Fast, Accurate, and Scalable Numerical Wave Propagation: Enhancement by Deep Learning

A DISSERTATION PRESENTED BY

Luis Kaiser

TO THE INSTITUTE OF MATHEMATICS
Julius-Maximilians-Universität Würzburg

FOR THE DEGREE OF
Master of Science
IN THE SUBJECT OF MATHEMATICS

JANUARY 2024

The Thesis Committee for Luis Kaiser certifies that
this is the approved version of the following dissertation:

Fast, Accurate, and Scalable Numerical Wave Propagation:

Enhancement by Deep Learning

Committee:

Prof. Dr. Christian Klingenberg

Prof. Dr. Yen-Hsi Richard Tsai

Thesis advisors: Prof. Christian Klingenberg & Prof. Yen-Hsi Richard Tsai

Fast, Accurate, and Scalable Numerical Wave Propagation: Enhancement by Deep Learning

Summary

In a variety of scientific and engineering domains, ranging from seismic modeling to medical imaging, the need for precise and efficient solutions for high-frequency wave propagation holds great significance. However, traditional numerical solvers face a trade-off between accuracy and computational efficiency, often failing to capture non-negligible fine-scale dynamics without incurring significant computational costs.

Recent advances in wave modeling use sufficiently accurate fine solver outputs to train neural networks that enhance the accuracy of a computationally efficient but inaccurate coarse solver. While most existing methods are limited to smooth wave speeds or do not preserve physical properties, enhancing fast numerical solvers can offer stable wave propagation across diverse media with multiscale speeds. A stable and sufficiently accurate enhanced solver allows the use of Parareal, a parallel-in-time algorithm to further reduce the wall-clock computation time for numerical simulations, and provide a mechanism to refine scientific machine learning solutions.

In this thesis, we build upon the work of Nguyen and Tsai [1] and present a novel unified system that integrates a numerical solver with deep learning components into an end-to-end framework. In the proposed setting, we investigate advancements to the neural network architecture, improve the sampling strategy for the wave solution manifold to include time-dependent dynamics, and employ the Parareal scheme and regularization methods within this novel setup. Our results show that the cohesive structure significantly improves performance without sacrificing speed, and demonstrate the importance of temporal dynamics for accurate wave propagation.

Schnelle, Akkurate und Skalierbare Numerische Wellenausbreitung: Erweiterung durch Deep Learning

Zusammenfassung

In Bereichen wie Seismik und medizinischer Bildgebung ist die präzise und schnelle Berechnung von hochfrequenten Wellen essentiell. Jedoch stoßen konventionelle numerische Ansätze schnell an Grenzen: Bei feingranularen Dynamiken erzeugen sie oft nicht zu vernachlässigende instabile Resultate oder hohe Rechenlast.

Neueste Wellenmodellierungsmethoden nutzen daher ausreichend genaue numerische Algorithmen zum Training neuronaler Netze, um die Genauigkeit von recheneffizienten grobmaschigen Verfahren zu erhöhen. Während die meisten konventionallen Ansätze entweder glatte Wellengeschwindigkeiten voraussetzen oder physikalische Eigenschaften missachten, können optimierte schnelle Löser stabilere Wellenausbreitungen in multiskaligen Medien bieten. Ein stabiler und ausreichend genauer verbesserter Löser ermöglicht den Einsatz von Parareal, ein parallel-in-time Algorithmus, um die Berechnungszeit von numerischen Simulationen weiter zu reduziert. Dies bietet ein Mechanismus zur Verfeinerung von scientific machine learning Lösungen.

In dieser Masterarbeit bauen wir auf die Arbeit von Nguyen und Tsai [1] auf und stellen ein end-to-end System vor, das einen numerischen Löser mit deep learning Komponenten in eine vereinheitlichte Struktur integriert. Im vorgestellten Setup optimieren wir die Netzwerkarchitektur, verbessern das Stichprobenverfahren der Wellenlösungs-Mannigfaltigkeit, um zeitabhängige Dynamiken zu berücksichtigen, und setzen das Parareal-Verfahren und Regularisierungsmethoden in dieser neuen Konfiguration ein. Unsere Ergebnisse zeigen eine signifikante Leistungssteigerung ohne Geschwindigkeitsverlust durch den ganzheitliche Ansatz, und unterstreichen die Bedeutung von zeitabhängigen Merkmalen für eine zuverlässige Wellenberechnung.

Acknowledgments

Throughout the writing of this thesis, as well as the entire duration of my studies, I have received a great deal of support and assistance.

I would like to express my deepest appreciation to Prof. Dr. Christian Klingenberg (The University of Wuerzburg) for his excellent guidance and expertise throughout the course of my master's thesis.

I am extremely grateful to Professor Prof. Dr. Yen-Hsi Richard Tsai (The University of Texas at Austin) for providing his great mentorship, welcoming me into his research group and enabling me to conduct research at The University of Texas at Austin.

Lastly, I would like to thank the research groups of both my advisors for interesting ideas and discussion, as well as providing me with valuable feedback on my thesis. I also thank Texas Advanced Computing Center (TACC) for the computing resources.

Contents

1 Introduction

Wave propagation in media with complex materials and environments is a pivotal problem in fields such as hydrocarbon exploration, medical imaging, and non-destructive testing. For instance, when searching natural gas, such calculations form the forward part of a numerical method for solving the inverse problem of geophysical inversion. By accurately propagating acoustic waves and analyzing the reflections and transmissions generated by media discontinuities, it becomes possible to characterize underground formations. Given the numerical approach and extensive data requirements involved in most algorithms, frequent solving of the forward problem is required. Consequently, an efficient wave propagator is indispensable to make solving the inverse problem practical.

We consider a second-order linear wave equation,

$$\begin{aligned} &u_{tt} = c^2(x)\,\Delta u, \quad x \in [-1,1]^2, 0 \le t < T,\\ &u(x,0) = u_0(x),\\ &u_t(x,0) = p_0(x), \end{aligned} \tag{1.1}$$

with respect to the space variable x and time t. $c(x) \in \mathbb{R}^2$ is the piecewise smooth wave speed in the 2D space. We impose either absorbing or periodic boundary conditions at the domain boundary. Numerically propagating waves in complex, non-uniform media with high accuracy is a challenging task. Traditional numerical computations often demand a fine spatial and temporal representation to accurately capture the propagation of high-frequency waves, handle erratic variations in the wave speed, and guarantee stability. However, this increased computational cost can present severe challenges for use in large-scale computations.

Recent advances in deep learning suggest that the techniques may be promising to improve the efficiency of wave propagation simulations. These techniques involve the training of neural networks to complement traditional numerical solvers, resulting in a reduction of wall-clock time, an increase in solution precision, and enhanced robustness against outliers.

Problem. While some approaches yield remarkable results using deep learning techniques for wave simulation, the focus of these studies primarily lies in the design of neural networks and datasets for a specific task [1, 2, 3]. However, generalized conclusions are limited because practical systems oftentimes demand preceding analysis for complex, discontinuous media or a detailed tuning of inputs [2, 4]. Similarly, well-established numerical solvers [5, 6] are avoided to prioritize speed; as a result, the predictions deviate from the physics described by the wave equation, and even small outliers may cause instability. Especially when wave propagation needs to be simulated over an extended period, these methods can diverge since they are trained on shorter time horizons and lack stabilization.

Objective. Combining a numerical solver with neural network components to solve the wave equation efficiently and accurately across a variety of wave speed profiles is a central point of our research. We take a first step by expanding the method of Nguyen and Tsai [1] and build an end-to-end model that enhances a computationally cheap numerical solver through deep learning. By considering the entire process from input to output, the unified system can learn complex relationships between individual components, eliminating the need for manual pre-processing of input data or intermediate steps.

Approach and Contribution. An efficient numerical solver $\mathcal{G}_{\Delta t}\mathfrak{u} \equiv \mathcal{G}_{\Delta t}[\mathfrak{u}, c]$ is used to propagate a wave $\mathfrak{u}(x,t) = (u, \partial_t u)$ for a time step $t + \Delta t$ on a medium described by the wave speed $c(x)$. This method is computationally cheap since the wave advancements are computed on a coarse grid; however, it is consistently less accurate than an expensive fine solver $\mathcal{F}_{\Delta t}\mathfrak{u} \equiv \mathcal{F}_{\Delta t}[\mathfrak{u}, c]$. Consequently, the solutions may exhibit numerical dispersion errors and missing high-fidelity details as a consequence of under-resolving the media and the wave fields.

We use a restriction operator $\mathcal{R}$ which transforms functions from a fine grid to a coarse grid. Additionally, for mapping coarse grid functions to a fine grid, a prolongation (e.g. interpolation) operator $\mathcal{I}$ is used. We can now define the efficient low-fidelity propagator $\Psi_{\Delta t} := \mathcal{I}G_{\Delta t}\mathcal{R}$. $\Psi_{\Delta t}$ takes a wave field $\mathfrak{u}$ defined on the fine grid, propagates it on a coarser grid, and returns the resulting wave field on the fine grid. As we mentioned above, $\Psi_{\Delta t}$ under-resolves the wave field on the fine grid in heterogeneous wave media. Therefore, we deploy a more elaborate technique to augment the accuracy of coarse solver ($\mathcal{G}_{\Delta t}$), as presented in [1, 7, 8]. We construct different variants that integrate either one or two neural network components with $\mathcal{G}_{\Delta t}$ end-to-end, which can be expressed as $\Psi_{\Delta t}[\mathfrak{u}, c, \theta] \equiv \Psi^{\theta}_{\Delta t} \in \{\Psi^{\theta,0}_{\Delta t} := \mathcal{I}^{\theta}\mathcal{G}_{\Delta t}\mathcal{R}^{0}, \Psi^{\theta_1,\theta_2}_{\Delta t} := \mathcal{I}^{\theta_1}\mathcal{G}_{\Delta t}\mathcal{R}^{\theta_2}\}$. The superscript 0 indicates interpolation, and θ indicates a neural network.

