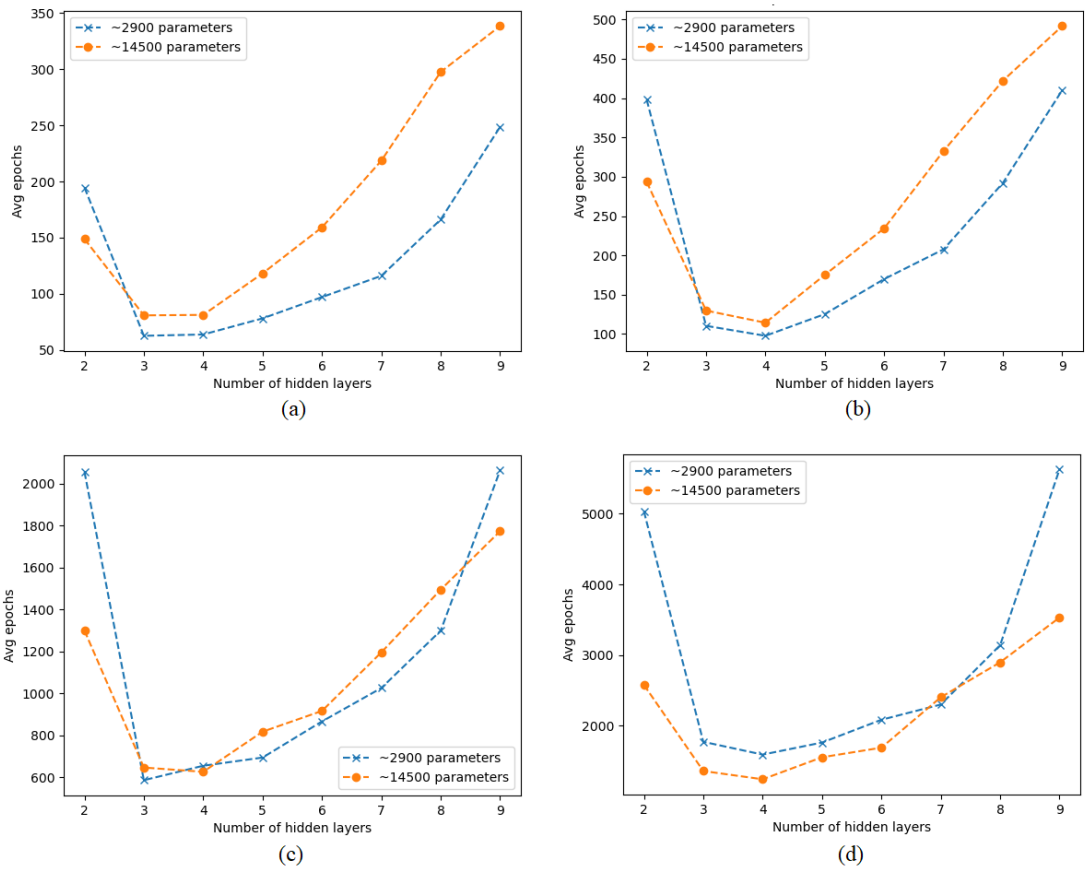

Figure 4.16 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average number of epochs and number of hidden layers when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

For $\omega = \pi$, the results from a neural network with 2900 parameters show that the hidden layer with the lowest number of epochs are 3, 4, 3, and 4 layers, and with the lowest average time are 3, 4, 3, and 4 layers for $E = \{1.0\%, 0.5\%, 0.1\%, 0.05\%\}$, respectively. For 14500 parameters, the best hidden layer with the lowest number of

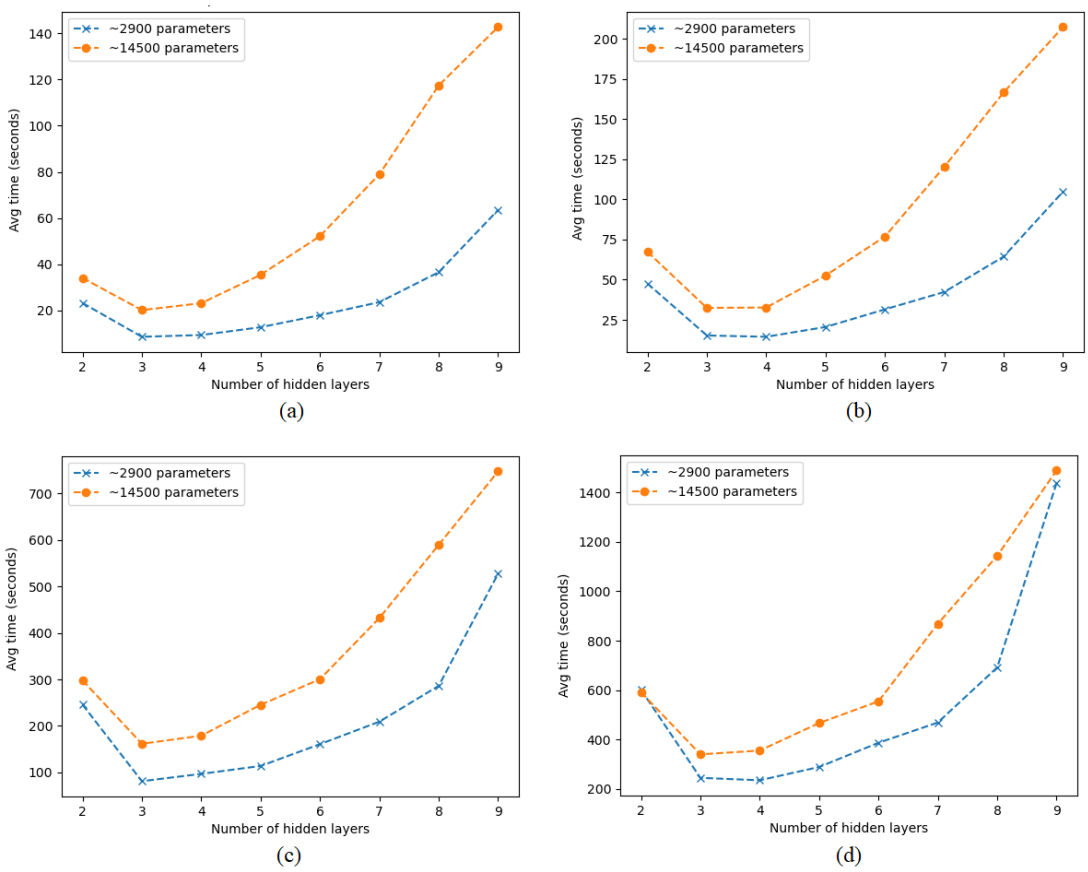

Figure 4.17 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average time and number of hidden layers when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

epochs are 3, 4, 4, and 4 layers, and with the lowest average time are 3, 3, 3, and 3 layers.

For $\omega = 3\pi$, the results from a neural network with 2900 parameters show that the hidden layer with the lowest number of epochs are 4, 5, 4, and 4 layers, and for the hidden layer with the lowest average time are 4, 4, 4, and 4 layers for $E = \{5.0\%, 1.0\%, 0.5\%, 0.2\%\}$, respectively. For 14500 parameters, the best hidden layer with the lowest number of epochs are 3, 4, 4, and 4 layers, and with the lowest average time are 3, 3, 4, and 4 layers.

