SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Approximating Solutions of Wave Equations Using DeepONets

Emin Mrkonja

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Approximating Solutions of Wave Equations Using DeepONets

Approximation von Lösungen von Wellengleichungen mit DeepONets

Author:	Emin Mrkonja
Supervisor:	Dr. Felix Dietrich
Submission Date:	15.05.2023

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

Munich, 15.05.2023 Emin Mrkonja

Acknowledgments

This Master's Thesis was completed under the guidance of my supervisor Dr. Felix Dietrich. His excellent professional knowledge, rigorous attitude toward science as well as modesty have had a great impact on me. From the topic selection to the final completion of the thesis, Dr. Felix has always given me patient guidance and careful help. Here, I would like to express my deepest gratitude to Dr. Felix Dietrich. Additionally, I am also thankful to the Scientific Computing Chair at Technical University of Munich to provide the opportunity for writing my Master's thesis.

Abstract

Structural defect detection is an important problem in civil engineering. Full Waveform Inversion (FWI) recently has been further developed to address this problem by emitting waves to the building attached with sensors and reconstructing sensor signals-much like a CT scan. However, decoding the defects from sensor signals is much time consuming and mathematically impossible, since the corresponding inverse problems are difficult to solve and usually are ill-posed in real engineering applications. To solve these issues, data-driven approaches from deep learning have been investigated by researchers. Data-driven surrogate models like DeepONets, Fourier Neural Operators, and PINNs show strong strengths in computational efficiency than the classical wave equation solvers. Additionally, a well-designed regularization network is also able to address the ill-posedness of wave inversion. Thus, it will be promising to solve the wave equation by combining the surrogate models and different Machine Learning approaches. In this master thesis, we focus on applying DeepONets architecture to wave equations and analyzing the results acquired by it. Initially, various information about wave equations and traditional solvers is introduced. Moreover, we get our data from simulations executed at high-capacity GPU servers. Different DeepONets architectures (Stacked and Unstacked) with various subnetworks (FCNN, CNN) are afterwards implemented to solve the equation. In the end, each approach is evaluated and subsequently, the best one is emphasized.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Background **3**

2.1 Wave Equation 3

2.2 Numerical Solvers 6

2.3 Neural Operators 8

2.3.1 Operator Learning 8

2.3.2 DeepONets 9

2.3.3 Fourier Neural Operator 14

2.3.4 Difference between FNOs and DeepONets 15

3 Approximation Solutions of Wave Equation Using DeepONets **17**

3.1 Selection of the right DeepONet 17

3.2 DeepONet Implementation 22

3.3 Metrics - Evaluating DeepONets 25

3.4 Results 26

3.4.1 Experiment 1 : Complete Wave Equation 26

3.4.2 Experiment 2 : Mapping from one to another timestep 43

4 Conclusion **61**

4.1 Summary 61

4.2 Discussion 61

4.3 Outlook 62

List of Figures **63**

List of Tables **65**

Bibliography **66**

1 Introduction

In this thesis, we focus on approximating the acoustic wave equation, a specific type of partial differential equation (PDE), using Deep Neural Networks. We explore the utilization of DeepONet, a network architecture proposed by Prof. Karniadakis from Brown University [25], which is designed to tackle the problem of learning nonlinear operators. This architecture consists of two parts, the Branch and Trunk neural networks, which work together to approximate the solution. Furthermore, we investigate the application of various neural networks (FNN, CNN) as components of DeepONet to evaluate its capabilities. This Master's Thesis serves as the first step in exploring methods for detecting cracks in solid materials, focusing on the forward pass of this process. In future approaches, Full-waveform inversion will be required for calculations. This thesis is organized into four sections: Introduction, Background, Approximation Solutions of Wave Equation using DeepONets, and Conclusion.

The Introduction provides an overview of the problem, the structure of the thesis, and the intentions of each section. The Background section offers a comprehensive review of related work and the knowledge required to get a better understanding of the topic. It begins with a description of the Wave Equation and related physics concepts, with a particular focus on the Acoustic Wave Equation, which is central to solving our problem. We discuss various initial and boundary conditions relevant to this topic and present our idea for using the Wave Equation to address the problem at hand.
In the next section 2.2, we explore state-of-the-art numerical solvers for approximating PDEs, with a primary focus on the Finite Difference Method. This method will be used to generate ground truth data for training our network. We discuss its application to the "perfect" Acoustic Wave Equation. Additionally, we present various metrics used for generating this data. Prof. Karniadakis' work [25] serves as the foundation for the DeepONet formulation and architecture.

The core of this thesis lies in the section on Approximation Solutions of Wave Equation Using DeepONets, where we detail the application of the presented methods to our specific problem. We begin by describing our training dataset, which is generated using numerical simulations as the state-of-the-art solution. We have two different datasets, one for the complete wave equation without cracks and one where the cracks

are modeled. We then present the full DeepONet architecture tailored to our problem and describe the Branch and Trunk networks' configurations.
In the following subsections, we explore different approaches for achieving optimal results in this domain. We discuss the various Branch and Trunk network configurations we examined and the challenges we encountered. This section concludes with a presentation of the metrics and experimental evaluations, providing a comprehensive understanding of the training and test losses, as well as a comparison of our approximation results with the actual wave equation propagation through time.
The final chapter concludes the work done in this thesis and suggests potential future work to improve results or apply the methods to real-life data.

We discovered that utilizing unstacked DeepONet demonstrates the potential for solving the wave equation problem and learning neural operators. The most promising architecture used convolutional layers within the Branch network, and Fully connected layers within the Trunk network, enabling us to generate reasonably accurate predictions on the testing dataset. Moreover, using Dropout layers on the dataset with cracks improved validation loss significantly. However, we have confirmed that the network can be affected by the overfitting problem, and this concern should be explored in future work.

2 Background

In this chapter, we aim to present all related work that has been conducted and which will be required to achieve the solution to our problem statement. We will put the most focus on defining the wave equation from a physics perspective. Additionally, we define the acoustic wave equation, which serves as the basis for our example. Furthermore, we will discuss the available numerical solvers for solving partial differential equations. In our case, as a state-of-the-art solution, we will employ the finite difference method. Finally, we will explain the concept of the DeepONet and Fourier Neural Operator (FNO) and provide an overview of their respective architecture.

2.1 Wave Equation

The wave equation is a mathematical model for waves that describes the behavior of waves in specified medium. It is a second-order partial differential equation that governs the propagation of waves by relating the double time derivative to the second derivative of a function with respect to all other spatial variables. The wave equation has applications in different fields, including acoustics electromagnetism, and fluid mechanics [31].

The wave equation can be derived from Maxwell's equations, which describe the behavior of electromagnetic waves [31]. However, it also can be utilized to model other types of waves, such as sound waves or water waves. The equation is a second-order partial differential equation, meaning that it includes both a second-order time derivative and a spatial derivative of the function.

In this particular case, our focus will be on the two-dimensional wave equation, which is an important concept in the study of wave propagation and the behavior of waves, particularly for the case we are presenting. We will use x and y as the spatial variables and t as the time variable so that the wave function is represented as $u(x, y, t)$. This representation enables us to investigate the behavior of waves in two-dimensional space, such as the propagation of surface waves with bodies with material boundaries [32].

For a better understanding of the wave equation, we need to define the Laplace Operator. The Laplace Operator is a differential operator and can be denoted as $\Delta^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. It can be applicable in many areas of mathematics and physics, including the study of partial differential equations and electromagnetism [39].

The Laplace Operator measures the difference between a function's value at a given point and the average value of the function at the neighboring points. Intuitively, it represents the evaluation of the curvature of the function in the spatial domain. In the context of our example with the wave equation, it characterizes the spatial behavior of the wave.

The wave equation can be written as $u_{tt} = c^2\Delta^2 u$ [19].
The c in our case refers to the constant speed of the wave propagated, and u is the function we already mentioned. This equation describes the relationship between the temporal acceleration of the wave function $u(x, y, t)$ and the spatial Laplacian of the same function.

For easier understanding, the Laplacian can be seen as a measure of how much one point of the function $u(x, y, t)$ differs from its neighboring points. For example, if $\Delta^2 u$ is positive, that implies that it is smaller than the average value of u at the neighboring points [19]. In the context of the wave equation, the Laplacian provides insight into the spatial distribution of energy within the wave, and how this energy propagates over time.

In conclusion, the two-dimensional wave equation is an important tool for studying the behavior of waves in various physical systems. By using the Laplace Operator, we can gain a better understanding of the spatial and temporal properties of the wave function $u(x, y, t)$, which eventually helps us to analyze and predict the behavior of waves in different scenarios.

In order to have the solution properly posed, additional boundary and initial conditions are imposed, which tend to describe the environment in which our wave equation should be propagated. Moreover, these conditions should enable the uniqueness of the solution. On the one hand, if we define the region Ω as an open, connected set with a piecewise smooth boundary $\delta\Omega$, then we can describe the boundary condition as an additional equation that defines the value of the function u and a subset of its derivatives within $\delta\Omega$.
It could be specified as

$$u = f(x, y)$$

on $\delta\Omega$ or

$$u_x = g(x, y)$$

on $\delta\Omega$ as the boundary condition [19].

On the other hand, the initial condition determines the value of the function u and its derivatives at the initial timestep t_0.
Similarly, they could be defined as

$$u(x, y, t_0) = f(x, y)$$

on Ω or

$$u_t(x, y, t_0) = f(x, y)$$

on Ω [19].

Typically, boundary conditions for partial differential equations can be classified into three distinct categories:

- Cauchy conditions,
- Dirichlet conditions,
- Neumann conditions.

The Cauchy condition defines the values of the function u and several of its normal derivatives within a specified smooth coordinate area in the space of all independent variables. For a partial differential equation (PDE) of dimension k, the values of u and its first $k - 1$ derivatives should be defined within the specified area. Cauchy conditions are particularly relevant for hyperbolic PDEs, as they help in providing a well-posed initial boundary value problem.

The Dirichlet condition specifies the value of the function u on the boundary $\delta\Omega$ of the region. It can be thought of as a constraint imposed on the data at the boundary. This condition is particularly useful when the boundary values of the solution are known, which then influences the behavior of the solution within the domain.

The Neumann condition defines the value of the normal derivative of the function u on the boundary $\delta\Omega$. It is often applied when the rate of change of the function normal to the boundary is known, rather than the function's value itself. The Neumann condition characterizes the flow across the boundary.

These conditions can also be combined to create mixed boundary conditions, which help ensure the existence of unique and well-behaved solutions to the mathematical problem. The choice of boundary conditions depends on the physical context of the problem and the desired properties of the solution. Properly specifying boundary conditions is essential for guaranteeing that the resulting mathematical problem is well-posed and that its solution accurately represents the underlying physical phenomenon [19].

2.2 Numerical Solvers

In this section, we will describe numerical solvers for Partial Differential Equations (PDEs), highlighting their key principles, advantages, and applications in solving the wave equation problem. The primary focus will be on the Finite Difference Method(FDM).

The FDM is a widely used technique for solving Partial Differential Equations (PDEs) due to their simplicity and flexibility in solving various types of problems. FDM works by approximating the differential operator by replacing the derivatives in the equation with differential quotients. The domain is discretized in both space and time, and the approximation is calculated at those points in either space or time [26].

To illustrate the FDM, let's consider the wave equation, which is formulated as

$$\frac{\partial^2 u}{\partial t^2}(x,y,t) = \alpha^2 \left(\frac{\partial^2 u}{\partial x^2}(x,y,t) + \frac{\partial^2 u}{\partial y^2}(x,y,t) \right).$$

The wave equation describes the movement of a wave function u in time and space with a propagation speed of α. The discretization process transforms the continuous wave equation into a set of algebraic equations, which can be combined to simulate wave propagation [2].

In the discretization process, we define h_x and h_y as the spatial grid spacings and h_t as the timestep between two evaluations. By approximating the second derivatives using finite differences, we obtain

$$\begin{aligned}
\frac{\partial^2 u}{\partial x^2} &= \frac{u(i+1,j,t) - 2u(i,j,t) + u(i-1,j,t)}{h_x}, \\
\frac{\partial^2 u}{\partial y^2} &= \frac{u(i,j+1,t) - 2u(i,j,t) + u(i,j-1,t)}{h_y}, \\
\frac{\partial^2 u}{\partial t^2} &= \frac{u(i,j,t+\Delta t) - 2u(i,j,t) + u(i,j,t-\Delta t)}{h_t}.
\end{aligned}$$

Here, $u(i,j,t)$ represents the value of the wave equation at time t, and spatial grid points (ih_x, jh_y), where we assume that $h_x = h_y$. By substituting these approximations into the wave equation, by following [23], we can derive the following iterative scheme:

$$\begin{aligned} u(i,j,t+\Delta t) = \; & 2u(i,j,t) - u(i,j,t-\Delta t) + \alpha^2 \tfrac{h_t}{h_x}(u(i+1,j,t) - 2u(i,j,t) + u(i-1,j,t) \\ & + u(i,j+1,t) - 2u(i,j,t) + u(i,j-1,t)). \end{aligned}$$

The obtained grid and grid points typically resemble the environment illustrated in Figure 2.1.

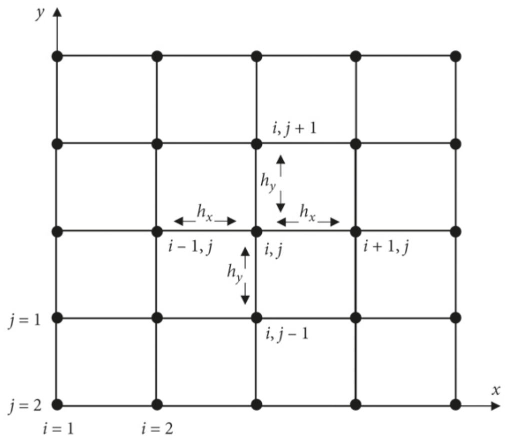


Figure 2.1: Finite Difference Method Grid, taken from [2].

This discretization of the wave equation allows for iterative calculations of $u(i,j,t+\Delta t)$ at each spatial grid point, which depends on the previous solutions of $u(i,j,t)$ and $u(i,j,t-\Delta t)$.

Besides the discretization of the wave equation, it is essential to discretize the initial and boundary conditions using the FDM. In the case of Dirichlet boundary conditions, the values at the boundary points should be equal to the given values, and the FDM is used to approximate the derivatives at the interior points. For Neumann boundary conditions, the FDM approximates the derivatives at the boundary points, which are subsequently employed to obtain the equations necessary for solving unknown values at interior grid points.

An important aspect of the FDM is that the accuracy of the solutions depends on the grid spacing h. Smaller grid spacings generally yield more accurate solutions, but they may also increase the computational cost of the simulations. When implementing the FDM, it is important to have a balance between accuracy and computational efficiency.

In conclusion, the finite difference method provides an effective and widely used approach for solving partial differential equations, such as the wave equation. By discretizing the domain and carefully considering initial and boundary conditions, the FDM allows for accurate and efficient numerical simulations of wave propagation and other physical phenomena.

2.3 Neural Operators

In this section, we aim to provide a comprehensive introduction to the concept of Operator Learning, depicting its fundamental characteristics and the underlying principles that govern its functioning. This discussion will encompass an overview of the key components and theoretical frameworks that contribute to the development of this field.

Subsequently, we explore the specific neural networks that have emerged within the domain of Operator Learning, namely DeepONets and Fourier Neural Operators, as well as engage in a comparative analysis to elaborate on the differences between DeepONets and Fourier Neural Operators.

2.3.1 Operator Learning

In the context of Operator Learning, physical systems comprising multiple interdependent functions are often considered. These functions typically predict or influence one another. A representative example of such a problem statement involves a physical system governed by a partial differential equation. The solution to this equation is expressed by the function $u(x, y, t)$, which includes a forcing term $f(x, y, t)$, initial conditions $u_0(x, y)$, and boundary conditions $u_b(x, y, t)$. Here, the variables x and y denote the spatial domain, while t signifies the time domain.

Let us represent the input function as v, defined on the domain $D \subset \mathbb{R}^d$, such that

$$v : D \ni x \mapsto v(x) \in \mathbb{R}.$$

Similarly, following [27], let the output function u be defined on the domain $D' \subset \mathbb{R}^{d'}$:

$$u : D' \ni \xi \mapsto u(\xi) \in \mathbb{R}.$$

Let V and U represent the spaces of v and u, respectively, while D and D' denote the two distinct domains. Given this problem setup, we can now define the mapping from the input function v to the output function u by means of an operator G:

$$G : V \ni v \mapsto u \in \mathbb{R}.$$

In order to facilitate the operator's learning of the mapping between the two function spaces, it is imperative to define an appropriate function to minimize the approximation error. This function, referred to as the error metric, describes the difference between the predicted and true output values. A popular choice for this error metric is the L^2 norm, which is particularly well-suited for approximating continuous functions. Given a vector X of the form:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.1}$$

the L^2 norm can be defined as:

$$|\mathbf{X}| = \sqrt{\sum_{k=1}^{n} |x_k^2|}.$$

This norm measures the Euclidean distance between two points in the function space, providing a measure of the overall error magnitude.

Because we need to work with our data numerically, we can assume that our access to these evaluations is pointwise. This means that the available data comprises discrete samples, rather than continuous functions. Therefore, in the subsequent subsection, we will elaborate on various approaches to handling different dataset setups, taking into consideration the discretized nature of the data. Moreover, Chapter 3 will discuss the specifics of the data collection process employed in this study and discuss how the obtained data is incorporated into the DeepONets architecture. This will include an examination of the preprocessing steps, data normalization, and the procedure for partitioning the data into training, validation, and testing datasets, all of which are of big importance for the effective training and evaluation of the DeepONets model.

2.3.2 DeepONets

In this subsection, we will introduce the concept of DeepONets, a deep learning-based approach to approximate nonlinear operators. We will discuss the underlying principles, architecture, and applications of DeepONets, highlighting their advantages in addressing complex mathematical problems. The content of this section is primarily based on the paper by Lu et al. (2021) [25].

The primary foundation for this thesis is based on the DeepONet Deep Neural Network introduced in the work by Lu et al. (2021) [25]. This approach focuses on learning nonlinear operators for identifying differential equations, which is grounded in the universal approximation theorem of operators [8]. In this section, we will review the content of this paper and its applications in greater detail.

On one hand, according to the universal approximation theorem, it has been proven that neural networks can be utilized to approximate any continuous function to an arbitrarily high accuracy, provided that no constraint is imposed on the depth and width of the hidden layers [9]. On the other hand, the works published in [7], [37], [30], [25], and [6] assert that any nonlinear continuous functional (a mapping from a space of functions into real numbers) and nonlinear operator (a mapping from a space of functions into another space of functions) can be approximated with a single hidden layer. In this paper, the operator G is related to an input function u, and consequently, $G(u)$ corresponds to the output function. Assuming that y is a point in the domain of $G(u)$, the output value of the function $G \circ u(y)$ is a real number. Therefore, the neural network in this case takes two input values: one for the function u and the other for the grid point y. Since the function u is continuous, it is necessary for this input to be discretized and then considered as the input value. To accomplish this, the values at finite locations $x_1, x_2, ..., x_n$ are taken, and we refer to them as "sensors". A better overview of this configuration can be observed in Figure 2.2.

