School of Computation, Information and Technology - Informatics

Technische Universität München

Master's Thesis in Informatics

Iterative Sampling of Deep Operator Networks

Osman Utku Özbudak

School of Computation, Information and Technology - Informatics

Technische Universität München

Master's Thesis in Informatics

Iterative Sampling of Deep Operator Networks

Iteratives Abtasten von tiefen Operatornetzwerken

Author:	Osman Utku Özbudak
Examiner:	Dr. Felix Dietrich
Advisor:	Iryna Burak, M.Sc.
Submission Date:	January 15th, 2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

January 15th, 2024 Osman Utku Özbudak

Acknowledgments

I am grateful to my supervisor, Dr. Felix Dietrich, and my advisor, Iryna Burak, for their insights and assistance throughout the process of writing this thesis.

Abstract

The advancement of neural network models for effectively solving partial differential equations is essential in accurately representing physical systems. While conventional training methods primarily use gradient-based optimizers, the Sampling Where It Matters (SWIM) technique offers an innovative weight sampling approach to training neural networks. This sampling approach associates weights and biases in hidden layers with specific points in input data. This thesis focuses on adapting SWIM sampling for DeepONet, a neural network model that combines neural networks with integral operators for efficient learning from a variety of datasets. In particular, the thesis aims to apply SWIM for weight sampling in DeepONet, in the context of partial differential equations. To achieve this, DeepONet is restructured to be a fully-connected network, based on the assumption of the orthogonality of its components. An iterative sampling method is then utilized for adjusting DeepONet's weights, aiming to evaluate the efficiency and applicability of SWIM in this context. The study includes the derivation, testing, and analysis of this iterative sampling approach using SWIM on DeepONet, with its performance assessed on one-dimensional Burgers' and Wave equation datasets.

Contents

Acknowledgements **iv**

Abstract **v**

1. Introduction **1**

2. State of the Art **4**
2.1. PDEs and Neural Networks 4
2.1.1. Partial Differential Equations 4
2.1.2. Physics-Informed Neural Networks (PINNs) 6
2.2. Deep Neural Operators 7
2.2.1. DeepONet 8
2.2.2. Extensions of DeepONet 9
2.2.3. Fourier Neural Operators 10
2.3. Weight Sampling 11
2.3.1. Extreme Learning Machines 11
2.3.2. Sampling Where It Matters (SWIM) 13

3. Iterative Sampling of Deep Neural Operators **15**
3.1. Sampled POD-DeepONet 16
3.2. Iterative Sampling of DeepONet 17
3.3. Experiments 20
3.3.1. Burgers' Equation 20
3.3.2. Wave Equation 30

4. Conclusion **40**
4.1. Summary 40
4.2. Discussion 40
4.3. Outlook 41

Bibliography **41**

Appendix **46**

A. Technical Specifications and Dataset Generation Details **46**
A.1. Technical Specifications 46
A.2. Dataset Generation Details 46
A.2.1. Burgers' Dataset 46
A.2.2. Wave Dataset 47

List of Figures **49**

List of Tables **50**

1. Introduction

The concept of neural networks has its roots in the 1940s when McCulloch and Pitts first introduced a computational model for neural networks based on mathematics and algorithms [22]. This was a milestone in the evolution of the artificial intelligence, which marked the inception of neural networks that we know today. Their model set the foundation for subsequent developments in the field, making it possible for the advanced neural network architectures.

Today, neural networks have become influential across applications of artificial intelligence. Inspired by the human brain's interconnected neural structure, these sophisticated computational models have demonstrated remarkable success across numerous fields. Their application ranges from image and speech recognition tasks to more complicated challenges like natural language processing [17]. These advancements have propelled the field of artificial intelligence into a new era, where machines can perform complex tasks that were once thought to be exclusive to human intelligence.

Stochastic gradient descent (SGD) [27] is a pivotal optimization algorithm in the successful implementation of neural networks. Its simplicity and effectiveness have made it a preferred choice, especially in large-scale dataset applications. However, SGD is not flawless. It is known to be sensitive to the learning rate, and the necessity for manual fine-tuning often becomes a challenge to its performance. Furthermore, SGD utilizes backpropagation [28], which is an algorithm that calculates the gradient of the loss function with respect to the network's weights. While backpropagation is crucial for optimizing the weights of the network during training, it can be computationally expensive, particularly in complex and deep network structures. This computational demand is an important drawback in scenarios involving large and complex models. Additionally, SGD may struggle with non-convex functions or datasets with irregular patterns. Adam [16], another widely used gradient-based optimization algorithm, offers improvements over SGD, such as adaptive learning rate adjustments. However, besides the burden of backpropagation, Adam also has some other drawbacks, such as increased memory consumption for storing the first and second moments of the gradients [2]. Given these challenges, the search for more efficient and robust optimization methods continues to be an active area of research in deep learning. These methods aim to enhance model performance while mitigating computational and memory demands. These methods not only enhance the performance of neu-

ral networks but also broaden their applicability in solving real-world problems, ranging from intricate scientific computations to everyday applications in technology and industry.

Although neural networks have proven to be powerful tools in the field of machine learning, traditional training methods such as SGD and Adam may still face limitations, especially when dealing with large-scale and complex data. To address these challenges, researchers have explored alternative neural network training approaches, such as weight sampling. A recent study in this area is the SWIM algorithm (Sampling Where It Matters) [6]. This method differs from earlier sampling approaches like Extreme Learning Machines (ELMs) [15], as it directly associates weights and biases in hidden layers with specific points in the input data, therefore creating a stronger link between the network and the data. The SWIM algorithm's distinctive approach aims to improve the neural network's capabilities by sampling its weights. According to experiments discussed in the SWIM study, SWIM sampling has the potential to reduce computational complexity while preserving efficiency in neural network training.

The primary focus of this thesis is to explore and assess the efficiency of an iterative sampling approach using SWIM sampling for DeepONet [20]. DeepONet is a neural network model that uniquely combines integral operators with neural networks to effectively learn from datasets, which shows particular proficiency in handling tasks related to partial differential equations (PDEs).

While the original SWIM study has explored weight sampling for variants like POD-DeepONet [21], this thesis concentrates on implementing and testing the iterative sampling approach directly on the standard DeepONet model. The aim is to investigate whether this iterative sampling approach using SWIM is effective for DeepONet. To comprehensively test and evaluate this methodology, datasets based on Burgers' and Wave equations are used. The iterative sampling process is applied to train and test DeepONet on these specific problems, providing insights into the practical applicability and potential benefits of iterative sampling in complex PDE scenarios. This investigation will also explore how the iterative SWIM sampling method impacts the accuracy and learning efficiency of sampled DeepONet.

The structure of the remainder of this thesis is as follows: Section 2 explores PDEs and examines some of the state-of-the-art neural network solutions for solving these equations, including a comprehensive review of neural operators such as DeepONet, its extensions, and other advanced models like Fourier Neural Operators. This section also discusses the principles of weight sampling with a focus on the SWIM methodology. Section 3 provides a detailed analysis of the weight sampling method, specifically concentrating on the implementation and details of iterative sampling using SWIM for DeepONet. In addition, this section presents and discusses the experimental results obtained using Burgers' and Wave equation datasets, offering a thorough examination of their significance. Finally,

Section 4 concludes the thesis with a discussion on the outcomes of the study and their broader implications.

2. State of the Art

In this chapter, the examination focuses on the core concepts and techniques that support this thesis. The exploration begins with PDEs and current state-of-the-art neural network approaches for solving them. Subsequent investigation covers deep neural operators. Lastly, attention turns to the sampling weights of deep neural networks. This exploration provides the necessary background to comprehend the significance and implications of the work presented in this thesis.

2.1. PDEs and Neural Networks

PDEs stand as a fundamental mathematical tool that possesses a range of applications across scientific domains, including the field of artificial neural networks. In essence, PDEs are equations that involve rates of change with respect to continuous variables. Thus, this makes them an ideal tool for modelling systems that display spatial and temporal variability. These systems can range from fluid flow and heat conduction to data-driven domains like image processing.

2.1.1. Partial Differential Equations

PDEs are used to describe various events in physics and they involve functions of several variables and their partial derivatives. At their core, they express the relation between an unknown function and its partial derivatives. A PDE can be expressed generally in the form

$$F\left(x_1, x_2, ..., x_n, u, \frac{\partial u}{\partial x_1}, ..., \frac{\partial u}{\partial x_n}, \frac{\partial^2 u}{\partial x_1^2}, ..., \frac{\partial^2 u}{\partial x_n^2}, ..., \frac{\partial^k u}{\partial x_n^k}\right) = 0, \tag{2.1}$$

where F is a given function, u is the unknown function which depends on n independent variables $x_1, x_2, ..., x_n$ and the equation involves up to k-th order partial derivatives of u [29]. Finding a solution to the PDE in Equation 2.1 means determining a function $u(x_1, x_2, ..., x_n)$ that makes the equation true for all applicable values of $x_1, x_2, ..., x_n$. Since PDEs can describe a variety of concepts, the solutions to these equations can also represent different outcomes or states.

The conventional methods of solving PDEs, while efficient for lower-dimensional problems, encounter substantial challenges when applied to high-dimensional PDEs. These classical techniques include finite difference methods, finite element methods, and spectral methods.

Finite difference methods involve discretizing the continuous problem by replacing derivatives with approximate finite differences. This method is straightforward and intuitive for problems with a small number of dimensions. However, as the number of dimensions increases, the number of grid points required for a precise approximation grows exponentially, which leads to a significant increase in computational cost and memory requirements.

Finite element methods decompose the domain into smaller, simpler parts known as finite elements. The PDE is then reconstructed as a system of algebraic equations over these elements. Finite element methods is highly flexible and can be applied to complex geometries and irregular domains. Nevertheless, similar to finite difference methods, finite element methods also suffers from the curse of dimensionality [4, 5] in higher-dimensional spaces, where the complexity and computational demand increase rapidly.

Spectral methods use global basis functions, typically trigonometric polynomials or orthogonal polynomials, to approximate the solution. These methods are known for their high accuracy in smooth problems and are particularly effective for periodic domains. However, they are less effective for problems with sharp gradients or discontinuities. Moreover, their performance degrades in higher dimensions due to the exponential increase in the number of basis functions required.

All these methods face the common challenge of the curse of dimensionality, which refers to the exponential growth in computational resources and time required as the number of dimensions in the problem space increases. Additionally, the accuracy of these traditional methods tends to decrease with the increase in dimensionality. Consequently, these problems necessitate the exploration of more computationally efficient and accurate alternatives and thereby allowing the usage of neural networks in this domain. Neural networks have the ability to learn representations in high-dimensional spaces efficiently, and they can potentially overcome the curse of dimensionality that impedes traditional PDE solving methods.

2.1.2. Physics-Informed Neural Networks (PINNs)

In the context of the challenges faced by traditional PDE solving methods, as previously discussed, the domain of scientific machine learning has been evolving to address these limitations. Traditional techniques in machine learning, such as convolutional and recurrent neural networks, often struggle to effectively handle complex physical or engineering systems. The primary reason for this struggle lies in the fact that these techniques typically do not incorporate the underlying principles or governing laws of these systems. As a result, the outcomes achieved by these models tend to be less precise and robust, particularly when dealing with problems characterized by complex physical phenomena.

Physics-informed neural networks (PINNs) [25, 26] have been developed to overcome this challenge and improve the handling of PDEs. The important advantage of PINNs is that they can work with small datasets, and they can be used to solve a wide range of engineering problems, making them a flexible tool for various applications. Unlike the traditional PDE solving methods that try to fit models to many pairs of states and values, PINNs use a different approach. They take into account the mathematical principles of the physical or engineering systems under study to find solutions to PDEs. This is a way of letting the basic physics of the problem guide the solution. The general form of the PDEs considered by PINNs is

$$u_t + \mathcal{N}[u; \lambda] = 0, x \in \Omega, t \in [0, T],$$

where $u(t, x)$ is the latent solution, $\mathcal{N}[.; \lambda]$ is a nonlinear differential operator parameterized by λ, and $\Omega \in \mathbb{R}^{\mathbb{D}}$.

The process works by constructing a loss function that considers both the PDE and the boundary conditions associated with it. In PINNs, the MSE (mean-squared error) loss function is composed of two parts: the data loss, denoted as MSE_u, and the PDE residual loss, denoted as MSE_f.

The data loss MSE_u measures how well the predicted solution matches the known values at initial and boundary points. For the data loss, let $u(t_u^i, x_u^i)$ be the neural network's solution at the i-th initial or boundary location, u^i is the actual solution at that location and t_u^i and x_u^i represent the time and space coordinates at i-th initial or boundary points, respectively. The sum is taken over N_u, which is the total number of initial and boundary points. The data loss MSE_u is then defined as

$$\text{MSE}_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2,$$

whose resulting value measures the difference between the known and the predicted solution at these points.

The PDE residual loss is represented as MSE_f and it is used to maintain the PDE's structure at selected points within the domain. For this purpose, the residual $f(t, x)$ is defined as

$$f := u_t + \mathcal{N}[u; \lambda],$$

where u_t denotes the partial derivative of the function u with respect to time t, and $\mathcal{N}[u]$ is a nonlinear operator acting on the function u. The parameter of the differential operator, denoted as λ, is transformed into a parameter of the network. The residual f measures the difference between the temporal dynamics of u and its nonlinear interactions. Given this, the PDE residual loss MSE_f is then defined as

$$\text{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2,$$

where $f(t_f^i, x_f^i)$ is the PDE residual at the i-th collocation point, t_f^i and x_f^i are its time and space coordinates, and N_f is the total number of these points. The overall loss function used to train the neural network is a combination of these two loss measurements:

$$\text{MSE} = \text{MSE}_u + \text{MSE}_f.$$

By reducing MSE loss, PINNs can find an approximate solution to the PDE that respects both the given boundary conditions and the physics of the problem as captured by the PDE.

2.2. Deep Neural Operators

After exploring PINNs and their ability to incorporate physical laws directly into the learning process, it becomes evident that neural networks hold significant potential in representing complex systems governed by PDEs. The Universal Function Approximation Theorem [9, 11] is a fundamental theorem in deep learning, stating their ability of neural networks to approximate functions accurately. Yet, their capabilities extend even further. Another theorem says that a single-layer neural network can approximate nonlinear continuous operators [7, 8]. These operators act as mappings, transitioning from one functional space to another function space. Deep neural operators are built to learn those nonlinear operators from data.

In essence, PDEs are inherently operator equations, and neural operators specialize at approximating these operators. Therefore, these networks provide a powerful tool for understanding and solving PDEs, which is crucial in scientific machine learning.