In a supervised learning framework, we aim to reduce the discrepancy between $\mathcal{G}_{\Delta t}$ and the ground truth wave field, while the outputs from the fine solver ($\mathcal{F}_{\Delta t}$) provide the training labels:

$$\mathfrak{u}_{n+1} := \mathfrak{u}(x, t + \Delta t) = \mathcal{F}_{\Delta t}\mathfrak{u}_n \approx \Psi^{\theta}_{\Delta t}\mathfrak{u}_n. \tag{1.2}$$

The models are parameterized by the family of initial wave fields $\mathfrak{F}_{\mathfrak{u}_0}$ and the family of wave speeds $\mathfrak{F}_c$. Therefore, the wave solution manifold is defined as

$$\mathcal{M} := \{\mathfrak{u}(x, t; u_0) \mid \mathfrak{u}(x, t; u_0) \text{ solves Eq. (1.1) for } c \in \mathfrak{F}_c, \mathfrak{u}_0 \in \mathfrak{F}_{\mathfrak{u}_0}\}. \tag{1.3}$$

Thus, $\forall \mathfrak{u}_0 \in \mathcal{M}$, it holds that $\mathfrak{u}(x, t; u_0) \in \mathcal{M}$.

Our key contributions include:

(i) Enhancements to the wave propagation setup from [1] by integrating neural networks and numerical simulations across diverse media conditions into a cohesive structure for optimized component interplay. This includes refinements to both the training algorithm and data generation techniques, as well as a systematic evaluation process.

(ii) The stability of the model, including both the numerical scheme and the deep learning inference, is improved through a refined sampling of the wave solution manifold $\mathcal{M}$, thereby approximating temporal propagation features.

(iii) Absorbing boundaries are added to simulate waves that exit the domain without reflecting back. This is realistic for certain types of problems, including seismic analysis or acoustics.

(iv) In a large-scale analysis ($\approx$ 6,552 GPU hours), we investigate the efficacy of the end-to-end structure, modifications to the deep learning architecture, loss function enhancements to capture time-dependent features, Parareal [9], and regularization methods, adapted for the end-to-end framework. We further provide insights gained about hyperparameters and training using fANOVA [10], and systematically assess numerical configurations.

This thesis is structured as follows: First, we provide an overview of the different areas of numerical wave propagation aided by deep learning and a review of its significant scientific publications is given in Chapter 2. Section 3.1 contains an introduction to the wave equation and a numerical approximation scheme. The supervised learning framework of training the wave propagator, and deep learning theory is explained in Section 3.2. Chapter 4 describes the experiment datasets and the architecture of the models. We discuss the results of the evaluation in Chapter 5. Lastly, a conclusion and thoughts on future work are given in Chapter 6.

2 Related Work

Gradient-based techniques for solving Partial Differential Equations (PDEs) have been an active area of research. Recent work by Rassini et al. [6] focuses on the application of Physics-Informed Neural Networks (PINNs) on the entire simulation of forward and inverse problems. PINN have physical constraints within the loss function, usually in the form of PDEs. Thus, errors caused by fully deep learning architectures are avoided, and regressors can extrapolate accurately beyond the training data boundaries. However, the results in [11] suggest that the loss landscape of PINN architectures are often ill-conditioned due to a limited expressiveness. Moreover, in the context of the 2D wave equation, PINN models frequently require refinements for any change in the medium [2, 3]. Particularly for inverse problems where details about the medium are unknown, the lack of adaptability is inadequate for general application.

[7, 8] aim to find alternative ways to integrate physical understanding into the model's setup. In their work, Convolutional Neural Networks (CNNs) enhance low-fidelity numerical solutions, particularly fluid dynamic simulations, and the Helmholtz equation, in a data-driven manner. The basic idea is to intersperse classical numerical iterations with neural network corrections trained by a fine grid solver. [1] adopted their setup for the 2D acoustic wave propagation problem. They employed a CNN resembling an autoencoder architecture, augmented with skip connections, to advance waves in challenging media.

These studies serve as a proof-of-concept and successfully establish stable wave propagation simulators. More specifically, the presented CNN-based mechanisms conserve the total energy of the system (i.e., symplectic) and are unaffected by the direction of time (i.e., time-reversible). Yet the authors have not explored the robustness against outliers, the effect of various model components on the accuracy, or absorbing boundary conditions in greater depth. Factors like optimal numerical configurations, such as the grid resolution, and deep learning specifics, such as the type of neural network architecture or the training algorithm, also remain unexplored. Consequently, no reliable conclusions can be made about their speed advantages or stability. In addition, the numerical method is not embedded in the neural network. Taking advantage of correlations between system components can lead to further accuracy improvements.

Many studies, including [12, 13], demonstrate the benefits of Parareal, a parallel-in-time algorithm for solving time-dependent PDEs, to correct a coarse solver through a sophisticated wave propagator. The setup presented by Nguyen and Tsai [1, 14] deploys this scheme to add back missing high-frequency components during online computation through a 'self-improving' looping mechanism. Our deep learning setup follows their design: Parareal iterations are used to iteratively refine the computed time series of wave fields. Therefore, we aim to introduce more samples beyond the training set than other approaches [7, 8]. While existing models achieve good results using finite-difference time-domain modeling, their dataset is limited to single time-step wave advancements. Furthermore, this motivates the study of an end-to-end system that utilizes the Parareal scheme, which iteratively alters the computed temporal dynamics.

Beyond related work. Following the philosophy of PINNs, our model incorporates physical domain knowledge to propagate waves end-to-end. However, to overcome the limitations of PINNs, we have adapted the modular framework of [1] by fusing a symplectic, time-reversible numerical solver along with CNNs. This single cohesive system allows us to refine the traditional supervised learning process: The model is applied to its previous solution and learns from a sequence of wave advancements.

At this time, few if any models deploy a deep learning method to solve the 2D wave equation with absorbing boundary conditions. Thus, we can achieve significant reduction in computation time relative to existing methods due to parallel computation and the fast and precise processing of neural networks.

The wave propagation algorithms under discussion are applied to different learning problems and test environments, which differ significantly in scale and nature. Such variability undermines reliable conclusions about their performance. A comparative study of the efficacy of different architectures, training algorithms, and optimization techniques, such as Parareal, was missing. This thesis bridges that gap and approaches the challenge of increasing efficiency without compromising on accuracy.

3 Theory

The purpose of this chapter is to review initial-boundary value problems for the 2D wave equation and introduce the core concepts of our proposed end-to-end model. In particular, we will explain numerical approximations for the wave equation, study the utilized building blocks of our neural networks, training refinements and the Parareal scheme.

3.1 The Wave Equation

The wave equation is a second-order linear Partial Differential Equation (PDE) that describes movements of an acoustic, electromagnetic, or seismic waves. In this simple model, waves mainly advance in the direction of their oscillations. The evolution of wave displacement u as a function of space $x \in [-1,1]^2$ and time $0 \leq t < T$ takes the form

$$u_{tt} = c(x)^2 \, \Delta u, \quad x \in [-1,1]^2, 0 \leq t < T. \tag{3.1}$$

In the following, Δ is the Laplacian for the spatial dimensions x_1 and x_2, i.e., $\Delta u := \frac{\partial u^2}{\partial x_1^2} + \frac{\partial u^2}{\partial x_2^2}$. $c(x) \in \mathbb{R}^2$ defines the speed of wave propagation and is determined by the medium. Two initial conditions $u(x,0) = u_0(x)$ and $u_t(x,0) = p_0(x)$ are usually required for an initial value problem to have a unique solution, as the equation is of second order with respect to time.

3.1.1 Wave Representation

Energy Semi-Norm. Given the linearity of the system, the energy semi-norm can be used to compare wave fields:

$$E[\mathfrak{u}] := \frac{1}{2} \int_{[-1,1]^2} |\nabla u|^2 + c^{-2} |u_t|^2 \, dx. \tag{3.2}$$

With respect to this semi-norm, the wave equation is well-posed and wave propagation is stable concerning fluctuations in the wave field [15]. Figure 3.1 illustrates an exemplary

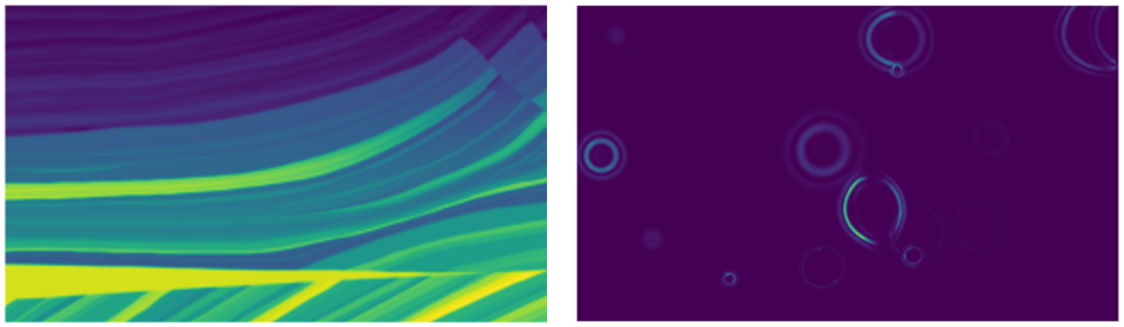

Figure 3.1: Wave propagation of acoustic waves in a medium in energy semi-norm. A numerical algorithm is deployed to solve the wave equation. The left image shows rock strata. Different colors represent different velocities; the brighter the pixels, the faster the waves propagate. The right image shows how waves propagate in this medium in two dimensions, while colors represent intensity measured in decibels. The waves are initialized by a Gaussian pulse (see Eq. (4.1)) and changed to the energy semi-norm representation (see Eq. (3.2)).

signal (right image) propagated through a medium (left image).

We propose that numerical dispersion errors can be reduced through convolutions of the wave energy components $(\nabla u, c^{-1}u_t)$ of the given wave field $\mathfrak{u} := (u, u_t)$. The mapping of physical components into their corresponding energy components is defined as

$$\Lambda : (u, u_t) \mapsto (\nabla u, c^{-1}u_t), \tag{3.3}$$

with $\Lambda^\dagger$ as the pseudo-inverse (cf. Section 1.3 in [1]). In [16, 17, 18], the authors demonstrate that utilizing the energy form for seismic imaging methods can enhance results compared to standard optimization measures. Furthermore, the use of energy components has been shown to improve convergence when training neural networks (see e.g. [1]). The mathematical methodology is explained in Appendix 1 in [1]. Chapter 4 provides a schema visualizing the wave argument transitions.