At 2 hidden layers, the network is also too shallow and cannot learn the data as well. The vanishing gradient problem starts to occur at around 5 hidden layers and greater. When we look at the performance of a neural network with 14500 parameters,

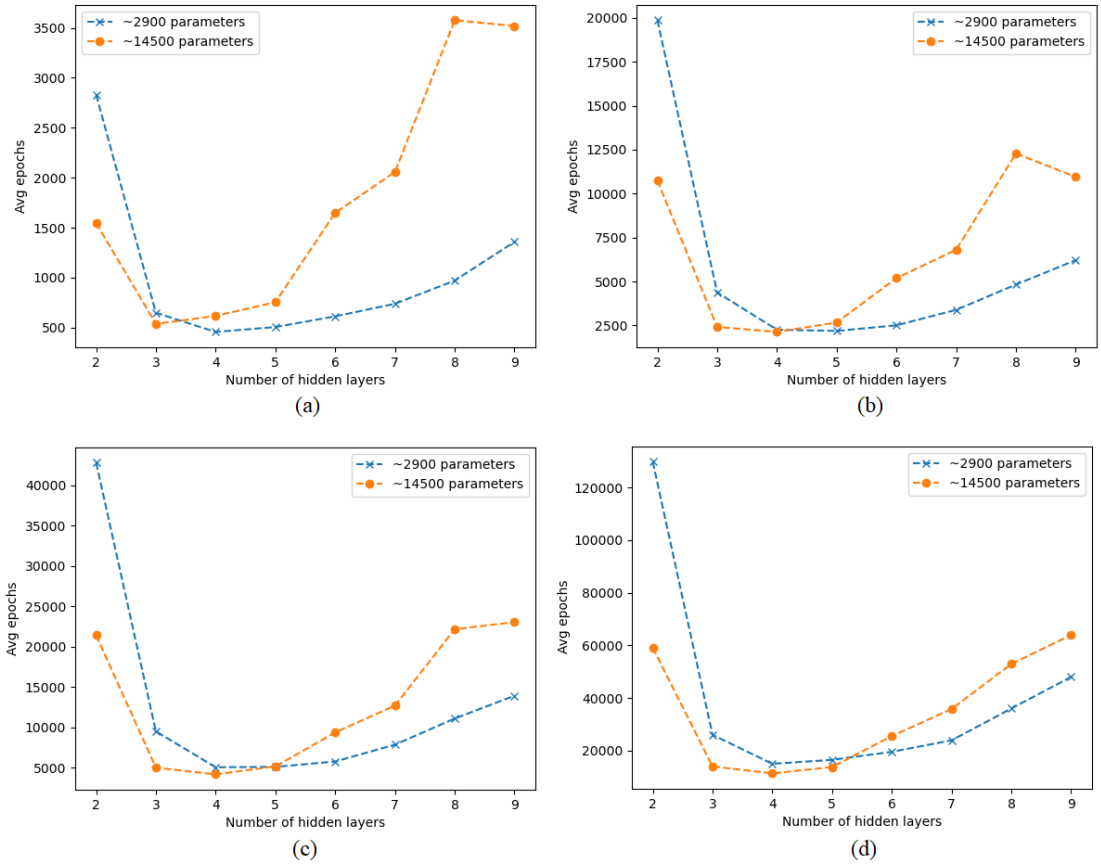

Figure 4.18 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average number of epochs and number of hidden layers when relative error equal to $5.0\%, 1.0\%, 0.5\%, 0.2\%$ for (a), (b), (c), and (d), respectively.

surprisingly the average number of epochs is higher when compared to 2900 parameters especially when the number of hidden layers is larger than 5 layers. The results seem to suggest that as we increase the total number of parameters at least for the mixed residual method, the model performance would eventually start to deteriorate especially when the number of hidden layers is larger than 5 layers.

In summary, for both $\omega = \pi$ and 3π, the best hidden layers are also in the range of $3 - 4$ layers. Moreover, the results show that when we take the average time per epoch of each hidden layer into account, the best hidden layer is shifted to 3 layers.

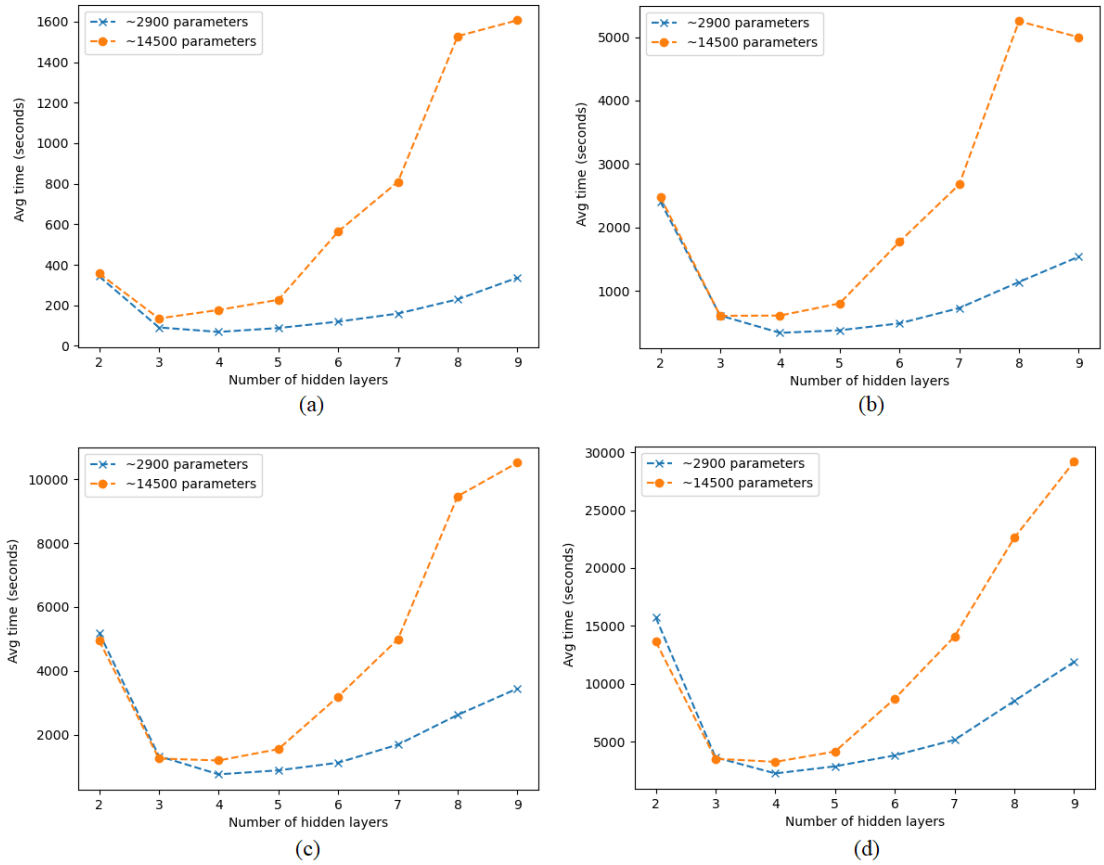

Figure 4.19 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average time and number of hidden layers when relative error equal to $5.0\%, 1.0\%, 0.5\%, 0.2\%$ for (a), (b), (c), and (d), respectively.

4.6.4 Total Number of Parameters : Average Time per Epoch

In the section of finding optimal total number of parameters, we train neural networks with different total number of parameters by fixing the number of hidden layers which is chosen to be 3 layers. The total number of parameters of the network is changed by changing the number of neurons per layer. For the exact boundary condition with $\omega = \pi, 3\pi$, we vary the number of neurons per layer from 15 to 240 neurons which gives us 540 to 116640 parameters. For the inexact boundary condition with $\omega = \pi, 3\pi$, we also vary the number of neurons per layer from 20 to 110 neurons which gives us 960 to 25080 parameters. The average number of epochs are calculated using 50 randomly initialized neural networks. The average time per epoch is calculated using 1000 epochs.

The average time per epoch for exact boundary condition when $\omega = \pi$ and 3π is shown in Figure 4.20. For the inexact boundary condition, the average time per epoch when $\omega = \pi$ and 3π is shown in Figure 4.21.

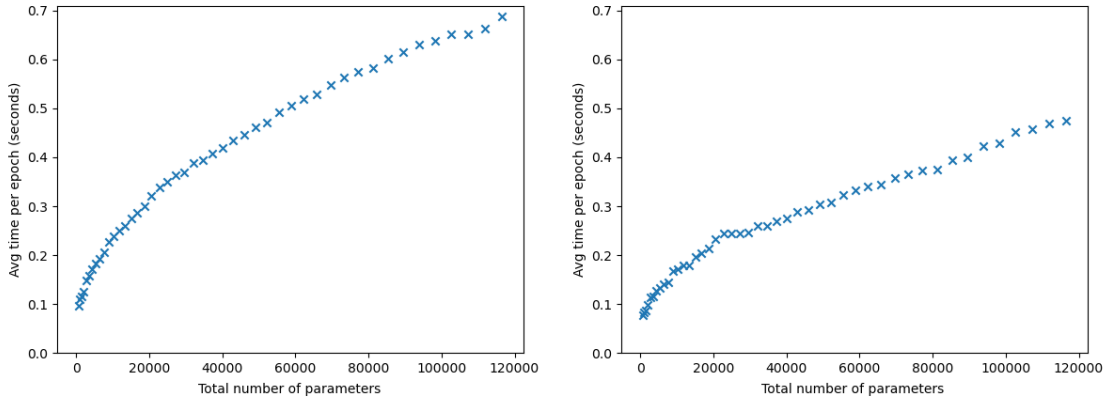

Figure 4.20 Exact boundary condition (Rectangular domain) : Average time per epoch for different total number of parameters, (a) $\omega = \pi$. (b) $\omega = 3\pi$.