2.2.1. DeepONet

DeepONet [20] was introduced in 2019 and it is the first published neural network for learning nonlinear operators. Throughout this thesis, DeepONet serves as a baseline model. Specifically, it is employed to expand the weight sampling methodology discussed in Section 2.3, which is the main focus of this thesis. Therefore, understanding the specifics of DeepONet is critical to subsequent work.

The underlying structure of DeepONet involves an operator G, which accepts an input function u. The operator generates an output function which is denoted as $G(u)$. For any specific point y within the domain of $G(u)$, the output $G(u)(y)$ represents a real number. To represent the input functions discretely for the network, DeepONet uses function values at a finite number of locations named as "sensors". These sensors ensure a consistent representation for the network by evaluating the values of the function at specific points. Consequently, the input to network consists of two elements: the input function u and the grid y. The output of the network is $G(u)(y)$.

In the DeepONet architecture, there is a trunk network. The trunk network operates on the grid y, generating a vector $[t_1, \ldots, t_k]^\top \in \mathbb{R}^p$. In parallel, there are p branch networks, each processing the input function u. These networks output a series of scalar values, each denoted as b_k for $k = 1, 2, \ldots, p$, where each $b_k \in \mathbb{R}$. Furthermore, the DeepONet includes a bias term represented as $b_0 \in \mathbb{R}$. The prediction is then constructed by the combination of the outputs from the trunk and branch networks, integrated with the bias term:

$$G(u)(y) = \sum_{k=1}^{p} b_k(u) t_k(y) + b_0.$$

In practical applications, the inclusion of p branch networks can lead to challenges such as increased computational cost. To address this challenge, the authors introduced two variations of DeepONet: the stacked and unstacked versions. The stacked DeepONet is the configuration we have discussed so far, which includes p branch networks. In contrast, the unstacked DeepONet simplifies the architecture by employing just a single branch network. Stacked and unstacked DeepONet architectures are illustrated in Figure 2.1 and Figure 2.2, respectively.

One of the key advantages of DeepONet over traditional networks is its significant reduction in generalization error. This error measures difference between the training data and unseen test data. DeepONet has demonstrated low generalization error across different ordinary and partial differential equation problems [20]. The success of DeepONet in performing a small generalization error is related to its inductive bias. Inductive bias refers to the prior knowledge encoded into machine learning systems to ensure a reduced generalization error. In the case of DeepONet, the output $G(u)(y)$ incorporates two independent

inputs u and y. This structure leads to the use of two distinct networks (the branch network and the trunk network). In essence, $G(u)(y)$ can be interpreted as a function of y that is conditioned on u.

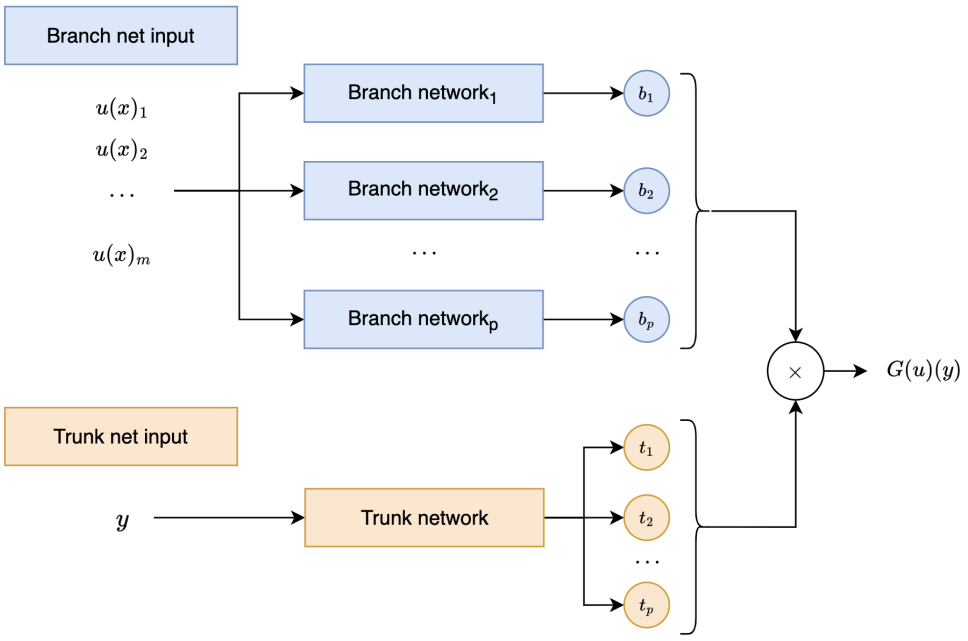

Figure 2.1.: Stacked DeepONet architecture includes p branch networks and a trunk network.

2.2.2. Extensions of DeepONet

An important variant of DeepONet is called POD-DeepONet, which is presented in [21]. This approach applies proper orthogonal decomposition (POD) to the training data to determine the basis. With this setup, the computed POD basis acts as the trunk network, leaving only the branch network to learn the coefficients for the POD basis. The resulting output can be expressed as:

$$G(u)(y) = \sum_{k=1}^{p} b_k(u)\psi_k(y) + \psi_0(y) \tag{2.2}$$

where $\psi_0(y)$ represents the mean of the solutions at y, $[b_1, b_2, ..., b_p]$ is the p branch network outputs, and $[\psi_1, \psi_2, ..., \psi_p]$ denote the p previously computed POD components for y.

Besides POD-DeepONet, DeepONet can be adjusted to better solve different problems by adding prior knowledge. There are two main ways to do this. In the trunk network,

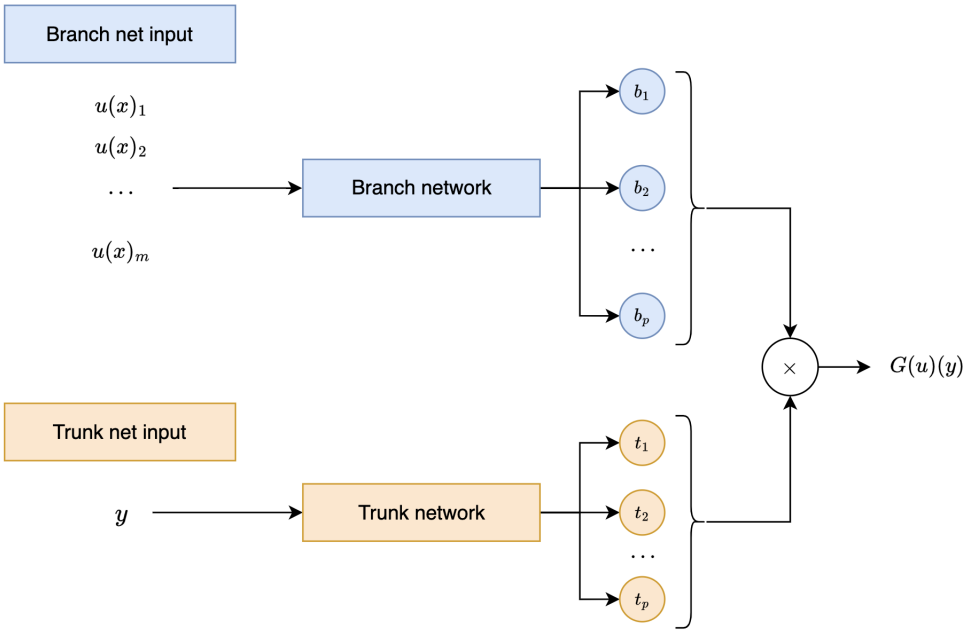

Figure 2.2.: Unstacked DeepONet architecture includes one branch network and one trunk network.

feature expansions help capture system behaviors more clearly. For instance, in [10], for the problems with oscillating solutions the authors proposed expanding the input y using harmonic features to capture prior knowledge

$$y \mapsto (y, \cos(y), \sin(y), \cos(2y), \sin(2y), \ldots) ,$$

and then expanded input can then be used in the trunk network. In the branch network, extra input functions can be added to handle information that changes with time and space. An example of this is a study on fluid flow where an extra input helped to include periodic boundary conditions [19].

2.2.3. Fourier Neural Operators

Besides DeepONet, Fourier Neural Operator (FNO) [18] is another significant study in the area of neural operators, especially recognized for their way of learning continuous functions. FNOs innovative is to use of the Fourier transform in learning functions in the frequency domain.

The Fourier transform is a fundamental concept in signal processing and applied mathematics, which transforms a function of time (or space) into a function of frequency. This

transformation is particularly useful for examining functions that are easier to understand in terms of frequency, rather than time or space. FNOs utilize this technique to offer a new perspective on handling complex functions, which makes it especially useful in situations where the frequency aspects of a function are key.

FNOs start by applying a Fourier transform to the input functions, which converts them into the frequency domain. This is an important step as it helps the FNO understand and learn the complex interactions between different frequencies. After this, the network learns to map these frequency relationships. The process concludes with an inverse Fourier transform, turning the output back into its original space. This method allows FNOs to effectively approximate complex functional mappings.

In general, FNOs represent an important study in the field of neural operators. Their use of the Fourier transform to map functions makes them a powerful tool for solving complex PDE problems.

2.3. Weight Sampling

Optimizing neural networks iteratively through gradient-based optimizers often introduces a time-intensive challenge, especially when dealing with high-dimensional datasets containing vast amounts of data. The training speed of these neural networks is often constrained by the available computational resources such as CPU or GPU. This constraint arises from the inherent characteristics of backpropagation. Using weight sampling as a training methodology for neural networks offers an alternative to gradient-based optimization.

2.3.1. Extreme Learning Machines

A notable study in weight sampling domain is Extreme Learning Machines (ELMs) [15]. ELMs operate on single-layer feedforward neural networks (SLFN) and the primary distinction of ELMs from traditional neural networks is that they sample input weights and biases from a distribution that is independent of the provided dataset, in other words, they are determined in a data-agnostic manner. Only the output weights are adjusted during the training process.

In ELMs, given a training dataset consisting of N arbitrary distinct samples (x_i, t_i), where $x_i = [x_{i1}, x_{i2}, \ldots, x_{in}]^\top \in \mathbb{R}^n$ represents the input vector and $t_i = [t_{i1}, t_{i2}, \ldots, t_{im}]^\top \in \mathbb{R}^m$ is the corresponding target output, the standard SLFNs with L hidden neurons and activation function $g(x)$ are modeled as

$$\sum_{i=1}^{L} \beta_i g(w_i \cdot x_j + b_i) = t_j, \quad j = 1, \ldots, N,$$

where $w_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^\top$ is the weight vector connecting the i-th hidden neuron to the input neurons, $\beta_i = [\beta_{i1}, \beta_{i2}, \ldots, \beta_{im}]^\top$ is the weight vector connecting the i-th hidden neuron to the output neurons, and b_i is the bias of the i-th hidden neuron. This equation can be written compactly as

$$\mathbf{H}\beta = \mathbf{T},$$

where $\mathbf{H}(w_1, \ldots, w_L, b_1, \ldots, b_L, x_1, \ldots, x_N) =$

$$\begin{bmatrix} g(w_1 \cdot x_1 + b_1) & \ldots & g(w_L \cdot x_1 + b_L) \\ \vdots & \ddots & \vdots \\ g(w_1 \cdot x_N + b_1) & \ldots & g(w_L \cdot x_N + b_L) \end{bmatrix}_{N \times L},$$

$$\beta = \begin{bmatrix} \beta_1^\top \\ \beta_2^\top \\ \vdots \\ \beta_L^\top \end{bmatrix}_{L \times m}, \quad \mathbf{T} = \begin{bmatrix} t_1^\top \\ t_2^\top \\ \vdots \\ t_N^\top \end{bmatrix}_{N \times m}.$$

The training objective of ELMs is to find the best β that minimizes the difference between the predicted output and the actual target. This optimization problem can be denoted as

$$||\mathbf{H}\beta - \mathbf{T}|| = \min_{\beta} ||\mathbf{H}\beta - \mathbf{T}||.$$

A common approach to solve this optimization problem is using the Moore-Penrose pseudoinverse, leading to the solution:

$$\beta^* = \mathbf{H}^\dagger \mathbf{T},$$

where $\mathbf{H}^\dagger$ is the pseudoinverse of the hidden layer output matrix $\mathbf{H}$.

ELMs offer a convenient approach to neural network training by fixing input weights and biases and adjusting only the output weights. This methodology results in rapid training times while maintaining competitive performance. According to the previously held studies, ELMs can outperform support vector machines in classification and regression tasks [14, 13, 12]. However, one of the inherent limitations of ELMs is that the input weights and biases are sampled in a data-agnostic manner, which might not always capture the underlying patterns or nuances of the dataset effectively.

2.3.2. Sampling Where It Matters (SWIM)

Recently, a new weight sampling method known as SWIM sampling was introduced [6]. SWIM stands for "Sampling Where It Matters". These networks are based on the idea of random feature networks [24, 23]. Random feature networks transform input data into a lower-dimensional space using predetermined functions and select weights without considering the specific attributes of the data. However, SWIM sampling differentiate themselves from ELMs and standard random feature networks. While ELMs and random feature networks are generally limited to use in SLFNs, SWIM sampling offer the flexibility to be applied across a varying number of hidden layers. Moreover, in contrast to ELMs and random feature networks, SWIM sampling utilize both input and output data to sample weights, especially in areas with larger gradients. In SWIM sampling, every weight and bias in hidden layers is associated with two specific points from the input space. The weight is determined by the difference between these points, and the bias is influenced by the interaction between this weight and one of the points. Building on SWIM sampling, the primary objective of this thesis is to apply SWIM methodology for sampling the weights of DeepONet in an iterative approach.

For the mathematical representation of the SWIM sampling, let Φ be a neural network with L hidden layers. The input space is denoted as $X \subseteq R^D$. The weights and biases for each layer l, where $l = 1, \ldots, L+1$, are represented by W_l and b_l respectively. The activation function is given by $g : R \mapsto R$. The output for the l-th layer of the neural network can be expressed as $\Phi^l(x) = g(W_l\Phi^{l-1}(x) - b_l)$, with $\Phi^0(x) = x$ for all $x \in X$. The number of neurons in the lth layer is represented by N_l. Here, $N_0 = D$ signifies the input dimension, and N_{L+1} denotes the output dimension. The notation $w_{l,i}$ refers to the i-th row of W_l, and similarly, $b_{l,i}$ stands for the ith entry of b_l.