Discretized Energy Semi-Norm. The discretized energy semi-norm on the spatial grid $h\mathbb{Z}^2 \cap [-1,1]^2$, with $h = 1/(n+1)$ for $n \in \mathbb{N}$, is given by

$$E_h[\mathfrak{u}] = \|\mathfrak{u}\|_{E_h} := \sum_{x_{i,j} \in h\mathbb{Z}^2 \cap [-1,1]^2} (\|\nabla_h u(x_{i,j})\|_2^2 + |c^{-1}(x_{i,j})\partial_t u(x_{i,j})|^2)h^2. \tag{3.4}$$

We calculate the discrete derivatives, $\nabla_h u(x_{i,j})$ and $\partial_t u(x_{i,j})$, at interior points using second-order central differencing and employ first-order one-sided differences at the boundary. Appendix C provides a description of central difference approximation.

3.1.2 Boundary Conditions

In numerical simulations, boundary conditions are imposed to define the behavior of the solution at the boundaries of the computational domain. Figure 3.2 illustrates two types of boundary conditions for the acoustic wave equation that are relevant in this thesis.

Periodic Boundary Condition

The periodic boundary condition enforces standard periodicity and is deployed in all implementations of [1]. In the 2D case, the solution repeats itself periodically in both directions with the same velocity,

$$\begin{aligned} & u((-1,k),t) = u((1,k),t) \\ \text{and} \quad & u((k,-1),t) = u((k,1),t),\ \forall k \in [-1,1),\ 0 \le t < T. \end{aligned} \tag{3.5}$$

While this boundary condition makes computation simpler, it does not accurately represent the physical world. However, since waves travel at finite speed, periodic boundary conditions will not affect the propagation of waves in the interior of the domain before they reach the boundary.

Absorbing Boundary Conditions

Absorbing boundary conditions are designed to simulate an infinite computational domain by absorbing outgoing waves at the boundaries. While a perfectly matched layer [19] is an absorbing region and significantly reduces reflections more than comparable methods, we chose absorbing boundary conditions for simplicity. In 1977, Engquist and Majda [20] presented one way of enforcing absorbing boundary conditions for the 2D wave equation:

For simplicity, let $c(x) = 1$ and we only consider the half-space $x_i \ge 0\ \forall i \in \{1, 2\}$. We impose absorbing boundary conditions at points $x_1 = 0$ for solutions of the wave equation $u = \frac{\partial u^2}{\partial t^2} - \frac{\partial u^2}{\partial x_1^2} - \frac{\partial u^2}{\partial x_2^2}$. Waves advancing to the left, i.e., towards $x_1 = 0$, take the form

$$u(x_1, x_2, t) = e^{i(\sqrt{\xi^2-\omega^2}x_1+\xi t+\omega x_2)} \tag{3.6}$$

assuming that $\xi^2 - w^2 > 0$, where $\xi > 0$ stands for frequency, and $\omega/t = \sin\theta$ with θ representing the angle of incidence of the wave upon the boundary $x_1 = 0$.

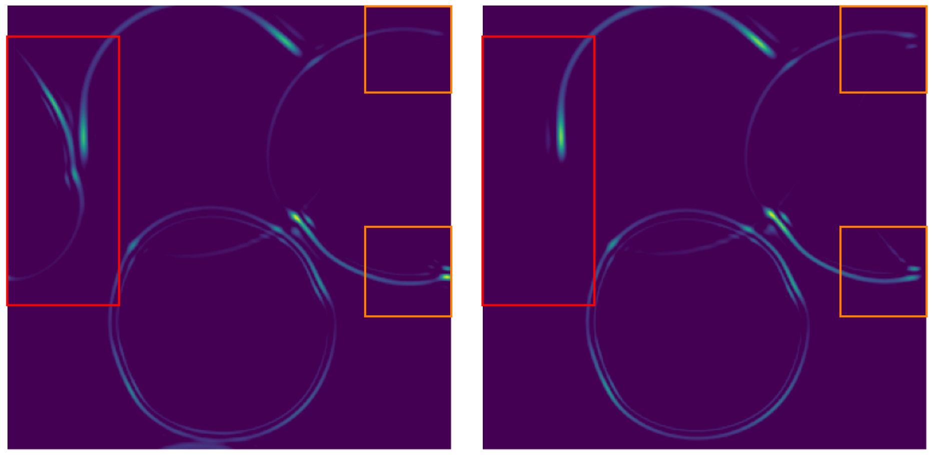
Figure 3.2: Difference in energy semi-norm when using different boundary conditions. We can see that for the same initial condition and medium, periodic boundaries (left image) re-introduce waves that hit the boundary on the opposite side, while absorbing boundaries (right image) do not further propagate these waves (see red rectangles). A crop of the medium in Figure 3.1 and the same numerical algorithm are used to solve the wave equation. Note that numerical wave field integrations with absorbing boundary conditions will cause small but significant reflections at the boundary (see orange rectangles). This can be observed on the right side of the right image and is caused by inaccuracies in the algorithm.

For constant (ω, ξ),

$$(\frac{d}{dx_1} - i\sqrt{\xi^2 - \omega^2})\ u\ \Big|_{x_1=0} = 0 \tag{3.7}$$

is the first-order boundary condition that annihilates left-moving waves. Summing the displacements, we get

$$u(x_1, x_2, t) = \int \int_{\sqrt{\xi^2-\omega^2}>0} e^{i(\ \sqrt{\xi^2-\omega^2}x_1+\xi t+\omega x_2\ )}\ \widehat{u}(0, \xi, \omega)\ \mathrm{d}\xi\ \mathrm{d}\omega, \tag{3.8}$$

which is a more general wave package advancing to the left, where $\widehat{u}$ represents the Fourier transform (cf. Section A) in (t, x_2). In the physical space, the above corresponds to

$$(\frac{d}{dx_1} - \sqrt{\frac{\partial^2}{\partial t^2} - \frac{\partial^2}{\partial x_2^2}})\ u\ \Big|_{x_1=0} = 0. \tag{3.9}$$

Under Fourier transform, it follows

$$i\xi \leftrightarrow \frac{\partial}{\partial t} \quad \text{and} \quad i\omega \leftrightarrow \frac{\partial}{\partial x_2} \tag{3.10}$$

for the theoretical non-local boundary condition seen above. By writing $i\sqrt{\xi^2 - \omega^2}$ in the form $i\xi\sqrt{1 - (\omega^2/\xi^2)}$, substituting $x = \omega/\xi$, multiplying by powers of $i\xi$ and applying

Eq. (3.10), we can approximate $\sqrt{1+x}$ using the second Taylor expansion (cf. Section B) for $x = 0$:

$$u_{x_1 t} - u_{tt} + \frac{1}{2} u_{x_2 x_2} \Big|_{x_1=0} = 0. \tag{3.11}$$

It is shown that a higher-order Taylor expansion leads to an ill-posed problem, which is why Engquist and Majda apply the Padé approximation [21]:

$$\sqrt{1+x} = 1 + \frac{x}{2 + x/2} + \mathcal{O}(|x|^3). \tag{3.12}$$

This leads to the third-order boundary condition that produces a zero reflection coefficient

$$u_{x_1 tt} - u_{ttt} - \frac{1}{4} u_{x_1 x_2 x_2} + \frac{3}{4} u_{t x_2 x_2} \Big|_{x_1=0} = 0. \tag{3.13}$$

Nevertheless, when the incident wave arriving at the boundary is not orthogonal to the boundary, then these absorbing boundary conditions do not completely eliminate reflection. Experiments in [20] show that for waves with a 45° angle of incidence, Eq. (3.11) reflects approximately 3% of the amplitude of the incident wave, while Eq. (3.13) causes reflections of 0.5%. Higher-order boundary conditions produce less reflection but make the computations significantly more expensive.

3.1.3 Approximation of Wave Propagators

Following the setup of Section 1.1 and 1.3 in [1], we describe an efficient, time-reversible $\mathcal{G}_{\Delta t^\star}$ and an accurate $\mathcal{F}_{\Delta t^\star}$ to numerically advance 2D waves through a medium $c(x) \in \mathbb{R}^2$. $\mathcal{G}_{\Delta t^\star}$ operates on a coarser grid than $\mathcal{F}_{\Delta t^\star}$ and is therefore computationally cheaper, but less accurate. They both propagate the given wave field $\mathfrak{u} \equiv \mathfrak{u}(x,t) \equiv (u, u_t)$ from t to $t + \Delta t^\star$, while $\mathcal{G}_{\Delta t^\star}$ uses fewer steps than $\mathcal{F}_{\Delta t^\star}$. The choice of grid spacing, time step, and discretization method directly impacts the solver's convergence, accuracy, and computational efficiency.

To define the solvers, we need to discretize the spatial and temporal domains. Let $Q_h u$ denote a numerical approximation of Δu, i.e.,

$$\partial_{tt} u(x,t) \approx c^2(x) Q_h u(x,t). \tag{3.14}$$

The approximation $(u, u_t)_t \approx (u_t, c^2 Q_h u)$ can be solved by a time integrator. With the spatial $(\Delta x, \delta x)$ and temporal spacing $(\Delta t, \delta t)$ on uniform Cartesian grids, we denote

- $\mathcal{G}_{\Delta t^\star} := (\mathcal{S}^{Q_h}_{\Delta x,\Delta t})^M$ with $\Delta t^\star = M\Delta t$, which operates on the lower resolution grid, $\Delta x\mathbb{Z}^2 \times \Delta t\mathbb{Z}^+$. Q_h is characterized using a central differencing scheme of second order for computing partial derivatives.
- $\mathcal{F}_{\Delta t^\star} := (\mathcal{S}^{Q_h}_{\delta x,\delta t})^m$ with $\Delta t^\star = m\delta t$, which operates on the higher resolution grid, $\delta x\mathbb{Z}^2 \times \delta t\mathbb{Z}^+$. Q_h is either a central differencing technique of second order or a spectral method to approximate Δu. We assume that $\mathcal{F}_{\Delta t^\star}$ is sufficiently accurate for the wave speed.