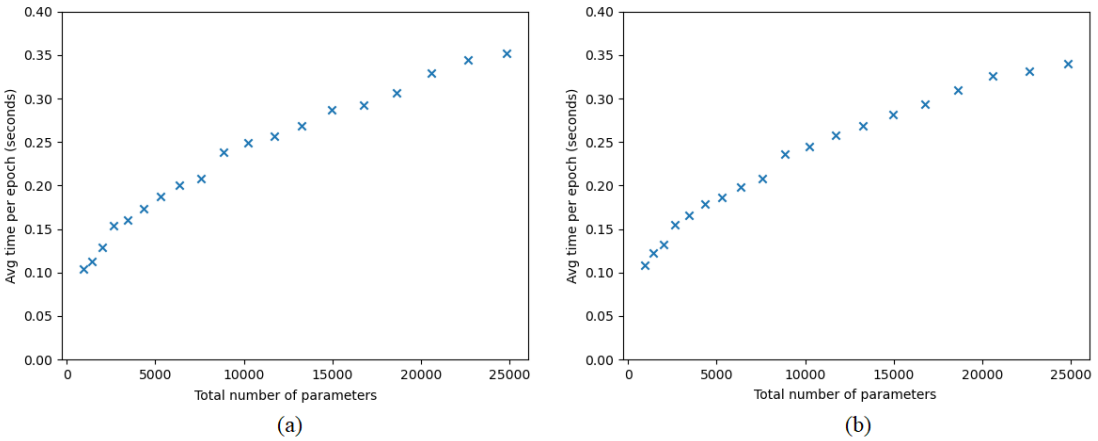

Figure 4.21 Inexact boundary condition (Rectangular domain) : Average time per epoch for different total number of parameters, (a) $\omega = \pi$. (b) $\omega = 3\pi$.

The results show that the average time per epoch increases with the total number of parameters nonlinearly. The average time per epoch increases rapidly at the start, and starts to slow down. In the next section, we will show the results of the average number of epochs and average time to find the optimal total number of parameters.

4.6.5 Total Number of Parameters : Exact Boundary Condition

In this section, we show the average number of epochs and average time for different total numbers of parameters for the case of exact boundary condition. For $\omega = \pi$, the average number of epochs and average time are shown in Figure 4.22 and 4.23, respectively. For $\omega = 3\pi$, the average number of epochs and average time are shown in Figure 4.24 and 4.25, respectively. In the graph of the average time, the vertical axis is the logarithm with base 10 of the average time.

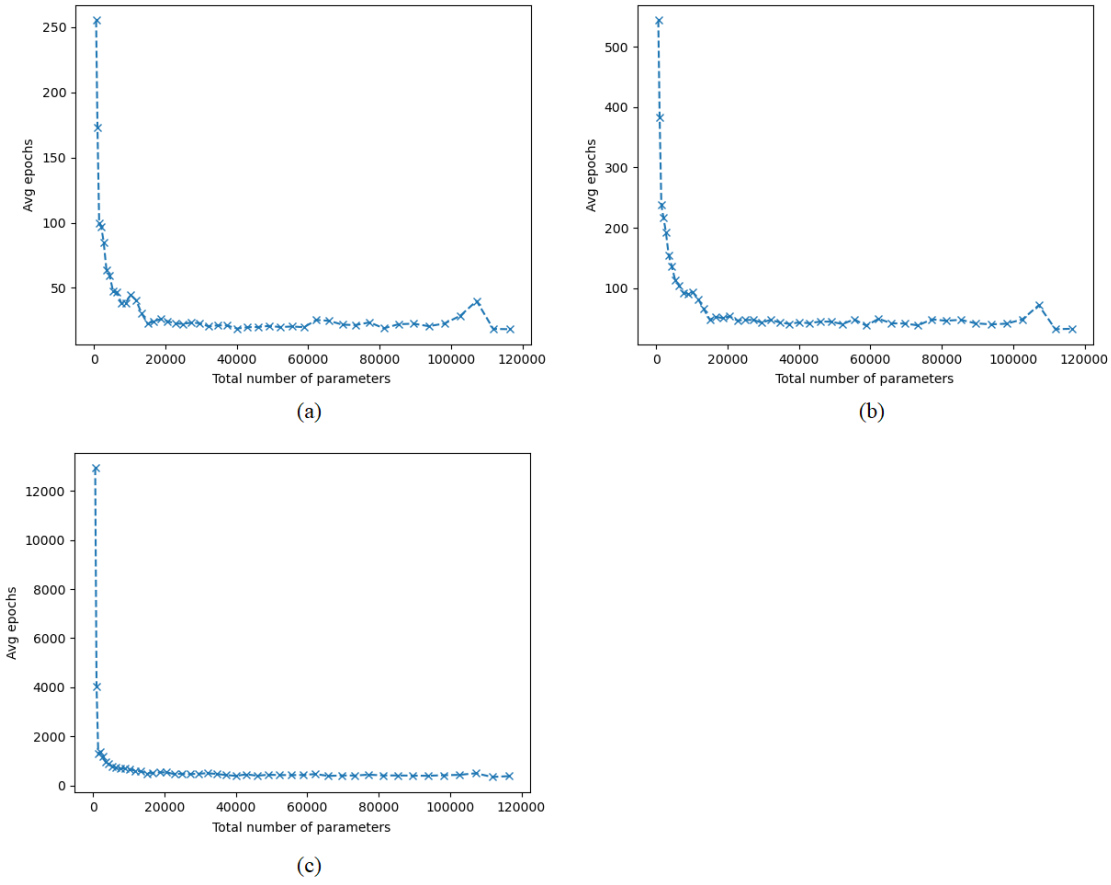

Figure 4.22 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average number of epochs and total number of parameters when relative error equal to $0.01\%, 0.005\%, 0.001\%$ for (a), (b), and (c), respectively.

By considering the average number of epochs, the results show that it roughly decreases with the total number of parameters both for $\omega = \pi, 3\pi$ indicating the increasing capability of the neural network. The results are consistent with what we expected.

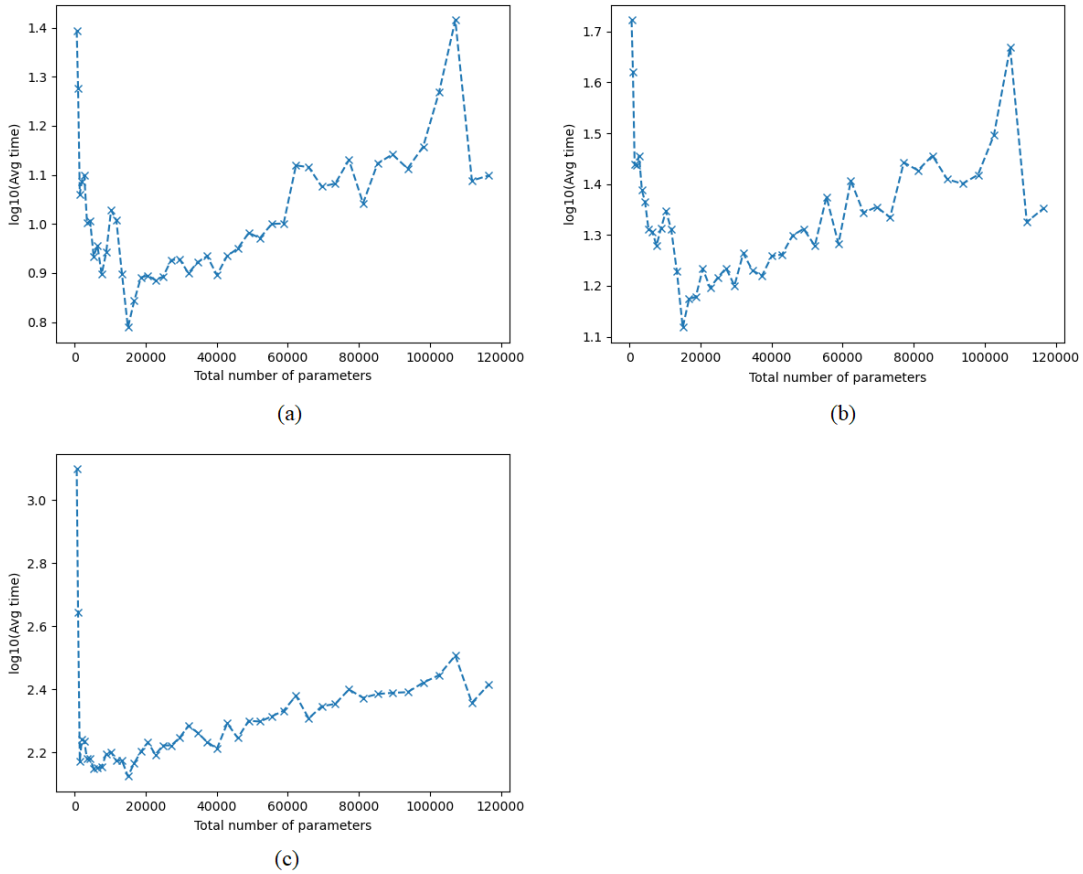

Figure 4.23 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average time and total number of parameters when relative error equal to $0.01\%, 0.005\%, 0.001\%$ for (a), (b), and (c), respectively.