Given a neural network Φ with L layers, for every layer l, from 1 to L, two points, $(x_{0,i}^{(1)}, x_{0,i}^{(2)})$ for $i = 1, \ldots, N_l$, are selected from the set $X \times X$ to determine a weight for that layer. The weight and bias are defined by:

$$w_{l,i} = s_1 \frac{x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}}{||x_{l-1,i}^{(2)} - x_{l-1,i}^{(1)}||^2}, \quad b_{l,i} = \langle w_{l,i}, x_{l-1,i}^{(1)} \rangle + s_2$$

where s_1 and s_2 are constants in $\mathbb{R}$. The values for s_1 and s_2 are determined based on the chosen activation function. Two activation functions are considered: ReLU and tanh. For the ReLU activation, the values are assigned as $s_1 = 1$ and $s_2 = 0$. This assignment ensures that $x^{(1)}$ maps to zero, while $s^{(2)}$ maps to one. In contrast, for the tanh activation, the values are assigned as $s_1 = 2s_2$ and $s_2 = \frac{ln(3)}{2}$. With this setting, $x^{(1)}$ and $x^{(2)}$ map to $\pm\frac{1}{2}$, respectively. With ReLU, these values help predict points between $x^{(1)}$ and $x^{(2)}$. For classification tasks with the tanh activation function, they determine if $x^{(1)}$ and $x^{(2)}$ are in different classes. After setting the weights and biases of the hidden layers, the only task

left is to solve an optimization problem to find the coefficients of a linear layer. This layer changes the output of the last hidden layer to the final output.

In the construction of SWIM sampling, a probability distribution is initially established over pairs of input data points, informed by known function values. Consequently, SWIM sampling offer a more data-centric weight sampling methodology, when compared to ELMs and random feature models. According to the evaluations, the SWIM method outperforms random feature models in accuracy on numerous cases. Moreover, since SWIM sampling do not optimize weights in a gradient-based manner, they achieve faster training speeds compared to conventional gradient-based training methods. Eliminating gradient-based optimization also removes the need for various hyperparameters, such as the learning rate, number of epochs, optimizer selection, and scheduling.

3. Iterative Sampling of Deep Neural Operators

DeepONet has shown its effectiveness in addressing partial differential equations [30]. There are extensions of DeepONet, like POD-DeepONet, which offer different approaches to standard DeepONet model. As discussed in Section 2, POD-DeepONet uses only the branch network during its training process. Instead of using a trunk network, it integrates a POD component that uses solutions of PDEs for computing POD. The outcomes from the branch network are combined with the results from the POD component. This integration effectively replaces the role of the trunk network found in the standard DeepONet framework.

DeepONet and POD-DeepONet, like many neural network architectures, are typically trained using gradient-based optimization algorithms such as SGD or Adam, which involve iterative weight adjustments. However, this standard training approach can encounter difficulties with high-dimensional datasets, a common scenario in neural network training. This situation highlights the need to address computational challenges in this area. Section 2 discussed the use of the SWIM algorithm as an innovative method for weight sampling in neural networks. This section is dedicated to implementing the SWIM methodology for training DeepONet via iterative sampling. The primary goal of this part of the thesis is to investigate how effective iterative sampling is, particularly when using SWIM, in the training process of DeepONet.

In the forthcoming discussion of iterative sampling in DeepONet and the sampled POD-DeepONet, the following notations will be used for dataset: let $\xi \in \mathbb{R}^m$ represent the grid of m points. Consider $v^i \in \mathbb{R}^m$ for $i = 1, \dots, N$ as the initial conditions and $u^i \in \mathbb{R}^m$ for $i = 1, \dots, N$ as the solutions, with N being the number of data samples. The initial conditions and the solutions can be expressed in matrix notation as follows: $V = (v^1, \dots, v^N)^\top \in \mathbb{R}^{N \times m}$, and $U = (u^1, \dots, u^N)^\top \in \mathbb{R}^{N \times m}$, respectively.

For the DeepONet model, let $b(v) \in \mathbb{R}^p$ represent the branch network and $t(\xi^j) \in \mathbb{R}^p$ denote the trunk network, while $b_0(\xi^j) \in \mathbb{R}$ is the bias component. When expressed in matrix form, these components are represented as: $B := b(V) \in \mathbb{R}^{N \times p}$, indicating the branch network matrix; $T := t(\xi) \in \mathbb{R}^{m \times p}$, showing the trunk network matrix; and $b_0 := b_0(\xi) \in \mathbb{R}^m$, which is the bias vector. In the context of the POD-DeepONet, in addition to the previously established notation, the POD components are represented as

$\Phi(\xi) = [\psi_1(\xi), \ldots, \psi_p(\xi)] \in \mathbb{R}^{m \times p}$, where these components correspond to the p largest POD modes, and $\psi_0(\xi) \in \mathbb{R}^m$ represents the mean of the solutions at grid ξ.

3.1. Sampled POD-DeepONet

In the study of SWIM, researchers conducted experiments using the SWIM algorithm to sample weights in different neural network architectures, including POD-DeepONet. The primary goal was to train POD-DeepONet by sampling its weights with the SWIM method. To facilitate sampling, the POD-DeepONet was transformed into a fully-connected network, utilizing the orthogonality of its components. An orthogonal matrix, sometimes known as an orthonormal matrix, is a square matrix composed of real numbers. Its unique feature is that its rows and columns are orthogonal, meaning they are perpendicular to each other. In cases where it's referred to as orthonormal, the rows and columns are also of unit length. The orthogonality feature that is used to restructure the DeepONet is $Q^\top Q = I_p$, where Q represents the matrix, $Q^\top$ is its transpose, and I stands for the identity matrix of size p.

The standard POD-DeepONet equation in matrix notation is

$$G(V)(\xi) = B\Phi^\top + \psi_0(\xi).$$

To derive the formulation for B, we start with the property of orthogonality of Φ, which states that $\Phi^\top \Phi = I_p$. This property allows us to manipulate the original equation. We start by subtracting $\psi_0(\xi)$ from both sides:

$$G(V)(\xi) - \psi_0(\xi) = B\Phi^\top.$$

Next, we multiply both sides of the equation by Φ:

$$(G(V)(\xi) - \psi_0(\xi))\Phi = B\Phi^\top \Phi.$$

Since $\Phi^\top \Phi = I_p$, we replace it in the equation:

$$(G(V)(\xi) - \psi_0(\xi))\Phi = BI_p.$$

The identity matrix I_p does not change the matrix it multiplies, so $BI_p = B$. Therefore, the equation simplifies to

$$B = (G(V)(\xi) - \psi_0(\xi))\Phi.$$

This restructured equation for the branch network B allows the transformation of training data to facilitate the sampling of a fully-connected network representing B. This approach to sampling POD-DeepONet using the SWIM method is elaborated in Section 2.3.

3.2. Iterative Sampling of DeepONet

This thesis concentrates on applying the SWIM sampling approach to the DeepONet architecture. In the POD-DeepONet framework, the focus is on training (or sampling) exclusively the branch network, while substituting the trunk network with a POD component. This approach is feasible thanks to a key modification: the revised equation for B facilitates the sampling process within a fully-connected network.

In the iterative sampling process for DeepONet, the original framework is maintained, meaning that both the trunk and branch networks are transformed into a fully-connected network configuration. This is achieved by assuming the orthogonality of the components under consideration for sampling. The process involves fixing one network at a time (either the trunk or the branch) while sampling the other network.

In the iterative sampling setting, the branch network B is sampled using the approach that is used in POD-DeepONet, as explained in previous subsection. The derivation for making the trunk network T a fully-connected network is again begins with the original DeepONet equation in matrix form

$$G(V)(\xi) = BT^\top + b_0.$$

To derive the formulation for T, we utilize the property of orthogonality of B, which implies $B^\top B = I$, where I is the identity matrix. This property is essential for manipulating the original equation. We begin by subtracting b_0 from both sides:

$$G(V)(\xi) - b_0 = BT^\top.$$

Next, we multiply both sides of the equation by $B^\top$:

$$B^\top(G(V)(\xi) - b_0) = B^\top BT^\top.$$

Given that $B^\top B = I$, we substitute it into the equation:

$$B^\top(G(V)(\xi) - b_0) = IT^\top.$$

Since multiplying by the identity matrix I does not change the matrix, the equation simplifies to:

$$T^\top = B^\top(G(V)(\xi) - b_0).$$

Taking the transpose of both sides, we arrive at the final expression for T:

$$T = (G(V)(\xi) - b_0)^\top B.$$

This formulation of T allows for the transformation in the context of the given equation, assuming the orthogonality of matrix B.

After restructuring the trunk network T and the branch network B, the DeepONet equation has been suitably adjusted for adaptation to SWIM sampling. The implemented iterative sampling algorithm primarily focuses on developing fully-connected structures within both T and B. This restructuring is crucial for the adaptation of SWIM sampling in the DeepONet framework.

The iterative sampling algorithm for DeepONet, as outlined in Algorithm 1, involves several steps to sample DeepONet effectively. Initially, the algorithm starts by defining two key elements: $b_{(0)}$ and $T_{(0)}$. Here, $b_{(0)}$ is the bias, which represents the mean of the solutions. Once set, b_0 remains fixed for the remainder of the process. $T_{(0)}$ signifies the p largest POD modes of the solutions U. For the POD computation, the solutions U are first centered by subtracting their mean. Then, the POD is computed using the `numpy.linalg.svd()` function, which yields the right singular matrix. From this matrix, the largest p modes are obtained as $T_{(0)}$.

After initializing $b_{(0)}$ as the mean of the solutions U and setting $T_{(0)}$ as the p largest POD modes of U, the iterative sampling process of the DeepONet algorithm begins. The process starts with an iteration counter k, beginning at 1. In each iteration, the branch network $B_{(k)}$ is first sampled using the previously updated trunk network $T_{(k-1)}$. After sampling $B_{(k)}$, its transformed version, $\hat{B}_{(k)} := B_{(k)}(V)$, is computed with the new weights, orthogonalized, and then reassigned to update $B_{(k)}$. The orthogonalization process is executed using the function `numpy.linalg.qr()`. This updated branch network is then used to sample the trunk network in the same iteration.

Proceeding within the same iteration, the trunk network $T_{(k)}$ is then sampled using the newly updated branch network $B_{(k)}$. Following its sampling, $T_{(k)}$ is subjected to a similar transformation and orthogonalization process, which leads to an updated version of $T_{(k)}$ that is prepared for the subsequent iteration. This procedure is carried out iteratively, with each cycle involving the sampling and orthogonalization of both the branch and trunk networks. The algorithm continues in this sequence until it satisfies a predefined convergence criterion or until it reaches the maximum number of iterations denoted as K.

In the iterative sampling process, the algorithm sets a maximum of $K = 10$ iterations and incorporates a patience mechanism to determine convergence. Convergence is achieved if there is no notable improvement in the network's performance across three consecutive iterations, which is measured against a predetermined tolerance level. The performance of the network is evaluated using the relative L2 mean loss. If the absolute difference in loss between successive iterations is smaller than a tolerance value of 10^{-6} for three consecutive iterations, the model is considered to have converged. Upon completion of the iterative sampling process, a final step is conducted where the branch network undergoes one last sampling update. This final step updates the branch network to its concluding

state, thus finalizing the iterative sampling process for the DeepONet, as outlined in Algorithm 1.

To summarize, the iterative sampling approach involves sampling the weights of both the trunk and branch networks through iterative processes. Initially, in the first iteration, the branch network is sampled using the POD modes from the solutions U, and this sampled branch is then used to sample the trunk network. In the following iterations, this cycle continues, using the continuously updated versions of both networks. Each iteration includes the sampling of both the trunk and branch networks. As hypothesized in this thesis, this method is expected to lead to the model's consistent improvement over time.

Algorithm 1 Iterative Sampling Algorithm for DeepONet. The process involves iteratively updating the branch and trunk networks of the DeepONet using the initial conditions V and PDE solutions U, respectively. The variable `last_iter` is initialized to track the last iteration index, providing a reference point in case the algorithm converges before reaching the maximum number of iterations K.

$b_{(0)} \leftarrow \frac{1}{N}\sum_{i=1}^{N} U^i$;
$T_{(0)} \leftarrow \text{POD}(U)$, retaining p largest components;
last_iter $\leftarrow 0$; ▷ Track the last iteration index in case of convergence
for $k = 1, 2, \ldots, K$ **do**
    last_iter $\leftarrow k$; ▷ Update last iteration index
    Sample $B_{(k)}$ using $T_{(k-1)}$;
    $\hat{B}_{(k)} \leftarrow B_{(k)}(V)$;
    $B_{(k)} \leftarrow \text{orthogonalize}(\hat{B}_{(k)})$;
    Sample $T_{(k)}$ using $B_{(k)}$;
    $\hat{T}_{(k)} \leftarrow T_{(k)}(\xi)$;
    $T_{(k)} \leftarrow \text{orthogonalize}(\hat{T}_{(k)})$;
    if converged **then**
        break;
    end if
end for
Sample $B_{(\text{last\_iter}+1)}$ using $T_{(\text{last\_iter})}$;

3.3. Experiments

This section of the thesis is dedicated to evaluating the performance of the proposed iterative sampling algorithm for DeepOnet through a series of designed experiments. These experiments were conducted using a variety of hyperparameters, carefully selected to span a comprehensive search space. The objective was to determine the optimal set of hyperparameters that yield the best performance for the algorithm. A thorough comparison of the results obtained with different hyperparameters was carried out to determine the most effective configurations.

The datasets for these experiments were created using Python programming. The upcoming subsections will provide detailed results of the addressed problems and explain the implementation techniques for dataset generation. A thorough analysis of these methods is significant for evaluating the iterative sampling method's effectiveness. This examination will analyze the method's practical use and performance across different experiments. The focus of this analysis is to assess the utility of SWIM sampling through the iterative sampling method for the DeepONet architecture, which offers key insights into its performance.

3.3.1. Burgers' Equation

For the experiments, the one-dimensional Burgers' equation [3] is one of the PDEs that is focused on. The Burgers' equation is a fundamental equation in fluid dynamics and it is used to describe systems with a viscous fluid that has both spatial and temporal properties. For example, in a system like a tube with a viscous fluid flowing inside, the equation can describe the speed of the fluid at different locations as time progresses. The Burgers' equation is described as

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \eta\frac{\partial^2 u}{\partial x^2},$$

where x and t represent the spatial and temporal coordinates, respectively. The term $u(x,t)$ denotes the fluid's speed at these coordinates and η is the viscosity coefficient. The viscosity coefficient η is a physical property of the fluid, and it acts to smooth out rapid changes in the fluid's speed, thus preventing discontinuities in the solution.