These two propagators take $\mathfrak{u}(x,t)$, with x defined on different Cartesian grids, and return an approximation of $\mathfrak{u}(x, t+\Delta t^\star)$ on the respective Cartesian grids. As the two solvers operate on different Cartesian grids with $\delta x < \Delta x$ and $\delta t < \Delta t$, we define extension using the restriction operator $\mathcal{R}$, which transforms functions from a fine grid $\delta x\mathbb{Z}^2 \times \delta t\mathbb{Z}^+$ to a coarse grid $\Delta x\mathbb{Z}^2 \;\times\; \delta t\mathbb{Z}^+$, while the prolongation operator $\mathcal{I}$ maps the inverse relation.

3.1.4 The Enhanced Propagators

Solutions of the wave propagators are either used to train the end-to-end model, or are integrated with deep learning components to offer stable wave propagation across diverse media with multiscale speeds. The enhanced model variants consist of:

(a) bilinear interpolations denoted as $\mathcal{R}^0$ and $\mathcal{I}^0$. Note that $\mathcal{I}^0\mathcal{R}^0\mathfrak{u} \neq \mathfrak{u}$, which is part of the error to be corrected with a more sophisticated approach. Appendix Section D contains a description of the bilinear interpolation algorithm.

(b) neural network components denoted as $\mathcal{R}^\theta \equiv \Lambda^\dagger\mathcal{R}^\theta_{\Delta t^\star}\Lambda$ and $\mathcal{I}^\theta \equiv \Lambda^\dagger\mathcal{I}^\theta_{\Delta t^\star}\Lambda$, while the lower index indicates that the neural networks are trained on the step size $\Delta t^\star$ (cf. Chapter 2 in [1]). For improved neural network inference, we use the transition operator Λ (cf. Eq. (3.3)) to transform physical wave fields (u, u_t) to energy component representations $(\nabla u, c^{-2}u_t)$. Section 3.2 serves as an overview of the deep learning methods used for the components above.

By reorganizing Eq. (1.2), we obtain the functions that the deep learning components seek to approximate:

$$\begin{aligned}
\Psi^{\theta,0}_{\Delta t^\star}[\mathfrak{u}, c] &:= \mathcal{I}^\theta\mathcal{G}_{\Delta t^\star}\mathcal{R}^0[\mathfrak{u}, c] \approx \mathcal{F}_{\Delta t^\star}\mathfrak{u} \\
\Psi^{\theta_1,\theta_2}_{\Delta t^\star}[\mathfrak{u}, c] &:= \mathcal{I}^{\theta_1}\mathcal{G}_{\Delta t^\star}\mathcal{R}^{\theta_2}[\mathfrak{u}, c] \approx \mathcal{F}_{\Delta t^\star}\mathfrak{u}.
\end{aligned} \tag{3.15}$$

In other words, $\mathcal{I}^\theta$ corrects the wave field computed by $\mathcal{G}_{\Delta t^\star}$. For $\Psi^{\theta_1,\theta_2}_{\Delta t^\star}$, the trainable parameters θ_1 and θ_2 are optimized together.

The model's performance depends on how the PDE is discretized, what numerical parameters are used, what numerical algorithm is applied for $\mathcal{G}_{\Delta t^\star}$ and $\mathcal{F}_{\Delta t^\star}$, and how we sample from the wave solution manifold $\mathcal{M}$ (cf. Eq. (1.3)). In the next section, we will explain the utilized numerical methods in our setup.

3.1.5 Traditional Numerical Schemes for the Coarse Solver ($\mathcal{G}_{\Delta t}$) and Fine Solver ($\mathcal{F}_{\Delta t}$)

Through discretizing both the spatial and time interval of the PDE, we can approximate the positions and velocities of particles at successive discrete points at finite number of steps. This transforms the PDE into a system of linear equations, which can then be solved numerically using difference equations. Since our problem involves complex manifolds and irregular shapes, this method allows for the subdivision of these domains into smaller and simpler parts, making it easier to solve the PDEs in these regions.

Velocity Verlet in Time and Central Difference in Space ($\mathcal{G}_{\Delta t}$)

One robust and popular algorithm for molecular dynamics simulations is attributed to Verlet [22]. To simplify the notation, we define the velocity $\frac{\partial \tilde{u}}{\partial t} = \tilde{v}$ and the acceleration $\frac{\partial \tilde{v}}{\partial t} = \tilde{a}$ of the wave $\tilde{u}$. Many finite-difference time-domain integration algorithms can be understood by expanding $\tilde{u}_{n+1} \coloneqq \tilde{u}(t_n + \Delta t)$ in a Taylor series up to the second order:

$$\tilde{u}_{n+1} = \tilde{u}_n + \tilde{v}_n \Delta t + \frac{\Delta t^2}{2} \tilde{a}_n. \tag{3.16}$$

Next, we define the velocity at the next time step by the central difference formula as

$$\tilde{v}_{n+1} = \frac{\tilde{u}_{n+2} - \tilde{u}_n}{2\Delta t}. \tag{3.17}$$

Applying the above equations twice, we obtain

$$\tilde{v}_{n+1} = \tilde{v}_n + \frac{\tilde{a}_{n+1} + \tilde{a}_n}{2\Delta t}. \tag{3.18}$$

Combining Eq. (3.18) and Eq. (3.16), we derive the fundamental velocity Verlet scheme implemented to numerically solve the wave equation. In our context, the velocity form of the Verlet time integrator can then be written as

$$\begin{aligned} \tilde{u}(x, t + \Delta t) &= \tilde{u}(x, t) + \Delta t\ \tilde{v}(x, t) + \Delta t \lambda c^2(x)\ Q_{\Delta x} \tilde{u}(x, t) \\ \tilde{v}(x, t + \Delta t) &= \tilde{v}(x, t) + \lambda [Q_{\Delta x} \tilde{u}(x, t) + Q_{\Delta x} \tilde{u}(x, t + \Delta t)], \end{aligned} \tag{3.19}$$

with $\lambda = \frac{\Delta t}{2\Delta x^2}$, while $\tilde{v}$ approximates $\tilde{u}_t$. While this scheme is computationally efficient, symplectic, and time-reversible, it is less accurate than other, more expensive approaches. The total accumulated error has an order of magnitude of $\mathcal{O}((\Delta t)^2) + \mathcal{O}((\Delta x)^2)$. To achieve a specific level of accuracy with small numerical dispersion errros, the velocity Verlet method may require a much smaller step size.

RK4 Pseudo-Spectral Method ($\mathcal{F}_{\Delta t}$)

To apply the pseudo-spectral method for spatial discretization, we use a Runge-Kutta method as an iterative scheme that showed good result in related work [1]. The Runge-Kutta method was first developed in 1900 by Carl Runge and Wilhelm Kutta [23, 24]. The scheme used in this thesis is as follows: While higher-order Runge-Kutta methods can provide better solutions, the Runge-Kutta of forth-order method (RK4) method is primarily used in $\mathcal{F}_{\Delta t}$ to trade-off accuracy, computational complexity, and convergence. By denoting the velocity as $\tilde{u}_t \coloneqq \tilde{v}$, RK4 is defined using the following iterative formula:

$$\begin{aligned}
k_1^{\tilde{u}} &= \tilde{v}_n \\
k_1^{\tilde{v}} &= c^2(x)\ \Delta_{\delta x}\tilde{u}_n \\
\\
k_2^{\tilde{u}} &= \tilde{v}_n + \frac{\delta x}{2}k_1^{\tilde{v}} \\
k_2^{\tilde{v}} &= c^2(x)\ \Delta_{\delta x}(\tilde{u}_n + \frac{\delta x}{2}k_1^{\tilde{u}}) \\
\\
k_3^{\tilde{u}} &= \tilde{v}_n + \frac{\delta x}{2}k_2^{\tilde{v}} \\
k_3^{\tilde{v}} &= c^2(x)\ \Delta_{\delta x}(\tilde{u}_n + \frac{\delta x}{2}k_2^{\tilde{u}}) \\
\\
k_4^{\tilde{u}} &= \tilde{v}_n + k_3^{\tilde{v}} \\
k_4^{\tilde{v}} &= c^2(x)\ \Delta_{\delta x}(\tilde{u}_n + k_3^{\tilde{u}})
\end{aligned} \tag{3.20}$$

and

$$\begin{aligned}
\tilde{u}_{n+1} &= \tilde{u}_n + \frac{1}{6}(k_1^{\tilde{u}} + 2k_2^{\tilde{u}} + 2k_3^{\tilde{u}} + k_4^{\tilde{u}})\ \delta t \\
\tilde{v}_{n+1} &= \tilde{v}_n + \frac{1}{6}(k_1^{\tilde{v}} + 2k_2^{\tilde{v}} + 2k_3^{\tilde{v}} + k_4^{\tilde{v}})\ \delta t
\end{aligned} \tag{3.21}$$

We efficiently evaluate the second-order spatial derivatives $\Delta_{\delta x}$ via the Fourier pseudo-spectral method:

(i) We first compute the 2D Fourier transform of the displacement function $\tilde{u}$, which converts the function from its spatial domain representation to its frequency domain equivalent. This transformation is accomplished using the Fast Fourier Transform (FFT) algorithm, as described in Appendix A. The advantage of this conversion is that differentiation can be performed more efficiently in the frequency domain.

(ii) We then multiply the result by the square of the wavenumber. A wavenumber k represents the spatial frequency of the function in the x_1 and x_2 directions; it is given by $k = \frac{2\pi}{n \cdot \delta x}$, where n denotes the number of grid points in the domain. This operation computes the second-order spatial derivatives, as differentiation in the frequency domain corresponds to multiplication by the wavenumber.

(iii) Lastly, we compute the inverse 2D Fourier transform to obtain the second-order spatial derivatives in the physical domain.

This scheme is only suitable for PDEs with periodic boundary conditions. To overcome this issue, we first apply $\mathcal{F}_{\Delta t}$ to a larger domain and then crop the image to simulate an infinite computational domain. Since solutions of $\mathcal{F}_{\Delta t}$ are only used for creating the training data, increasing the domain size does not affect speed. Second, computations are not symplectic, which means that the energy is not conserved over long time intervals. Consequently, in problems where long-term energy conservation is crucial, the velocity Verlet method may be a more suitable choice.