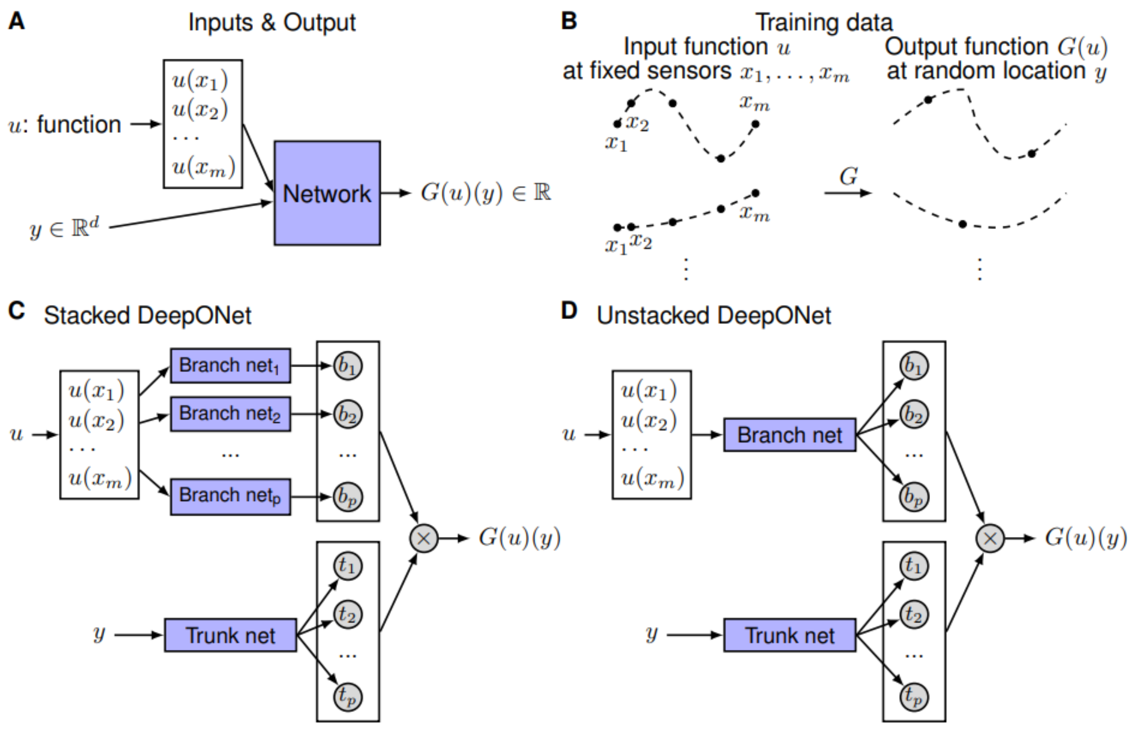


Figure 2.2: Illustration of the problem setup and architecture of DeepONets, taken from [25].

In this figure, we can identify several different architectures. For example, the stacked DeepONet architecture takes the function evaluations at m locations as input and implements a separate branch net for each of these values. In contrast, the unstacked DeepONet feeds the Branch net with the values $[u(x_1), u(x_2), ..., u(x_m)]$ as a single vector input. Furthermore, in Figure 2.2(A), the trunk net input y and inputs of the function u are directly propagated through a common network.

In the following text, we present the theorem from Chen's paper [8]:

Theorem 1 (Universal Approximation Theorem For Operator) *Suppose that δ is a continuous non-polynomial function, X is a Banach Space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in X and $\mathbb{R}^d$, respectively, V is a compact set in $C(K_1)$, G is a nonlinear continuous operator, which maps V into $C(K_2)$. Then for any $\epsilon > 0$, there are positive integers n, p, m, constants*

$c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}, w_k \in \mathbb{R}^d, x_j \in K_1$, i=1,...,n, k=1,...p, j=1,...,m, such that

$$|G(u)(y) - \sum_{k=1}^{p} \underbrace{\sum_{i=1}^{n} c_i^k \sigma(\sum_{j=1}^{m} \xi_{ij}^k u(x_j) + \theta_i^k)}_{branch} \underbrace{\sigma(w_k \cdot y + \zeta_k)}_{trunk} | < \epsilon$$

holds for all $u \in V$ and $y \in K_2$.

The theorem stated above suggests that neural networks have potential applications for learning nonlinear operators directly from evaluated data. The works presented in papers [16], [24], and [5] discuss various challenges that arise during the training of neural networks. Consequently, three main factors should be considered: approximation, optimization, and generalization. High-quality neural networks typically exhibit low prediction error, easy network training, i.e., low optimization error, and robust performance on unseen data, i.e., low generalization error.

Various neural network architectures have been applied to problems concerning dynamic systems, including Fully Connected Neural Networks (FNNs), Recurrent Neural Networks (RNNs), Residual Networks, and Autoencoders, as presented in papers [36], [25], [35], and [10], respectively. The selection of the appropriate architecture depends on the problem at hand, and in many cases, a combination of architectures may offer an optimal solution. Another approach involves using input or output functions as an image, enabling the use of CNN architecture to learn image-to-image mapping [40, 43]. We now provide a more detailed examination of the branch and trunk branches that constitute DeepONet. The branch sub-network takes evaluations at m "sensors" specific to the problem. These can be represented as a vector $[u(x_1), u(x_2), ..., u(x_m)]^T$, which serves as input and propagates through the initial network, yielding a specific p-dimensional output $[b_1, b_2, ..., b_p]^T$. In addition to the branch network, the trunk input only monitors inputs at grid points. This input can be d-dimensional, allowing it to be adjusted for different dimensional problems or to accommodate additional variables. This sub-network also outputs a p-dimensional vector, which can be written as $[t_1, t_2, ..., t_p]^T$. It is essential to note that these two sub-network outputs must have the same dimension, as they need to be merged for the complete DeepONet to be trained, as depicted in Figure 2.2.

Subsequently, the output of the DeepONet looks like the following

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

Although Theorem 1 states that bias is not necessary to achieve sufficient results, using bias can improve generalization error, which in turn enhances overall performance on unseen data. The proposed formula for the final prediction, assuming $b_0 \in \mathbb{R}$, can be presented as

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

According to the paper [25], in practice, the dimension to which sub-networks are propagated is p=10 or more. Additionally, their research group implemented both versions of "unstacked DeepONet" and "stacked DeepONet," and this framework is available as a Python library designed for scientific machine learning. The paper that supports this library, named DeepXDE, is presented in the work [28].

However, the library is constantly updated with new functionalities, and support for the Pytorch Framework [34] is still under development at this time. Therefore, in this master's thesis, we implement the network "from scratch." In addition to the proposal of DeepONet architecture, the authors of this work describe the generation of data for specific problems. They consider two function spaces: Gaussian Random Field (GRF) and orthogonal (Chebyshev) polynomials.

One additional issue that could arise during the implementation of DeepONets is the number of dimensions. Because the inputs are sensor values at different points in the grid, the network cannot be directly applied to grids that are discretized differently from the one on which the model was trained. This problem can be solved by down-sampling or up-sampling the grids depending on the case presented. Furthermore, one approach can involve inputting an additional layer at the branch network, which will scale the image appropriately before training the image further. However, an approach introduced in the paper [3] proposes PCA-based operator approximation, so that the function space becomes finite-dimensionalized. This discretization, as well as graph-based ones, are discussed in more detail in the paper [18]. Graph Neural Operator (GNO) is a neural operator that approximates integrals by combining a Nyström approximation with domain truncation and is based on the idea of graph neural networks. This network was originally constructed and proposed by [22]. Additional approaches presented include Low-rank Neural Operators (LNO) and Multiple Graph Neural Networks (MGNO), which utilize the idea of imposing the kernel of a tensor product form and combining it with graph neural networks.

2.3.3 Fourier Neural Operator

Fourier Neural Operator (FNO) is a deep learning method that combines the power of neural networks and the Fourier transform to approximate nonlinear operators. The FNO method was introduced in a paper by Li et al. in 2020 [21].

In traditional numerical methods, PDEs are usually discretized into a finite difference or finite element system and then solved numerically. This can be computationally expensive, especially for high-dimensional problems. FNO offers an alternative approach that is faster and more accurate in certain cases.

FNO works by representing the solution to a PDE as a neural network that takes the input data (e.g., boundary conditions) and outputs the user-defined query points in the domain. The neural network architecture is designed to exploit the Fourier transform to efficiently encode the spatial information of the solution. The Fourier transform breaks down the solution into a set of frequency components, and the mapping in those components is learned by the neural network.

The architecture of an FNO can be depicted similarly to the one in Figure 2.3. Initially, the input function $a(x)$ is lifted to a higher-dimensional channel space using a neural network. Subsequently, multiple integral operators and activation functions are applied. In the next step, the neural network, represented as Q in Figure 2.3, is used to project the target dimension to the desired one. These results can then be used to train the complete neural operator with the ground truths *u(x)*. The subnetwork shown in 2.3(b) depicts the Fourier layers, which are the main characteristics of the Fourier Neural Operator. At the top of this architecture, a Fourier transformation F, a linear transformation R on the lower Fourier modes, filtration on the higher modes, and finally the inverse Fourier transformation F^{-1} are applied. In the lower part, a local linear transformation W is applied, and these two parts are then dot-multiplied before using the activation function.

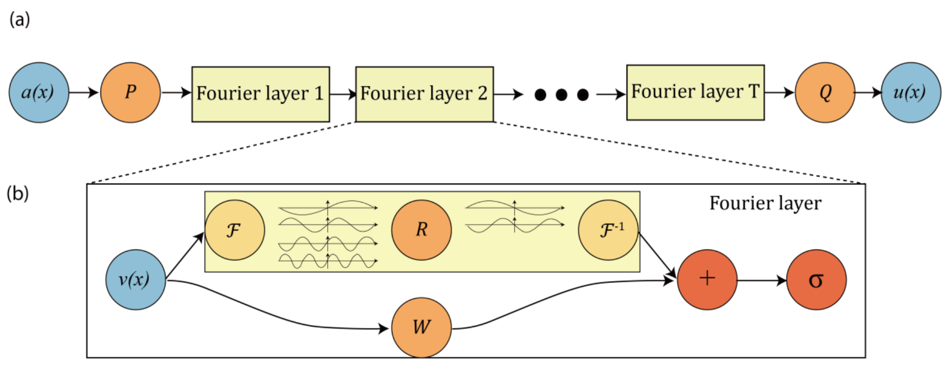


Figure 2.3: Illustration of the problem setup and architecture of FNO, taken from [21].

The FNO method has proven effective in solving a wide range of PDEs, including the heat equation, wave equation, and Schrödinger equation. It has also been applied to problems in fluid dynamics and image processing [21]. It can be considered a natural extension of the idea of making neural operator function spaces finite-dimensional.

Overall, FNO represents a promising new direction for using deep learning to solve PDEs, with the potential to offer significant improvements in accuracy and efficiency over traditional numerical methods.

2.3.4 Difference between FNOs and DeepONets

The main difference between DeepONets and FNOs is the representation of the input space. DeepONets use an input space that is not in the finite dimension and can be observed differently depending on the number of grid points used during the simulation. On the other hand, FNOs use Fourier Transformation to propagate input values through the neural network.

Let us now introduce the Discretization-Invariant models [18]. For a mathematical model to be discretization-invariant, we need a model with a fixed number of parameters that satisfy the following rules:

- It can be applied to any discretization of the input function, i.e., any set of points in the input domain can be used.
- It can be evaluated at any points that lie in the output domain.
- It converges to a continuum operator as the discretization is refined.

Property \ Model	CNNs	DeepONets	Interpolation	Neural Operators
Discretization Invariance	✗	✗	✓	✓
Is the output a function?	✗	✓	✓	✓
Can the output be evaluated at any point?	✗	✓	✓	✓
Input at any point	✓	✗	✓	✓
Universal Approximation	✗	✓	✗	✓

Figure 2.4: Comparison of Approximation Techniques, taken from [18].

The first two requirements are standard, while the third enables consistency in the limit as the refinement of the discretization is increased. Thus, it imposes the requirement that the number of parameters is fixed. If the number of parameters is unbounded, then the limit of discretization becomes unbounded.

Figure 2.4 compares four different Approximation Techniques: CNNs, DeepONets, Interpolation, and Neural Operators. In addition to the properties discussed in the previous section, the property of the Universal Approximation Theorem for the Operator is also evaluated. The reference to this theorem can be found in the DeepONets section 2.3.2.

3 Approximation Solutions of Wave Equation Using DeepONets

In this chapter, we first discuss the various considerations involved in the selection of appropriate DeepONet architecture. Afterwards, we present the architecture, complete implementation, and training of the DeepONet network for the problem at hand, as well as the numerical simulations of the wave equation used for dataset generation and experiments conducted.

3.1 Selection of the right DeepONet

In this subsection, we discuss the typical problem formulation associated with the utilization of DeepONets, as well as examine various considerations when implementing and employing these networks. Furthermore, we expound upon the methodology selected in our work and the rationale behind this choice.

The research article presented by [27] proposes multiple potential metrics for enhancing the efficacy of DeepONet networks and explores possible improvements. Additionally, the authors implement various benchmarks and compared the results with those of Fourier Neural Operators, as shown in Section 2.3.4. As previously discussed, the theorem published by [8] asserts that the universal operator approximation utilizing a single-layer neural network is achievable, given theoretically infinite computational resources.

The implementation of Deep Operator Networks (DeepONets), as introduced in the paper [27], involves several key considerations and modifications to enhance the model's performance and capabilities:

- The inclusion of new features that enhance the trunk and branch networks' capacity for representation and allow for the approximation of increasingly complex functions.
- Hard limitations are imposed for Dirichlet and periodic boundary conditions where necessary. These limitations help the model better reflect the underlying

physics of the issue, producing predictions that are more precise and physically consistent.

- The creation of a unique extension termed POD-DeepONet, which feeds the trunk network data from the training dataset's Proper Orthogonal Decomposition (POD) modes. By utilizing the data's natural low-dimensional structures, this method improves the model's capacity for learning.

- Analysis and optimization of DeepONet scaling to enhance accuracy. By investigating the scalability of the model, potential bottlenecks and the learning process can be identified, leading to improved performance.

- Introduction of a fast implementation strategy, drawing inspiration from Fourier Neural Operators (FNOs). This approach aims to reduce computational complexity and expedite the training and inference processes, making DeepONets more practical for large-scale problems and real-time applications.

In the study presented in the paper [25], four slightly distinct versions of DeepONets were developed to explore various architectural choices and their implications on the model's performance. To enable a numerical analysis of the data, the input function u is discretized and represented as values at points within a two-dimensional grid. The function u is partitioned into a set of locations $x_1, x_2, ..., x_n$, where each point corresponds to the point-wise evaluation of the function u. The trunk network receives the exact coordinates of these points as input, along with the time step, depending on the problem statement. The output of this network can be expressed as:

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

where $b_0 \in \mathbb{R}$ represents a bias term. The points $b_1, b_2, ..., b_p$ correspond to the p outputs of the branch network, while $t_1, t_2, ..., t_p$ denote the p outputs of the trunk network. This type of architecture, when employed for solving problems using DeepONets, is referred to as "Vanilla" DeepONet. By examining the performance of various architectural designs and configurations, one can identify the most effective approaches for specific problem domains and further advance the state-of-the-art in Operator Learning.

In order to enhance the prediction performance of neural networks, it can be useful to consider multiple factors that can contribute to a more comprehensive understanding of the problem. One approach involves considering not only the dataset but also the prior knowledge about the underlying system, which exists in many application domains. The method of incorporating additional parameters into the existing dataset is known as ***Feature expansion***, which seeks to improve the model's learning capacity by leveraging domain-specific information.

Feature expansion can prove beneficial when encoding prior knowledge directly into the DeepONet by modifying its architecture. While there are numerous approaches to achieving this, we will discuss some of the techniques proposed in the paper [27] that exemplify the benefits of incorporating domain-specific knowledge into the neural network architecture.

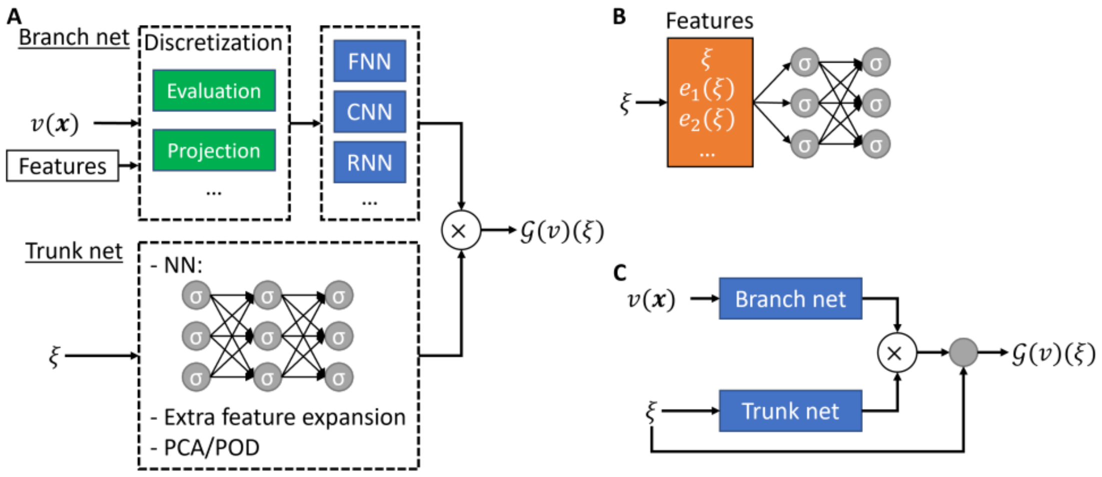


Figure 3.1: **Architecture of DeepONet. (A)** DeepONet architecture. If the trunk net is a feed-forward neural network, then it is a vanilla DeepONet. **(B)** Feature expansion of the trunk-net input. Periodic BCs can also be strictly imposed into the DeepONet by using the Fourier feature expansion. **(C)** Dirichlet BCs are strictly enforced in DeepONet by modifying the network output, taken from [27].

In papers examining Physics-Informed Neural Networks (PINNs) [41] and [42], the concept of feature expansion was initially introduced as a means to improve the model's ability to capture complex relationships between input and output variables. This approach involved extending the input of the function $u(\xi)$ to the expression $(e_1(\xi), e_2(\xi), ...)$ for the input of the trunk network. By doing so, the network can better adapt to the intrinsic structure of the problem and generate more accurate predictions. A visualization of this process can be observed in Figure 3.1B, which demonstrates the application of feature expansion to the trunk network input.

The utilization of feature expansion, as demonstrated in the example provided by

[20], addresses problems involving oscillating solutions by applying a harmonic feature expansion on the input ξ of the trunk network.

The transformation is defined as:

$$\xi \mapsto (\xi, cos(\xi), sin(\xi), cos(2\xi), sin(2\xi), ...).$$

This expanded input is subsequently used as the actual input for the trunk network, effectively increasing its capacity to model complex patterns and relationships. Another approach to enhance the performance of the network involves leveraging historical data as a feature. However, in our work, we encountered challenges with many inputs being equal to or very close to zero, prompting us to explore alternative methods.

We proposed a method involving the downsampling of input condition values with dimensions (x, y). These values, represented as images, are downsampled to sizes of $(\frac{x}{2}, \frac{y}{2})$ and $(\frac{x}{4}, \frac{y}{4})$. Subsequently, the downsampled images are upsampled back to the original image size and stacked into a single matrix with three channels. This three-channel matrix is then employed as the input for the branch network, effectively adding an additional layer of information to enhance the model's learning capacity.

Regarding the feature expansion of the branch network, as depicted in Figure 3.1A, an additional input function can be considered to serve as a branch input.

The next proposition for improving the performance of the DeepONet for certain applications where the imposition of the system's boundary conditions is essential. The primary objective is to configure the network in such a manner that the boundary conditions are integrated during the execution of the product between the branch and trunk networks, ensuring compliance with the boundary conditions. While this approach is not employed in the current study, an explanation of this concept can be found in the paper [27].

The "Vanilla" DeepONet uses the trunk network to autonomously learn the basis of the output function from the data. In contrast, the concept proposed by [27] introduces the POD-DeepONet, in which the basis is computed through the application of proper orthogonal decomposition (POD) on the training data. Initially, the existing mean of the training data is eliminated, followed by the calculation of the POD. Subsequently, the POD basis is used as input for the trunk network, while the branch network is exclusively used for learning the coefficients of the POD basis (Figure 3.1A).
The output value can be expressed as follows:

$$G(v)(\xi) = \sum_{k=1}^{p} b_k(v)\phi_k(\xi) + \phi_0(\xi),$$

where $\phi_0(\xi)$ denotes the mean function of $u(\xi)$, calculated using the training dataset.

Furthermore, the variables $b_1, b_2, ..., b_p$ represent the outputs of the branch network, while $\phi_0, \phi_1, ..., \phi_p$ constitute the p precomputed POD modes of the function $u(\xi)$.

Another concept originates from the challenge of vanishing or exploding gradients of the variance, which prevent the effective training of neural networks [11][29]. To solve this issue, various initialization methods have been developed, one of which is the Glorot initialization method [11]. In our work, we opted to utilize this particular method for initializing the neural network during the training process of our DeepONet.

In a separate paper, [13] presented the He initialization for ReLU activation functions as an alternative approach to facilitate the efficient training of DeepONets. Although this type of initialization offers potential benefits, we did not incorporate it into our current work. Nonetheless, it could be considered for implementation in future research endeavors.