By considering the average time, the total number of parameters with the lowest average time is 14960, 14960, and 14960 parameters, respectively. For $\omega = 3\pi$, the total number of parameters with the lowest average time is 7560, 7560, 34580, and 34580 parameters for $E = \{1\%, 0.5\%, 0.1\%, 0.05\%\}$, respectively.

Both results show that for a large total number of parameters the average time per epoch is too high when compared to the average number of epochs. It also shows that the optimal total number of parameters will depend on the value of relative error in which low total number of parameters corresponds to high relative error.

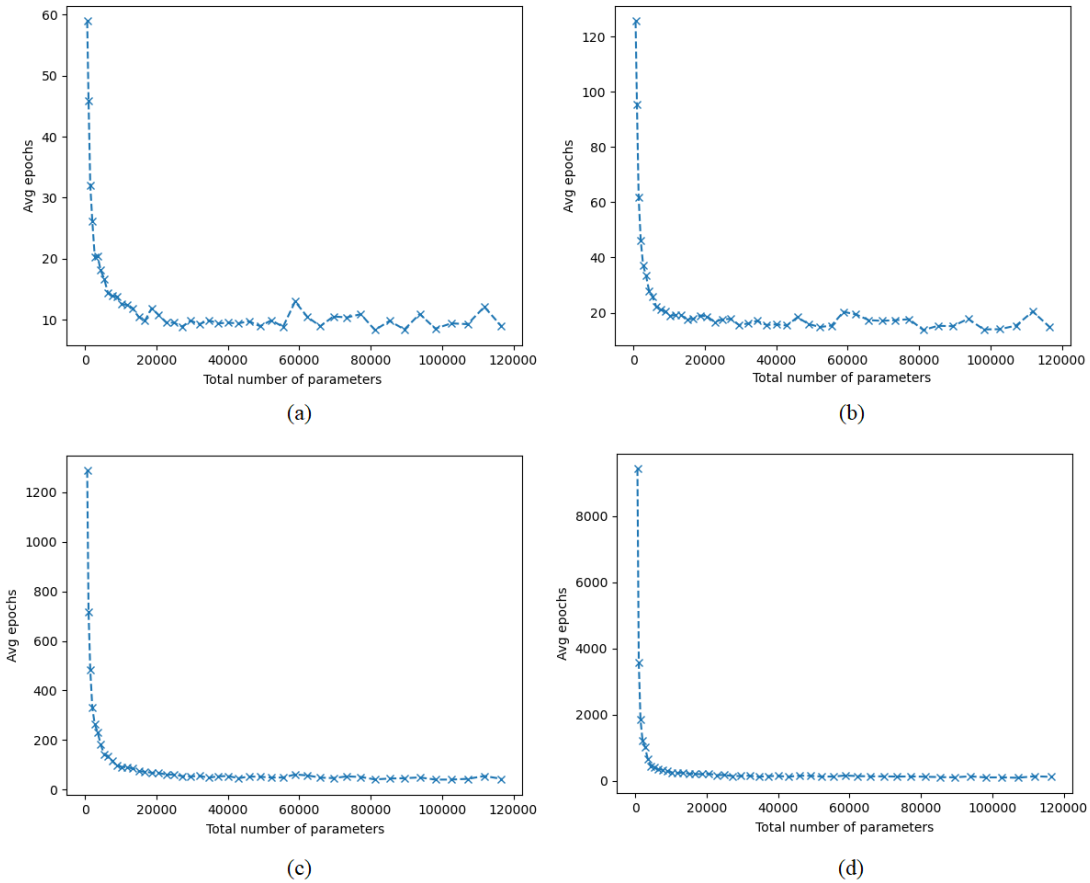

Figure 4.24 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average number of epochs and total number of parameters when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

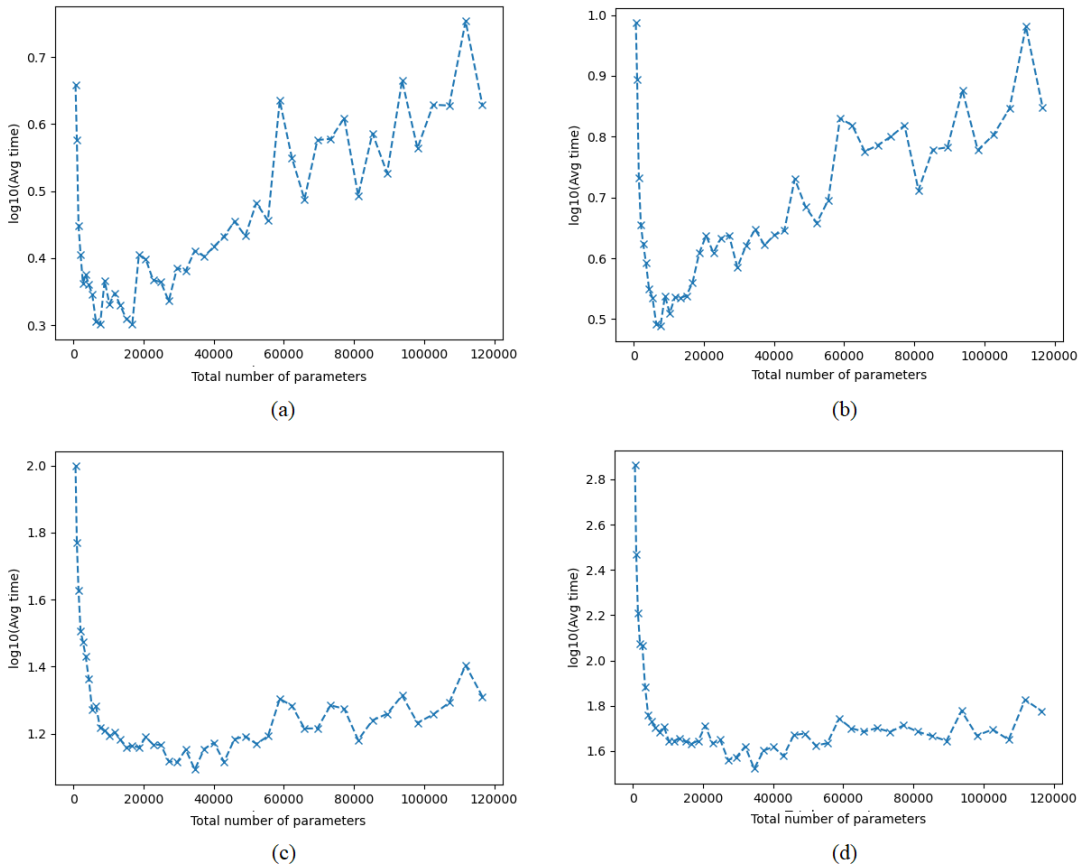

Figure 4.25 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average time and total number of parameters when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

4.6.6 Total Number of Parameters : Inexact Boundary Condition

In this section, we show the average number of epochs and average time for different total numbers of parameters for the case of inexact boundary condition. For $\omega = \pi$, the average number of epochs and average time are shown in Figure 4.26 and 4.27, respectively. For $\omega = 3\pi$, the average number of epochs and average time are shown in Figure 4.28 and 4.29, respectively. In the graph of the average time, the vertical axis is the logarithm with base 10 of the average time.