Stability Conditions

Numerically simulating waves, especially on discontinuous media with a high rate of change, presents a challenging task. The convergence and results of both wave propagation schemes depend on the composition of spatial and temporal parameters. The parameters from $\mathcal{G}_{\Delta t}$ serve to illustrate the condition, but the same principle applies to $\mathcal{F}_{\Delta t}$. Choosing overly large parameters leads to the accumulation of errors and can cause rapid oscillations in the simulation.

A necessary condition for convergence is that the parameters are selected according to the stability criterion outlined by Courant, Friedrichs, and Lewy (CFL) [25]: The domain of dependence of the PDE must lie within the domain of dependence of the numerical method. This is because correct solutions cannot be computed if the information that determines the solution is not accessible.

For explicit schemes addressing hyperbolic problems, the CFL condition constrains the time step $c\Delta t$ not to exceed one spatial step Δx. This condition requires a numerical scheme to be aligned with the wave speed; the easiest way to ensure adherence to the

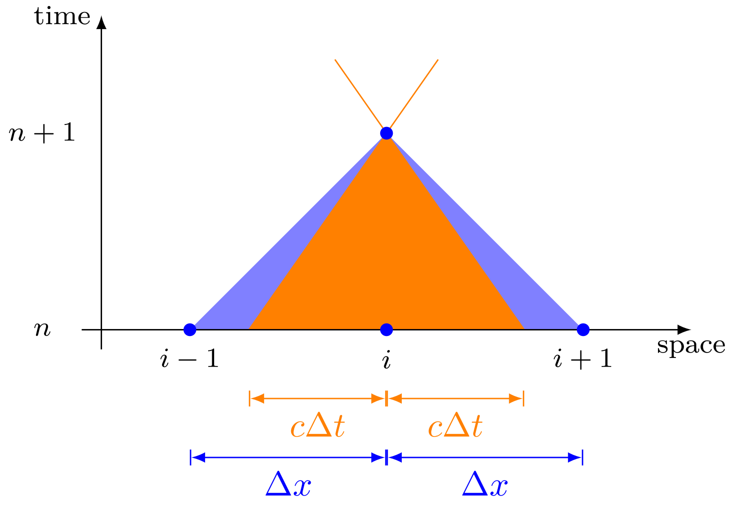

Figure 3.3: Physical interpretation of the CFL condition for the 1D wave equation. The orange region represents the true solution to the continuous 1D PDE, while the orange lines are defined by $\Delta x/\Delta t = c$ and $\Delta x/\Delta t = -c$. The blue region shows the domain of dependence of the numerical method, e.g., the velocity Verlet algorithm from Eq. (3.19). Since the set of points that influence the solution $x(i, n+1)$ includes all the physical information from the previous time step, the parameters fulfill the CFL condition.

CFL condition is to universally use the fastest wave speed and the smallest time step. In other words, the numerical wave speed $\Delta x/\Delta t$ must be at least as fast as the physical wave speed c. This ansatz is expressed using the Courant number $\mathcal{C}$:

$$\frac{c\Delta t}{\Delta x} = \mathcal{C} \leq \mathcal{C}_{max} = 1. \tag{3.22}$$

Note that the CFL criterion is necessary, but not sufficient. Other methodologies, such as the Von Neumann analysis [26], may yield a more comprehensive understanding of stability in the explicit scheme. Figure 3.3 visualizes the domains of dependence in one dimension.

3.2 Introduction to Deep Learning

Among various machine learning paradigms, neural networks are especially contributing to the state-of-the-art for predicting wave propagation as emphasized in [27, 28, 29]. Neural networks consist of interconnected layers of artificial neurons, which are computational units that can learn to approximate complex functions and relationships from training data. The central relationship we aim to establish is between the input features $(u_n, (u_t)_n, c(x))$

and the target output $(u_{n+1}, (u_t)_{n+1})$. Note that our setup is inspired by Section 1.1 in [1], but diverges in employing a different training algorithm and dataset, as detailed in Chapter 4. Most of the formulas below are taken from [30, 31, 32, 33].

3.2.1 Supervised Learning of Wave Propagators

The paradigm in machine learning, where the model learns a relationship between inputs and labeled output targets, is called supervised learning. In this framework, a dataset composed of pairs of inputs and corresponding outputs is used to guide the learning process. The primary objective of supervised learning is to construct a model that can generalize and make accurate predictions for unseen data, leveraging patterns inherent in the training data.

In the context of this thesis, one data point corresponds to an object $x = (\nabla u_n, c^{-2}(u_n)_t, c) \in \mathbb{R}^{(h\times w\times 4)}$ with h (height) and w (width) representing the dimensions of the image. The label for this wave propagation problem corresponding to an object x is then given by $y = (\nabla u_{n+1}, c^{-2}(u_{n+1})_t) \in \mathbb{R}^{(h\times w\times 3)}$. Given a set of N paired training points $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, we attempt to learn a function $\tilde{f} : \mathbb{R}^{(h\times w\times 4)} \to \mathbb{R}^{(h\times w\times 3)}$ that meets the objective prescribed in Eq. (1.2) for unseen test data. Consequently, this can be seen as approximating an unknown function $f : x \mapsto y$ with the objective $\tilde{f}(x) \approx y = f(x)$, i.e., learn nontrivial alignments between x and y, while (x, y) belong to a certain distribution. In Subsection 4.3.1, we present different variants of $\tilde{f}$ in our setup, also denoted as $\Psi^{\theta}_{\Delta t^\star}$. This model consists of one or two neural networks, denoted by θ. The following section serves as an introduction to feed-forward networks, usually referred to as ‘vanilla’ neural networks.

3.2.2 Feed-Forward Neural Networks

Feed-forward neural networks are a type of artificial neural network architecture that consists of multiple layers of neurons organized as seen in Figure 3.4. In mathematical terms, layers are structured as an ordered arrangement of alternating linear and nonlinear operations, with each operation being characterized by a set of adjustable, learnable parameters. We flatten the multi-dimensional input variable x to a vector $\tilde{x}$ to obtain the first hidden layer $A^{(1)}$,

$$A^{(1)} = \sigma(W^{(0)}\tilde{x} + B^{(0)}). \tag{3.23}$$

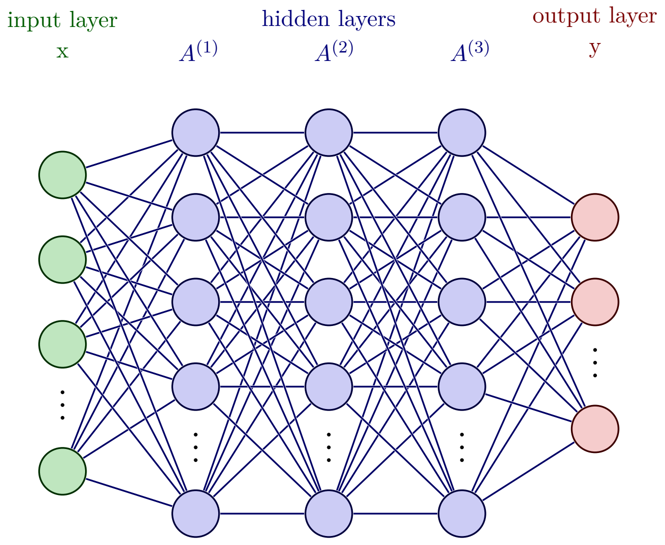

Figure 3.4: Sample feed-forward neural network architecture. Neurons are represented by circles and each neuron in the previous layer is fully pairwise connected with all neurons in the adjacent layer. I.e., it is called a fully connected network. Neurons within a single layer share no connections. This drawing is a modification of [34].

The utilized parameters are:

Weights. $w_j^{(i)} \in W^{(i)}$ represents a single weight at position j of the weight matrix $W^{(i)}$ in the i-th layer. For example, $W^{(0)}$ are the weights from the input layer to the first hidden layer. We chose to randomly initialize the weights according to the Kaiming Uniform Initialization [35]

$$w \sim \mathcal{U}(-\sqrt{(1/n)}, \sqrt{(1/n)}), \tag{3.24}$$

where n is the number of input units in the layer and $\mathcal{U}$ is the uniform distribution. This choice is motivated by the findings in [36]: To avoid that the magnitudes of input signals are reduced or increase exponentially, the weights are initialized in a way that the variance of the outputs from each neuron remains the same throughout the network layers during the forward pass.

Biases. To help the neural network better fit the data, the bias B is an additional parameter that is added to each layer to shift the output by a scalar. The biases $B^{(i)}$ are initialized either as zero (for the last layer) or according to Eq. (3.24) (for all other layers).

Activation. The activation function σ is applied to the sum of weighted inputs and biases. This plays a critical role in the ability of neural networks to learn complex, non-linear decision boundaries. The activation function predominantly used in our setup is the Rectified Linear Unit (ReLU) function $\sigma(x) = \max(0, x)$; it sets all negative input values to zero, while positive input values remain unchanged. We chose the ReLU activation function for all of our layers due to its simplicity, computational efficiency, and success in related work [37]. Note that, unlike other layers in regression tasks, the output layer neurons most commonly do not have an activation function to enable negative values.

Finally, we characterize the trainable weights and biases θ of the learning function f_θ related to the network components $\mathcal{I}^\theta$ and $\mathcal{R}^\theta$, as

$$\theta \coloneqq \{W^{(0)}, B^{(0)}, \dots, W^{(N-1)}, B^{(N-1)}\}. \tag{3.25}$$

where N is the number of hidden layers in the neural network. Training aims to adjust the parameters θ iteratively using an optimization algorithm in such a way that the output becomes increasingly accurate.