An additional challenge arises when the output function cannot be represented as a single function. Assuming there are n output functions, the paper [27] proposes the following approaches:

1. Training n distinct DeepONets, with each being responsible for only one function output.
2. Dividing the branch and trunk network into n groups, where the output of the kth group represents the kth solution. This implies that every kth segment of the output weights corresponds to the kth function output.
3. Similarly to the second approach, the branch network can be divided into k groups while sharing the trunk network, thereby separating the k different function outputs.
4. Similarly, the trunk network inputs can be divided into k groups and share the branch network.

In our case, for instance, we have multiple initial condition grids as input, which need to be mapped to another grid at a different timestep. However, the output grid has an output size of (x, y), and since we consider the function evaluation of the function at each point on the grid, we needed to address a similar issue. Therefore, we adopted the idea of dividing the grid in each timestep by a multiple of x and y to obtain a single function output. These outputs are utilized as inputs to our training data, and subsequently, only one DeepONet is trained.

It is also worth mentioning a scenario in which the efficiency of computational cost and memory usage should be considered. By observing these factors, important benefits could be achieved in terms of accelerating the model training process.
In summary, this subsection has presented various considerations that were taken into

account while seeking the most suitable features for implementing DeepONet. In the following subsection, our attention will be directed toward the actual implementation of the branch and trunk networks, wherein we will discuss the specific network architecture and discuss the training process of our models.

3.2 DeepONet Implementation

In this subsection, we discuss the architecture, comprehensive implementation, and training of the DeepONet network for our problem. We present the architecture of both branch and trunk networks, as well as the input and output specifications of the DeepONet implementation. It is important to note that the framework employed for training the model is PyTorch [34], and we have implemented the entire network independently.

The process of selecting the optimal functioning model is divided into three distinct stages: data preparation, model selection, and prediction generation. Initially, we load our data into the dataset, where we also conduct some preprocessing depending on the case and experiment with its impact on the network's performance. Consequently, the input on the branch network is either a tensor with the shape $(1 \text{ x } X \text{ x } Y)$ or $(3 \text{ x } X \text{ x } Y)$. We will now examine the first case of input and investigate its network architecture. For the trunk network, we consider only the inputs of tensors with the shape $(1 \text{ x } m)$, where m represents the number of parameters taken into consideration. Depending on the experiments detailed in Sections 3.4.1 and 3.4.2, the input of the branch network can represent either a single point on the grid used in the training sample or, additionally, the timestep. Furthermore, we divided our dataset into batches of size 4 to accelerate the model training process, achieve better generalization, and prevent overfitting.

As is every usage of DeepONet, we consider two distinct networks(branch and trunk) that are combined to produce the final result. For the implementation of the branch network, we employ Convolutional Neural Network (CNN) layers [33] to effectively extract the underlying structure of the input signals.

Table 3.1: Overview of Branch Network Convolution Layers

DeepONet Branch Network			
Input Channels	Output Channels	Kernel Size	Stride
1	8	5	2
8	16	3	2
16	8	2	1
8	4	3	2

An overview of all the convolutional layers used in our model is presented in Table 3.1. Between all convolutional layers, we employ the ReLU activation function. Subsequently, the tensor obtained after propagating through these layers is converted into a one-dimensional tensor, which then passes through an additional Fully Connected Layer (FC Layer). This tensor is subsequently utilized for multiplication with the trunk network. The overall architecture of the DeepONet used in this implementation can be seen in Figure 3.2.

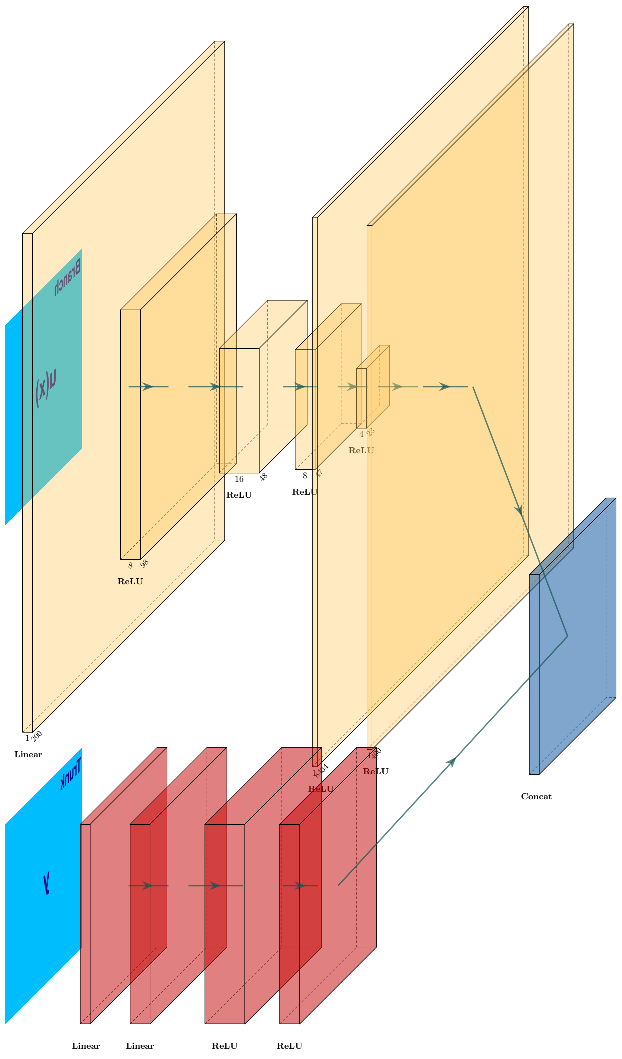


Figure 3.2: **Architecture of DeepONet.** Branch Network uses Convolution Layers, while Trunk Network uses Fully Connected Layers.

The trunk network receives a one-dimensional tensor as input, and we propagate it through three fully connected layers before merging it with the branch network.

In order to improve the training of the model and achieve convergence, we employ Xavier Initialization (alternatively known as Glorot Initialization) for initializing the weights of the neural network. The chosen loss function for calculating the difference between our model's predictions during training and the corresponding ground truths, which we seek to minimize, is the Mean Squared Error (MSE). The MSE loss function is known to be a popular choice in regression problems because it aims to effectively penalize large errors and ensure a smoother convergence to the optimal solution. For the optimization algorithm in our training process, we utilize the Adam Optimizer, a widely-used, adaptive gradient-based optimization technique. The parameters specified for the Adam Optimizer are delineated in Table 3.2.

Table 3.2: Parameters used for Adam Optimizer during training

Adam Optimizer parameters		
Learning Rate	Betas	Epsilon
0.00001	(0.9, 0.999)	1e-8

In the context of the Adam optimizer, the beta values represent the coefficients for computing the running averages of both the gradient and its square. These coefficients influence the step sizes during the optimization process. The epsilon term serves as the term added to the denominator to improve numerical stability.

On one hand, in our implementation, we adhere to the default settings for beta values and epsilon, as defined in the PyTorch implementation of the Adam optimizer. This decision is based on the extensive empirical success of these default settings across a wide range of problems. On the other hand, we alter the learning rate from the typical value by setting it to be significantly smaller than usual. This choice was motivated by the challenges we encountered during training, where numerous wave propagation values were approximately zero. By utilizing a smaller learning rate, our model can gradually adjust its parameters and better overcome this issue, ultimately leading to more accurate predictions and improved generalization.

3.3 Metrics - Evaluating DeepONets

In completing the final step of our process, it is required to identify a suitable method for evaluating the performance of DeepONets. Therefore, this section presents a thorough discussion of the appropriate metrics that can be employed to ensure a fair comparison of the model's performance.

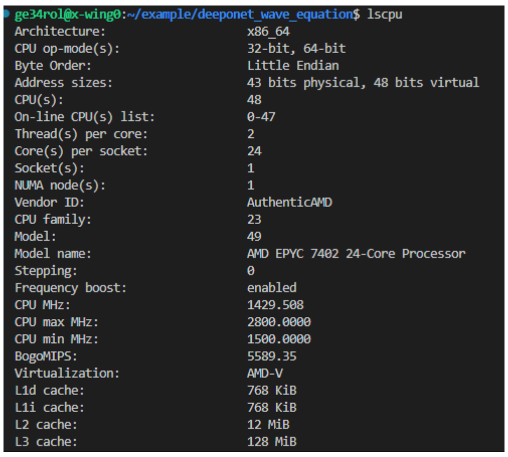
In this work, the decision was made to adopt the L2 metric, also known as the Euclidean distance, as the primary criterion for evaluating the performance and efficacy of our proposed model. The L2 metric offers an advantageous method for effectively and intuitively quantifying and representing the degree of similarity between the generated outputs of the model and the actual ground truth values obtained from the dataset under examination.

The selection of the L2 metric as the evaluation method was based on several notable benefits that it provides. Firstly, the L2 metric is robust to minor variations and potential errors that may be present in the data or the model output, therefore enabling a more reliable and stable evaluation of the model's performance. Secondly, the L2 metric can be easily and fast calculated, which offers a huge advantage when performing extensive assessments of the model.

The performance of our model, as evaluated using this metric, will be demonstrated through a series of experiments that will be presented in the next section.

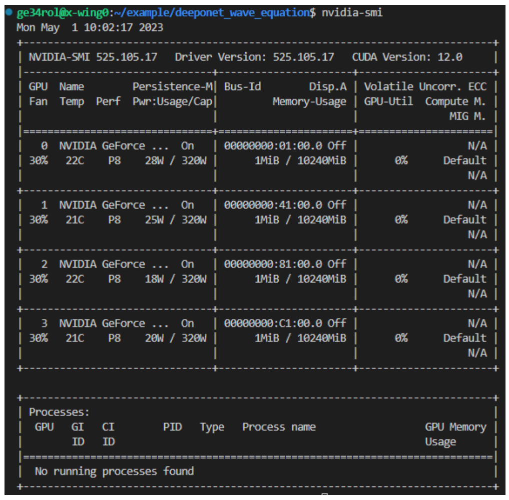
3.4 Results

This section presents a comprehensive examination of the various DeepONet architectures employed on our datasets, as well as a thorough analysis of the approaches that did not meet the necessary requirements. Our experimentation is conducted on two discrete datasets, which will be introduced in this Chapter. Following this, we implement distinct yet analogous models to tackle the complexities within the datasets. Furthermore, we elucidate the conceptual framework and design of the training procedure, as well as the hyperparameter optimization is undertaken. Subsequently, we exhibit the outcomes of each experiment and discuss the potential improvements.
More information about the hardware employed for the training process can be found in Figure 3.3.



```
ge34rol@x-wing0:~/example/deeponet_wave_equation$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   43 bits physical, 48 bits virtual
CPU(s):                          48
On-line CPU(s) list:             0-47
Thread(s) per core:              2
Core(s) per socket:              24
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       AuthenticAMD
CPU family:                      23
Model:                           49
Model name:                      AMD EPYC 7402 24-Core Processor
Stepping:                        0
Frequency boost:                 enabled
CPU MHz:                         1429.508
CPU max MHz:                     2800.0000
CPU min MHz:                     1500.0000
BogoMIPS:                        5589.35
Virtualization:                  AMD-V
L1d cache:                       768 KiB
L1i cache:                       768 KiB
L2 cache:                        12 MiB
L3 cache:                        128 MiB
```

(a) CPU Information



```
ge34rol@x-wing0:~/example/deeponet_wave_equation$ nvidia-smi
Mon May  1 10:02:17 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:01:00.0 Off |                  N/A |
| 30%   22C    P8    28W / 320W |      1MiB / 10240MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  On   | 00000000:41:00.0 Off |                  N/A |
| 30%   21C    P8    25W / 320W |      1MiB / 10240MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  NVIDIA GeForce ...  On   | 00000000:81:00.0 Off |                  N/A |
| 30%   22C    P8    18W / 320W |      1MiB / 10240MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  NVIDIA GeForce ...  On   | 00000000:C1:00.0 Off |                  N/A |
| 30%   21C    P8    20W / 320W |      1MiB / 10240MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

(b) GPU Information

Figure 3.3: The specification of the system used for experiments.

3.4.1 Experiment 1 : Complete Wave Equation

In this subsection, we initially describe the first numerical simulations and the dataset derived from them. We simulate the propagation of the wave equation over time, which is defined as:

$$\frac{\partial^2 u}{\partial t^2}(x, y, t) = \alpha^2 (\frac{\partial^2 u}{\partial x^2}(x, y, t) + \frac{\partial^2 u}{\partial y^2}(x, y, t)).$$

Here, x and y represent the points in the spatial domain, while t represents the timestep during which the wave is being propagated. The signal evaluated at the given position

and timestep is denoted by variable u, and α represents the velocity of the wave. The velocity is usually dependent on the spatial parameters, but we define it as constant. This solution is highly inspired by the work [4].
The initial condition defined for spatial variables x, y, and the timestep $t = 0$ is as follows:

$$u(x, y, t = 0) = 0.2e^{-((x-1)^2/0.1+(y-1)^2/0.1)}.$$

The spatial domains are divided into 100 areas for both x and y, with a spatial increment of 0.05 in both directions. Consequently, they lie within the array $[0, 0.05, 0.10, ..., 4.95]$. The wave speed is defined as a constant with $\alpha = 1$. The duration of the simulation is set to 5 seconds, with each timestep equivalent to 0.005 seconds. The complete simulation, therefore, has 1000 timesteps, and the propagation of the wave at different timesteps in the domain. The Neumann boundary condition is selected for this simulation. The simulation is conducted employing the formula presented in Section 2.2.

The dataset produced as a result of the numerical simulation can be found in Figure 3.4, which provides a comprehensive visualization of the wave function values throughout the simulation. These values are sampled at regular intervals of 62 timesteps, facilitating a detailed analysis of the temporal evolution of the wave. Consequently, this allows for a more thorough understanding of the wave's behavior and properties as they change over time.

To create these visualizations, the widely-used Matplotlib library [14] was employed, specifically utilizing heatmaps to effectively display the variations in the wave function values. This choice of representation ensures that the data is both accessible and easily interpretable for the user for the examination of the simulation results.

In the heatmap, the color scheme is designed to highlight the differences in magnitude, with black representing values that are close to zero and yellow signifying increased magnitude. This color-coding system allows for a more intuitive understanding of the distribution of wave function values within the simulation, as well as the identification of any notable patterns or trends that may emerge as the wave evolves.

In this work, our primary goal was to systematically examine the performance of our proposed model by initiating the training process from a single timestep and progressively incorporating subsequent timesteps. This approach allowed us to examine the model's adaptability and learning capabilities over time. One of the key objectives was to assess the model's performance on unseen timesteps that followed the final step in the training dataset, thereby evaluating its generalizability.

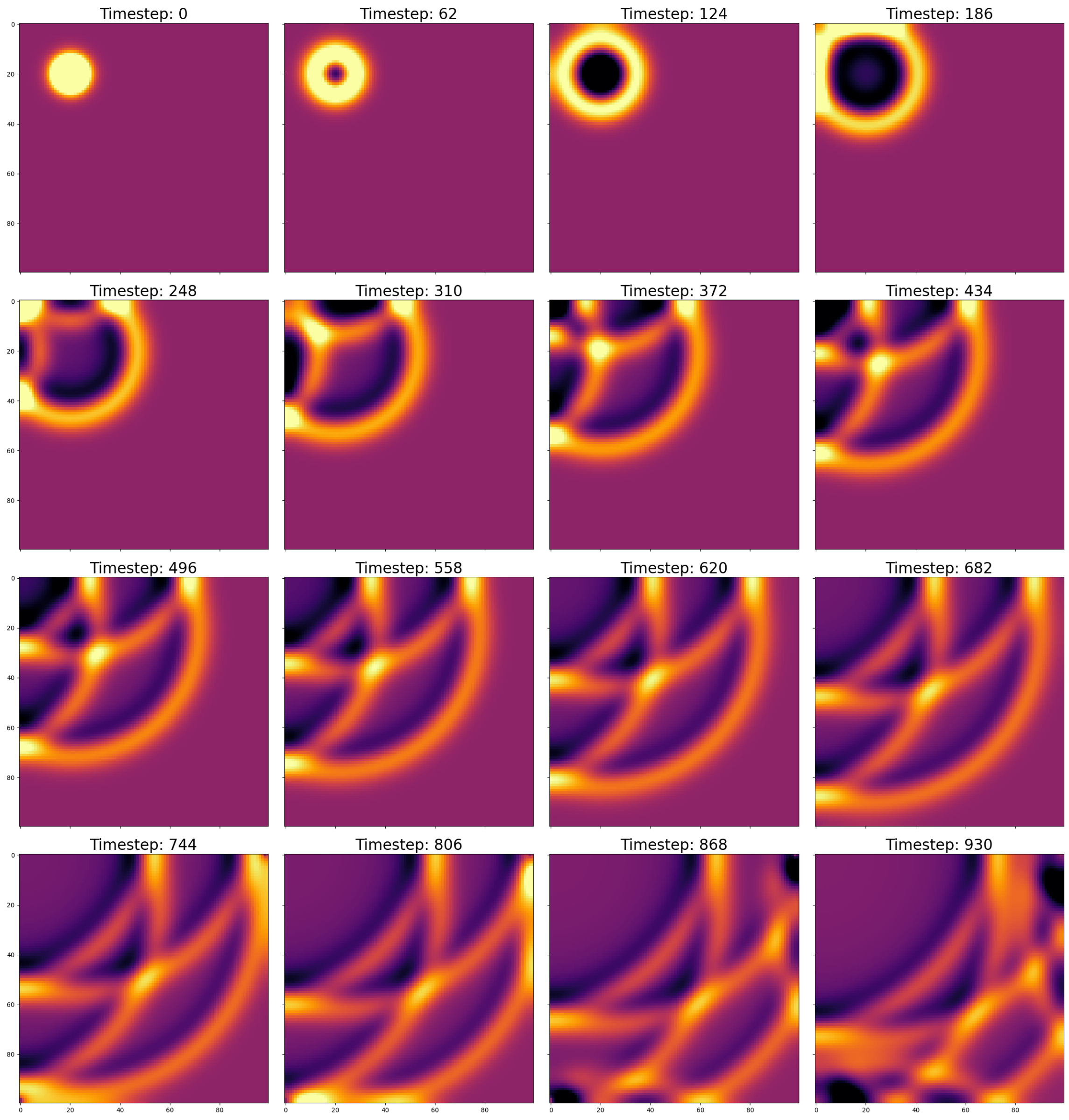


Figure 3.4: Wave function propagation during the time.

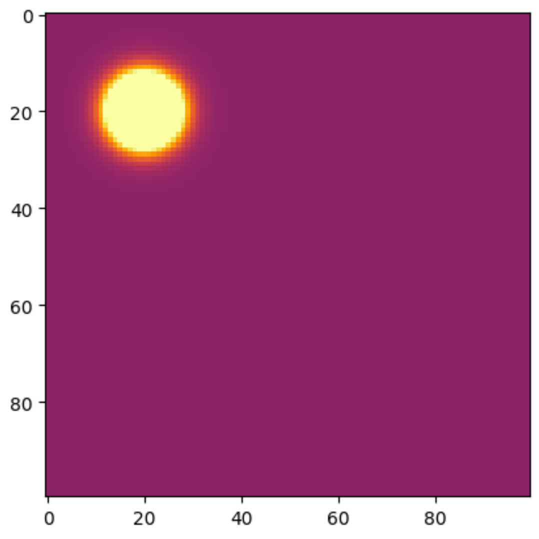
The first experiment is based on the wave equation simulation where we generate our dataset which is described previously.

For the model presented in this work, we selected Timestep 200 as the starting point of our training dataset. This choice was based on the consideration of minimizing the number of values equal to zero in the dataset. The values of the wave propagated at this timestep are comprehensively illustrated in Figure 3.5, which can be regarded as the initial condition for our training process. These initial values provide a baseline for further model development.

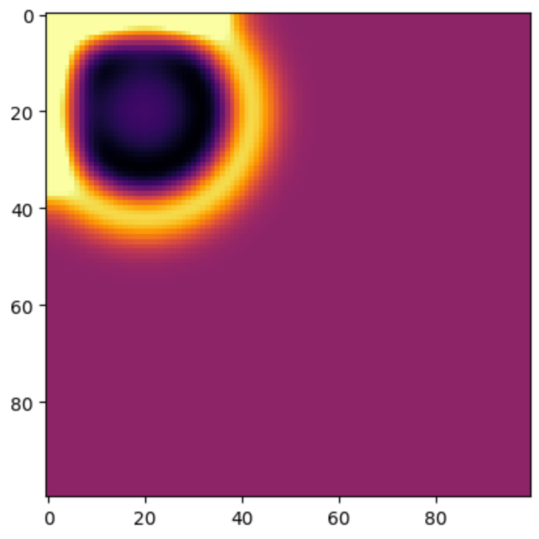
Subsequently, we methodically incorporated the following 200 timesteps into our training dataset, specifically selecting the steps with indices [200, 201, ..., 399].

Initially, our primary focus was on utilizing 50 timesteps as the test dataset. This selection was intended to provide a representative sample for evaluating the model's performance in a defined setting. However, we also extended our analysis to include the entire dataset, aiming to investigate the impact of training on 200 samples with respect to the prediction performance on the remaining 600 timesteps (ranging from 400 to 999). This extensive evaluation allowed us to thoroughly assess the model's capabilities and identify potential areas for improvement or further research.

By adopting this extensive approach, we sought to provide a thorough and rigorous evaluation of our proposed model.



(a) TIMESTEP 0



(b) TIMESTEP 200

Figure 3.5: Different Timesteps considered as the initial condition.

All different matrices for each timestep are stored using the Numpy library [12].

In order to incorporate our solution into the Pytorch framework, we develop the new Dataset which aims to the values of each entry of the dataset and then store them as the Tensors. Afterwards, in order to enable the model to better generalize data and not to learn in a specific order, we use the shuffle method, which executes the randomizing of the access indices.
In addition to considering only the initial conditions assessed at each grid point, we addressed the issue of numerous points exhibiting values close to zero by employing downsampling techniques. Specifically, we utilized downsampling scales of 0.5 and 0.25 to mitigate this problem. Subsequently, we upsampled the data to restore the original values and introduced the input via three distinct channels, potentially enhancing the network's learning capabilities. This can then be used to serve as our new dataset.

As a result, in these instances, the input can be characterized as a matrix with dimensions $(3, 100, 100)$, effectively representing the spatial and multi-scale information.

In the analysis of branch networks, a critical aspect involves accurately representing the spatial and temporal components of the network. For this purpose, we adopt a vector notation that incorporates the precise coordinate values (x_i, y_i) and the corresponding time step (t_i), depending on its usage in the context of the dataset being examined. This representation facilitates a complete understanding of the network and its various attributes.

For this dataset, we employ the following vector format:

$$[x_i, y_i, t_i]$$

This format encompasses the spatial coordinates (x_i, y_i) and the temporal component (t_i). It captures the dynamics of the branch network, accounting for changes over time.

During the practical implementation, we opted to employ the Unstacked DeepONet architecture, as depicted in Figure 2.2. This choice was made primarily because the initial condition serves as the sole input to the branch network, thereby allowing us to obtain the evaluation of each point at the precise timestep specified.

The code snippet responsible for generating the input vector is as follows:

```
// loop through the whole grid
for i in range(100):
    for j in range(100):
    // set initial condition at the timestep 200
    initial_condition_array[i][i] = u[i][j][200]
```

In this context, the matrix 'u' represents all the evaluated values of the wave equation, propagated throughout the spacetime domain.

Furthermore, the output variable for each data sample is derived from the value of the wave function, evaluated precisely at the grid points and the timestep defined within the trunk network. Consequently, for each timestep of the simulation, we obtain a set of values in the training dataset with dimensions corresponding to the product of x_{size} and y_{size}.

The primary objective of this experiment was to design a DeepONet capable of operating with a small dataset, which could then be further refined and developed. We initiated the process by working with a dataset that had a batch size of one, wherein each entry in the group of entries utilized the initial condition as input for the branch network and a single grid coordinate as input for the trunk network. This was employed as a single input entry in the training data. For output, we employed the evaluation at input coordinates, thus allowing us to configure the initial architecture for our DeepONet.

The initial architecture comprised of Fully Connected Layers for both branch and trunk networks. While we experimented with various neuron sizes in each layer, we were unable to easily identify a reasonably large network capable of learning our patterns. In our assessment, the primary challenge for the network was the prevalence of near-zero values in the dataset. Consequently, we explored several potential solutions to this issue.

Our initial approach included transforming the input data to reduce the proportion of zero values. To achieve this, we applied the downsampling and upsampling techniques previously described in this work. We attempted to downsample the input image of the initial condition by coefficients of 0.5 and 0.25, followed by upsampling to the original size and conversion to a single tensor with three channels. This method exhibited significant benefits, as our network began to learn patterns and overcame the persistent issue of zero tensor outputs. To implement this function in PyTorch, we utilized the provided function *TORCH.NN.FUNCTIONAL.INTERPOLATE* with suitably defined inputs, scale factor, and mode. We persisted in exploring further potential improvements.

However, the complexity of the problem necessitated a more complex network that could be supported solely by Fully Connected Layers. Thus, we explored alternative network types. It is important to note that for the trunk network, which featured a one-dimensional vector representing a point on the coordinate system, we maintained the use of Fully Connected Layers and experimented with varying sizes of this FNN network. Contrary to initial expectations that the trunk network would not require a large size, we had to implement a reasonably large neural network in order to successfully captured our operator pattern.

Subsequently, we investigated the potential influence of incorporating convolutional neural layers into the overall network. Initially, we used the input as the initial condition with a single channel, without applying the prior transformation. As a result, we started with a single-channel input and assessed various architectures to evaluate their performance. Our experiments indicated that the most promising outcomes were achieved with approximately three distinct convolutional layers. It is important to acknowledge that an increased number of layers might result in impractical training times, thereby constraining exploration opportunities. A comprehensive depiction of the initial experimental model's architecture can be found in Table 3.3. In this overview, we illustrate the size of the training dataset, as our initial goal was to attain good performance on a smaller dataset. Additionally, we employed varying numbers of epochs to train our network to observe the impact of the new architecture on performance and subsequently determine if the experiment showed promising results. Initially, we also intended to implement the Xavier Initialization for several convolutional layers.

Table 3.3: Configuration for the First Experiment

First experiment	
Training Dataset Size	30
Epochs	5
Batch Size	1
Transformation	True
Initialization	Xavier Initialization for 2 convolution layers
Optimizer	SGD(lr=0.0001, momementum=0.8)
Loss function	Mean Square Error

Branch Network	Conv2d(in_channels=3, out_channels=3, kernel_size=80, stride=1)
	Conv2d(in _channels=3, out_channels=3, kernel_size=10, stride=1)
	Conv2d(in_channels=3, out_channels=3, kernel_size=10, stride=2)
	Linear(12, 16)

Trunk Network	Linear(3, 8)
	Linear(8, 8)
	Linear(8, 16)

To determine whether the architecture needs further exploration or modification, we also plotted the ground truth of the solution alongside the prediction generated by the model during training with the presented configuration. This prediction is illustrated in Figure 3.6.

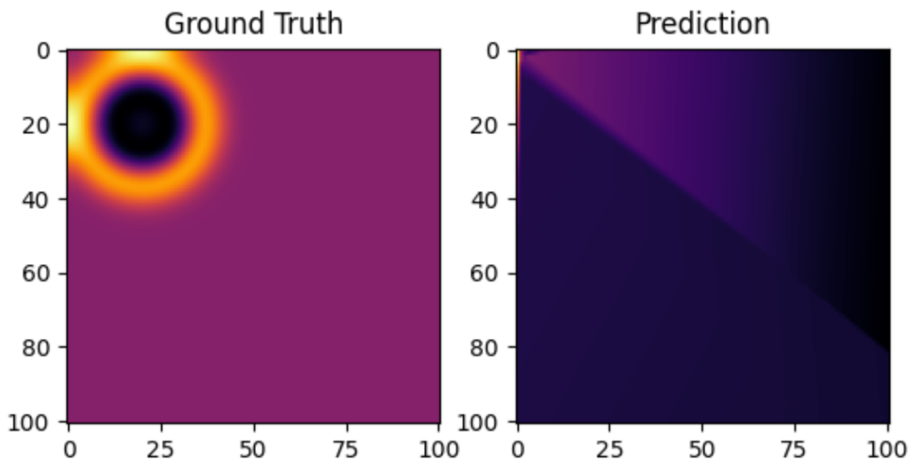


Figure 3.6: Failed Prediction for First Experiment.

Typically, the number of channels in the convolutional layers we used did not exceed 16, but we had to increase it as we observed that the model underperformed when trained on a smaller number of channels, such as up to 3. Incorporating larger channel sizes would result in a substantial increase in training time, which we leave for future work. Regarding the kernel size in filters, we primarily employed sizes ranging from 2 to 5, although we also experimented with larger kernel sizes without observing significant improvements. In terms of stride parameters in convolutions, we aimed to maintain a range of 1 to 2 to avoid excessive filter compression in a single step. While we did not observe any notable performance differences between data with and without down- and upsampling transformations, this method could be employed in future iterations without negatively affecting results.

The trunk network maintained its original fully connected layers format, and we did not implement any significant modifications in this aspect. Another critical factor that substantially influenced our model was the incorporation of weight initialization, specifically the Xavier (Glorot) method explained in the previous chapters. After propagating the weights through the convolutional layers in the branch network, we collapsed the tensor into a one-dimensional vector form. This one-dimensional tensor was then propagated through an additional fully connected layer to achieve the same dimensions as the trunk network's output. Moreover, the activation functions employed for handling non-linear relationships were ReLU [1]. We did not conduct any significant experiments concerning the modification of the activation functions.

An additional consideration for the hyperparameters used in our model involved identifying the most suitable optimizer and loss function. Initially, we explored the basic stochastic gradient descent (SGD) with various parameters. During our experiments, we concluded that the learning rate needed to be significantly smaller than in typical models due to the dataset's specific characteristics. Consequently, the optimal parameters for this type of optimizer were a learning rate of 0.00001 and a momentum of 0.8.

However, the model's performance did not meet our expectations, which necessitated the exploration of an alternative solution. We subsequently used the well-known Adam Optimizer [17]. Certain parameters required consideration during the utilization of the Adam optimizer, namely the learning rate, betas, and epsilon.

As mentioned previously, the Adam optimizer's performance with a larger learning rate was unsatisfactory. Consequently, we employed the same learning rate as in Stochastic Gradient Descent, specifically 0.00001. We also experimented with various values for the hyperparameters betas, the initial decay rates used when estimating the first and second moments of the gradient. These rates are squared at the conclusion of each training step. However, our results indicated superior performance when the values were greater than or equal to 0.9. Therefore, we opted to use the default betas in the PyTorch implementation, betas=(0.9, 0.999). For the epsilon configuration, we selected the value eps=1e−08.

Details regarding the experiment conducted in this manner are provided in Table 3.4. The configuration was largely similar to that of the previous experiment, with the exception of the optimizer. In this instance, the Adam optimizer was utilized, resulting in a notable improvement. Figure 3.7 illustrates the visible improvement in learning the initial input. The prediction successfully captured the overall area of the wave propagation. Despite these advancements, the model's performance on the test data remained suboptimal.