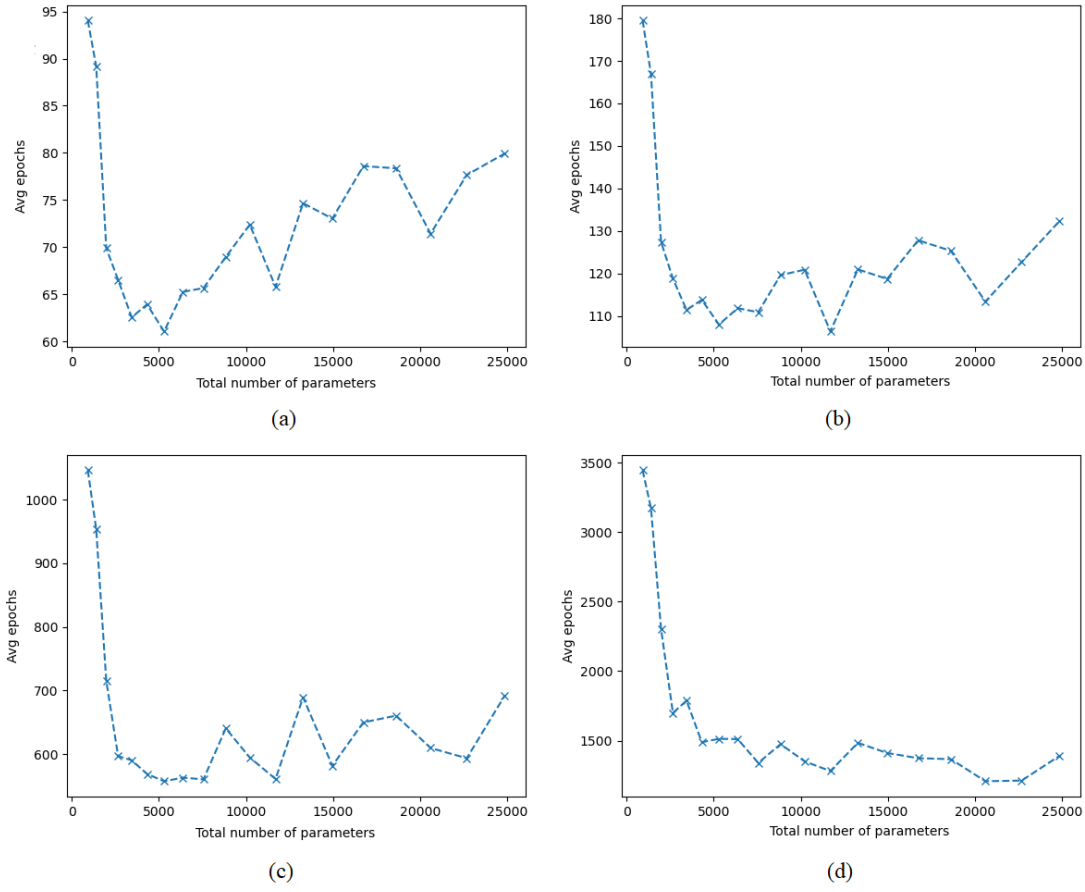

Figure 4.26 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average number of epochs and total number of parameters when relative error equal to $1\%, 0.5\%, 0.1\%, 0.05\%$ for (a), (b), (c), and (d), respectively.

For $\omega = \pi$, the average number of epochs when $E = 1\%$ will decrease until the total number of parameters equal to 5400 parameters, and will start to in-

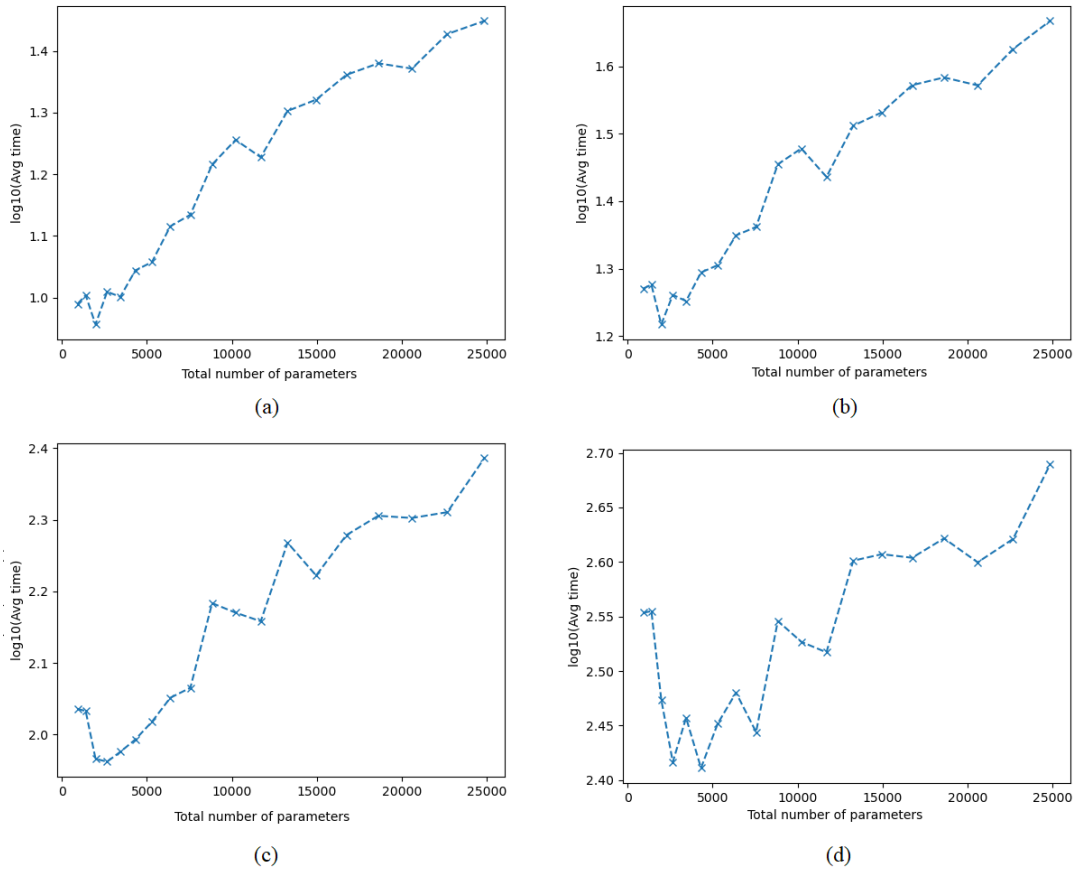

Figure 4.27 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Relationship between average time and total number of parameters when relative error equal to $0.01\%, 0.005\%, 0.001\%$ for (a), (b), and (c), respectively.

crease steadily. When $E = 0.5\%, 0.1\%$, the trend is less noticeable. Finally when $E = 0.05\%$, the average number of epochs decrease with the total number of parameters, but we expect to see the deterioration of the performance if we increase the total number of parameters further. By considering the average time, The total number of parameters with the lowest average time is 2040, 2040, 2730, and 4410 parameters for $E = \{1\%, 0.5\%, 0.1\%, 0.05\%\}$, respectively.

For $\omega = 3\pi$, when $E = 5\%$, the average number of epochs decreases until the total number of parameters equal to 5400 parameters, and will start to increase steadily. For other relative errors, the average number of epochs seem to decrease with the total number of parameters. By considering the average time, The total number of