Burgers' equation is one of the equations used to generate a dataset for training and testing purposes in this thesis. Our dataset for the Burgers' equation is generated within the spatial interval $[0, 2\pi]$. The equation is solved under periodic boundary conditions, meaning $u(0,t) = u(2\pi,t)$ for all t. Various initial conditions are denoted as $u(x, t=0)$ and they are formed using an inverse Fourier transform of normally distributed coefficients. Therefore, variety of different scenarios are produced. The Burgers' equation is then evolved from $t = 0$ to $t = 1$ with a viscosity coefficient of 0.1. The initial conditions $u(x, t = 0)$ and

the corresponding solutions at $t = 1$, denoted as $u(x, t = 1)$, are plotted for a selection of samples in Figure 3.1. The different colors in the graphs correspond to different samples in the dataset. This variety in data allows for extensive and rigorous training and testing of the models. It also introduces a challenge for the DeepONet model to effectively represent the data.

The generated dataset contains 15000 samples, where each sample includes an initial condition and its corresponding evolved state, distributed over a grid of 256 points. Each initial condition is evolved in accordance with the Burgers' equation.

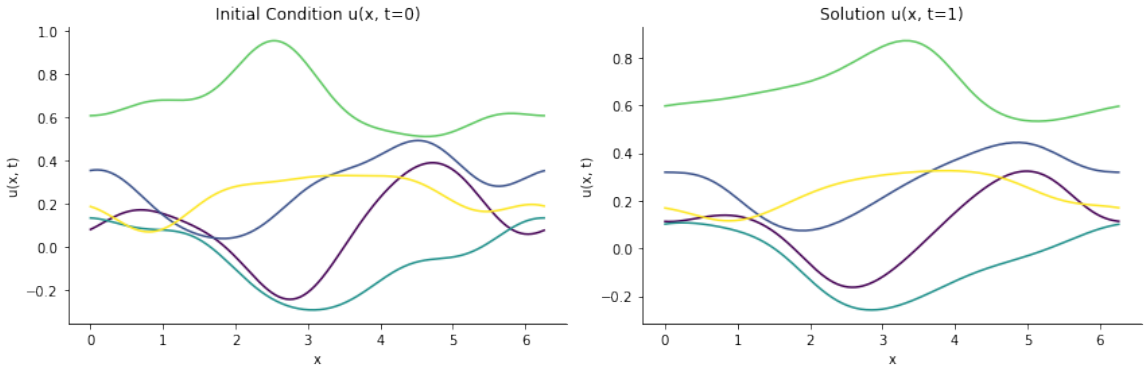

Figure 3.1.: The left figure displays the initial conditions $u(x, t = 0)$, while the right figure displays the solutions at $t = 1$ after the Burgers' equation has evolved the system. Each line in the graph corresponds to a distinct sample. Matching colors between the two figures indicate the progression of the same sample from its initial state at $t = 0$ to its evolved state at $t = 1$.

The iterative sampling approach in DeepONet involves a unique configuration for both the branch and trunk networks, each including a dense layer and a linear layer. The dense layer is characterized by various hyperparameters, including its width, activation function, and parameter sampler. On the other hand, the linear layer's primary hyperparameter is the regularization scale. Another important aspect of this iterative sampling method for DeepONet is the number of POD modes used in the initial iteration. This choice plays an important role in the algorithm's performance. The list of the hyperparameters, along with their respective search spaces, is shown in Table 3.1.

Table 3.1.: The list of the hyperparameters and their values for tuning in DeepONet, applicable to both the trunk and branch networks. Each network is configured with layers that share the same hyperparameters and value ranges. Specifically, the dense layers in both branch and trunk networks employ matching activation functions and parameter samplers (e.g., 'relu' or 'tanh' for both hyperparameters). Furthermore, the width of the dense layers and the regularization scale for the linear layers are chosen to be the same across both networks. For instance, if a dense layer width of 1024 is selected, it will be 1024 for both the trunk and branch networks, and similarly, a chosen regularization scale like 10^{-4} will be applied equally to both networks' linear layer.

Hyperparameter	**Values**
Dense Layer Width	256, 512, 1024, 2048
Dense Layer Activation	tanh, relu
Dense Layer Parameter Sampler	tanh, relu
Linear Layer Regularization Scale	$10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}$

In Figure 3.2, we explore the impact of activation functions, layer width, and the number of POD modes on the relative L2 loss on training and test sets. Each graph within the figure corresponds to a distinct combination of activation function and layer width. For each of these combinations, the figure illustrates the lowest training loss achieved across various POD mode scenarios. For the corresponding experiments with the lowest training loss, their test performance is also shown. It is important to note that the term 'width' in this context refers to the width of both the branch network and the trunk network. Additionally, the 'activation' mentioned represents the functions utilized for both activation and parameter sampler hyperparameters.

The examination of the results shows a clear difference in performance based on the number of POD modes used. In particular, using only 2 or 4 of the largest POD modes in the initial iteration results in poorer performance of the sampled DeepONet. The training loss is significantly higher—about 10^3 times greater with 2 POD modes and 10^2 times higher with 4 POD modes—compared to cases where at least 8 largest POD modes are utilized. The observed poorer performance of the sampled DeepONet with only 2 or 4 of the largest POD modes can be related to the limited data representation capacity of such a low number of modes. When only a few modes are used, the model may not capture enough of the essential characteristics and complexities in the dataset. This leads to a significant loss in the ability to accurately model the PDEs, which results in a higher training loss. In contrast, using a larger number of POD modes, such as 8 or more, allows for a more comprehensive representation of the data, which in turn leads to improved model performance and lower training loss. The test loss results follow the same pattern with the training loss, but it is observed that test loss is higher than the training loss for the experiments, which indicates overfitting.

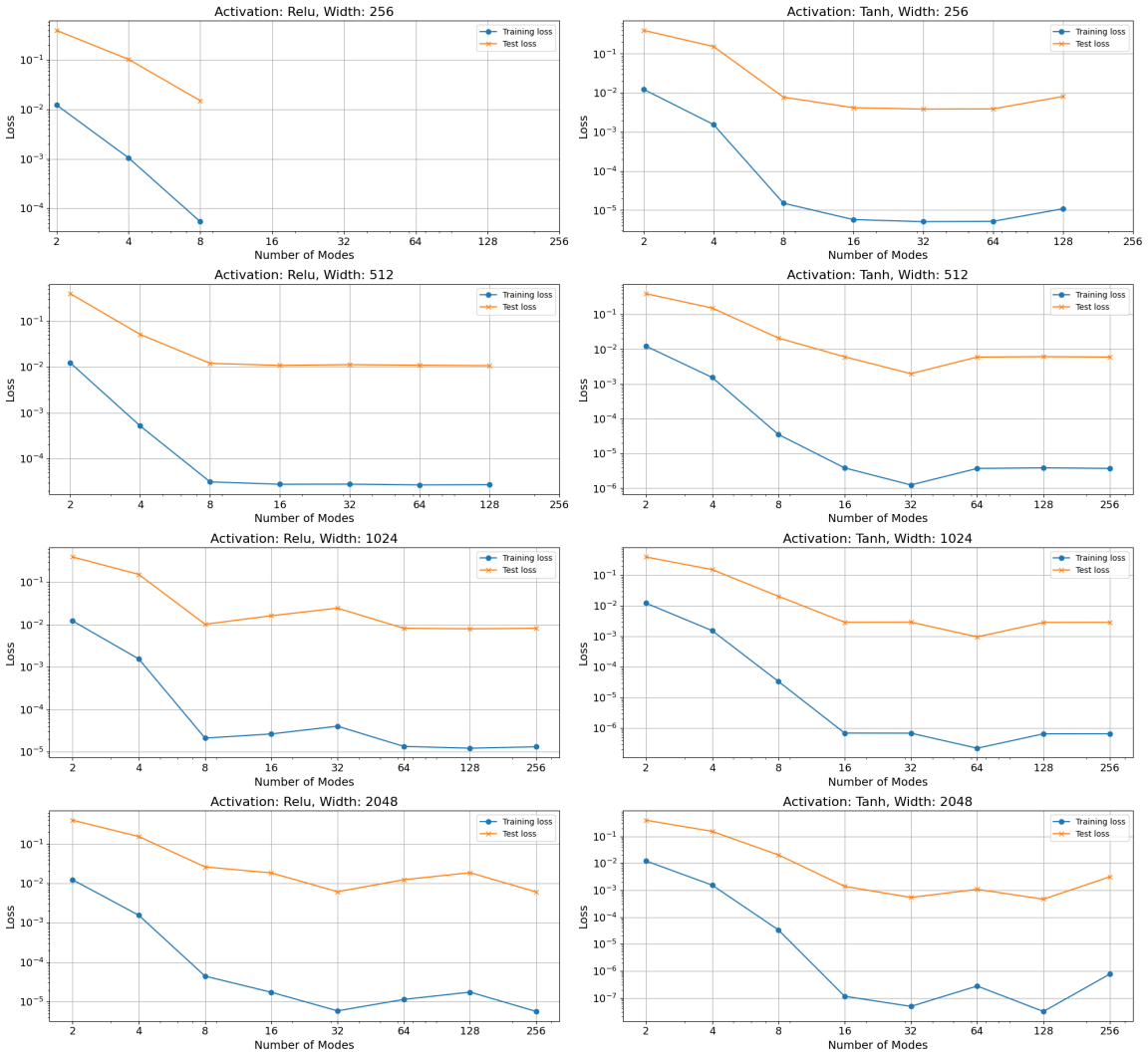

Figure 3.2.: The figure shows the various configurations tested using the Burgers' dataset, focusing on the interaction between activation functions, layer width and number of POD modes. Each individual plot corresponds to a unique combination of activation function and layer width. The x-axis shows the number of POD modes, while the y-axis, which is on logarithmic scale, indicates the lowest training loss achieved in each specific setting (blue line). For the corresponding experiments with the lowest training losses, their test loss is shown with orange line. Notably, there are gaps in the data where results are absent, which indicates that in those particular combinations of settings, the experiments did not to converge.

A noticeable trend appears when the number of largest POD modes used is 16 or higher: the tanh activation function consistently outperforms the relu activation across all configurations. The lowest training loss is observed when the dense layers of both networks have a width of 2048, the model employs 128 largest POD modes with tanh activation and parameter sampler, and regularization scale of 10^{-10} is applied to the linear layer.

Particularly, the combination of tanh activation and parameter sampler results in more effective performance than relu in cases where the number of POD modes is 16 or more. While there are exceptions, the training loss tends to decrease as the layer width increases, except in scenarios with 2 or 4 POD modes. The training performance improves with an increasing number of POD modes, but the optimal performance is observed with 128 POD modes, rather than the maximum of 256 POD modes. Figure 3.3 illustrates these trends, showing the impact of the number of POD modes and layer width on training performance for different activation functions and the parameter sampler. This figure offers a clearer visualization of the effects and comparison of different POD modes. Understanding the influence of POD modes is crucial, as it directly affects the performance of the sampled DeepONet. For the relu activation function, an enhancement in training performance is observed with an increase in layer width, particularly when utilizing 8 or more POD modes. On the other hand, with tanh activation, there is a general trend of improved training performance with 16 or more POD modes, although exceptions to this pattern do occur.

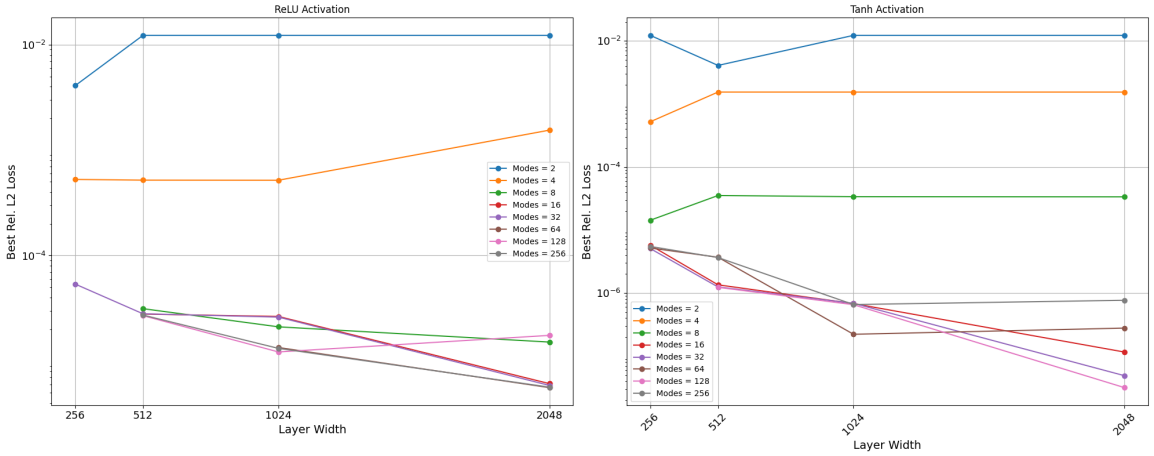

Figure 3.3.: The x-axis shows the layer width, and the y-axis displays the best relative L2 training loss on a logarithmic scale for the Burgers' dataset. Each line represents a different number of POD modes. The left part illustrates results for the relu activation, and the right part for the tanh activation.

In Figure 3.4, the relationship between the number of POD modes and the total training time of the model is shown. The graph shows that, with a few exceptions, the overall training time generally increases as the number of POD modes increases. Notably, the training time for 8 POD modes is greater than for 16 and 32, but beyond these points, the trend of increasing time with more POD modes continues. This pattern suggests a trade-off inherent in sampled DeepONet training. While incorporating more POD modes can potentially boost model performance, it concurrently escalates computational demands. The graph displays a clear trend of escalating training time with the increase in modes, which shows the balance between computational efficiency and model complexity. This trend indicates that the choice of the number of POD modes is a critical decision point in the design of the network, as it directly impacts the practicality of the model in terms of both efficiency and resources.