3.2.3 Training

Loss Function

Mathematically, the learning process involves minimizing a loss function l, which measures the difference between the predicted outputs $f_\theta(x_i)$ and the true outputs y_i for all the examples in the training dataset $\mathcal{D}$:

$$\min_{\theta \in \mathbb{R}^{m \times n}} \mathcal{L}(f_\theta; \mathcal{D}) = \sum_i l(f_\theta(x_i), y_i), \ \forall i \in \{1, \dots, |\mathcal{D}|\}. \tag{3.26}$$

Here, $l(y_i, f_\theta(x_i))$ represents the loss incurred for a single input-output pair (x_i, y_i). The intention is to find the function f_θ that minimizes the overall loss $\mathcal{L}(f_\theta; \mathcal{D})$.

The choice of loss function depends on the problem domain and the desired properties of the trained system. We chose the L2 loss, which is one of the most common loss functions used in supervised learning. It is defined as:

$$\min_{\theta \in \mathbb{R}^{m \times n}} \mathcal{L}(f_\theta; \mathcal{D}) = \sum_i (f_\theta(x_i) - y_i)^2, \ \forall i \in \{1, \dots, |\mathcal{D}|\}. \tag{3.27}$$

Due to the squared term, the L2 loss is more sensitive to outliers than other loss functions. This means that the performance will be strongly influenced by large errors. The loss function is convex with the premise that $\theta^\star \coloneqq \arg\min_\theta \mathcal{L}(\theta; \{(x_i, y_i)\}_{i=1}^N)$, which means that it has a unique global minimum. This property simplifies optimization, as gradient-based methods can be applied to find the optimal solution without getting trapped in local minima. Subsection 3.2.4 details the refinement of the standard L2 loss function (Eq. (3.27)).

Gradient-based Optimization

The parameters θ are optimized by calculating the gradient of the loss function $\nabla_\theta \mathcal{L}$. This gradient is then used to minimize the loss function by adjusting the model's parameters in the direction of the negative gradient of the loss function. A popular choice for training neural networks is to feed data in batches, allowing parallel computations and averaging of the gradients. For each epoch, one shuffles the data and divides the training set into smaller subsets (also called 'mini-batches') with a batch size m, each containing m data samples. For each mini-batch, the gradient of the loss function $J(\theta)$ are computed with respect to the parameters for the mini-batch:

$$\nabla J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \mathcal{L}(\theta; (x_i, y_i)) \tag{3.28}$$

According to the learning rate α, each iteration of gradient descent updates can be written as

$$\theta^{t+1} = \theta^t - \alpha \nabla J(\theta) \tag{3.29}$$

where t denotes the iteration in gradient descent [38]. A common variation of the mini-batch gradient descent optimization algorithm is Adam [39] which is predominant in various domains [40].

$\frac{\partial \mathcal{L}}{\partial \theta}$ is computed by automatic differentiation and 'backpropagation'. In detail, the fine-tuning is done for each weight $\partial w_{ij}^{(k)}$ and bias, simplified to $\partial w_{0i}^{(k)}$, which is connecting neuron i in layer $(k-1)$ to neuron j in layer k. By applying the chain rule and using the activation of a neuron $a_i^k \coloneqq \sum_j w_{ij}^k a_{i-1}$, we define

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \frac{\partial \mathcal{L}}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k}. \tag{3.30}$$

The impact of the weight on the activation is denoted as the output o_i^{k-1} of node i in the

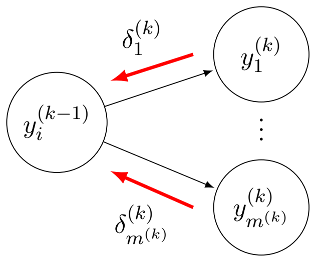

Figure 3.5: Backpropagation of errors. Here, the red arrows show the backpropagation graph. δ_k^i represents errors that are propagated back from layer (k) to layer $(k-1)$. $m^{(k)}$ corresponds to the amount of nodes y_i^k in layer (k). This illustration is an adaptation of [42].

previous layer $k-1$,

$$\delta_j^k = \frac{\partial \mathcal{L}}{\partial a_j^k}, \qquad \frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}(\sum_r w_{rj}^k o_r^{k-1}) = o_i^{k-1}. \tag{3.31}$$

Thus, the partial derivative of the loss function can be written as

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}. \tag{3.32}$$

As depicted in Figure 3.5, we then iteratively compute the gradient of the loss function and move backward from the output layer to the input layer through the computation graph. It is important to note that computations of the error term δ_j^k depend on the chosen activation and loss function. For more detailed information, please refer to the work by Hecht-Nielsen [41].

We adjust the parameters θ by iterating over all training points in $\mathcal{D}$ multiple times. An epoch refers to a complete pass through the entire dataset. For each epoch, the data points are usually shuffled and grouped to create mini-batches. This sampling strategy, known as mini-batch gradient descent, has shown good results [40] because it leverages parallelism in modern hardware and reduces the variance of the gradient.

3.2.4 (Weighted) Multi-Step Training

During evaluation, the end-to-end model $\Psi_{\Delta t}^{\theta}$ is applied multiple times to advance waves over the duration Δt. It is natural to leverage time-dependent features also during training. This is why we introduce a multi-step training strategy, in which the algorithm is applied to itself iteratively to learn time-dependent dynamics inherent in the data. The novel technique modifies Eq. (1.2) and changes the wave solution manifold $\mathcal{M}$ (cf. Eq. (1.3)). For k time steps, we can denote:

$$\mathfrak{u}_{n+k} := u(t_n + k\Delta t) = (F_{\Delta t})^k \mathfrak{u}_n \approx (\Psi^{\theta}_{\Delta t})^k \mathfrak{u}_n. \tag{3.33}$$

Similar to the mini-batch approach in Subsection 3.2.3, we then randomly join these sequences to form batches. The losses for the samples within the batch are summed across the batch to form a single loss value. By computing the gradient with respect to consecutive losses, the gradient flows through the entire computation graph across multiple time steps. This encourages the system to learn longer-term dependencies and capture more complex temporal patterns. The enhanced dataset is denoted as $\mathcal{D}^m$.

A schematic of the multi-step data generation process can be seen in Figure 3.6. For each initial condition $\mathfrak{u}_0$, $\mathcal{F}_{\Delta t}$ is applied N times with solutions denoted as $\mathfrak{u}_n, \forall n \in U_1 := \{0, \dots, N\}$. In a random order, the end-to-end model $\Psi^{\theta}_{\Delta t} \in \{\Psi^{\theta,0}_{\Delta t}, \Psi^{\theta_1,\theta_2}_{\Delta t}\}$ is applied to every $\mathfrak{u}_n$ for a random amount of steps $k \in U_2 := \{1, \dots, N-n\}$. Formally, the optimization problem can therefore be described as:

$$\min_{\theta \in \mathbb{R}^{m \times n}} \mathcal{L}^m(\Psi^{\theta}_{\Delta t}; \mathcal{D}^m) = \min_{\theta \in \mathbb{R}^{m \times n}} \frac{1}{|\mathcal{D}^m|} \sum_{\mathfrak{u}_0} \sum_{n \sim \mathcal{U}_1 \setminus \{N\}} \sum_{\substack{k \sim \mathcal{U}_2 \\ n < k \leq N-n}} \|(\Psi^{\theta}_{\Delta t})^k \mathfrak{u}_n - (\mathcal{F}_{\Delta t})^k \mathfrak{u}_n\|^2_{E_h}. \tag{3.34}$$

We draw both n and k from the uniform random distributions U_1 and U_2, respectively.

Weighted Approach

Building upon the multi-step training strategy, we explore alternative sampling strategies for the variables n and k. The central idea is to stabilize training and accelerate convergence by weighting individual losses: For optimal results, the training horizon must be sufficiently long such that propagation sequences encompass an adequate number of patterns. At the same time, wave trajectories during training should exhibit a certain degree of similarity to allow for more efficient pattern recognition. However, in the model's initial, untrained phase, feature variations can be extreme, leading to imprecise gradient estimations. Therefore, rather than drawing $k \in \mathcal{U}_2$ from a uniform distribution, we select values for k according to a truncated normal distribution. This process is motivated by other studies [43, 44] that similarly weight the loss function to achieve training improvements.

Let X be a random variable following a normal distribution with mean μ and standard deviation σ,

$$X \sim N(\mu, \sigma^2), \tag{3.35}$$

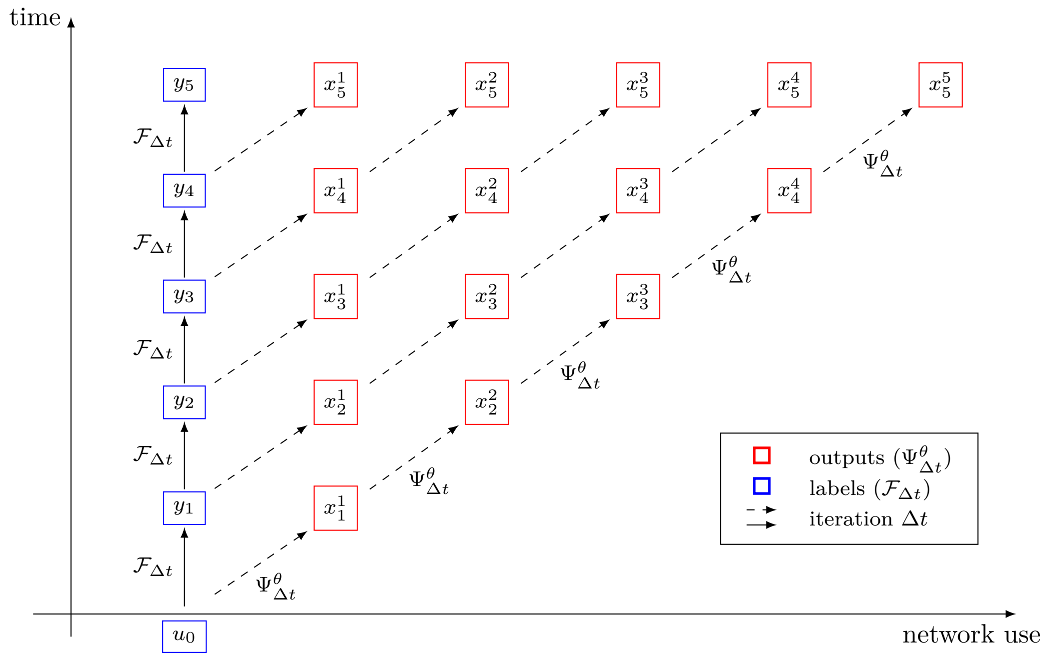
time
network use
u_0
y_1
y_2
y_3
y_4
y_5
$\mathcal{F}_{\Delta t}$
$\Psi^{\theta}_{\Delta t}$
x_1^1
x_2^1 x_2^2
x_3^1 x_3^2 x_3^3
x_4^1 x_4^2 x_4^3 x_4^4
x_5^1 x_5^2 x_5^3 x_5^4 x_5^5
outputs ($\Psi^{\theta}_{\Delta t}$)
labels ($\mathcal{F}_{\Delta t}$)
iteration Δt