Table 3.4: Configuration for the Second Experiment

Second experiment	
Training Dataset Size	30
Epochs	100
Batch Size	1
Transformation	True
Initialization	Xavier Initialization for 2 convolution layers
Optimizer	Adam(lr=0.001, betas=(0.9, 0.999), eps=1e-08)
Loss function	Mean Square Error

Branch Network	Conv2d(in_channels=3, out_channels=8, kernel_size=70, stride=1)
	Conv2d(in _channels=8, out_channels=4, kernel_size=10, stride=2)
	Conv2d(in_channels=4, out_channels=4, kernel_size=10, stride=2)
	Linear(16, 16)

Trunk Network	Linear(3, 8)
	Linear(8, 8)
	Linear(8, 16)

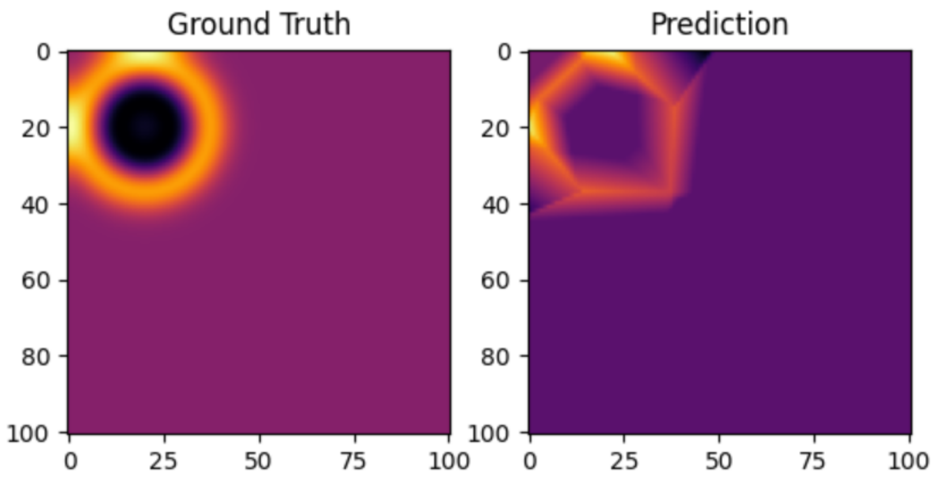


Figure 3.7: Failed Prediction 2.

During the training of our model, we encountered the challenge of extended processing time. Completing a single pass through the entire training dataset, known as an epoch, took over an hour. This necessitated the identification of a new approach to address this issue.

An optimal strategy involved increasing the number of samples used during a single forward pass of our network, before updating the weights. This was achieved by dividing the dataset into smaller subsets, known as batches, and presenting these as a new dataset.

We experimented with various batch sizes, all being powers of two, and ultimately concluded that to avoid overfitting, the batch size should not be excessively large. As a result, our model was trained using a batch size of four.

The experiments were then conducted with this batch size, facilitating a more efficient hyperparameter tuning process. The configuration of the experiment, as displayed in Table 3.5, provides specifics for this process. However, in contrast to previous experiments, we applied the Xavier Initialization to all weights within the neural network.

In this experiment, we refrained from performing the transformation as we previously did, as we aimed to observe its performance with the original data. Additionally, we increased the size of the linear layers in the trunk network, leading us to conclude that a significantly larger number of neurons in these layers would be required.

We trained this model for 100 epochs. It performed satisfactorily on the training data.

However, it produced unsatisfactory results when applied to the test data. Despite this, the results showed a significant improvement, and the model began to capture the wave propagation over time and its movement. An example of the model's performance on a single data entry from the test dataset is displayed in Figure 3.8.

Table 3.5: Configuration for the Third Experiment

Third experiment	
Training Dataset Size	30
Epochs	100
Batch Size	4
Transformation	False
Initialization	Xavier Initialization
Optimizer	Adam(lr=0.001, betas=(0.9, 0.999), eps=1e-08)
Loss function	Mean Square Error

Branch Network	Conv2d(in_channels=1, out_channels=8, kernel_size=70, stride=1)
	Conv2d(in _channels=8, out_channels=4, kernel_size=10, stride=2)
	Conv2d(in_channels=4, out_channels=4, kernel_size=10, stride=2)
	Linear(16, 32)

Trunk Network	Linear(12, 8)
	Linear(8, 8)
	Linear(8, 32)

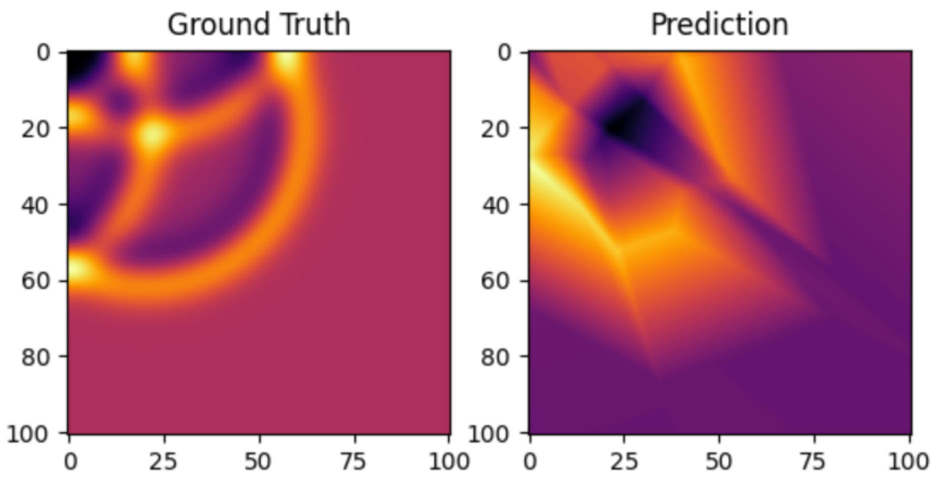


Figure 3.8: Failed Prediction on Test Data.

In order to improve the performance of our model, we decided to continue the training process with a larger dataset. This expanded dataset incorporated 200 different timesteps from wave propagation. We maintained the batch size of four and continued to use Xavier initialization. As previously noted in this work, we sought to avoid issues with values around zero by applying a smaller learning rate, thereby ensuring we did not overlook the optimal solution.

Furthermore, after experimenting with different architectures, we opted to proceed with the one depicted in Table 3.6. A notable change was the increase in the size of the linear layers, which significantly contributed to the improvement of the model's performance.

Table 3.6: Configuration for the Final Experiment

Final experiment	
Training Dataset Size	200
Epochs	15
Batch Size	4
Transformation	False
Initialization	Xavier Initialization
Optimizer	Adam(lr=0.00001, betas=(0.9, 0.999), eps=1e-08)
Loss function	Mean Square Error

Branch Network	Conv2d(in_channels=1, out_channels=8, kernel_size=5, stride=2)
	Conv2d(in _channels=8, out_channels=16, kernel_size=3, stride=2)
	Conv2d(in_channels=16, out_channels=8, kernel_size=2, stride=1)
	Conv2d(in _channels=8, out_channels=4, kernel_size=3, stride=2)
	Linear(1600, 400)

Trunk Network	Linear(12, 1024)
	Linear(1025, 512)
	Linear(512, 400)

To assess the performance of our model at various timesteps, we provide multiple figures to illustrate the predictions generated by the model. On one hand, Figures 3.9, 3.10, and 3.11 represent the predictions of our model on the timesteps 201, 298, and 333 respectively, which are all stored in our training dataset. On the other hand, Figures 3.12, 3.13, 3.14, and 3.15 depict the evaluation of how our model performs on the samples of timesteps 400, 450, and 550 respectively.

All these entries are contained within our test dataset, and we can conclude that our model successfully learned the initial pattern of wave propagation over time. Additionally, we observe that as we progress further in time, our model's performance declines compared to the initial timesteps immediately following timestep 399 (which concludes our training data). This issue could potentially be attributed to overfitting, which we did not prevent as effectively as possible. Hence, future work may consider employing different mechanisms.

One approach could involve training the model for fewer epochs, sacrificing performance on the training data but potentially improving performance on the test data. Furthermore, we could increase the size of the dataset, providing our model with more timesteps necessary to learn how the wave would react when reaching all boundaries, thereby addressing these cases.

Another suggestion would be to incorporate dropout layers into our architecture, which randomly set the input units to 0 with a frequency of a defined rate at each step during training time, thus preventing overfitting [38].

Lastly, to stabilize the learning process and reduce the number of training epochs required to train the network, we could employ the batch normalization technique [15].

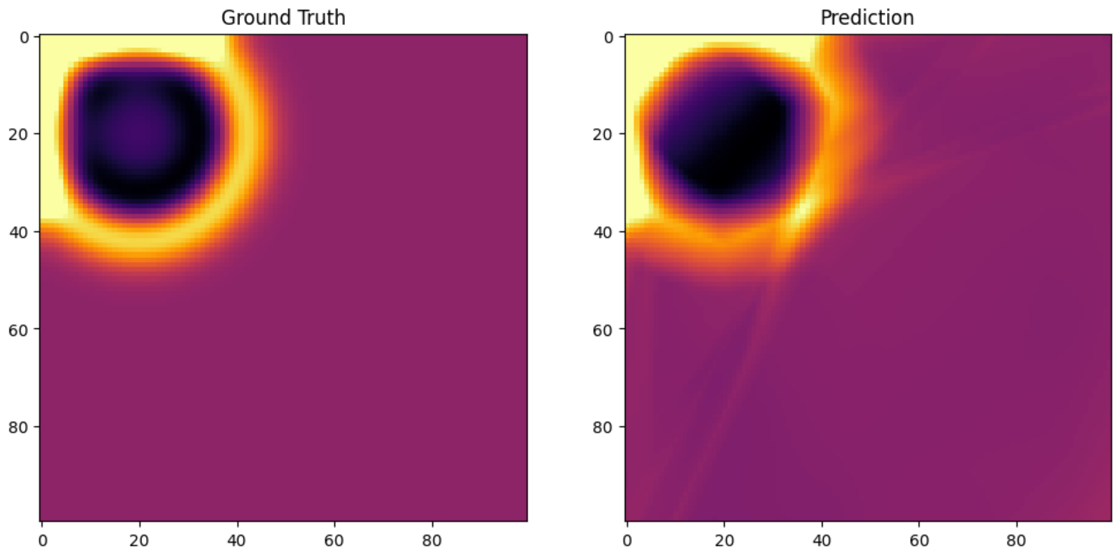


Figure 3.9: Timestep: 201

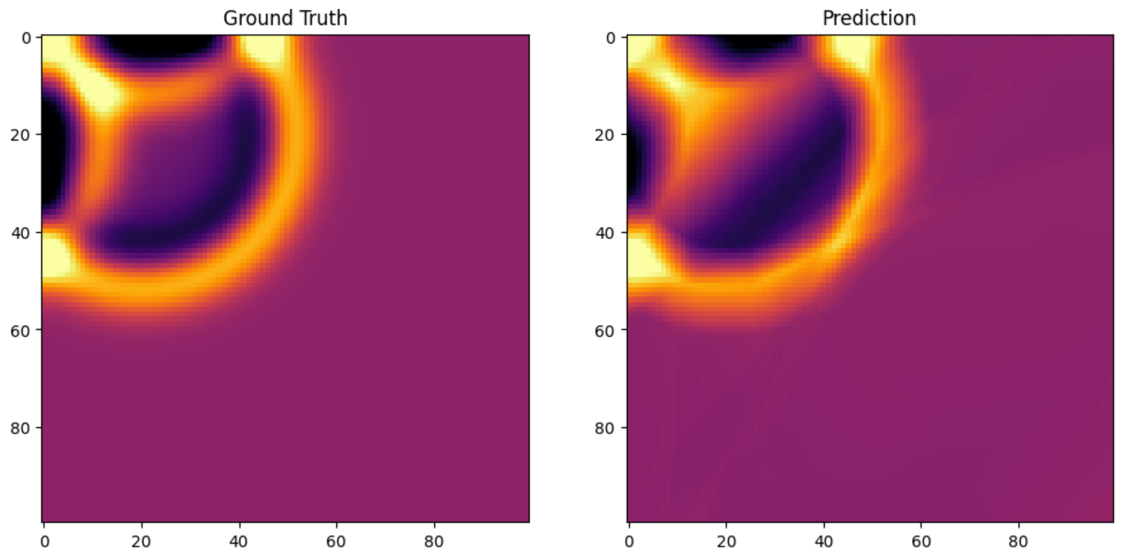


Figure 3.10: Timestep: 298

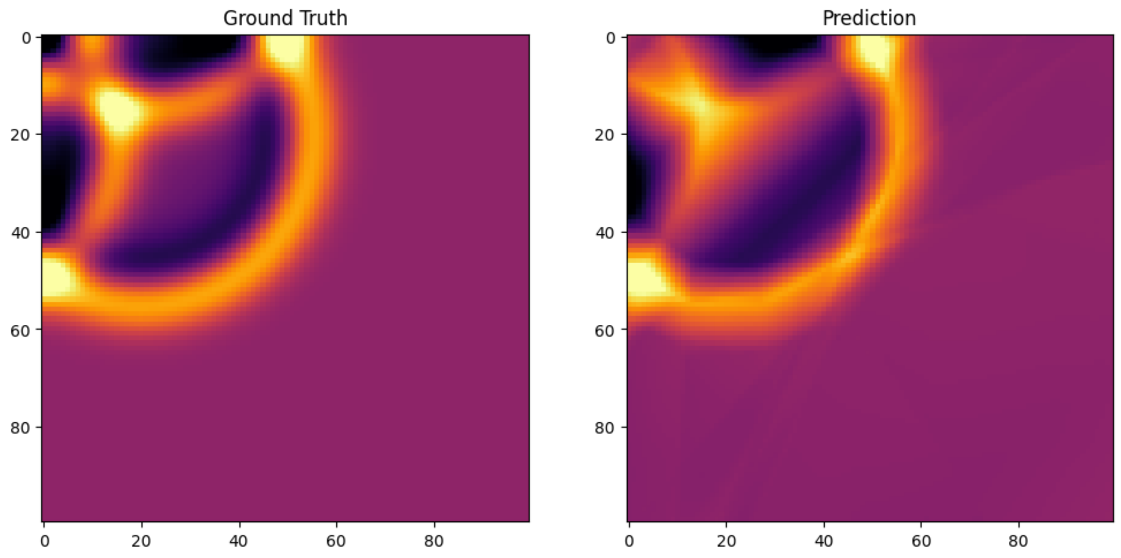


Figure 3.11: Timestep: 333

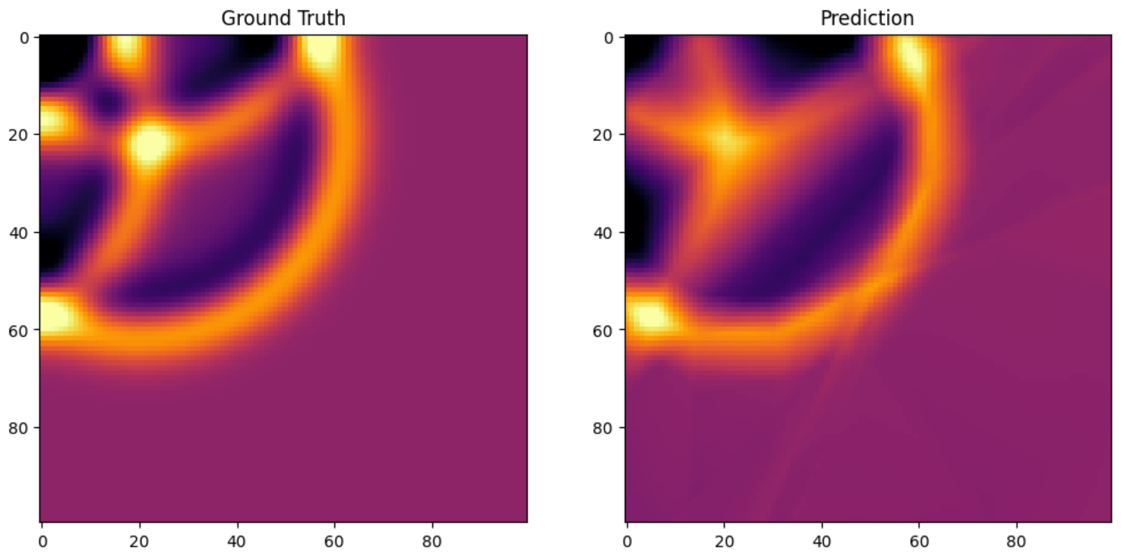


Figure 3.12: Timestep: 400

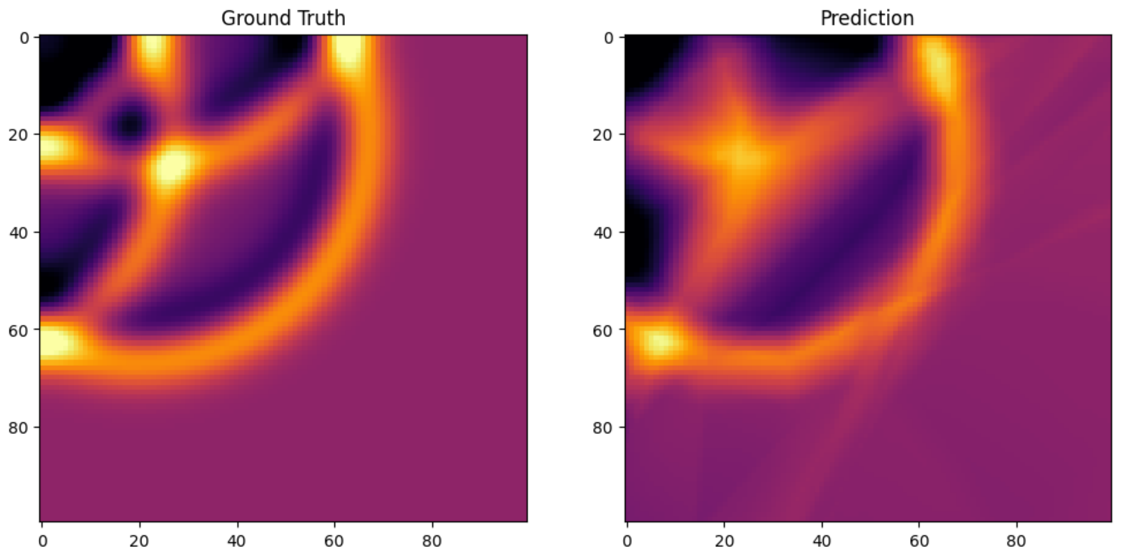


Figure 3.13: Timestep: 450

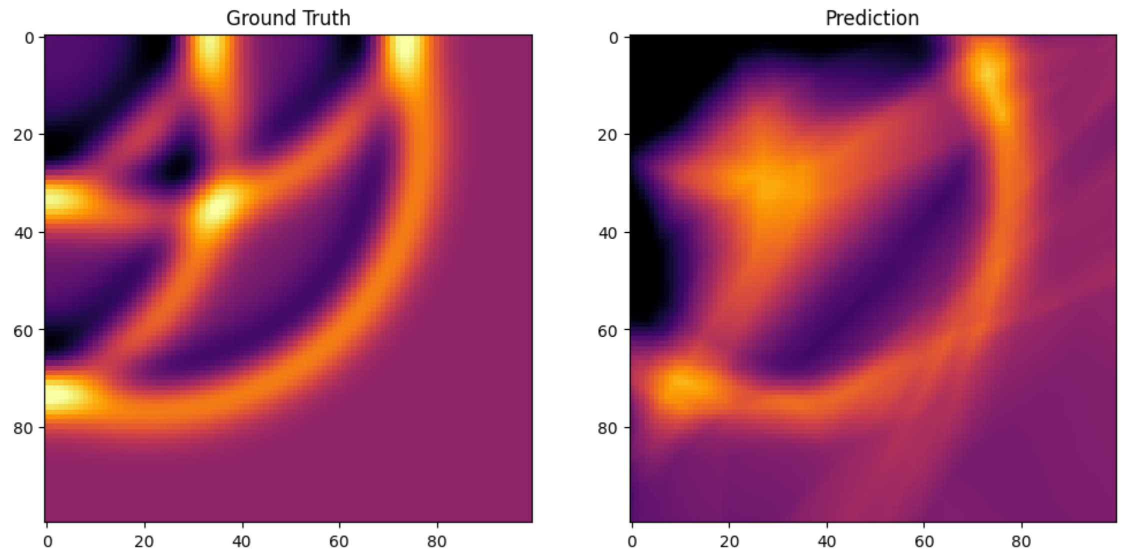


Figure 3.14: Timestep: 550

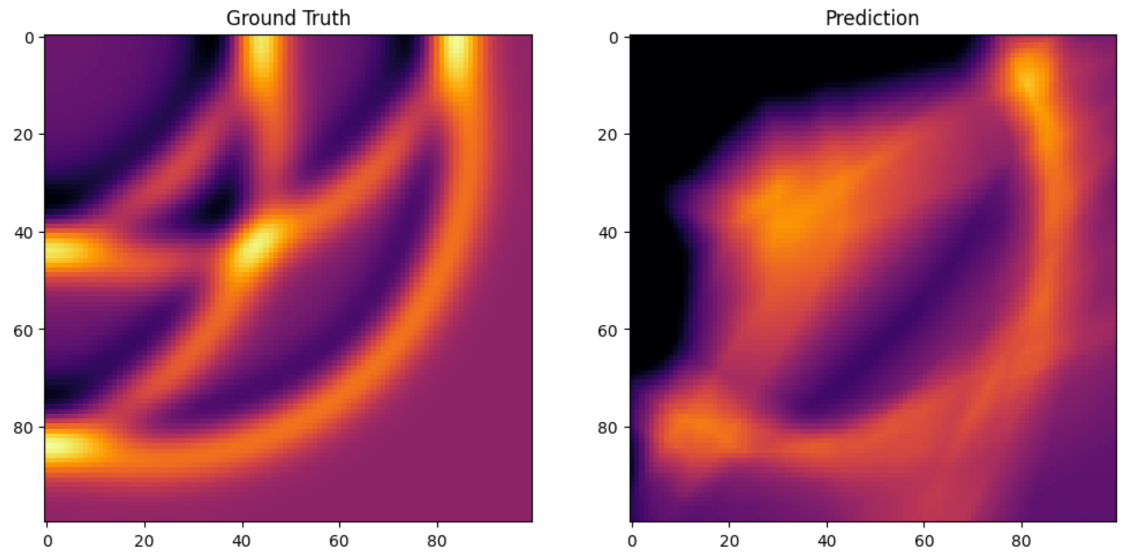


Figure 3.15: Timestep: 650

3.4.2 Experiment 2 : Mapping from one to another timestep

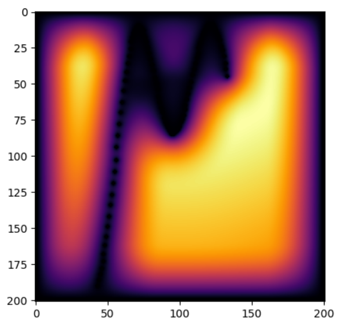
In this subsection, we describe the methodology employed for modeling cracks in the surrogate model of wave propagation through the medium. This dataset was kindly provided by Rahul Manavalan, and the source code can be accessed on his GitHub repository at `https://github.com/dynamic-queries/FullWaveformInversion.jl`.

In this numerical simulation, various cracks are modeled, and the wave is propagated for 100 timesteps. Figure 3.16 displays the different modeled cracks, with 50 distinct instances present in our dataset. Taking these initial conditions into account, we capture the different evaluations of the wave function at timestep 50. Consequently, these evaluations will be considered as ground truth outputs for their corresponding initial conditions.
These outputs are illustrated in Figure 3.17.

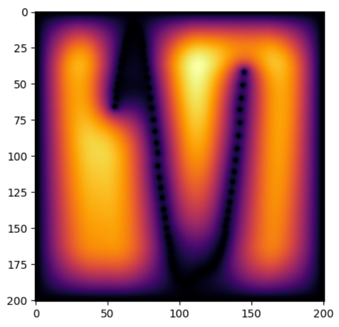
This experiment presents a different set of challenges to address. Our primary objective is to identify and understand the various neural operators that exist between distinct timesteps within this particular dataset. To achieve this, we have embarked on an exploration of the ways in which the initial time step of the simulation can be associated with the different timesteps observed throughout the entire simulation process.

As the basis for our initial condition, we utilize the first sample in the simulation, which serves as a representation of the modeled crack. In order to generate accurate and reliable results, we evaluate the values for each point on the grid and subsequently align them with the appropriate surrogate model. This approach is necessary to accommodate the described variations in crack configurations that may be encountered. This model is designed to address the complexities and challenges that are inherent in the diverse crack formations that may be encountered.

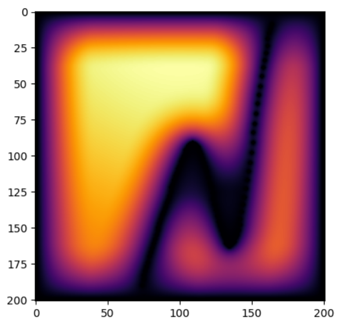