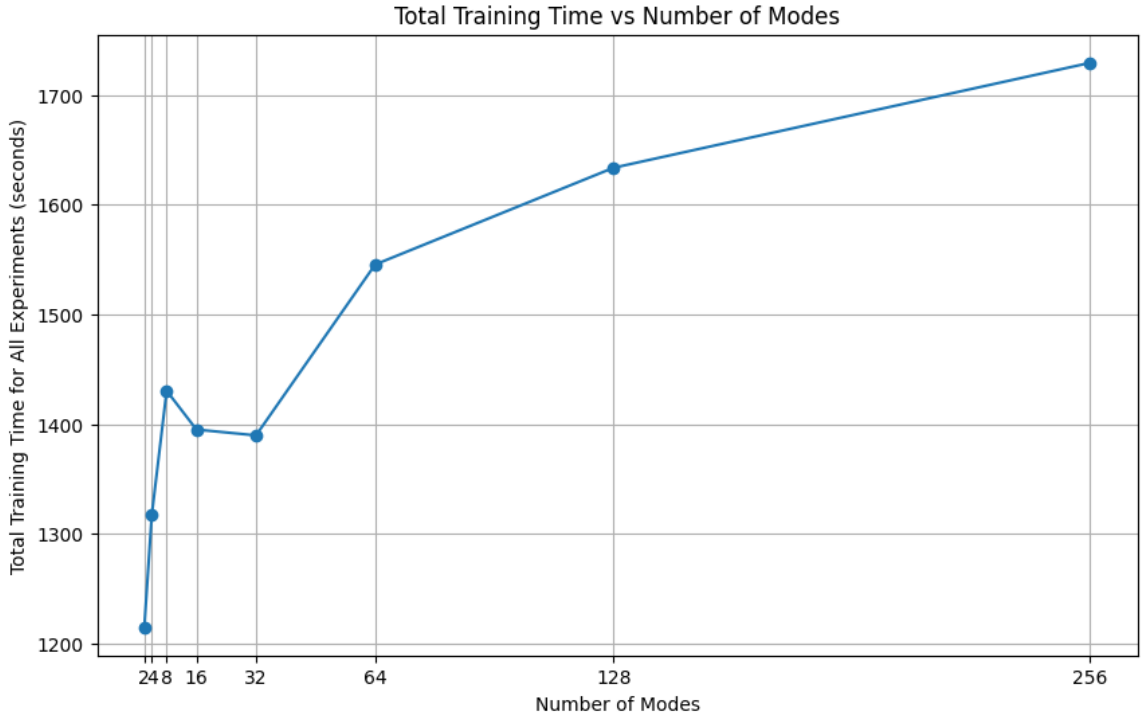

Figure 3.4.: Cumulative training time for the same number of experiments across different POD mode settings for Burgers' dataset. The number of POD modes is shown on the x-axis and the total time in seconds is shown on the y-axis.

The influence of the regularization scale in the linear layer on training performance is shown in Figure 3.5. The results show that for configurations using 2, 4, or 8 POD modes during the initial iteration, the regularization scale has little to no impact on the training

performance, suggesting these settings are insensitive to this hyperparameter. However, as the number of POD modes increases, choosing the right regularization scale becomes important. With some exceptions, a scale of 10^{-4} generally leads to the poorest training results, showing that this level of regularization might be insufficient for model generalization. On the other hand, scales of 10^{-8} or 10^{-10} tend to produce the most favorable training results. The optimal scale, however, seems to depend on the number of POD modes used, which shows the relationship between model complexity and the need for effective regularization.

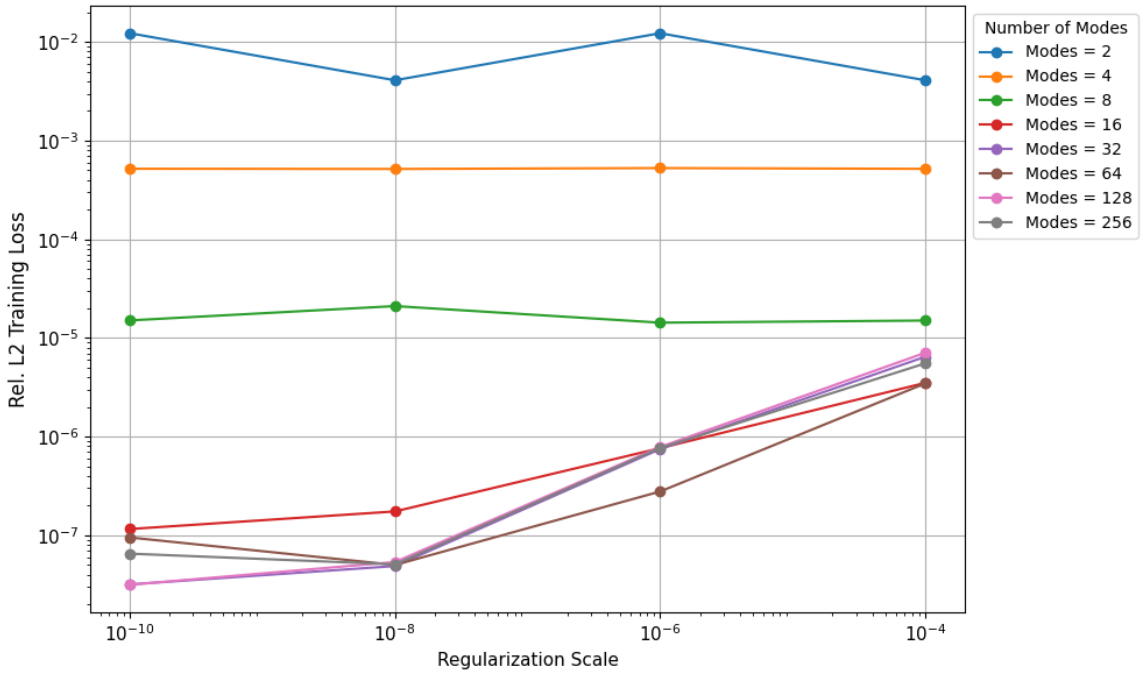

Figure 3.5.: Impact of the regularization scale in the linear layer on training performance for Burgers' dataset. The x-axis denotes the tested regularization scale values, while the y-axis shows the minimum training loss achieved for each respective POD mode setting, which is displayed on a logarithmic scale. Each line in the graph represents a different setting of the number of POD modes as established in the initial iteration.

The analyses conducted so far have primarily focused on the best training performance of DeepONet through iterative sampling. Figure 3.6 illustrates the network's best test performance using the one-dimensional Burgers' equation dataset. The best training results were obtained with a setup of 128 POD modes, a layer width of 2048, 'tanh' as the activation function and parameter sampler, and a regularization scale of 10^{-10}. However,

the network demonstrated its most effective performance in testing with a different configuration: 16 POD modes while maintaining a layer width of 2048, 'tanh' activation and parameter sampling, and a regularization scale of 10^{-10} in the linear layer. The test performance analysis indicates varied optimal configurations depending on the number of POD modes. Specifically, a layer width of 1024 yielded the lowest test loss for 64 and 256 POD modes. In contrast, a wider layer of 2048 was optimal for the remaining POD mode settings. Regarding the regularization scale in the linear layer, a scale of 10^{-8} was most beneficial for setups with 32 and 64 POD modes. For other configurations with different POD mode settings, the best test performance observed when a regularization scale of 10^{-10} was applied.

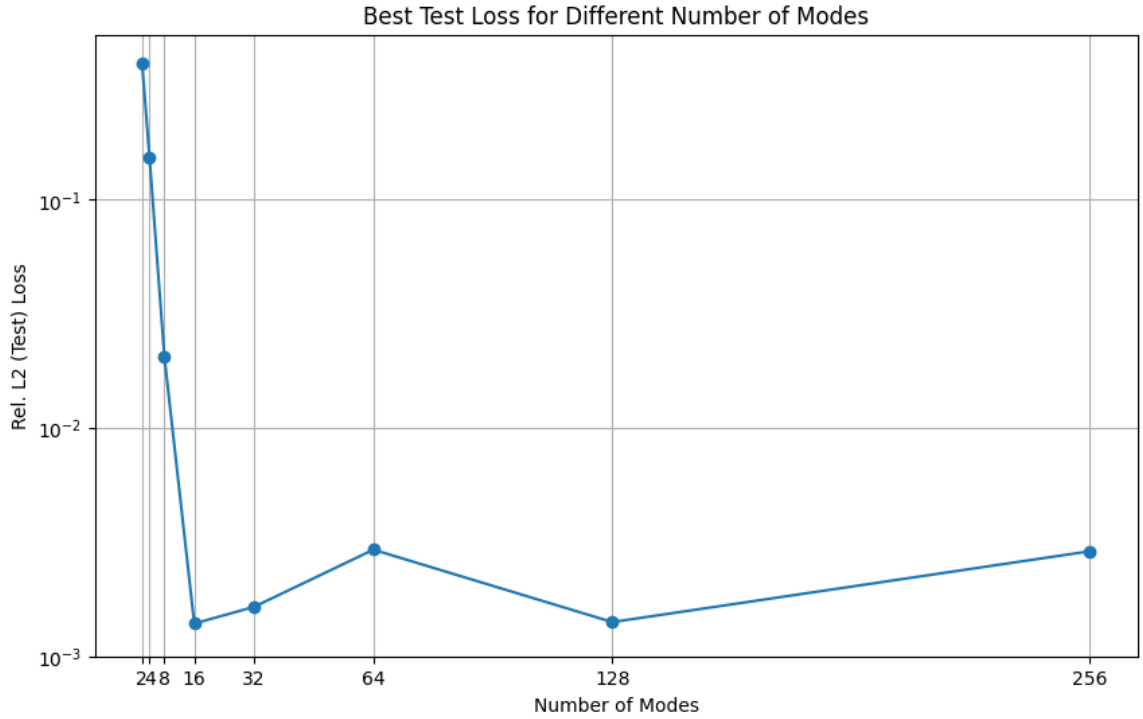

Figure 3.6.: Model performance assessment on the one-dimensional Burgers' test dataset. The x-axis indicates the number of POD modes employed during the initial iteration of iterative sampling, while the y-axis shows the lowest test loss achieved among all experiments and configurations, shown on a logarithmic scale.

As stated previously, the main goal of this thesis is to explore the effectiveness of the iterative sampling approach in the context of the DeepONet. This necessitates a thorough

review of each iteration to understand the dynamics of this method. As discussed, Figure 3.6 displayed the best test performances across various POD mode configurations. To assess the efficiency of the iterative approach for the sampled DeepONet, these results are analyzed in detail. Figure 3.7 represents the training and testing losses at each iteration for the experiments that achieved the most optimal test performance.

This figure reveals that the iterative sampling approach improves the network's efficiency multiple times, particularly with 32 and 128 POD modes in the initial iteration. Notably, in the scenario using 32 POD modes, the iterative method reduces the loss measurement multiple times beyond the initial iteration. Similarly, with 128 POD modes, the approach improves the learning performance of the network multiple times. For other scenarios, the iterative sampling decreased the loss only once (in the second iteration). For some scenarios, the figure additionally shows that after several iterations, the training loss becomes lower than the test loss, suggesting an instance of overfitting. This indicates that while the iterative sampling method may improve learning, it may lead to the model becoming too finely tuned to the training data. As a result, the network's ability to generalize to new, unseen data could be compromised, which denotes the importance of monitoring for overfitting in such iterative training processes.

To summarize, the first experiment involved using the iterative sampling method with the one-dimensional Burgers' equation on sampled DeepONet. An extensive search is carried out for the best hyperparameters to properly evaluate this iterative sampling. In this process, a series of experiments were conducted. For the majority of these experiments, the iterative sampling improved the network performance only once, which happened in the second iteration. For these cases, doing more iterations did not improve the performance of the model. However, in some specific cases, like in experiments involving 32 or 128 POD modes, the performance of the sampled DeepONet did improve multiple times.

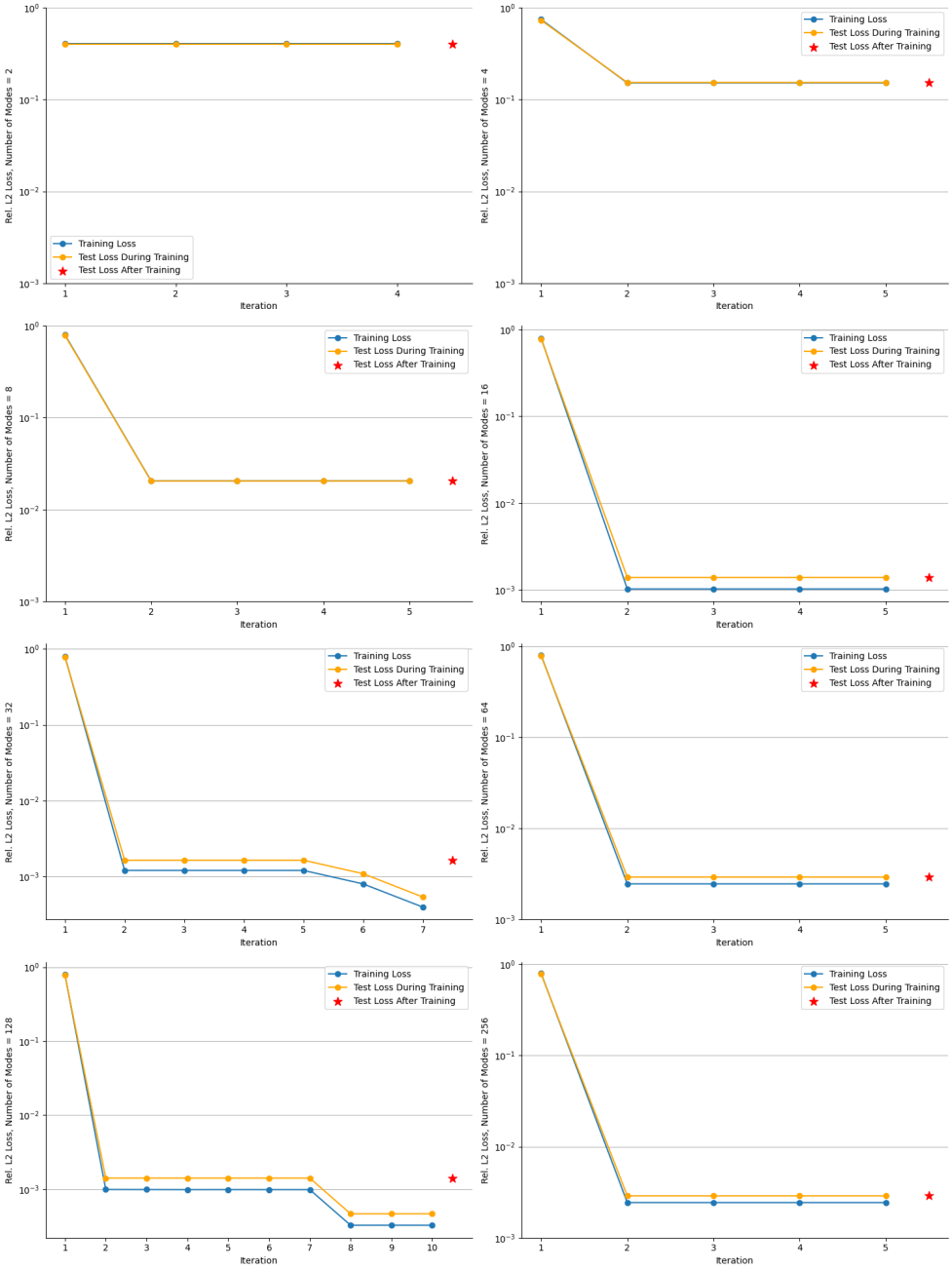

Figure 3.7.: Iterative sampling performance on the Burgers' dataset. This figure plots the evolution of training and test losses over the course of iterative sampling for the experiments shown in 3.6. The x-axis shows the iteration count, while the y-axis shows the loss values on logarithmic scale. The training loss is represented by the blue line, and the test loss during training is represented by the orange line. The final test loss, observed at the completion of training, is represented by a red star for each case of POD settings.