Figure 3.6: Multi-step loss function with labels (on the left in blue) and end-to-end model solutions (on the right in red) as used in experiment 2. The subscript of x and y indicates the elapsed propagation time $\Delta t \cdot n$, while the superscript denotes the number of times the solvers have been applied. Note that we show only five iterations for simplicity. Our strategy randomly samples points of the red point cloud, and computes the loss of sequences with their respective blue labels, i.e., $\sum_k \|x_n^k - y_n\|$. This figure differs from Figure 4 in [1] in that the model is applied to its preceding solution for several time steps.

where X is restricted to the sample space between a and b, represented as $-\infty < a < X < b < \infty$. The truncated normal distribution, denoted as $\mathrm{TN}(\mu, \sigma, a, b)$, has the following probability density function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{F(\frac{b-\mu}{\sigma}) - F(\frac{a-\mu}{\sigma})} \tag{3.36}$$

where F is the cumulative distribution function of the standard normal distribution. After every third epoch, the mean μ is increased by one to account for longer sequences.

In the earlier epochs, the end-to-end model $\Psi^{\theta}_{\Delta t}$ is applied for fewer steps. Consequently, we prioritize learning simpler and short-term patterns in the initial stages of training, while progressively capturing more complex and long-term dependencies as the training progresses through later epochs. In the later epochs, the end-to-end model $\Psi^{\theta}_{\Delta t}$ is applied for a greater number of steps; by focusing on minimizing the impact of errors in the early stages, the system becomes more robust and capable of maintaining its performance even when confronted with challenging media. We refer to this dataset as $\mathcal{D}^{w,m}$.

3.2.5 Regularization Methods

Overfitting occurs when a machine learning model becomes too closely adapted to the training data, resulting in reduced performance and poor generalization to new, unseen data. This issue often arises when the model is overly complex, or the training data is not sufficiently diverse to capture the underlying data distribution. We use the following methods for better generalization:

Dropout. During training, individual neurons are randomly removed from the hidden layer of the neural network [45]. That is, their contributions to the output layer's activation are temporarily nullified and the remaining weights are rescaled to maintain a consistent input scale for the subsequent layer. This forces the network to rely on a diverse set of neurons for making predictions.

Batch Normalization. In 2015, Ioffe and Szegedy [46] introduced Batch Normalization. The input to each layer is normalized separately for each feature map (or channel in Subsection 3.2.7) to ensure a consistent mean (usually 0) and variance (usually 1) throughout the training process. Stabilizing input distributions allows faster convergence, and better generalization due to more accurate gradients and a lower sensitivity to learning rates.

Data Augmentation. To artificially increase the size and diversity of a training dataset, a common technique is to create new samples through various transformations of the original data [47]. This is particularly important when the dataset is rather small compared to the model's complexity. In this thesis, we chose to randomly flip the data horizontally and vertically with data being generated in real-time to reduce memory usage, increasing our dataset by four.

Adding noise to the input images is another possible data augmentation method, because it is frequently necessary to assess the networks efficacy on data points beyond the training distribution, for example due to special media. [48] offers further insights.

Weight Decay. By adding a term to the loss function that is proportional to the sum of the squared weights (also known as L2 regularization), weight decay penalizes large weights during backpropagation. The added term encourages the model to learn smaller weights, resulting in a more robust algorithm that can better generalize to new data. The standard Adam optimizer handles weight decay by incorporating it into the adaptive learning rate calculation, which can sometimes cause issues with convergence and generalization. In contrast, AdamW [49] separates the weight decay and the learning rate update, applying weight decay directly to the weights. For this reason, the AdamW optimizer is used for training each variant.

3.2.6 Approximation Theory

A key concern in modern research is to study how complex functions can be closely represented by simpler terms. One popular approach is deep learning, which refers to the process of approximating an unknown target function by utilizing historical observations. To better understand the power and limitations of neural networks, we describe the conceptual approximation capabilities of several architectures.

The most notable benchmark is the Universal Approximation Theory, introduced by A. Barron [50, 51]: A feed-forward neural network (cf. Subsection 3.2.2) with a single hidden layer and a bounded non-linear activation can mimic any function with any desired non-zero amount of error, assuming sufficient hidden units and computing resources. Similarly, Yarotsky's work investigates the theoretical basis of deep neural networks and their capacity to approximate diverse function classes [52]. Yarotsky also gives perspectives on the complexity analysis related to the activation function and the network size. E.g., he establishes lower and upper error bounds for deep ReLU networks in a Sobolev space [53] and characterizes deeper networks to be a more accurate approximator than shallow networks.

In [54, 55], Zhou et al. underscore the potential of neural networks across various applications. In particular, they provide mathematical estimates for the approximation error required for the learning theory. Besides, a theoretical correlation between the regularity (smoothness) of the kernel[1] and the logarithmic rate of convergence is established. This gives insight into the design of kernels to make the learning algorithm converge faster. [56] expands the generalization ability of deep learning methods to CNNs (cf. Subsection 3.2.7), given that the depth is large enough and computer runtime is not restricted. Compared to vanilla feed-forward networks, when dealing with large dimensional data, deep CNNs also show efficiency benefits due to convolutional operators. Based on these findings, it comes naturally to integrate CNNs in our setup.

[1]A kernel enables the application of linear classifiers to solve problems with non-linear characteristics. This is achieved by transforming non-linear data into a higher-dimensional space.

3.2.7 Convolutional Neural Networks

Unlike feed-forward network architectures, CNNs use shared weights and local connections to exploit the 2D structure of input data. Thus, the network can learn spatially invariant features more efficiently, which is crucial for handling variations in wave forms and propagation paths. The architecture significantly reduces the number of parameters, leading to a more streamlined training process and faster network performance.

In CNNs, the primary building block is the convolution layer, which applies a series of filters to the input data. Mathematically, convolution is a linear and continuous operation for $K, f : \mathbb{R}^d \to \mathbb{F}$ that is defined as:

$$K * f(x) := \int K(x - \tau) f(\tau) d\tau. \tag{3.37}$$

We assume that the Lebesgue integral exists for almost every $x \in \mathbb{R}^d$.

In 2D convolutional layers, the continuous convolution operation is replaced by a discrete counterpart, which is defined for an input matrix A with dimensions (I_1, J_1) and a kernel K with dimensions (I_2, J_2) as:

$$C(m, n) = \sum_{i<I_1} \sum_{j<J_1} A(i, j) * K(m - i, n - j), \tag{3.38}$$

while $0 \leq i < I_1 + I_2 - 1$ and $0 \leq j < J_1 + J_2 - 1$. The resulting output is known as a feature map C, which captures the local patterns found in the input data. The kernel K is a 2D matrix that transforms the input by sliding across the input matrix A, performing element-wise multiplication and summation, as illustrated in Figure 3.7. The weights $k_{ij} \in K$ from the discretized convolution corresponds to the trainable parameters of a CNN.

One major advantage is that the operation is shift-invariant, which means that the behavior remains unchanged when its input is shifted. In image processing, shift-invariance implies that a network can identify input features, regardless of their position within the image. Unlike other methods that are limited to a specific resolution after initialization, CNNs can therefore efficiently process inputs with varying resolution and dimensionality.

Furthermore, three factors significantly impact the success of training:

Kernel Size. In Figure 3.7, the size of the kernel is three, and is generally referred to as the kernel size; larger kernel sizes result in more weights to optimize. Nonetheless, because the weights are shared across all positions, the total number of parameters is

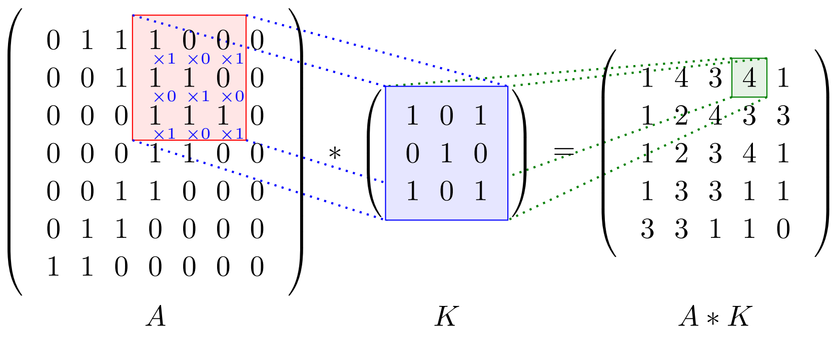

Figure 3.7: A visual representation of a convolutional layer. The kernel is placed over an element of the input image, which is then replaced by a weighted sum of itself and nearby pixels. The figure is derived from [34].

significantly lower than in other network architectures.

Padding. To handle specific boundary conditions and control the spatial resolution of the feature maps, padding adds extra pixels around the feature map before performing convolution. The most common strategy is to pad the input with rows and columns of zeros to maintain the spatial dimensions, while not influencing the values at the boundary and being computationally efficient. There are other padding methods, such as same-value padding, that are not discussed in this study.

Receptive Field. The region in the input image that affects a single neuron is defined as the receptive field. Having larger receptive fields allows the network to capture more hierarchical patterns in the input data, i.e., learn higher-order features as the input information is more widely spread across the image. This depends on the structure, kernel sizes, and depth of the network, which is explained in the next section. To ensure that the CNN can accurately approximate the solution operator of the wave equation, the product of the convolutional kernel width and the number of convolutional layers should be large enough to cover the entire domain of dependence. This ensures that the CNN can capture all relevant information and dependencies in the input data.