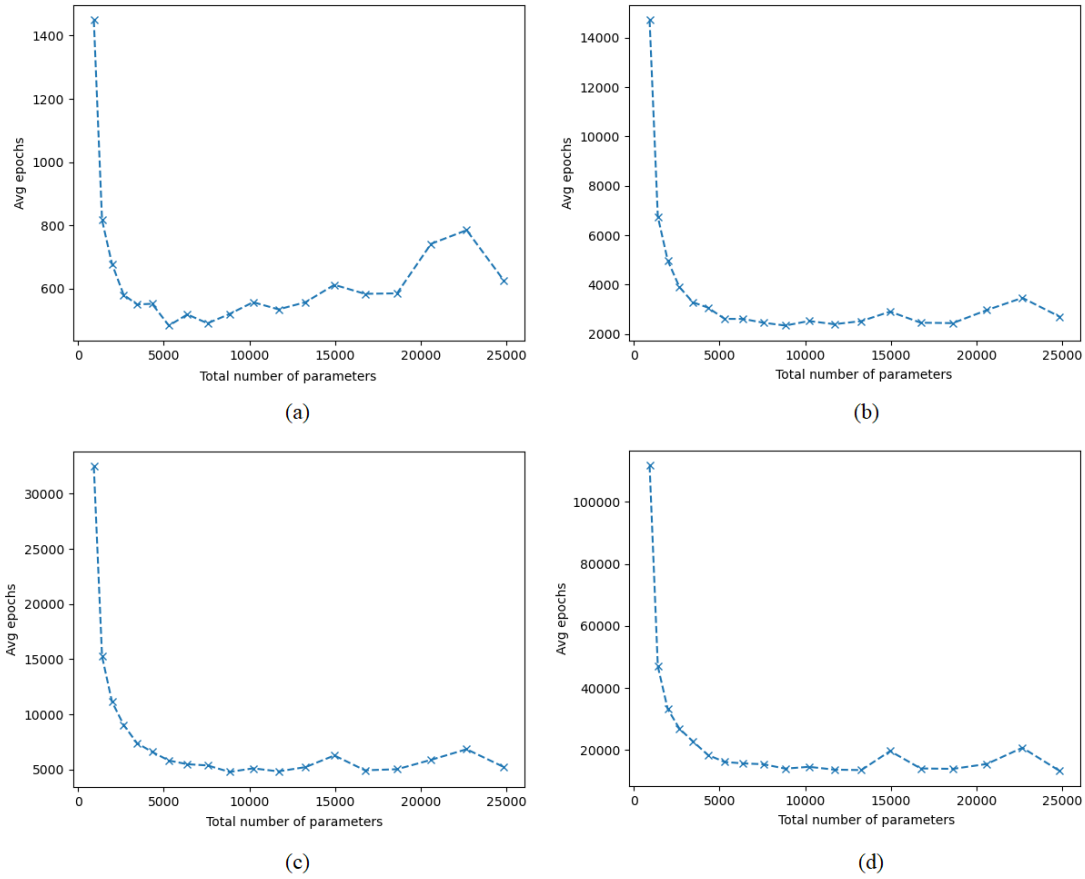

Figure 4.28 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average number of epochs and total number of parameters when relative error equal to $5.0\%, 1.0\%, 0.5\%, 0.2\%$ for (a), (b), (c), and (d), respectively.

parameters with the lowest average time is 2040, 5400, 5400, and 5400 parameters for $E = \{5\%, 1\%, 0.5\%, 0.2\%\}$, respectively.

The results show that a higher total number of parameters is inefficient when the average time per epoch is taken into account. Moreover, the optimal total number of parameters will increase with lower value of relative error.

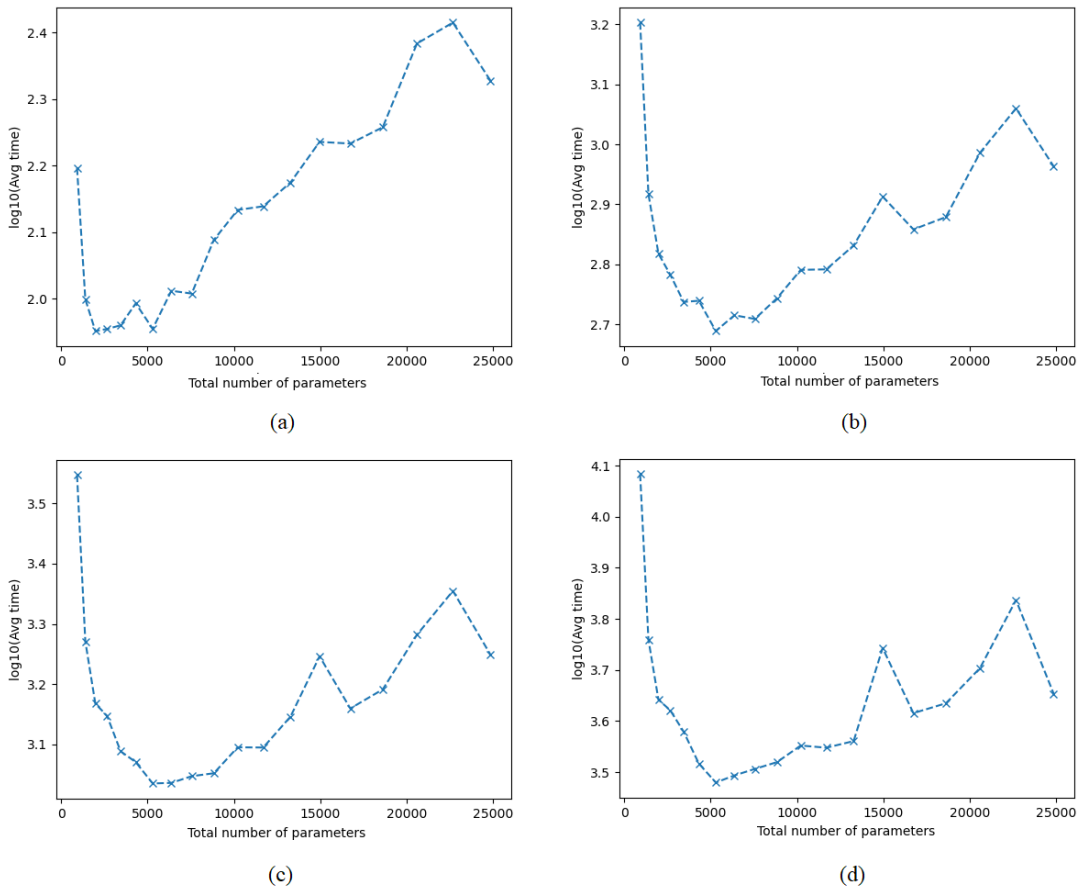

Figure 4.29 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Relationship between average time and totalnumber of parameters when relative error equal to $5.0\%, 1.0\%, 0.5\%, 0.2\%$ for (a), (b), (c), and (d), respectively.

CHAPTER V
CONCLUSION

In this thesis, we study the performance of different neural network methods for solving differential equations, and search for the optimal hyperparameters of the neural network using a genetic algorithm. The hyperparameters we searched for are activation function, optimization algorithm, and weight initialization. We also search for the optimal number of hidden layers, and total number of parameters. In this work, the neural network method is used to solve a 2D Laplace's equation on a rectangular domain. The boundary condition is chosen such that the analytical solution is shown in Equation 4.2. We are interested in the analytical solution with frequencies equal to π and 3π. The neural network is trained until a specified value of relative error is reached, and the number of epochs is recorded. The number of epochs together with the average time per epoch are the performance measurement of a neural network.

For the performance of different neural network methods in the case of exact boundary conditions on a rectangular domain, we are interested in the Lagaris and mixed residual method. The average time per epoch is comparable between two methods, and we can directly use the average number of epochs as an performance indicator. The results show that the Lagaris method has a lower average number of epochs and standard deviation across different relative errors, and frequencies ($\omega = \pi, 3\pi$). Moreover, the Lagaris method will perform increasingly better when the value of relative error gets lower.

For the case of inexact boundary conditions, we are interested in the deep Galerkin and mixed residual method. The average time per epoch is also comparable between two methods. On a rectangular, the mixed residual method has a slightly lower

average number of epochs and standard deviation across different relative errors when $\omega = \pi$. When $\omega = 3\pi$, both for average number of epochs and standard deviation, the mixed residual method outperforms the deep Galerkin method especially on the lower value of relative error. To further investigate the methods on an irregular domain, we consider a 2D Laplace's equation on a circular domain. The analytical solution is shown in Equation 4.10 with $m = 2, 3$. The results show that the mixed residual method has a slightly higher average number of epochs when $m = 2$. On the contrary when $m = 3$, the mixed residual method outperforms the deep Galerkin method on the average number of epochs. Interestingly, the mixed residual method has a lower value of standard deviation in both cases. We also consider a 2D Poisson's equation on a star shape domain. The mixed residual method has a higher average number of epochs, but a lower standard deviation. From the results, we expect the mixed residual method to outperform the deep Galerkin method when the shape of the solution becomes more complex.

Next, a genetic algorithm is used to find the optimal activation function, optimization algorithm, and weight initialization by considering the hyperparameters with the lowest number of epochs, and the hyperparameters with the greatest proportion in the last generation. The average time per epoch is comparable across different combinations of activation function and optimization algorithm. Therefore, we can directly measure the performance of the neural network using only the number of epochs.