As a result of our methodology, in these specific instances, the input can be characterized as a matrix with dimensions $(1, x_{size}, y_{size})$, which effectively represents the spatial and multi-scale information inherent in the dataset.



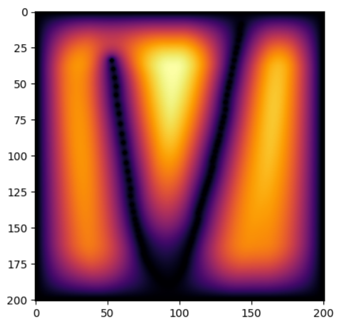
(a) Initial Condition 1



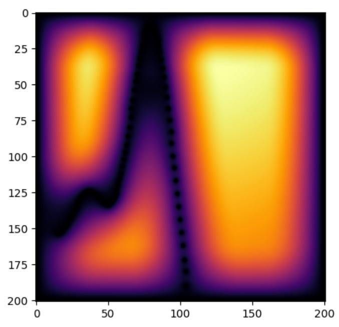
(b) Initial Condition 3



(c) Initial Condition 5

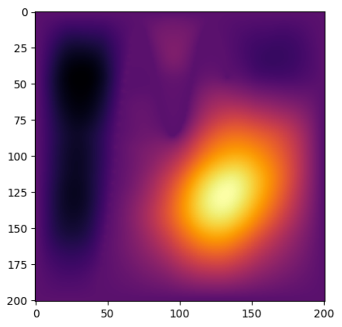


(d) Initial Condition 7

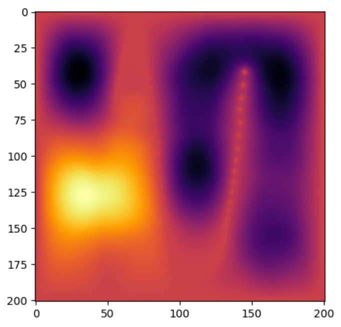


(e) Initial Condition 9

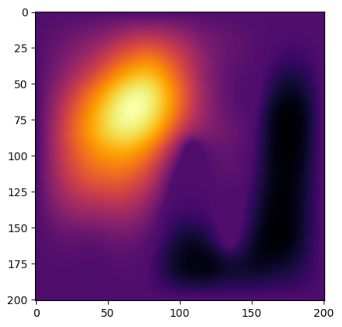
Figure 3.16: Different Initial Conditions representing modeled cracks.



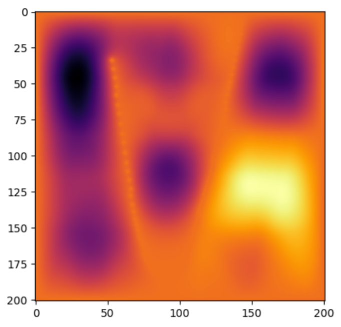
(a) Ground Truth 1



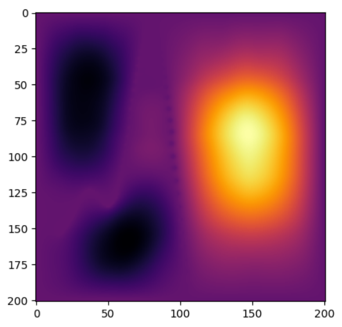
(b) Ground Truth 3



(c) Ground Truth 5



(d) Ground Truth 7



(e) Ground Truth 9

Figure 3.17: Ground Truths at the timestep 50 for respective Initial Conditions.

In order to ensure compatibility with the used PyTorch framework [34], it is necessary to convert our input values into Tensors. As the branch network input, we employ 30 distinct initial conditions of the sizes (1, 200, 200), each of which represents a unique crack formation in our surrogate model. For the trunk network input, we adopt a simplified vector format, specifically denoted as

$$[x_i, y_i].$$

In this particular case, the representation concentrates exclusively on the spatial coordinates (x_i, y_i), purposefully excluding the temporal aspect (t_i) due to its lack of relevance to the analysis of this specific dataset.

It is important to note that, within the context of vector notation, both x_i and y_i represent the grid points on a predefined scale. By utilizing a matrix with dimensions (200, 200), we aim to discretize the grid in a slightly different manner, allowing for the establishment of a distribution on the grid that ranges between 0 and 5. With these inputs serving as the foundation for the branch network, we represent the inputs as all possible combinations of the vector [0, 0.025, 0.050, ..., 4.975].

In order to train our model for this dataset, we employed similar concepts as presented in the first experiment. However, in this case, our objective was to learn the neural operator between two spaces. The first space was defined by the initial condition, representing the crack modeled with our surrogate model. For the second space, we initially used the wave function evaluated at timestep 50, intending to assess how our model performs with different cracks.

In this experiment, we utilized 30 distinct initial conditions and trained the model for 15 epochs. The batch size was set to four to accelerate the process, and no transformations were applied to the data. The weights were initialized using the Xavier Initialization method and the chosen optimizer was Adam, with the parameters depicted in Table 3.7. It is important to note here that we changed the value of the learning rate in the Adam optimizer to 0.001. Since our input data was presented on a grid with a shape of $(200, 200)$, we needed to make slight modifications to our network compared to the first experiment described in subsection 3.4.1.

Lastly, we employed the Mean Square Error as the loss function for training our model.

Table 3.7: Configuration for Operator Learning

Operator Learning Experiment	
Training Dataset Size	30
Epochs	15
Batch Size	4
Transformation	False
Initialization	Xavier Initialization
Optimizer	Adam(lr=0.0001, betas=(0.9, 0.999), eps=1e-08)
Loss function	Mean Square Error

Branch Network	Conv2d(in_channels=1, out_channels=8, kernel_size=5, stride=2)
	Conv2d(in _channels=8, out_channels=16, kernel_size=3, stride=2)
	Conv2d(in_channels=16, out_channels=8, kernel_size=2, stride=1)
	Conv2d(in _channels=8, out_channels=4, kernel_size=3, stride=2)
	Linear(8464, 400)

Trunk Network	Linear(8, 1024)
	Linear(1025, 512)
	Linear(512, 400)

In addition to the training data, we opted to use a validation set to evaluate the performance of the network and identify at which point our network encounters the issue of overfitting.

Initially, we aimed to divide our dataset into mini-batches of size 2000, and after training each batch of 2000, we calculated the average loss and plotted it. Upon examining the loss function illustrated in Figure 3.18, we can conclude that our model performed well, and the loss consistently decreased.

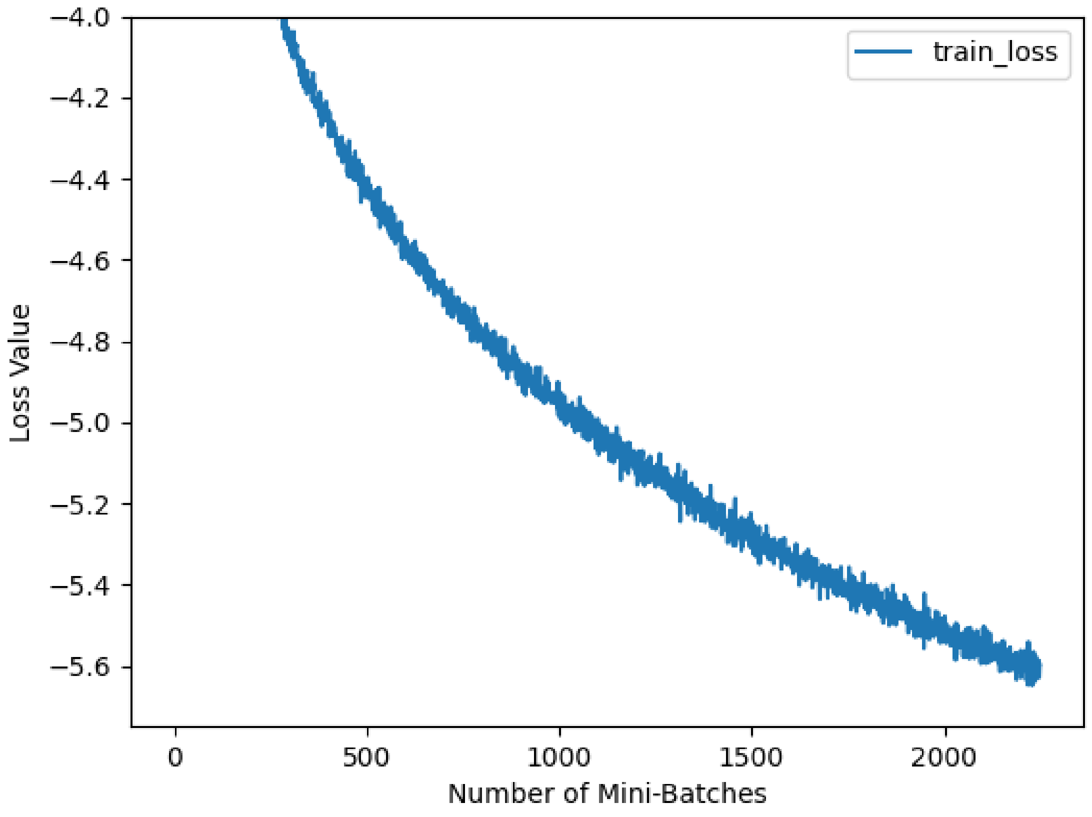


Figure 3.18: Logarithm of Training loss for 15 epochs.

To address the potential overfitting issue, we also utilized a validation set. We employed validation for five distinct timesteps and tracked the value of the loss function for both the training and validation sets. This loss function was calculated after one iteration through the entire dataset, thus recording the values after each epoch. Upon examining Figure 3.19, which displays these errors, we can conclude that the validation loss begins to increase significantly after the fifth epoch. Consequently, we will use the model saved after this iteration.

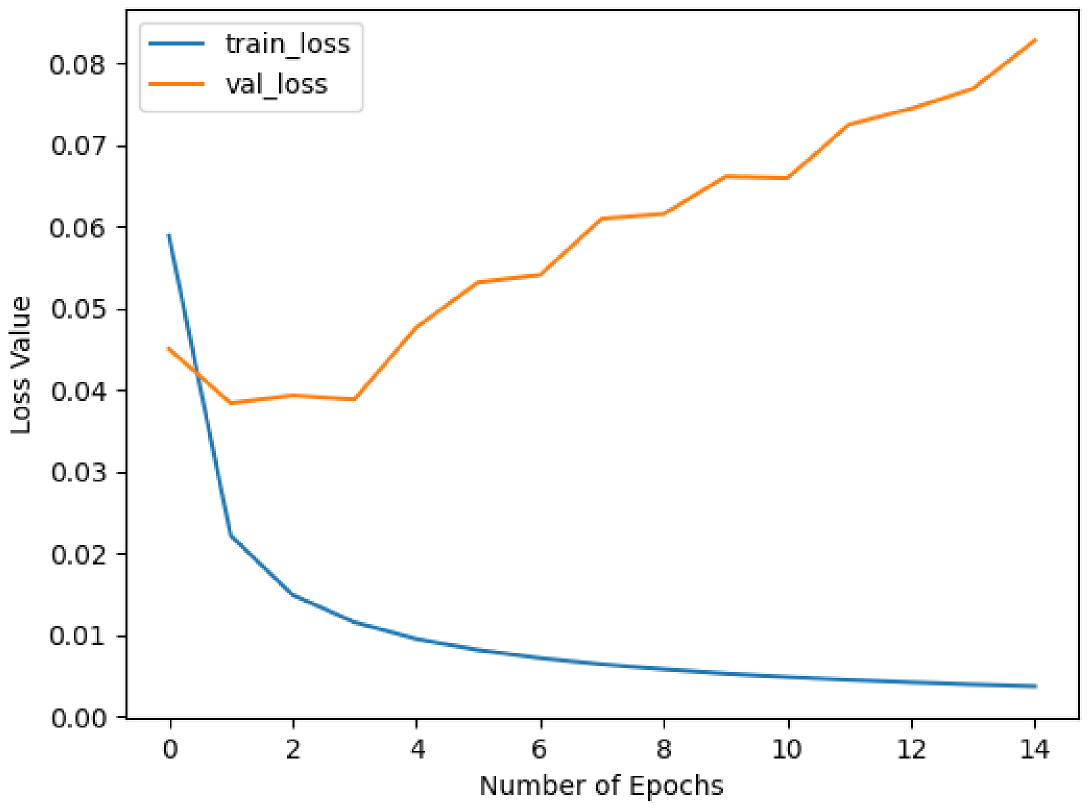


Figure 3.19: Training and Validation loss for 15 epochs.

For the evaluation of the performance of our model, we plot the predictions next to the actual ground truth of the wave function. We use three samples from the training, as well as from the test dataset, in order to also display the performance on the unseen data. On one hand, Figures 3.20, 3.22, and 3.23 represent the evaluations of the function for the entries used in the training dataset. On the other hand, Figures 3.24, 3.25, 3.26, and 3.27 display the performance of our trained model within the test data.

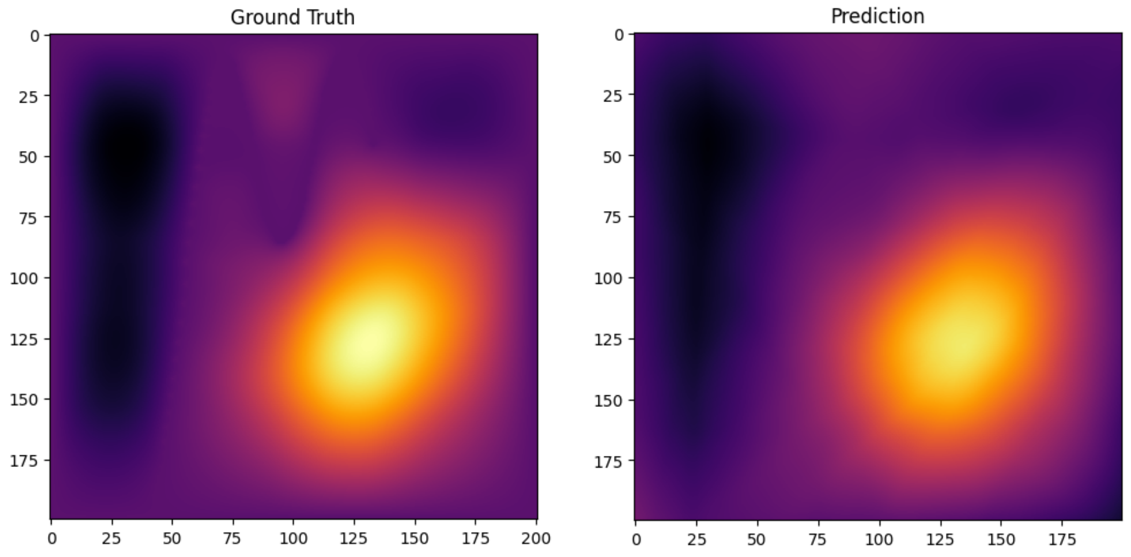


Figure 3.20: **Training Dataset;** Crack: 0

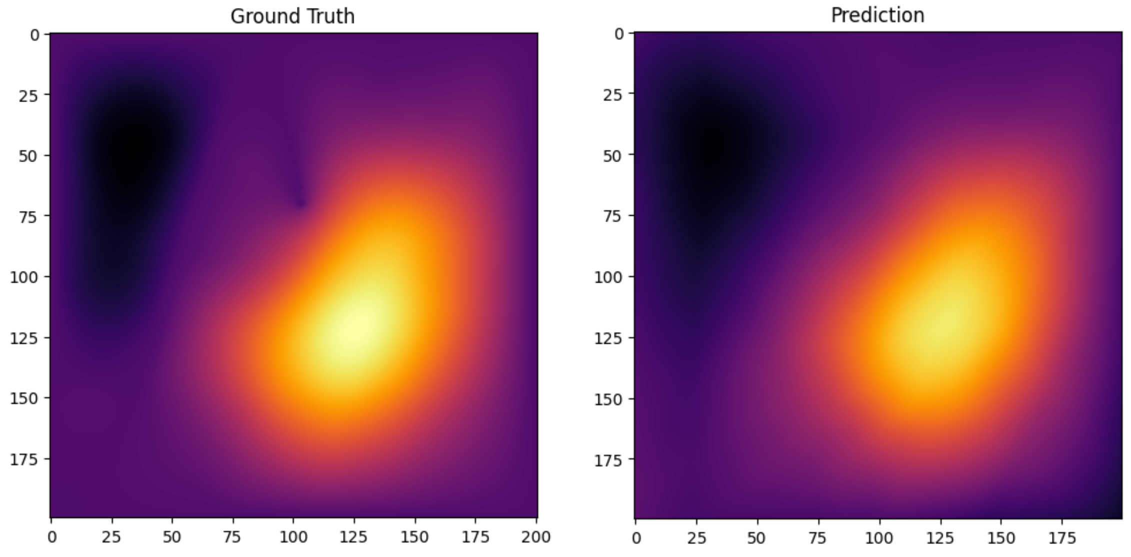


Figure 3.21: **Training Dataset;** Crack: 7

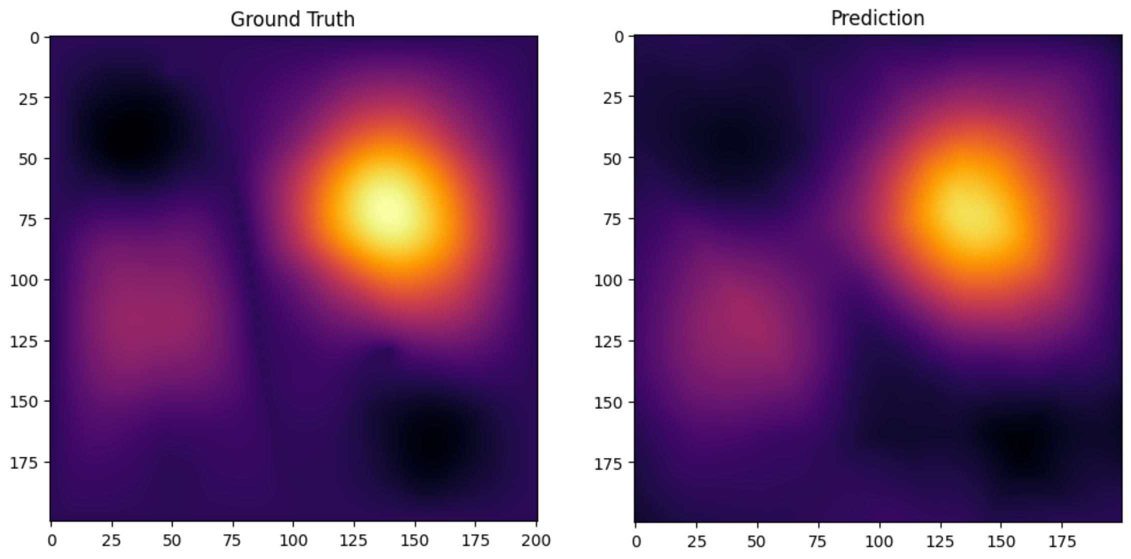


Figure 3.22: **Training Dataset;** Crack: 20

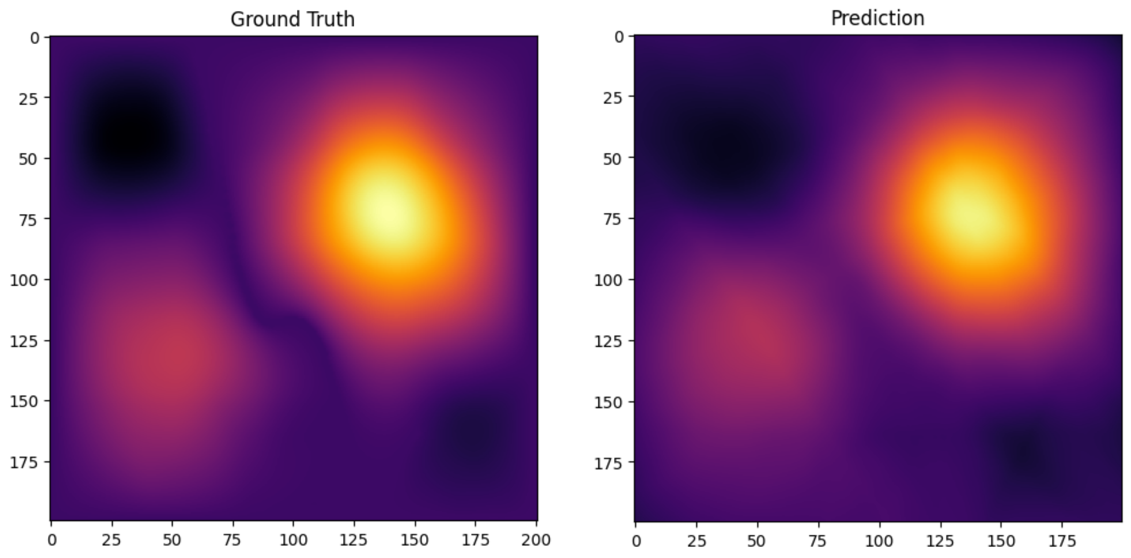


Figure 3.23: **Training Dataset;** Crack: 24

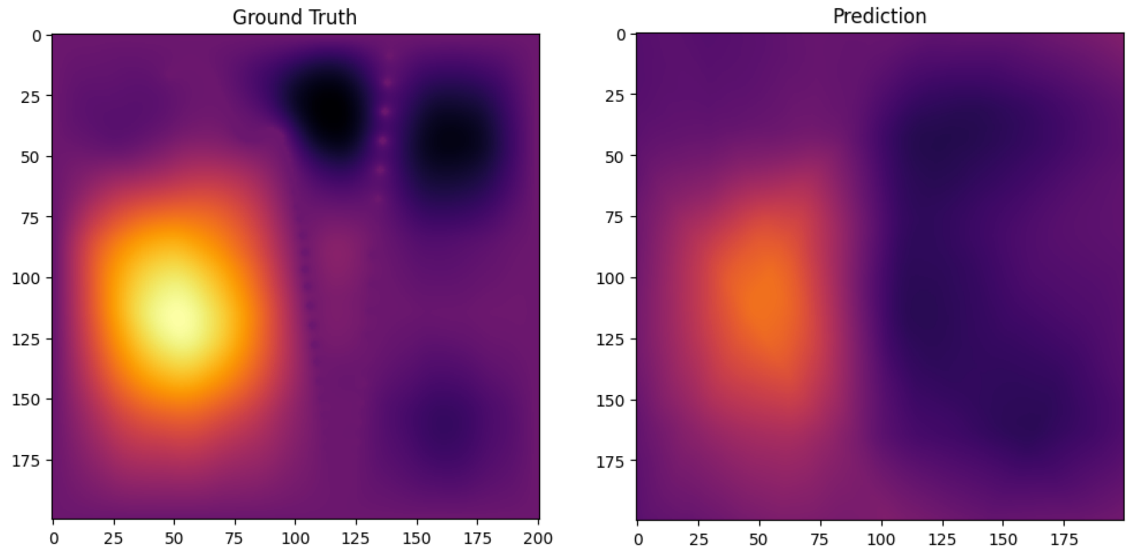


Figure 3.24: **Test Dataset;** Crack: 32

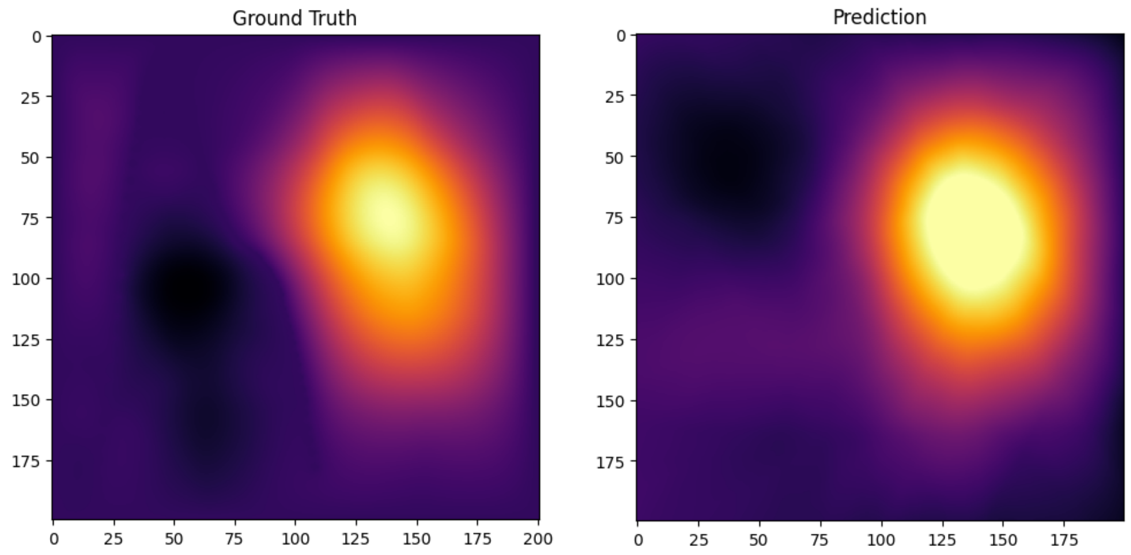


Figure 3.25: **Test Dataset;** Crack: 35

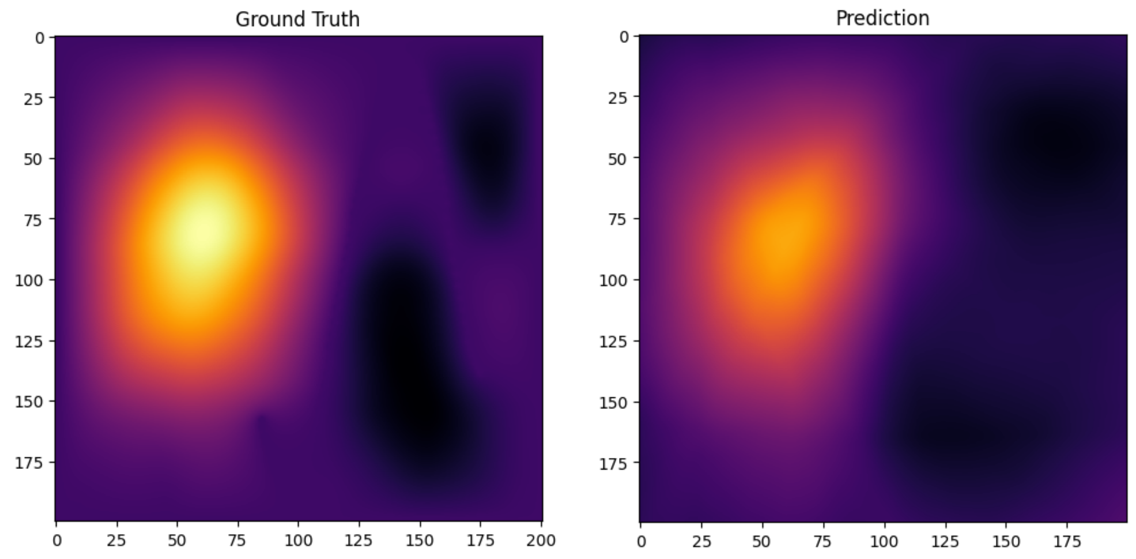


Figure 3.26: **Test Dataset;** Crack: 43

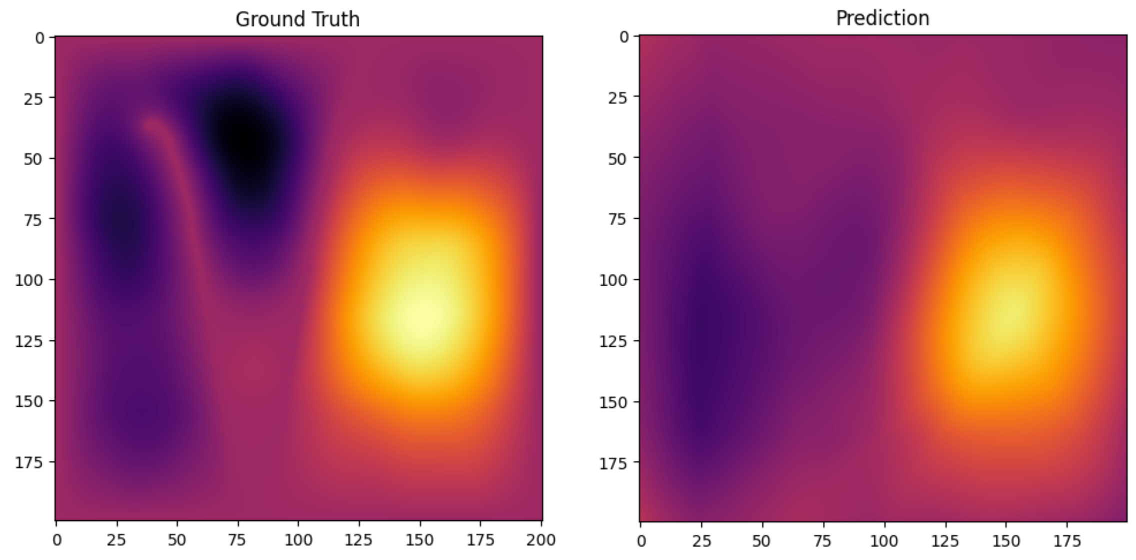


Figure 3.27: **Test Dataset;** Crack: 48

When comparing the performance of the model on training and test data, we can observe that the evaluation of the function on the training data is considerably better than that on the test data. Nonetheless, it is evident that the primary patterns within the wave propagation across different timesteps are learned, and the performance on the test data is rather satisfactory. Our model particularly excels in distinguishing between larger values (depicted in yellow) and smaller values (represented in black). Although this demonstrates that the primary barrier in the crack is well-captured, we still encounter challenges in making accurate predictions within the section divided by the crack. Since we have shown that the loss function consistently decreases, we could approach the overfitted model and achieve better performance at the crack with a similar formation. A more significant issue arises with cracks that are entirely different from those in the training data.

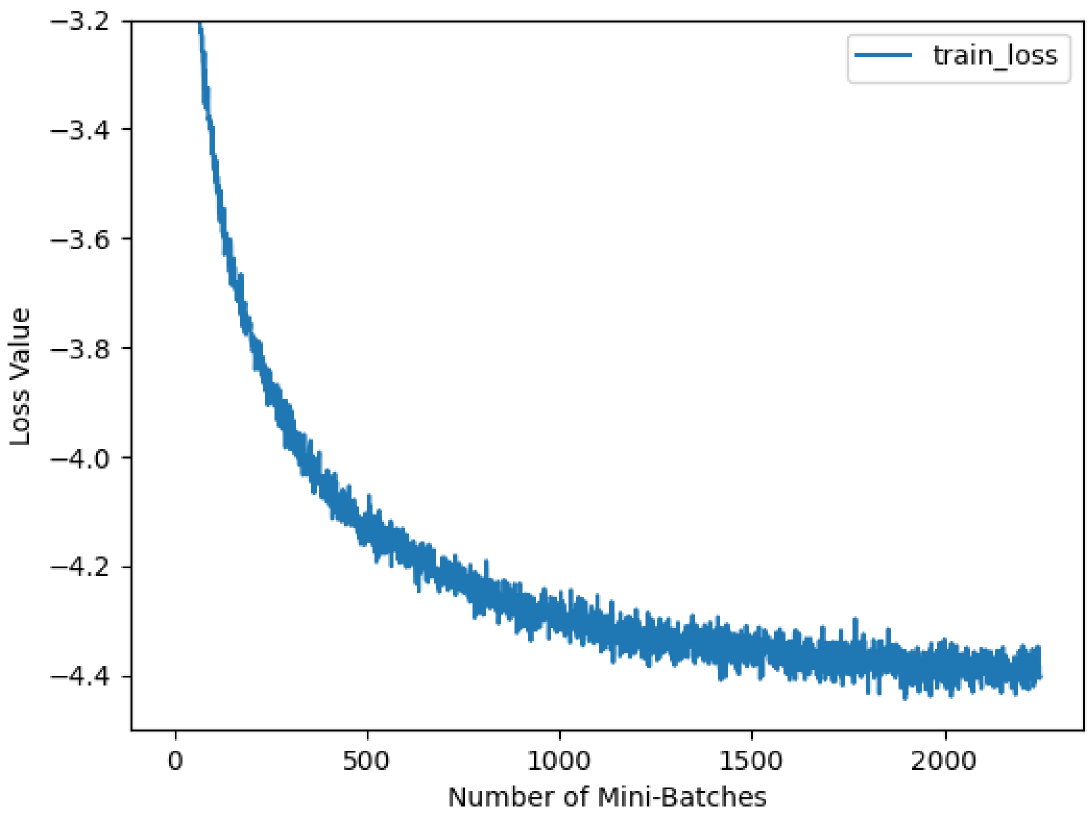


Figure 3.28: Logarithm of Training loss for 15 epochs (With Dropout layer).

In order to prevent overfitting, Dropout layers are incorporated into the network, and the model is retrained with the modified architecture. In the branch network, a Dropout layer with a defined probability of $p = 0.5$ is applied after the second convolution layer, as well as after the final Linear layer, with an identical probability. Additionally, a similar Dropout layer is applied to the trunk network following the second linear layer. Upon training this network for 15 epochs, Figures 3.28 and 3.29 depict the loss function values for the mini-batches and after each epoch, respectively. It can be concluded that the Dropout layer significantly reduces the validation loss for the model and improves the stability of model training.

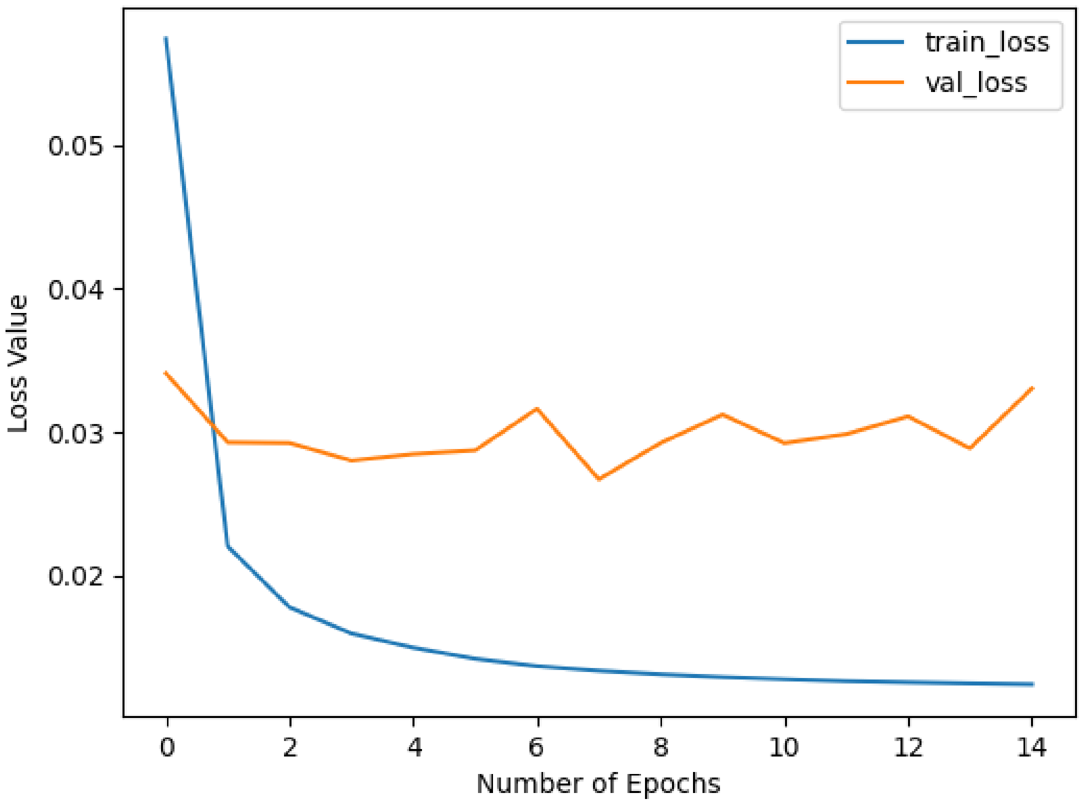


Figure 3.29: Training and Validation loss for 15 epochs (with Dropout layer).

Furthermore, two predictions are plotted, as illustrated in Figures 3.30 and 3.31. These predictions are derived from the model generated after the 8th epoch, as this model displays the lowest validation loss. In comparison with the preceding model, it can be concluded that the current model demonstrates better performance, with the improvements slightly visible in the plots.

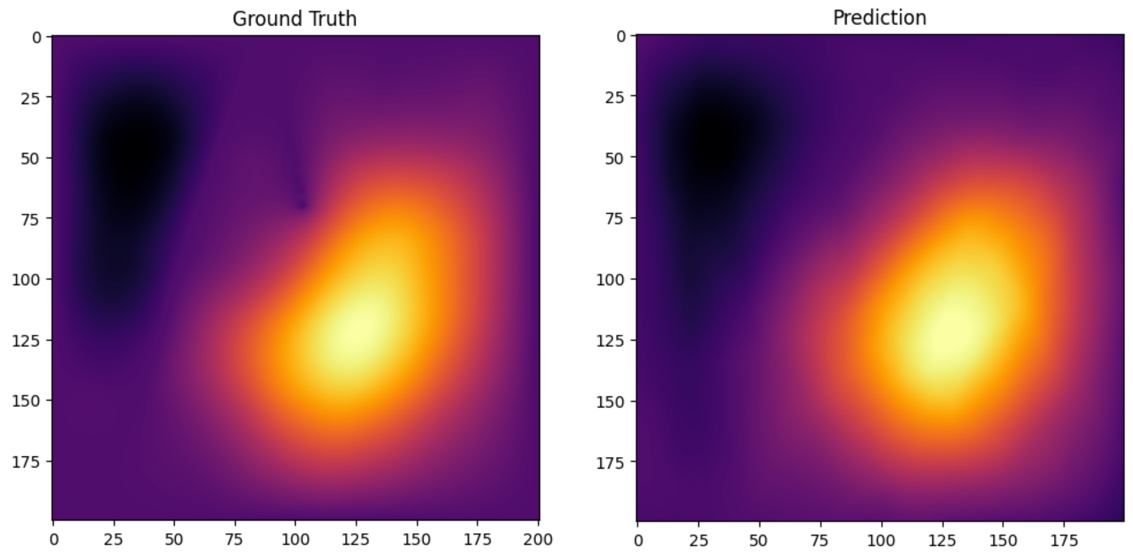


Figure 3.30: **Training Dataset;** Crack: 7 (with Dropout layer).

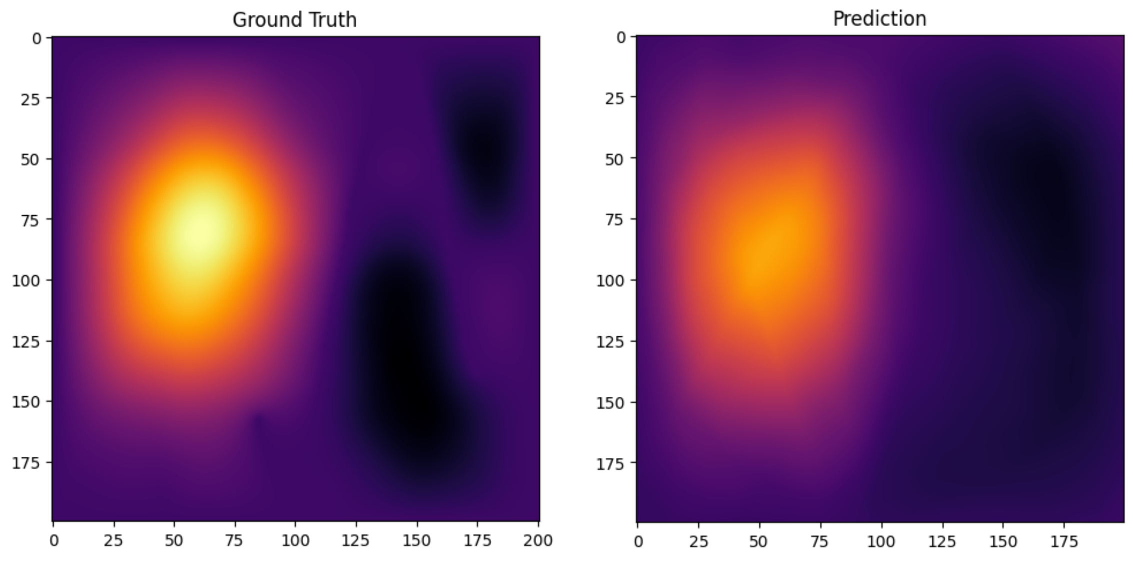