3.3.2. Wave Equation

The second experiment uses the Wave equation, which is an important equation in physics for examining wave motion. This equation is useful in understanding various wave types, including sound and light, and proves beneficial in diverse fields ranging from engineering to the study of natural phenomena like earthquakes [1]. The Wave equation in its fundamental form is

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u.$$

Here, u is a function that shows the position of the wave at any place and time, and c is the speed at which the wave travels. The term $\nabla^2 u$ describes the wave's shape and its spatial variations.

In this particular experiment, we focus on a one-dimensional version of this equation, which is

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

By simplifying the equation to one dimension, we are able to scrutinize the wave movements along a single line. This experiment uses reflective boundary conditions in the one-dimensional Wave equation to create a variety of wave patterns, each originating from distinct initial conditions. Reflective boundaries cause the wave to reflect back into the medium when it reaches the boundary, which simulates the effect of waves in a confined space.

In the experiment, a dataset is generated by computing solutions to the one-dimensional wave equation under various initial conditions. These solutions are obtained using the `py-pde` library, which is an effective solver for partial differential equations. The initial conditions for the wave equation are generated using a Gaussian function defined as $u_0(x) = Ae^{-(x+S)^2 \times \text{scale}}$. This formula ensures a rich variety of wave patterns by varying the amplitude A and the position shift S within set limits. And the $scale$ parameter controls the spread of the Gaussian function. The initial wave state u_0 is determined by applying this Gaussian function to a range of values. Alongside u_0, the initial velocity of the wave, denoted as v_0, is calculated as $v_0(x) = x \times u_0(x)$, which represents the initial speed of each point in the wave. This approach to generating initial conditions allows examining wave behaviors under various scenarios, and makes it challenging for the neural network model to learn. The representations of the initial wave and initial speed are shown in Figure 3.8, and the solutions are shown in Figure 3.9.

The dataset contains 1500 samples, where each sample includes an initial condition and its corresponding evolved state, distributed over a grid of 256 points. Each initial condition is evolved in accordance with the Wave equation.

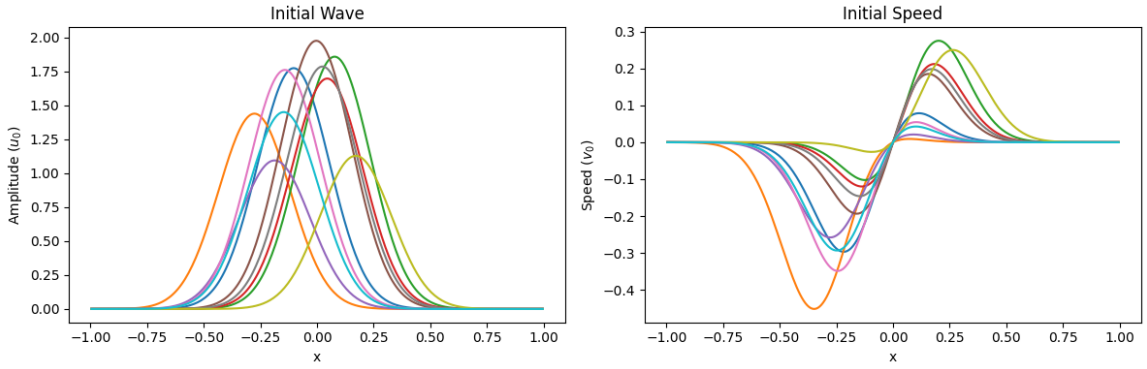

Figure 3.8.: Visualization of initial wave and speed parameters. On the left, the initial wave amplitude (y-axis) is plotted against the spatial domain (x-axis), showing the amplitude variations. On the right, the initial speed of the wave (y-axis) is shown, correlating to the same spatial domain (x-axis). The plots display the first 10 samples from a total of 1500. Each line in the plot represents a distinct sample, with color coding consistent across both visualizations.

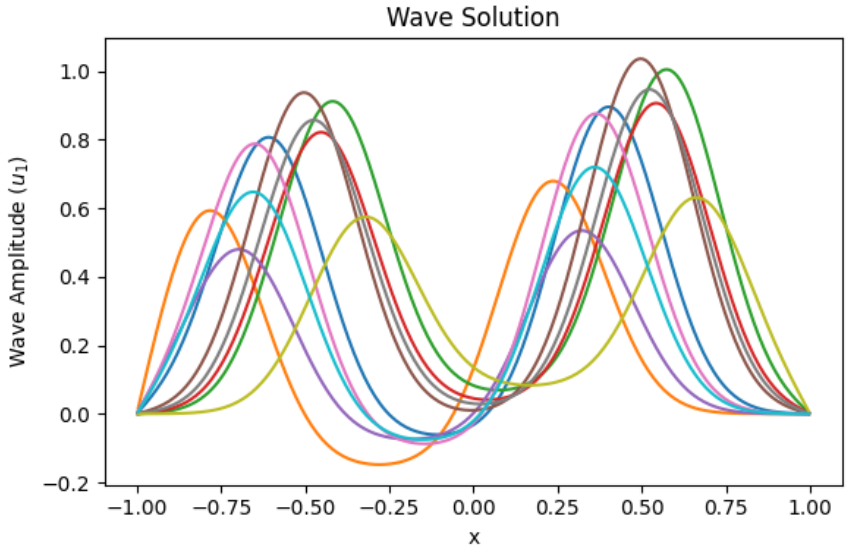

Figure 3.9.: Evolution of wave solutions derived from initial conditions. This plot presents the solutions of the wave equations based on the initial wave and speed conditions shown in Figure 3.8. The solutions are visualized over the same spatial domain on the x-axis, with the y-axis representing the amplitude of the wave over time. The color scheme is maintained from the initial conditions.

In the Wave equation experiment, a hyperparameter search similar to the one conducted for the Burgers' dataset was conducted. This search was aimed at evaluating iterative sampling in various settings. The hyperparameters and search space used are identical to those in the Burgers' experiment. The list and the range of these hyperparameters are detailed in Table 3.1.

In these experiments, the main goal is to investigate the effects of layer width, activation functions, and parameter samplers on the training efficiency of the sampled DeepONet. This investigation considers different numbers of POD modes used in the first iteration. Figure 3.10 demonstrates these connections. Each plot in the figure shows a combination of an activation function and layer width. Similar to the Burgers' experiment, 'width' here refers to the size of both the branch network and the trunk network. The term 'activation' is used to describe the functions for both activation and parameter sampling. It is noted that for cases with 2 or 4 POD modes, changes in layer width or activation functions have minimal impact on training results, likely due to the limited data representation capacity with such a small number of modes. However, for a higher number of POD modes, the relu activation demonstrates improved performance with increased layer width. On the contrary, the performance with tanh activation remains fairly uniform across various layer widths. The best training loss was achieved using 16 POD modes, unlike in the Burgers' experiment where the optimal training loss was reached with 128 POD modes. This was achieved with a layer width set at the maximum of 2048, using tanh activation and a parameter sampler, along with a regularization scale of 10^{-10} for the linear layer. Despite this variance in the number of POD modes, the other hyperparameters associated with the best training performance were consistent in both experiments. The pattern of test loss results mirrors that of the training loss, yet it is noticeable that the test loss exceeds the training loss in the experiments, which indicates overfitting.

The Burgers' experiment highlights the critical importance of the number of POD modes used in the initial iteration. This significance is also evident in the Wave experiment. To analyze the impact of the number of POD modes, Figure 3.11 offers insightful observations. With the relu activation function, the best training loss was achieved with 32 POD modes in the first iteration. Conversely, using the tanh activation function led to the lowest training loss with 16 POD modes. In both scenarios, the optimal training outcomes were reached with a layer width of 2048, the largest value in the considered range for layer width. These findings suggest that the number of POD modes chosen initially plays a critical role in the subsequent performance of the iterative sampling process. This conclusion is supported by results from both experiments across different datasets. It is important to note, however, that there is no straightforward formula linking an increase in POD modes to better training performance. Generally, performance tends to improve when using more than a minimal number of POD modes, such as 2 or 4. Nevertheless, for optimal results, it is crucial to fine-tune the number of POD modes for each specific problem and dataset.

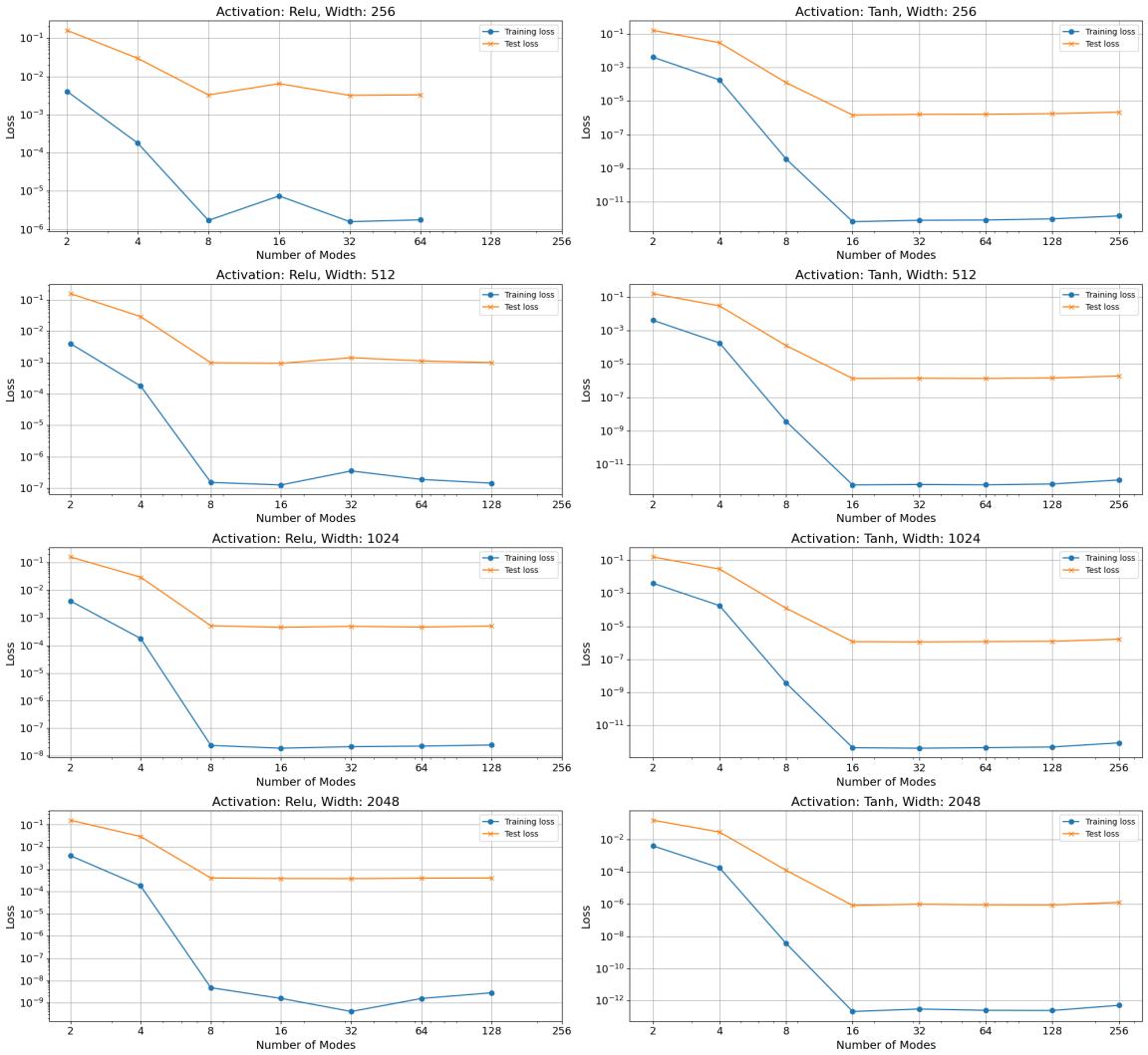

Figure 3.10.: The figure represents the various configurations tested using the Wave dataset, which focuses on the interaction between activation functions, layer width and number of POD modes. Each plot corresponds to a unique combination of activation function and layer width. The x-axis shows the number of POD modes, while the y-axis, which is on logarithmic scale, indicates the lowest training loss achieved in each specific setting. For the corresponding experiments with the lowest training losses, their test loss is shown with orange line. The gaps in the data where results are absent indicates that in those particular combinations of settings, the experiments did not to converge.

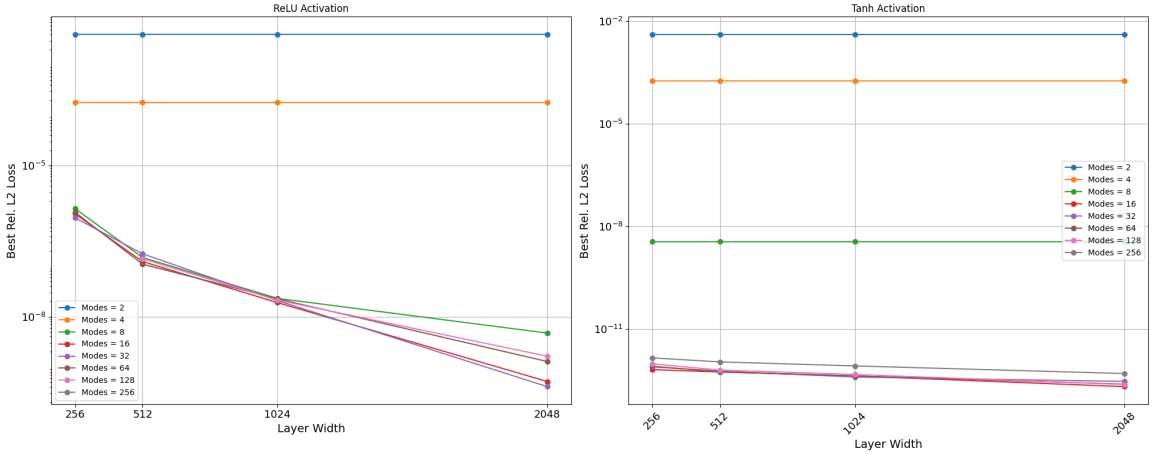

Figure 3.11.: The x-axis shows the layer width, and the y-axis displays the best relative L2 training loss on a logarithmic scale for Wave dataset. Each line represents a different number of POD modes. The left part illustrates results for the relu activation, and the right part for the tanh activation.