For the case of exact boundary condition on a rectangular domain, the optimal optimization algorithm and weight initialization are Nadam and He uniform, respectively. The optimal activation function when $\omega = \pi$ is a GELU function, and when $\omega = 3\pi$ is a Mish function. For the case of inexact boundary condition on a rectangular domain, the optimal optimization algorithm and weight initialization are Adam and Lecun uniform, respectively. The optimal activation function when $\omega = \pi$ is a GELU function. But when $\omega = 3\pi$, the optimal activation functions are a GELU function and a Mish function. The neural network that use a GELU function has a lower average number of epochs, but the one that uses a Mish function has a lower standard deviation.

For the optimal number of hidden layers, we train two sets of neural networks with $2-9$ hidden layers while the total number of parameters is fixed. The first one has the total number of parameters roughly equal to 2900 parameters. The second one has the total number of parameters roughly equal to 14500 parameters. The average time per epoch increases linearly with the number of hidden layers, and needs to be considered together with the average number of epochs.

By considering the average number of epochs, the optimal hidden layer is $3-5$ layers both for the case of exact and inexact boundary conditions. By considering the average time, the optimal hidden layer is $3-4$ layers for both conditions. The results are consistent across different solutions, relative errors, and two different total numbers of parameters. For the mixed residual method with the case of inexact boundary condition, we observe that when the number of hidden layers is greater than 5 layers, the average number of epochs increase when the total number of parameters equal to 14500 parameters. These results show that increasing the total number of parameters might not improve the capability of the neural network at least when the number of hidden layers is larger than 5 layers.

For the optimal total number of parameters, we train the neural networks with different total number of parameters while fixing the number of hidden layers which is chosen to be 3 layers. The total number of parameters is changed by changing the number of neurons per layer. For exact boundary conditions, we vary the number of neurons per layer from 15 to 240 neurons corresponding to 540 to 116640 parameters. For inexact boundary conditions, we vary the number of neurons per layer from 20 to 110 neurons corresponding to 960 to 25080 parameters. The average time per epoch increases nonlinearly with the total number of parameters, and is needed to be considered.

For the case of exact boundary condition, the average number of epochs decreases with the total number of parameters both for $\omega = \pi$ and 3π. By considering the average time, the optimal total number of parameters for $\omega = \pi$ is 14960 parameters across all different relative errors. For $\omega = 3\pi$, the optimal total number of parameters

is 7560, 7560, 34580, and 34580 parameters for $E = \{1\%, 0.5\%, 0.1\%, 0.05\%\}$, respectively. The results show that a higher total number of parameters is inefficient due to the large value of average time per epoch, even though the average number of epochs is lower.

For the case of inexact boundary condition, when $E = 1\%$ and $\omega = \pi$ or when $E = 5\%$ and $\omega = 3\pi$, the average number of epochs will decrease with the total number of parameters to a certain value, and will start to increase afterward. Unlike the case of exact boundary condition, this means that after a certain point the capability of the neural network will start to deteriorate if we further increase the total number of parameters. For low relative error, we also expect the average number of epochs to increase if we use a larger total number of parameters. By considering the average time, the optimal total number of parameters for $\omega = \pi$ is 2040, 2040, 2730, and 4410 parameters for $E = \{1\%, 0.5\%, 0.1\%, 0.05\%\}$, respectively. For $\omega = 3\pi$, the optimal total number of parameters is 2040, 5400, 5400, and 5400 parameters for $E = \{5\%, 1\%, 0.5\%, 0.2\%\}$, respectively. The results show that a higher total number of parameters is inefficient, and the optimal value seems to increase with the lower value of relative error.

REFERENCES

[1] J. Berg and K. Nyström, "A unified deep artificial neural network approach to partial differential equations in complex geometries," *Neurocomputing*, vol. 317, pp. 28–41, Nov 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092523121830794X

[2] G. D. Smith, *Numerical solution of partial differential equations : finite difference methods*, 3rd ed., ser. Oxford applied mathematics and computing science series. Oxford, England: Clarendon Press, 1985.

[3] T. J. R. Hughes, *The finite element method : linear static and dynamic finite element analysis*. Mineola [N.Y.]: Dover Publications, 2000.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec 1943. [Online]. Available: https://doi.org/10.1007/BF02478259

[5] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[7] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[8] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006, pMID: 16764513. [Online]. Available: https://doi.org/10.1162/neco.2006.18.7.1527

[9] L. O. Chua and L. Yang, "Cellular neural networks: theory," *IEEE Transactions on Circuits and Systems*, vol. 35, no. 10, pp. 1257–1272, 1988.

[10] J. C. Chedjou, K. Kyamakya, M. A. Latif, U. A. Khan, I. Moussa, and Do Trong Tuan, "Solving stiff ordinary differential equations and partial differential equations using analog computing based on cellular neural networks," in *2009 2nd International Workshop on Nonlinear Dynamics and Synchronization*, 2009, pp. 213–220.

[11] J. Takeuchi and Y. Kosugi, "Neural network representation of finite element method," *Neural Networks*, vol. 7, no. 2, pp. 389–395, Jan 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0893608094900310

[12] X. Li, J. Ouyang, Q. Li, and J. Ren, "Integration wavelet neural network for steady convection dominated diffusion problem," in *2010 Third International Conference on Information and Computing*, vol. 2, 2010, pp. 109–112.

[13] Jianyu Li, Siwei Luo, Yingjian Qi, and Yaping Huang, "Numerical solution of elliptic partial differential equation by growing radial basis function neural networks," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, vol. 1, 2003, pp. 85–90 vol.1.

[14] J. Park and I. W. Sandberg, "Approximation and radial-basis-function networks," *Neural Computation*, vol. 5, no. 2, pp. 305–316, Mar 1993. [Online]. Available: https://doi.org/10.1162/neco.1993.5.2.305

[15] N. Mai-Duy and T. Tran-Cong, "Numerical solution of differential equations using multiquadric radial basis function networks," *Neural Networks*, vol. 14, no. 2, pp. 185–199, Mar 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608000000952

[16] H. Lee and I. S. Kang, "Neural algorithm for solving differential equations," *Journal of Computational Physics*, vol. 91, no. 1, pp. 110–131, Nov 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002199919090007N

[17] L. Wang and J. M. Mendel, “Structured trainable networks for matrix algebra,” in *1990 IJCNN International Joint Conference on Neural Networks*, 1990, pp. 125–132 vol.2.

[18] R. Yentis and M. E. Zaghloul, “Vlsi implementation of locally connected neural network for solving partial differential equations,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 43, no. 8, pp. 687–690, 1996.

[19] A. J. Meade and A. A. Fernandez, “The numerical solution of linear ordinary differential equations by feedforward neural networks,” *Mathematical and Computer Modelling*, vol. 19, no. 12, pp. 1–25, Jun 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0895717794900957

[20] ——, “Solution of nonlinear ordinary differential equations by feedforward neural networks,” *Mathematical and Computer Modelling*, vol. 20, no. 9, pp. 19–44, Nov 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089571779400160X

[21] N. Yadav, *An Introduction to Neural Network Methods for Differential Equations*, ser. SpringerBriefs in Applied Sciences and Technology. Springer, 03 2015.

[22] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, Dec 1989. [Online]. Available: https://doi.org/10.1007/BF02551274

[23] K. Hornik, “Approximation capabilities of multilayer feedforward networks,” *Neural Networks*, vol. 4, no. 2, pp. 251–257, Jan 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/089360809190009T

[24] B. Csáji, “Approximation with artificial neural networks,” *Faculty of Sciences, Etvs Lornd University, Hungary*, 2001.

[25] I. E. Lagaris, A. Likas, and D. I. Fotiadis, “Artificial neural networks for solving ordinary and partial differential equations,” *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 987–1000, 1998.

[26] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou, "Neural-network methods for boundary value problems with irregular boundaries," *IEEE Transactions on Neural Networks*, vol. 11, no. 5, pp. 1041–1049, 2000.

[27] K. S. McFall and J. R. Mahan, "Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions," *IEEE Transactions on Neural Networks*, vol. 20, no. 8, pp. 1221–1233, 2009.

[28] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *Journal of Computational Physics*, vol. 375, pp. 1339–1364, Dec 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999118305527

[29] L. Lyu, Z. Zhang, M. Chen, and J. Chen, "Mim: A deep mixed residual method for solving high-order partial differential equations," 2020.

[30] Xin Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[31] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Computation*, vol. 2, no. 2, pp. 198–209, Jun 1990. [Online]. Available: https://doi.org/10.1162/neco.1990.2.2.198

[32] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 1. Morgan-Kaufmann, 1989, pp. 107–115. [Online]. Available: https://proceedings.neurips.cc/paper/1988/file/07e1cd7dca89a1678042477183b7ac3f-Paper.pdf

[33] A. Roy, L. S. Kim, and S. Mukhopadhyay, "A polynomial time algorithm for the construction and training of a class of multilayer perceptrons," *Neural Networks*, vol. 6, no. 4, pp. 535–545, Jan 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608005800577

[34] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, 1994.