Figure 3.31: **Test Dataset;** Crack: 43 (with Dropout layer).

Moreover, an additional experiment was conducted to assess the performance of the DeepONet architecture on a dataset containing mappings of wave-propagated values from time step 0 to 30. The new network was trained for 30 epochs, incorporating the same Dropout layers as previously described. Figures 3.32 and 3.33 present the loss function values for both mini-batch and epoch, respectively. Upon examination, it is evident that the training loss consistently declines, while the validation loss ultimately decreases and exhibits stability.

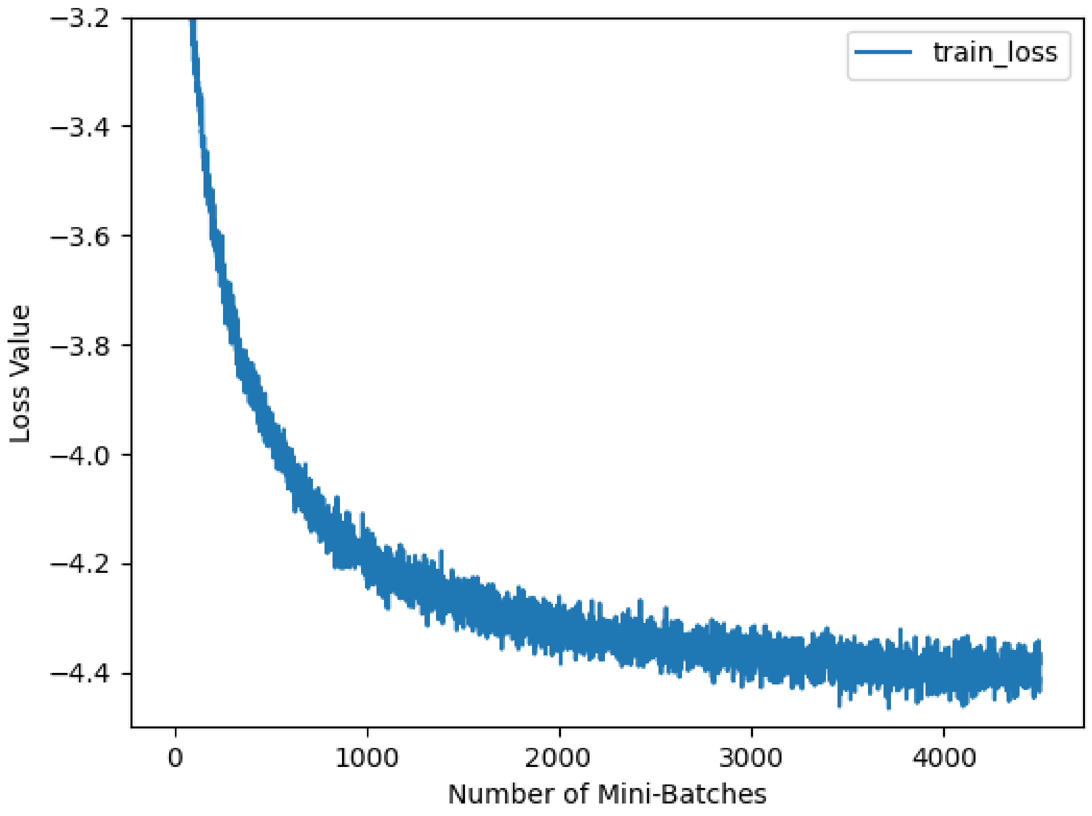


Figure 3.32: Logarithm of Training loss for 30 epochs (with Dropout layer, new mapping).

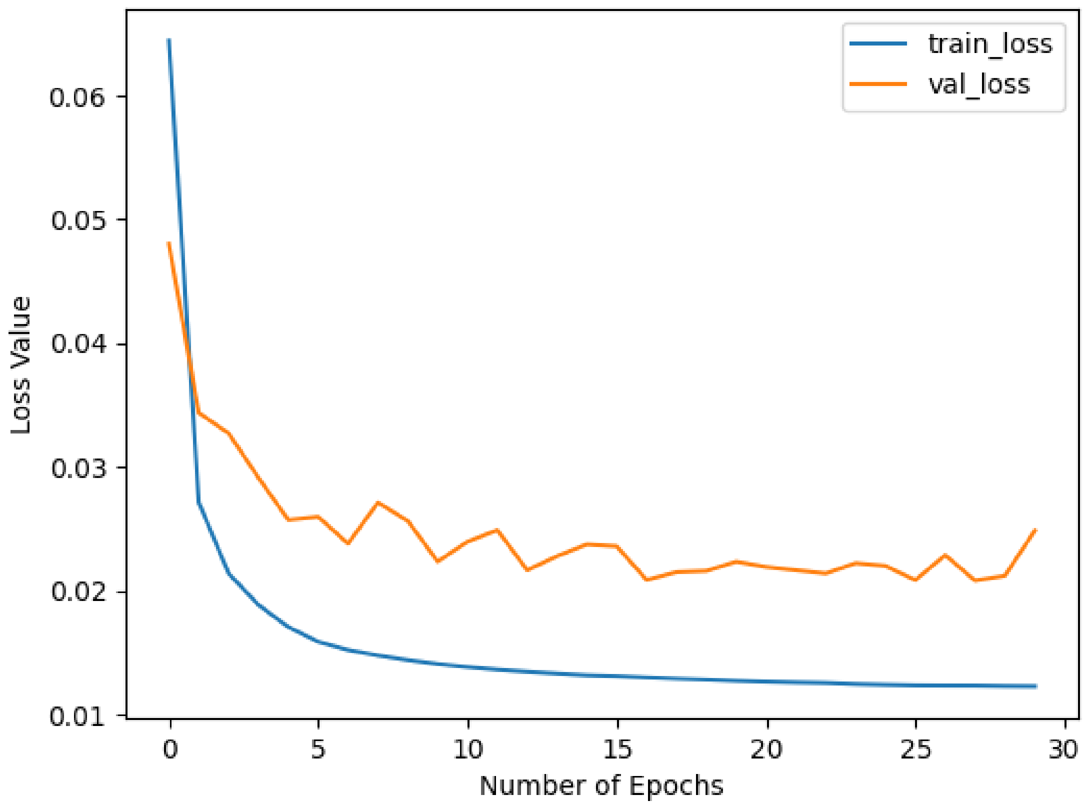


Figure 3.33: Training and Validation loss for 30 epochs (with Dropout layer, new mapping).

Two predictions are plotted, as illustrated in Figures 3.34 and 3.35. These predictions are derived from the model generated after the 17th epoch, as this model displays the lowest validation loss. It can be observed that the model effectively captures the contrast between areas with considerable differences in wave-propagated values.

On one hand, the performance on the training data is very good, and it is evident that the crack is accurately captured. On the other side, the performance on the test data does not meet expectations. However, this model demonstrates the promising potential for enhanced performance with future architectural adjustments.

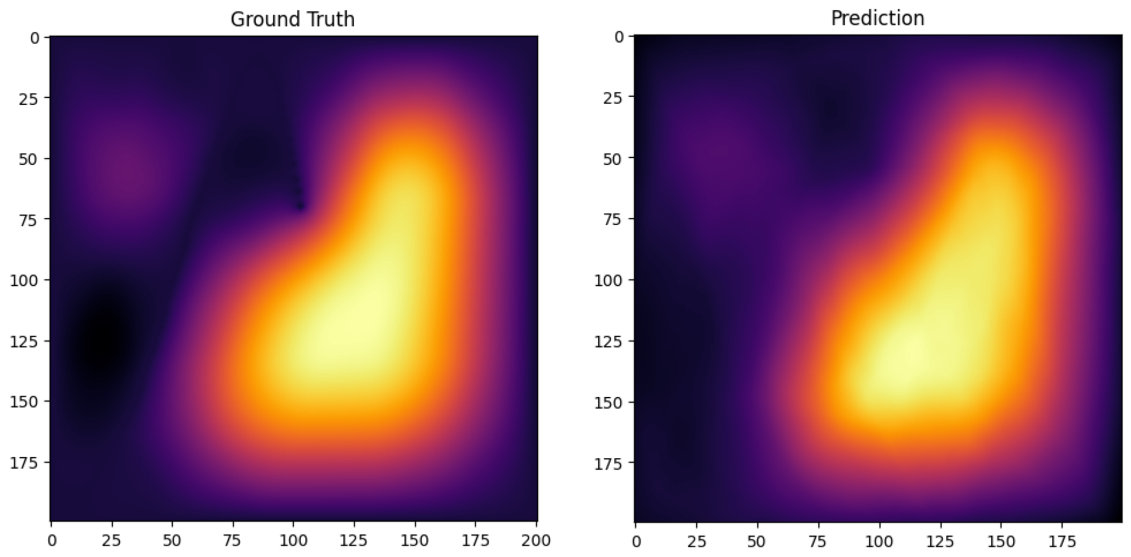


Figure 3.34: **Training Dataset;** Crack: 7 (with Dropout layer, new mapping).

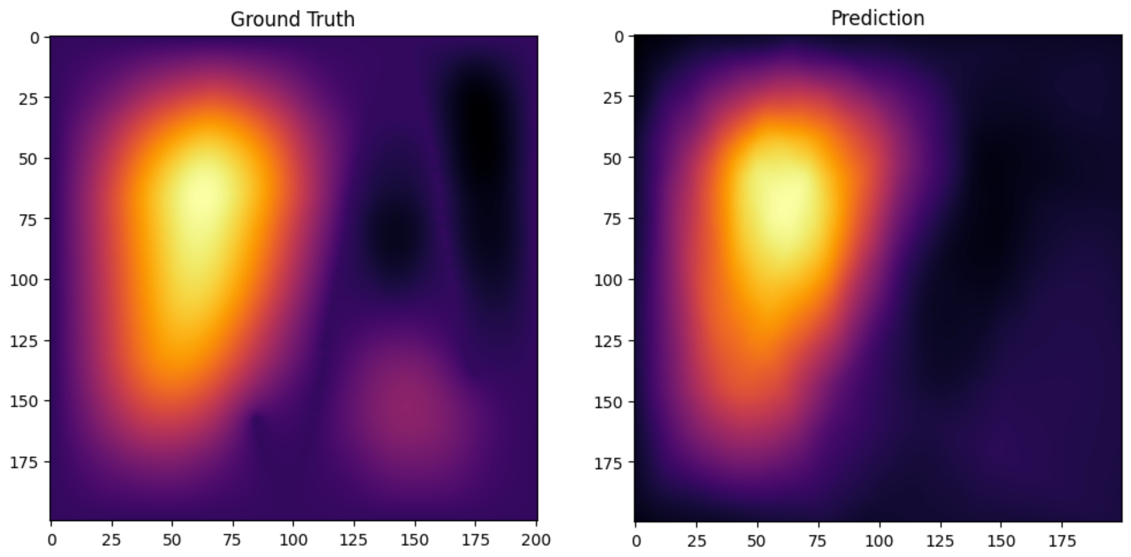


Figure 3.35: **Test Dataset;** Crack: 43 (with Dropout layer, new mapping).

To address the challenge of overfitting prevention, various techniques can be employed to enable extended training of the model across more epochs while preserving enhanced generalization on unseen data. This improvement is evident when multiple Dropout layers are applied, resulting in a notable reduction in validation loss. Additional strategies may include the implementation of Batch Normalization or data augmentation. Furthermore, an exploration into deepening the Branch network with a greater number of convolution filters and training the model on the modified architecture could be conducted.

In conclusion, the model demonstrates very good model's performance on the training samples, while encountering some issues with the test data.
However, it can be asserted that the DeepONet displays the potential to be utilized in problems with similar problem statements.

4 Conclusion

In this thesis, we implement the DeepONet neural network based on the paper by Lu et al. (2021) [25]. We evaluate the performance of our models in two different experiments, the first with complete wave propagation during the time and the second with the mapping of the spaces from one to another timestep.

4.1 Summary

Chapter 2 introduces the formulation of the problem statement for the wave equation and introduces the numerical method for solving partial differential equations, specifically the Finite Difference Method. Subsequently, the concept of neural operators is presented, outlining the basics of operator learning problem formulation. Here we aimed to introduce neural operators DeepONets [25] and FNOs [21], where we mostly concentrated on the DeepONet architecture, as this neural network type represents the core of the present work. We continue in Chapter 3 by discussing the idea of various DeepONet considerations when implementing the network. Furthermore, we present the two experiments which are executed and discuss the network implementation in each of them. Lastly, we plot the predictions and evaluate our model's performance.

4.2 Discussion

In the main section of this Master's thesis, various considerations and modifications are presented in order to improve the model's performance, and capabilities are discussed, primarily based on the paper [27]. These considerations include the addition of features in both branch and trunk networks, imposition of boundary conditions, utilization of Proper Orthogonal Decomposition, DeepONet scaling, and the adoption of fast implementation strategies, predominantly drawing inspiration from Fourier Neural Operators. Furthermore, the actual architecture of the DeepONet is introduced, detailing the parameters employed for training the network and the used metrics for evaluating the model's performance.
Throughout this work, two distinct experiments were conducted. The first experiment's dataset was obtained from the complete numerical simulation of the wave equation,

which was simulated for 1000 timesteps. Timesteps 200 to 399 were used as the training dataset and we evaluate the performance of our model. The DeepONet architecture utilized in this context initially explored Fully Connected Layers. The complete neural network was implemented "from scratch" using the Pytorch framework [34]. After concluding that the FCL implementation would not succeed, Convolutional layers were introduced to enhance performance, displaying significant improvement. Data augmentation for the branch network input was achieved through down and upsampling and showed potential, but was not used in all cases. Furthermore, the Adam optimizer demonstrated superior performance compared to the Stochastic Gradient Descent. The most influential change implemented in the model involved increasing the number of neurons in the Linear Layers of the trunk network, which exhibited greater capability in capturing wave movement over time. The model's evaluation revealed excellent performance on the training dataset and satisfactory performance on timesteps immediately following the training dataset. However, as the timesteps increased, the model's performance declined.
In the second experiment, a surrogate model was used, where various cracks within the medium were modeled. The model's performance was evaluated when applied as a neural operator for mapping the function of the wave propagated from one time step to another. The branch network input comprised different cracks modeled as initial conditions at time step 0, while the output was the function evaluated at time step 50. In addressing this problem, the model exhibited great performance, with the issue of overfitting. Validation loss increased, prompting the proposal to implement Dropout layers in the network. This consideration significantly improved the validation loss on the data and demonstrated the potential for addressing such problems. Additionally, we used the mapping timestep 0 to timestep 30, as our new dataset, and promising results were obtained.

4.3 Outlook

The DeepONet was capable of learning the problem statement for both experiments and displayed promising results for the application of the problems of similar statements. Although performance on unseen data was lacking, various techniques, such as Dropout layers, contributed to significant improvement. Additional strategies may include the implementation of Batch Normalization or data augmentation. Further exploration into deepening the Branch network with a greater number of convolution filters and training the model on the modified architecture could be conducted, as well as additional hyperparameter tuning. The DeepONet has potential for future application in similar problems, and may be used to solve more complex, multi-dimensional problems.

List of Figures

2.1 Finite Difference Method Grid, taken from [2]. 7
2.2 Illustration of the problem setup and architecture of DeepONets, taken from [25]. 11
2.3 Illustration of the problem setup and architecture of FNO, taken from [21]. 15
2.4 Comparison of Approximation Techniques, taken from [18]. 16

3.1 **Architecture of DeepONet. (A)** DeepONet architecture. If the trunk net is a feed-forward neural network, then it is a vanilla DeepONet. **(B)** Feature expansion of the trunk-net input. Periodic BCs can also be strictly imposed into the DeepONet by using the Fourier feature expansion. **(C)** Dirichlet BCs are strictly enforced in DeepONet by modifying the network output, taken from [27]. 19
3.2 **Architecture of DeepONet.** Branch Network uses Convolution Layers, while Trunk Network uses Fully Connected Layers. 23
3.3 The specification of the system used for experiments. 26
3.4 Wave function propagation during the time. 28
3.5 Different Timesteps considered as the initial condition. 29
3.6 Failed Prediction for First Experiment. 33
3.7 Failed Prediction 2. 35
3.8 Failed Prediction on Test Data. 37
3.9 Timestep: 201 39
3.10 Timestep: 298 40
3.11 Timestep: 333 40
3.12 Timestep: 400 41
3.13 Timestep: 450 41
3.14 Timestep: 550 42
3.15 Timestep: 650 42
3.16 Different Initial Conditions representing modeled cracks. 44