Figure 3.12.: Total training time for the same number of experiments across various POD mode settings for the Wave dataset. The number of POD modes is shown on the x-axis and the total time (in seconds) is shown on the y-axis.

As it is observed, the number of POD modes is s a critical hyperparameter for the iterative sampling of DeepONet. However, it is important to note that increasing the number of POD modes, while beneficial, also comes with a cost in terms of computational resources. Using a relatively larger number of modes tends to increase training time, making it more computationally demanding. As illustrated in Figure 3.12, the aggregate training time for all experiments escalates with the increase in the number of POD modes, starting from 4 POD modes. An exception occurs when only 2 POD modes are used, where the total training time exceeds that of experiments with 4, 8, and 16 modes.

The significance of the regularization scale in the linear layers of the branch and trunk networks is a critical aspect of the sampling process. Figure 3.13 illustrates the lowest training loss results for each regularization scale value within the search space, relative to the number of POD mode settings. When employing a small number of POD modes, such as 2 or 4, it becomes apparent that the regularization scale has little or no impact on the model's performance. This observation highlights once again the essential role of POD modes in determining model efficiency. In contrast, with an increased number of POD modes, the importance of the regularization scale escalates. Optimal training performances are typically achieved with a regularization scale of 10^{-10}. In some scenarios, using a scale of 10^{-8} yields performance nearly comparable to that achieved with a scale of 10^{-10}.

The analysis of the sampled DeepONet's hyperparameters is further extended to its performance on the test set to offer additional insights into the model. Figure 3.14 displays the model's performance on the test set for the Wave dataset. The optimal training performance was observed with 16 POD modes. Yet, when it comes to predicting unseen test data, the model demonstrates its best performance with 256 POD modes in the initial iteration. The additional hyperparameters contributing to this optimal performance include a layer width of 2048, the use of tanh activation and parameter sampler, and a regularization scale of 10^{-10} in the linear layer. Different configurations of POD modes reveal that the model's effectiveness on unseen data diminishes when a smaller number of POD modes, such as 2, 4, and 8, are used. The range between 16 and 128 POD modes also delivers promising results for unseen data. However, in cases where accuracy on unseen data is critical, utilizing the maximum number of POD modes provides the best performance for unseen data.

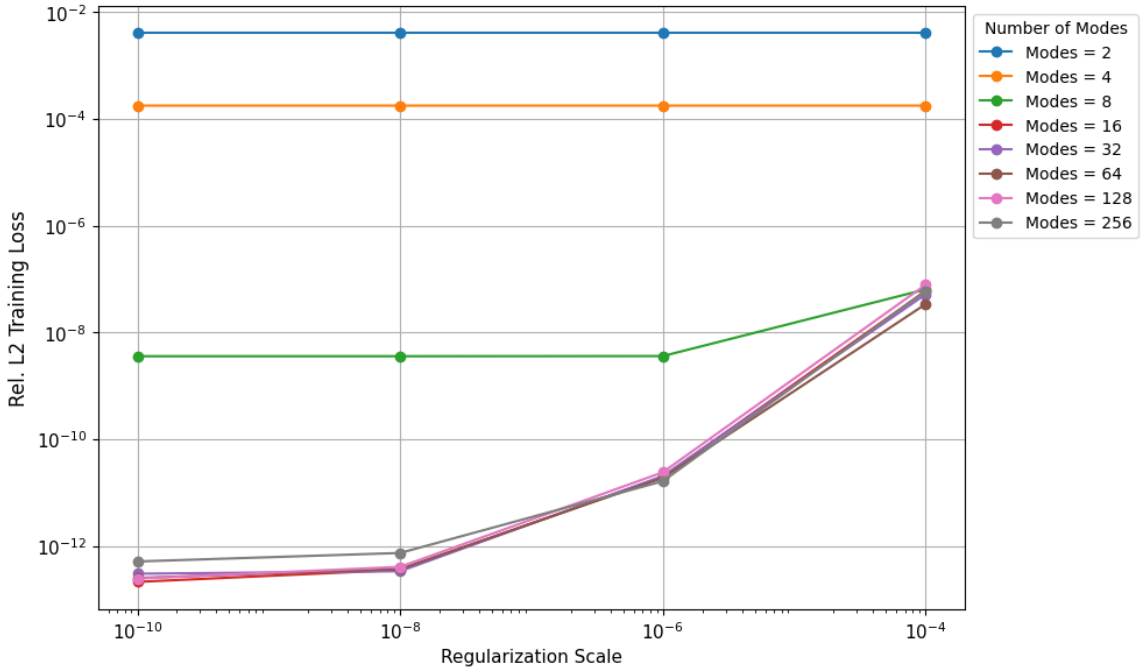

Figure 3.13.: Impact of regularization scale in the linear layer on training efficiency for the Wave dataset. The x-axis represents the various regularization scale values tested, while the y-axis, using a logarithmic scale, displays the lowest training loss achieved for each respective POD mode setting. Each line in the graph corresponds to a distinct setting of POD modes number as determined in the initial iteration.

In the context of the results shown in Figure 3.14, which highlights the experiments yielding the best test results, it is essential to closely examine the behavior of the iterative sampling process throughout its iterations. This analysis is important for the main goal of the thesis, which aims to evaluate the practicality and effectiveness of applying iterative sampling to a DeepONet model using SWIM sampling. Figure 3.16 illustrates the trajectories of both training and test losses during the sampling process. The final loss value on the test dataset is highlighted with a red star on the graph. A significant observation from this analysis is that additional iterations after the first one do not appear to enhance the model's performance. In other words, iterative sampling was able to improve the performance only once, which occured in the second iteration. Additional iterations beyond this point did not reduce the loss. This lack of improvement might be related to the relative simplicity of the Wave equation for the network, even though the dataset was designed to be challenging. The network's rapid adaptation and learning of the appropriate weights for this dataset supports this hypothesis. This conclusion is further supported by the comparison with results from an earlier experiment using the Burgers' dataset, where iterative

sampling improved the model's performance multiple times in certain conditions.

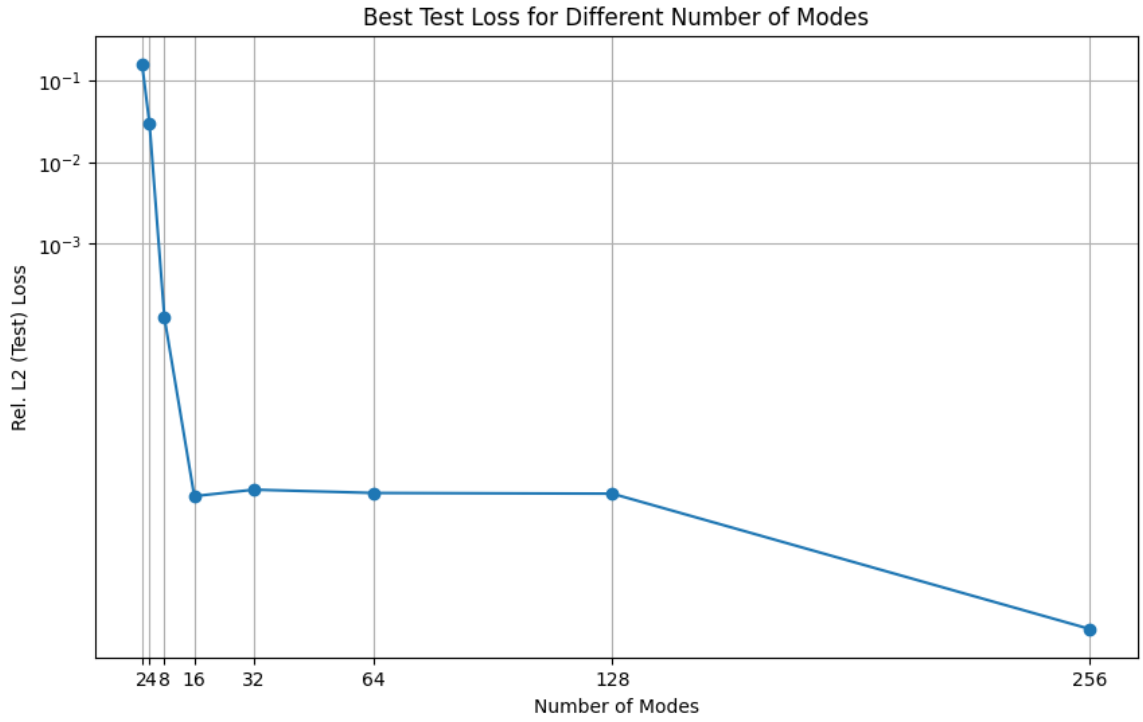

Figure 3.14.: Model performance assessment on the one-dimensional Wave test dataset. The x-axis shows the number of POD modes used during the initial iteration of iterative sampling, while the y-axis shows the lowest test loss achieved among all experiments and configurations, shown on a logarithmic scale.

In the experiments resulting in the best test performances, it was observed that iterative sampling contributed to an improvement in the model's performance only once. However, this observation does not imply an absence of multiple improvements across other experiments. In a specific instance, iterative sampling did indeed improve the model's performance multiple times, despite the fact that this particular experiment was not the most successful in terms of training or testing outcomes. The experiment utilized hyperparameter configurations of 128 POD modes, tanh as the activation function and parameter sampler, and a regularization scale of 10^{-6} in the linear layer. Details of this experiment are presented in Figure 3.15. The figure reveals that iterative sampling decreased both the training and test losses during the training phase. However, the final test loss after training did not align with these improvements seen during training.

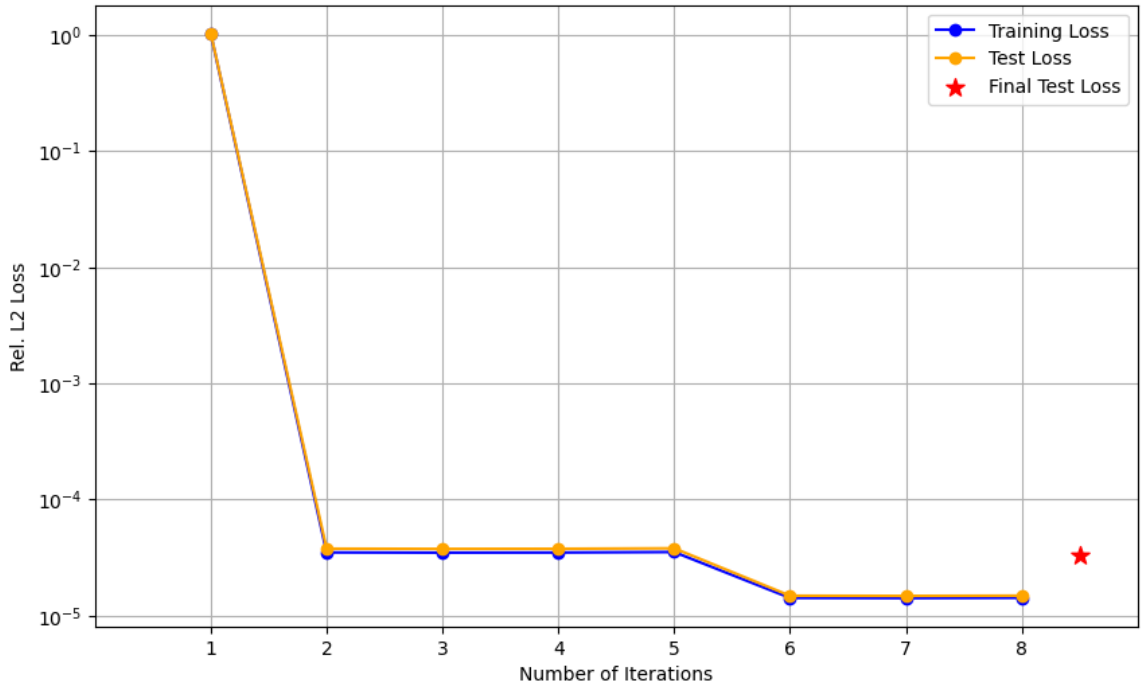

Figure 3.15.: Experiment showing iterative sampling's multiple improvement in Wave dataset. The x-axis shows the number of iterations, while the y-axis shows the logarithmically scaled loss values. The red star shows the final test loss.

In the experiment with the Wave equation dataset, the best results were primarily achieved through the iteration following the initial sampling and further iterations did not enhance the model's performance in most cases. Specifically for the Wave dataset, the model quickly adapted to the appropriate weights during the second sampling. The absence of further improvement in subsequent iterations could be related to the relative simplicity of the Wave equation dataset for the model to learn.