[35] J. R. Koza and J. P. Rice, “Genetic generation of both the weights and architecture for a neural network,” in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. ii, 1991, pp. 397–404 vol.2.

[36] S. Bornholdt and D. Graudenz, “General asymmetric neural networks and structure design by genetic algorithms: a learning rule for temporal patterns,” in *Proceedings of IEEE Systems Man and Cybernetics Conference - SMC*, vol. 2, 1993, pp. 595–600 vol.2.

[37] M. Mandischer, “Evolving recurrent neural networks with non-binary encoding,” in *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, vol. 2, 1995, pp. 584–589 vol.2.

[38] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, “Evolving the topology of large scale deep neural networks,” in *Genetic Programming*, M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni, and P. García-Sánchez, Eds. Cham: Springer International Publishing, 2018, pp. 19–34.

[39] A. Lopez-Rincon, A. Tonda, M. Elati, O. Schwander, B. Piwowarski, and P. Gallinari, “Evolutionary optimization of convolutional neural networks for cancer mirna biomarkers classification,” *Applied Soft Computing*, vol. 65, pp. 91–100, Apr 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1568494617307615

[40] J. Kiefer and J. Wolfowitz, “Stochastic estimation of the maximum of a regression function,” *Ann. Math. Statist.*, vol. 23, no. 3, pp. 462–466, Sep 1952. [Online]. Available: https://doi.org/10.1214/aoms/1177729392

[41] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, “Incorporating second-order functional knowledge for better option pricing,” in *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp, Eds., vol. 13. MIT Press, 2001, pp. 472–478. [Online]. Available: https://proceedings.neurips.cc/paper/2000/file/44968aece94f667e4095002d140b5896-Paper.pdf

[42] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” 2017.

[43] D. Misra, “Mish: A self regularized non-monotonic activation function,” 2020.

[44] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2020.

[45] S. K. Roy, S. Manna, S. R. Dubey, and B. B. Chaudhuri, "Lisht: Non-parametric linearly scaled hyperbolic tangent activation function for neural networks," 2020.

[46] S. Ruder, "An overview of gradient descent optimization algorithms," 2017.

[47] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul 2011.

[48] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 2012.

[49] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[50] T. Dozat, "Incorporating nesterov momentum into adam," in *Proc. International-Conference on Learning Representations (ICLR)*, 2016.

[51] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," 2014.

[52] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998, p. 9–50.

[53] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: JMLR Workshop and Conference Proceedings, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

[54] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," 2015.

[55] L. N. Smith, "Cyclical learning rates for training neural networks," in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2017, pp. 464–472. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/WACV.2017.58

[56] F. Mattioli, D. Caetano, A. Cardoso, E. Naves, and E. Lamounier, “An experiment on the use of genetic algorithms for topology selection in deep learning,” *Journal of Electrical and Computer Engineering*, vol. 2019, p. 3217542, Jan 2019. [Online]. Available: https://doi.org/10.1155/2019/3217542

[57] Yee Leung, Yong Gao, and Zong-Ben Xu, “Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis,” *IEEE Transactions on Neural Networks*, vol. 8, no. 5, pp. 1165–1176, 1997.

APPENDICES

APPENDIX A
AVERAGE TIME PER EPOCH FOR ACTIVATION FUNCTION AND OPTIMIZATION ALGORITHM

Here, the average time per epoch for different activation functions and optimization algorithms is calculated by averaging over 1000 epochs. For exact boundary conditions with $\omega = \pi, 3\pi$, the results are shown in Table A.1 and A.2 respectively. For inexact boundary conditions with $\omega = \pi, 3\pi$, the results are shown in Table A.3 and A.4 respectively.

Table A.1 Exact boundary condition (Rectangular domain, $\omega = \pi$) : Average time per epoch for different activation function and optimization algorithm.

Average time per epochs (seconds) (Rectangular domain, $\omega = \pi$)							
	Tanh	Sigmoid	Swish	Softplus	Mish	GELU	LiSHT
Adamax	0.0604	0.0702	0.0922	0.0613	0.1020	0.1041	0.0823
Adam	0.0589	0.0714	0.0918	0.0596	0.1009	0.1032	0.0814
Nadam	0.0674	0.0768	0.0988	0.0671	0.1089	0.1124	0.0904
SGD	0.0588	0.0658	0.0886	0.0589	0.0998	0.1031	0.0813
RMSprop	0.0618	0.0673	0.0935	0.0622	0.1035	0.1071	0.0851
Adadelta	0.0602	0.0679	0.0922	0.0600	0.1016	0.1050	0.0828
Adagrad	0.0592	0.0691	0.0891	0.0598	0.1008	0.1044	0.0820

Table A.2 Exact boundary condition (Rectangular domain, $\omega = 3\pi$) : Average time per epoch for different activation function and optimization algorithm.

Average time per epochs (seconds) (Rectangular domain, $\omega = 3\pi$)							
	Tanh	Sigmoid	Swish	Softplus	Mish	GELU	LiSHT
Adamax	0.0613	0.0634	0.0859	0.0616	0.1037	0.1076	0.0846
Adam	0.0601	0.0627	0.0855	0.0609	0.1021	0.1066	0.0838
Nadam	0.0681	0.0734	0.0931	0.0701	0.1107	0.1160	0.0925
SGD	0.0617	0.0641	0.0916	0.0671	0.1067	0.1131	0.0869
RMSprop1	0.0636	0.0659	0.0885	0.0648	0.1055	0.1123	0.0877
Adadelta	0.0604	0.0633	0.0856	0.0608	0.1031	0.1082	0.0844
Adagrad	0.0600	0.0628	0.0853	0.0609	0.1024	0.1178	0.0835

Table A.3 Inexact boundary condition (Rectangular domain, $\omega = \pi$) : Average time per epoch for different activation function and optimization algorithm.

Average time per epochs (seconds) (Rectangular domain, $\omega = \pi$)							
	Tanh	Sigmoid	Swish	Softplus	Mish	GELU	LiSHT
Adamax	0.0717	0.0671	0.0929	0.0714	0.1049	0.1126	0.0925
Adam	0.0691	0.0665	0.0913	0.0705	0.1037	0.1107	0.0913
Nadam	0.0785	0.0758	0.1013	0.0804	0.1146	0.1221	0.1027
SGD	0.0660	0.0669	0.0913	0.0693	0.1033	0.1148	0.0896
RMSprop	0.0696	0.0736	0.0967	0.0751	0.1087	0.1177	0.0955
Adadelta	0.0665	0.0702	0.0920	0.0727	0.1048	0.1114	0.0934
Adagrad	0.0682	0.0688	0.0920	0.0713	0.1031	0.1116	0.0921

Table A.4 Inexact boundary condition (Rectangular domain, $\omega = 3\pi$) : Average time per epoch for different activation function and optimization algorithm.

Average time per epochs (seconds) (Rectangular domain, $\omega = 3\pi$)							
	Tanh	Sigmoid	Swish	Softplus	Mish	GELU	LiSHT
Adamax	0.0737	0.0734	0.1002	0.0726	0.1087	0.1178	0.0969
Adam	0.0712	0.0698	0.0966	0.0736	0.1069	0.1164	0.0930
Nadam	0.0834	0.0805	0.1044	0.0830	0.1166	0.1207	0.1014
SGD	0.0762	0.0757	0.1037	0.0796	0.1148	0.1255	0.1083
RMSprop	0.0758	0.0768	0.0987	0.0793	0.1088	0.1108	0.1001
Adadelta	0.0719	0.0728	0.0984	0.0759	0.1084	0.1184	0.0966
Adagrad	0.0717	0.0736	0.0955	0.0736	0.1106	0.1164	0.0926

BIOGRAPHY

NAME	Chaloemrat Boonthanawat
DATE OF BIRTH	14th February 1993
PLACE OF BIRTH	Bangkok, Thailand
INSTITUTIONS ATTENDED	Mahidol University, 2011-2014 Bachelor of Science (Physics) Mahidol University, 2015-2020 Master of Science (Physics)
HOME ADDRESS	Bang khae District, Bangkok, Thailand, 10160 Tel. +668-3138-1151 E-mail : chaloemrat.boo@gmail.com