3.17 Ground Truths at the timestep 50 for respective Initial Conditions. . . . 45
3.18 Logarithm of Training loss for 15 epochs. 48
3.19 Training and Validation loss for 15 epochs. 49
3.20 **Training Dataset;** Crack: 0 50

3.21 **Training Dataset;** Crack: 7 50
3.22 **Training Dataset;** Crack: 20 51
3.23 **Training Dataset;** Crack: 24 51
3.24 **Test Dataset;** Crack: 32 52
3.25 **Test Dataset;** Crack: 35 52
3.26 **Test Dataset;** Crack: 43 53
3.27 **Test Dataset;** Crack: 48 53
3.28 Logarithm of Training loss for 15 epochs (With Dropout layer). 54
3.29 Training and Validation loss for 15 epochs (with Dropout layer). 55
3.30 **Training Dataset;** Crack: 7 (with Dropout layer). 56
3.31 **Test Dataset;** Crack: 43 (with Dropout layer). 56
3.32 Logarithm of Training loss for 30 epochs (with Dropout layer, new mapping). 57
3.33 Training and Validation loss for 30 epochs (with Dropout layer, new mapping). 58
3.34 **Training Dataset;** Crack: 7 (with Dropout layer, new mapping). 59
3.35 **Test Dataset;** Crack: 43 (with Dropout layer, new mapping). 59

List of Tables

3.1 Overview of Branch Network Convolution Layers 22
3.2 Parameters used for Adam Optimizer during training 24
3.3 Configuration for the First Experiment 32
3.4 Configuration for the Second Experiment 35
3.5 Configuration for the Third Experiment 37
3.6 Configuration for the Final Experiment 38
3.7 Configuration for Operator Learning 47

Bibliography

[1] A. F. Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. 2019. arXiv: 1803.08375 [cs.NE].

[2] A. H. Ali, A. Jaber, M. Yaseen, M. RASHEED, O. Bazighifan, and T. Nofal. "A Comparison of Finite Difference and Finite Volume Methods with Numerical Simulations: Burgers Equation Model." In: *Complexity* 2022 (June 2022). DOI: 10.1155/2022/9367638.

[3] K. Bhattacharya, B. Hosseini, N. B. Kovachki, and A. M. Stuart. "Model Reduction and Neural Networks for Parametric PDEs." In: (2021). arXiv: 2005.03180 [math.NA].

[4] S. BINDER. *Wave equation simulations 1D/2D (équation de D'Alembert)*. https://github.com/sachabinder/wave_equation_simulations. 2021.

[5] L. Bottou and O. Bousquet. "The Tradeoffs of Large Scale Learning." In: 20 (2008). Ed. by J. Platt, D. Koller, Y. Singer, and S. Roweis, pp. 161–168.

[6] T. Chen and H. Chen. "Approximation capability to functions of several variables, nonlinear functionals, and operators by radial basis function neural networks." In: *IEEE transactions on neural networks* 6.4 (1995), pp. 904–910. ISSN: 1045-9227. DOI: 10.1109/72.392252.

[7] T. Chen and H. Chen. "Approximations of continuous functionals by neural networks with application to dynamic systems." In: *IEEE Transactions on Neural Networks* 4.6 (1993), pp. 910–918. DOI: 10.1109/72.286886.

[8] T. Chen and H. Chen. "Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems." In: *IEEE Transactions on Neural Networks* 6.4 (1995), pp. 911–917. DOI: 10.1109/72.392253.

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

[10] N. B. Erichson, M. Muehlebach, and M. W. Mahoney. "Physics-informed Autoencoders for Lyapunov-stable Fluid Flow Prediction." In: (2019). arXiv: 1905.10866 [physics.comp-ph].

[11] X. Glorot and Y. Bengio. “Understanding the difficulty of training deep feedforward neural networks.” In: *Journal of Machine Learning Research - Proceedings Track* 9 (Jan. 2010), pp. 249–256.

[12] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. “Array programming with NumPy.” In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`.

[13] K. He, X. Zhang, S. Ren, and J. Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: `1502.01852 [cs.CV]`.

[14] J. D. Hunter. “Matplotlib: A 2D graphics environment.” In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[15] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: `1502.03167 [cs.LG]`.

[16] P. Jin, L. Lu, Y. Tang, and G. E. Karniadakis. “Quantifying the generalization error in deep learning in terms of data distribution and neural network smoothness.” In: *Neural Networks* 130 (Oct. 2020), pp. 85–99. DOI: `10.1016/j.neunet.2020.06.024`.

[17] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`.

[18] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar. “Neural Operator: Learning Maps Between Function Spaces.” In: (2022). arXiv: `2108.08481 [cs.LG]`.

[19] M. P. Lamoureux. “The mathematics of PDEs and the wave equation.” In: (2006).

[20] P. C. D. Leoni, L. Lu, C. Meneveau, G. Karniadakis, and T. A. Zaki. *DeepONet prediction of linear instability waves in high-speed boundary layers*. 2021. arXiv: `2105.08697 [physics.flu-dyn]`.

[21] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. “Fourier Neural Operator for Parametric Partial Differential Equations.” In: (2021). arXiv: `2010.08895 [cs.LG]`.

[22] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. “Neural Operator: Graph Kernel Network for Partial Differential Equations.” In: (2020). arXiv: `2003.03485 [cs.LG]`.

[23] S. Linge and H. P. Langtangen. “Wave Equations.” In: *Finite Difference Computing with PDEs: A Modern Software Approach*. Cham: Springer International Publishing, 2017, pp. 93–205. ISBN: 978-3-319-55456-3. DOI: `10.1007/978-3-319-55456-3_2`.

[24] L. Lu. "Dying ReLU and Initialization: Theory and Numerical Examples." In: *Communications in Computational Physics* 28.5 (June 2020), pp. 1671–1706. DOI: 10.4208/cicp.oa-2020-0165.

[25] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis. "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators." In: *Nature Machine Intelligence* 3.3 (Mar. 2021), pp. 218–229. DOI: 10.1038/s42256-021-00302-5.

[26] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis. "The finite difference method." In: *Nature Machine Intelligence* 3.3 (Mar. 2021), pp. 218–229. DOI: 10.1038/s42256-021-00302-5.

[27] 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." In: *Computer Methods in Applied Mechanics and Engineering* 393 (Apr. 2022), p. 114778. DOI: 10.1016/j.cma.2022.114778.

[28] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis. "DeepXDE: A Deep Learning Library for Solving Differential Equations." In: *SIAM Review* 63.1 (Jan. 2021), pp. 208–228. DOI: 10.1137/19m1274067.

[29] L. Lu, Y. Shin, Y. Su, and G. Karniadakis. "Dying ReLU and Initialization: Theory and Numerical Examples." In: *Communications in Computational Physics* 28 (Nov. 2020), pp. 1671–1706. DOI: 10.4208/cicp.OA-2020-0165.

[30] H. N. Mhaskar and N. Hahm. "Neural Networks for Functional Approximation and System Identification." In: *Neural Computation* 9.1 (Jan. 1997), pp. 143–159. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.1.143. eprint: https://direct.mit.edu/neco/article-pdf/9/1/143/813389/neco.1997.9.1.143.pdf.

[31] D. Morin. *Waves*. Accessed: 11 May 2023. eprint: https://scholar.harvard.edu/david-morin/waves.

[32] A. Murdoch. "The propagation of surface waves in bodies with material boundaries." In: *Journal of the Mechanics and Physics of Solids* 24 (June 1976), pp. 137–146. DOI: 10.1016/0022-5096(76)90023-5.

[33] K. O'Shea and R. Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE].

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[35] T. Qin, K. Wu, and D. Xiu. "Data driven governing equations approximation using deep neural networks." In: *Journal of Computational Physics* 395 (Oct. 2019), pp. 620–635. DOI: `10.1016/j.jcp.2019.06.042`.

[36] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Multistep Neural Networks for Data-driven Discovery of Nonlinear Dynamical Systems." In: (2018). arXiv: `1801.01236 [math.DS]`.

[37] F. Rossi and B. Conan-Guez. "Functional multi-layer perceptron: a non-linear tool for functional data analysis." In: *Neural networks : the official journal of the International Neural Network Society* 18 1 (2007), pp. 45–60.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958.

[39] A. Strohmaier and A. Waters. "The relative trace formula in electromagnetic scattering and boundary layer operators." In: (2021). arXiv: `2111.15331 [math.AP]`.

[40] N. Winovich, K. Ramani, and G. Lin. "ConvPDE-UQ: Convolutional neural networks with quantified uncertainty for heterogeneous elliptic partial differential equations on varied domains." In: *Journal of Computational Physics* 394 (May 2019). DOI: `10.1016/j.jcp.2019.05.026`.

[41] A. Yazdani, L. Lu, M. Raissi, and G. E. Karniadakis. "Systems biology informed deep learning for inferring parameters and hidden dynamics." In: *PLOS Computational Biology* 16.11 (Nov. 2020), pp. 1–19. DOI: `10.1371/journal.pcbi.1007575`.

[42] D. Zhang, L. Guo, and G. E. Karniadakis. *Learning in Modal Space: Solving Time-Dependent Stochastic PDEs Using Physics-Informed Neural Networks*. 2019. arXiv: `1905.01205 [cs.LG]`.

[43] Y. Zhu, N. Zabaras, P.-S. Koutsourelakis, and P. Perdikaris. "Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data." In: *Journal of Computational Physics* 394 (Oct. 2019), pp. 56–81. DOI: `10.1016/j.jcp.2019.05.024`.