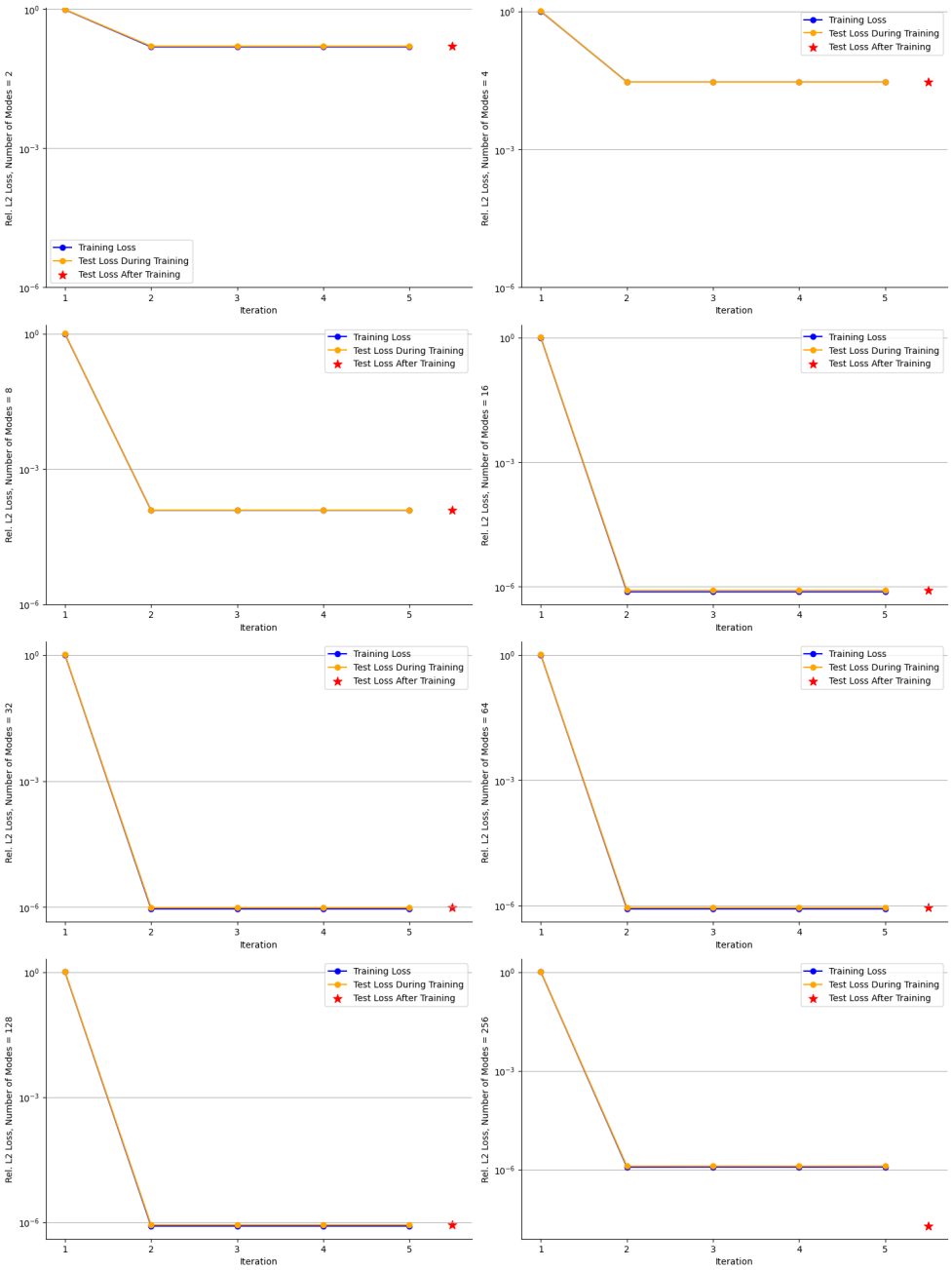

Figure 3.16.: Iterative sampling performance on the Wave dataset. This figure plots the evolution of training and test losses over the course of iterative sampling for the experiments shown in 3.14. The x-axis shows the iteration count, while the y-axis shows the loss values on logarithmic scale. The training loss is represented by the blue line, and the test loss during training is represented by the orange line. The final test loss, observed at the completion of training, is represented by a red star for each case of POD settings.

4. Conclusion

4.1. Summary

The final section of this thesis offers a detailed summary of the work conducted, discusses the results obtained, and presents an outlook on the implications. The primary aim of this thesis was to investigate the effectiveness of an iterative sampling approach for DeepONet, utilizing SWIM sampling. In Section 2, the thesis begins by introducing PDEs as the foundation for creating test benchmarks. It then explores various state-of-the-art methods for solving PDEs using neural networks. Subsequently, a detailed introduction of DeepONet and its variations, such as POD-DeepONet, is provided. The thesis also comprehensively introduces and discusses SWIM sampling, the principal sampling strategy used in this study. In Section 3, the proposed iterative sampling approach is thoroughly derived and elaborated. This is followed by two main experimental investigations using the one-dimensional Burgers' and Wave equations. The thesis thoroughly analyzes and discusses the performance of iterative sampling on these datasets, which highlights the approach's efficiency and potential use cases.

4.2. Discussion

The DeepONet was restructured into a fully-connected network by effectively utilizing the orthogonality of its branch and trunk networks. This restructured version was then trained and tested in various experiments under a range of hyperparameter settings. An extensive search was conducted to find the optimal hyperparameters for the model. This process allowed for a thorough analysis of the sampled DeepONet and offered insights into the SWIM sampling methodology. The experiments showed that for the Burgers' equation, iterative sampling improved DeepONet's performance. Notably, in certain scenarios, the network's performance improved multiple times across iterations, not just once. This repeated enhancement was most significant when using 32 and 128 POD modes, leading to the best test loss outcomes. However, in other settings with varying POD modes, the improvement due to iterative sampling was observed only once, immediately after the first iteration. It is important to mention that there were also experiments where iterative sampling led to multiple improvements in the model's performance, although these did not necessarily result in the best overall performance. In the second experiment, which focused on the Wave equation, the optimal model performance was achieved without the need for iterations beyond the second one. Iterative sampling reduced the loss only

once, following the initial sampling. The model rapidly adapted its weights to the Wave data through this sampling process, and no further improvements were noted afterward. Nonetheless, there were instances where iterative sampling did lead to multiple improvements in the model's performance, although these instances did not yield the best overall performance.

4.3. Outlook

The primary conclusion of this thesis is the demonstrated potential of iterative sampling in conjunction with SWIM sampling. The experiments revealed that for relatively complex equations, such as the Burgers', iterative sampling can enhance the model's performance multiple times. Conversely, for relatively simpler datasets like the Wave equation, the sampled DeepONet quickly adapted in the second iteration, and no subsequent improvements were occurred. This suggests that iterative sampling might be particularly beneficial for more complex problems, where more sampling rounds may be useful for better learning. Looking ahead, there is scope for further exploration of iterative sampling with various neural network architectures. Such research could provide a more comprehensive understanding of its effectiveness across a spectrum of complexities and applications. This exploration could offer new insights into optimizing sampled neural networks using SWIM for a wide range of complex and challenging problems.

Bibliography

[1] D. A. Angus. The one-way wave equation: A full-waveform tool for modeling seismic body wave phenomena. *Surveys in Geophysics*, 35(2):359–393, 03 2014.

[2] Rohan Anil, Vineet Gupta, Tomer Koren, and Yoram Singer. Memory-efficient adaptive optimization. *arXiv preprint arXiv:1901.11150*, 2019.

[3] C. Basdevant, M. Deville, P. Haldenwang, J. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, and A. Patera. Spectral and finite difference solutions of the burgers equation. *Computers & Fluids*, 14:23–41, 1986.

[4] Richard Ernest Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] Richard Ernest Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[6] Erik Lien Bolager, Iryna Burak, Chinmay Datar, Qing Sun, and Felix Dietrich. Sampling weights of deep neural networks. *arXiv preprint arXiv:2306.16830*, 2023.

[7] T Chen and H Chen. Approximation capability to functions of several variables, nonlinear functionals, and operators by radial basis function neural networks. *IEEE Transactions on Neural Networks*, 6(4):904–910, 1995.

[8] T Chen and H Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4):911–917, 1995.

[9] G Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[10] P. C. Di Leoni, L. Lu, C. Meneveau, G. Karniadakis, and T. A. Zaki. Deeponet prediction of linear instability waves in high-speed boundary layers. *arXiv preprint arXiv:2105.08697*, 2021.

[11] K Hornik, M Stinchcombe, and H White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[12] GB Huang. An insight into extreme learning machines: Random neurons, random features and kernels. *Cogn Comput*, 6:376–390, 2014.

[13] GB Huang. What are extreme learning machines? filling the gap between frank rosenblatt's dream and john von neumann's puzzle. *Cogn Comput*, 7:263–278, 2015.

[14] Guang-Bin Huang, Hongming Zhou, Xiaojian Ding, and Rui Zhang. Extreme learning machine for regression and multiclass classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(2):513–529, 2012.

[15] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: A new learning scheme of feedforward neural networks. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, volume 2, pages 985–990, Budapest, Hungary, 2004. IEEE.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[18] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[19] L. Lu, X. Meng, S. Cai, Z. Mao, S. Goswami, Z. Zhang, and G. E. Karniadakis. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. *Computer Methods in Applied Mechanics and Engineering*, 393:114778, 2022.

[20] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021.

[21] Lu Lu, Xuhui Meng, Shengze Cai, Zhiping Mao, Somdatta Goswami, Zhongqiang Zhang, and George Em Karniadakis. A comprehensive and fair comparison of two neural operators (with practical extensions) based on fair data. *Computer Methods in Applied Mechanics and Engineering*, 393:114778, April 2022.

[22] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[23] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007.

[24] Ali Rahimi and Benjamin Recht. Uniform approximation of functions with random bases. In *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, pages 555–561, Monticello, IL, USA, Sep 2008. IEEE.

[25] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[26] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

[27] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[28] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[29] Walter A. Strauss. *Partial Differential Equations: An Introduction*. John Wiley & Sons, 2007.

[30] Min Zhu, Handi Zhang, Anran Jiao, George Em Karniadakis, and Lu Lu. Reliable extrapolation of deep neural operators informed by physics or sparse observations. *arXiv:2212.06347 [cs.LG]*, 2022.

Appendix

A. Technical Specifications and Dataset Generation Details

A.1. Technical Specifications

The implementation of the datasets (one-dimensional Burgers' and Wave datasets) and the iterative sampling for DeepONet was conducted using Python version 3.9.12. The experiments were executed on a 2.3 GHz Dual-Core Intel Core i5 CPU with 8 GB of RAM. The complete code for these implementations, including the scripts for dataset generation and model training, is available in a GitHub repository: `https://github.com/utkuozbudak/thesis`.

A.2. Dataset Generation Details

A.2.1. Burgers' Dataset

The Python script developed for generating the Burgers' dataset combines numerical methods and random Fourier features to create a comprehensive dataset suitable for training and evaluating the performance of sampled DeepONet. To numerically solve Burgers' equation, we implement a function that computes these derivatives and applies the equation's dynamics. The spatial derivatives are calculated using NumPy's `gradient()` function.

The initial conditions for the Burgers' equation are generated using random Fourier features. This involves creating random coefficients following a normal distribution, which are then used in an inverse real Fourier transform to obtain the initial state of the system. These coefficients are scaled by the square of their respective mode numbers, and the first coefficient is set to be real.

The parameters for the dataset generation are carefully chosen to ensure a diverse and comprehensive dataset. These parameters and their values are shown in the Table A.1.

Table A.1.: Parameters for Burgers' Equation Dataset Generation

Parameter	Description	Value
`n_samples`	Number of data samples to generate	15000
`x_bounds`	Spatial domain bounds	(0, 2π)
`space_resolution`	Number of spatial points (resolution)	256
`n_coeffs`	Number of Fourier coefficients	5
`coeff_mean`	Mean for generating random coefficients	0
`coeff_scale`	Scale for generating random coefficients	5
`random_state`	Seed for reproducibility	42
`visc`	Viscosity for the Burgers' equation	0.1
`t_bounds`	Time interval for solving the equation	(0, 1)

Utilizing these parameters, a series of initial conditions is generated, each of which is then evolved over time using a numerical solver. The solver, employing the `solve_ivp` method from the SciPy package, integrates the Burgers' equation from $t = 0$ to $t = 1$, thereby simulating the temporal evolution of each initial state. This process results in a pair of data for each sample: the initial condition and its corresponding state at $t = 1$.

A.2.2. Wave Dataset

The process of generating a Wave dataset involves creating initial conditions and numerically solving the Wave equation using Python tools.

The simulation begins by defining a one-dimensional grid within specified spatial and temporal bounds. The spatial domain is set between -1 and 1, with a resolution of 256 spatial points. Temporally, the simulation spans from $t = 0$ to $t = 1$, divided into 20,000 time points. This grid serves as the foundation for solving the Wave equation.

The initial conditions for the wave profiles and their corresponding speeds are generated using a formula based on Gaussian functions. Specifically, the initial wave profile, u_0, is calculated as $u_0(x) = Ae^{-(x+S)^2 \times \text{scale}}$, where A represents the amplitude, S represents the shift, and $scale$ parameter controls the spread of the Gaussian function. The initial speed, v_0, is derived by multiplying x with $u_0(x)$. Amplitude and shift parameters are randomly sampled within the bounds of 1 to 2 and -0.3 to 0.3, respectively, for each of the 1500 wave functions generated.

Once the initial conditions are established, they are converted into field objects suitable for processing by the numerical solver. The solver then tackles the Wave equation, employing the parameters of wave speed (set to 0.5) and boundary conditions (value set to 0), to simulate the wave dynamics over the defined grid. Table A.2 shows all the parameters and their values used during the process.

Table A.2.: Parameters for Wave Equation Dataset Generation

Parameter	Description	Value
`n_fns`	Number of data samples	1500
`amplitude_bounds`	Bounds for the amplitude of initial waves	(1, 2)
`shift_bounds`	Bounds for the shift of initial waves	(-0.3, 0.3)
`scale`	Scale factor for Gaussian functions	20
`x_bounds`	Spatial domain bounds	(-1, 1)
`x_resolution`	Number of spatial points	256
`t_bounds`	Temporal domain bounds	(0, 1)
`t_resolution`	Number of time points	20000
`speed`	Speed of wave propagation	0.5

As a result of this process, each data point in the dataset comprises an initial condition and its corresponding solution.

List of Figures

2.1. Stacked DeepONet architecture 9
2.2. Unstacked DeepONet architecture 10

3.1. Initial conditions and solutions for Burgers' dataset 21
3.2. The relationship between layer width, activation and loss in different POD mode settings for Burgers' experiment. 23
3.3. Layer width vs. training loss by POD modes and activation types for Burgers' 24
3.4. Total training time vs. POD modes for Burgers' dataset 25
3.5. Impact of regularization scale on training loss across POD mode settings . . 26
3.6. Model performance on Burgers' test dataset 27
3.7. Iterative sampling performance on Burgers' dataset 29
3.8. Initial wave and initial speed examples of the Wave dataset 31
3.9. Example solutions of the Wave dataset 31
3.10. The relationship between layer width, activation and loss in different POD mode settings for Wave experiment. 33
3.11. Layer width vs. training loss on Wave dataset: Comparison of POD modes and activations 34
3.12. Total training time versus number of POD modes for Wave dataset 34
3.13. Effect of regularization scale on training performance in Wave dataset experiments 36
3.14. Model performance on the test set of the Wave dataset 37
3.15. Experiment in which iterative sampling improved performance multiple times on the Wave dataset 38
3.16. Iterative sampling performance on Wave dataset 39

List of Tables

3.1. List of hyperparameters and search space for DeepONet 22

A.1. Parameters for Burgers' Equation Dataset Generation 47
A.2. Parameters for Wave Equation Dataset Generation 48