Structure of a CNN

Similar to feed-forward neural network architectures, CNNs consist of layers of interconnected neurons and activation functions, which perform computations on input data and propagate the results through the network. The formula for a convolutional operation transitioning from the (k-1)-th layer to the k-th layer at a single local region, combined with the ReLU activation, can be expressed as

$$a_j^k = max(0, \sum_c K_j^{(k-1)} * a_c^{k-1} + B^{(k-1)}), \tag{3.39}$$

where c and j correspond to the channel's number.

Convolutional layers process components of the input locally instead of flattening the input. These components, also called channels, refer to the specific feature map within the input data. In our case, the channels contain all relevant information about the three energy components of the wave and the medium. Therefore, an input contains four channels $x \in \mathbb{R}^{(h \times w \times x \times 4)}$, while an output contains three channels $y \in \mathbb{R}^{(h \times w \times x \times 3)}$. A filter is convolved with each channel resulting in multiple feature maps that represent patterns at different locations in the input.

The feature maps are then combined, either through summation or stacking to form the output of the convolutional layer. The method of combining output maps is dependent on how the channels are grouped. Each group of channels is processed independently using a distinct set of convolutional filters. By varying the number of filters, one can control the number of trainable parameters and, thus, the approximation properties of the model.

Resizing Layers

Another common strategy to increase the amount of parameters is to reduce the resolution and increase the channel size. This increases the receptive field, which helps the CNN to learn more contextual information and hierarchical features from the input data. The way to change the resolution of an image is by introducing other types of layers between convolutional layers:

Pooling. The two most common types are max pooling and average pooling. In max pooling, a window of a specified size is moved across the feature map, and the maximum value within the window is selected. In average pooling, the average value within the window is used instead.

Striding. By changing the number of pixels by which the kernel is moved across the feature map during convolution, striding can modify the spatial dimensions of the output. For example, a stride of one means the filter moves one pixel at a time, resulting in a densely computed feature map that retains most of the spatial information from the input. A stride of two or larger causes the filter to skip pixels, leading to reduced spatial dimensions. Most modern CNNs are built using striding instead of pooling [57]; striding is applied directly to the convolutional layer to summarize the input region deploying a trainable set of weights, whereas pooling simply selects or averages values.

Transposed Convolution. It reverses the standard convolution process by sliding the input over the kernel and performing element-wise multiplication and summation (also called deconvolution). The dimensions of the output can then be adjusted through striding and padding of the layer.

Interpolation. Another approach is to decouple the up- or downsampling process from the convolution operation. The image is first resized using an interpolation technique such as bilinear interpolation (cf. Appendix D), and then passed through a convolutional layer for feature computation.

3.2.8 Architecture Enhancements

Residual Networks

A common problem during backpropagation is that gradients can vanish or explode as they go backward through the graph. This leads to slow or oscillating updates to the weights in earlier layers, making the learning process difficult or impossible to complete. To address this issue, Kaiming He et al. [58] add 'skip connections' to the computation graph. They allow the network to learn residual functions, which are the differences between the desired output and the input. This design enables easier learning of identity mappings, preserving the information from earlier layers and making it easier for gradients to bypass less important layers. As a result, ResNets, short for Residual Networks, can be trained to much greater depths compared to traditional deep networks, without experiencing performance degradation.

We adopt the notation in Veit et al. [59] to present ResNets mathematically: Let y_{l-1} be the input of the l-th residual module, where f_l is a sequence of the neural network layer. Thus, we can recursively define

$$y_l \equiv f_l(y_{l-1}) + y_{l-1}. \tag{3.40}$$

JNet Networks

We chose to focus on the 'JNet' architecture presented in Chapter 2 of [1], which, in turn, is inspired by the fully-convolutional UNets that were originally introduced by Long et al. [60]. However, JNets increase the overall resolution rather than simply restoring it. This shift in resolution more closely resembles the letter 'J'. It primarily consists of convolutional ResNet blocks that form an autoencoder structure. In the encoding phase, the resolution is reduced using striding layers while the channel size is increased to extract the features from the image. Subsequently, a decoding phase elevates the resolution beyond its original scale using bilinear interpolation in between convolutions, while simultaneously decreasing the channel size. Mathematically, this setup is equivalent to the component $\mathcal{I}^\theta$ from Eq. (3.15).

3.3 Parareal Algorithm

The Parareal algorithm, an iterative scheme that allows time parallelization for computational time-dependent problems, was first introduced by Lion et al. in 2001 [9]. Identical to Chapter 4 in [1], our implemented scheme iteratively refines the solution using the difference between the fine solver ($\mathcal{F}_{\Delta t}$) and the coarse solver ($\mathcal{G}_{\Delta t}$) for each subinterval Δt. In particular, missing high-frequency components can be corrected due to the transition to a lower grid resolution $\mathcal{R}$, as well as errors caused by a simpler numerical algorithm.

Parareal iterations are typically unstable for problems without dissipation, and in the case of a too inaccurate $\mathcal{G}_{\Delta t}$. Therefore, a more elaborate model, denoted by $\Psi^\theta_{\Delta t}$, is required for convergence. As demonstrated in related studies [61, 62, 63], the convergence in various contexts is influenced by the product of $\|\mathcal{F}_{\Delta t} - \Psi^\theta_{\Delta t}\|$ and $\frac{1-(\Psi^\theta_{\Delta t})^N}{1-\Psi^\theta_{\Delta t}}$. This depends on the magnitude of $\|\Psi^\theta_{\Delta t}\|$ and the size of the iterative system, N.

To derive the algorithm presented in Algorithm 1, we rearrange Eq. (1.2) for the time stepping of $\mathcal{F}_{\Delta t}$ into

$$\mathrm{u}_{n+1} \coloneqq \mathcal{F}_{\Delta t}\mathrm{u}_n = \mathcal{F}_{\Delta t}(\mathcal{I}\mathcal{R})\mathrm{u}_n + [\mathcal{F}_{\Delta t}\mathrm{u}_n - \mathcal{F}_{\Delta t}(\mathcal{I}\mathcal{R})\mathrm{u}_n]. \tag{3.41}$$

Formally, we replace $\mathcal{F}_{\Delta t}(\mathcal{I}\mathcal{R})\mathrm{u}_n$ by a computationally cheaper strategy in the fixed-point iteration. In our case, we aim to develop an efficient deep learning model for $\Psi^\theta_{\Delta t}$, trained by appropriately generated examples, to enhances a low-fidelity solver, $\mathcal{G}_{\Delta t^\star}$, end-to-end:

$$\mathrm{u}^{k+1}_{n+1} \coloneqq \Psi^\theta_{\Delta t}\mathrm{u}^{k+1}_n + [\mathcal{F}_{\Delta t}\mathrm{u}^k_n - \Psi^\theta_{\Delta t}\mathrm{u}^k_n], \quad k = 0, \ldots, K-1 \tag{3.42}$$

$$\mathrm{u}^0_{n+1} \coloneqq \Psi^\theta_{\Delta t}\mathrm{u}^0_n, \quad n = 0, \ldots, N-1. \tag{3.43}$$

We observe that the computationally expensive $\mathcal{F}_{\Delta t}\mathrm{u}^k_n$ on the right-hand side of Eq. (3.42) (also see line 7 and 8 in Algorithm 1) can be performed in parallel. Thus, for each iteration

Algorithm 1 Parareal Pseudo Algorithm

```
1: Sample initial condition and media $(\mathfrak{u}_0, c)$.
2: Initialize $\mathfrak{u}_n^k$ and $\mathfrak{u}_n^{\text{parareal}}$ for $0 \leq n \leq N$, $0 \leq k \leq K$.
3: Set $\mathfrak{u}_0^k = \mathfrak{u}_0 \ \forall k$.

4: for $n = 0$ to $N - 1$
5:     Compute initial guess $\mathfrak{u}_{n+1}^0 = \Psi_{\Delta t}^{\theta} \mathfrak{u}_n^0$.

6: for $k = 0$ to $K - 1$
7:     for $n = 0$ to $N - 1$ [in parallel]
8:         Compute Parareal term $\mathfrak{u}_{n+1}^{\text{parareal}} = \mathcal{F}_{\Delta t} \mathfrak{u}_n^k - \Psi_{\Delta t}^{\theta} \mathfrak{u}_n^k$.
9:     for $n = 0$ to $N - 1$
10:        Compute corrected solution $\mathfrak{u}_{n+1}^{k+1} = \Psi_{\Delta t}^{\theta} \mathfrak{u}_n^{k+1} + \mathfrak{u}_{n+1}^{\text{parareal}}$.

11: return $\mathfrak{u}_n^k$
```

in k, the computations for $\mathcal{F}_{\Delta t} \mathfrak{u}_n^k$ can be distributed to N different processors, which significantly speeds up the computation. We focus on the balance between efficiency and accuracy of the individual components $\mathcal{I}$ and $\mathcal{R}$. Therefore, a key goal is to investigate the deep learning components, $\mathcal{I}^\theta$ and $\mathcal{R}^\theta$, to enhance the stability and convergence of Parareal.

4 Evaluation Setup

The focus of this thesis is to empirically compare different variants that best simulate 2D wave propagation. For this reason, our experiments are structured to maintain simplicity in the setup and keep comparisons fair. The goal is to have a robust system that corrects errors caused by the low-fidelity solver using machine learning components. By keeping the error small enough, the parallel-in-time integration methods like Parareal can be applied to efficiently enhance the solution. Without a stable model $\Psi^{\theta}_{\Delta t}$, it is not possible to significantly reduce the computational time as the error will accumulate over time and cause the numerical algorithm to diverge.

The key error sources are the reduced grid size and the inferior accuracy of the low-fidelity $\mathcal{G}_{\Delta t}$ running on it. This causes wave field computation by $\mathcal{G}_{\Delta t}$ to lag behind and loose high-frequency components. For this reason, we study different model configurations in order to make reliable design choices even in the presence of strongly oscillating media. The central factors that affect the trade-off between computation time and accuracy we choose to tune are:

(i) by how much we scale down the spatial resolution of the input wave field (spatial grid spacing),

(ii) the step size of the solvers (temporal grid spacing),

(iii) the model setup, including the deep learning architecture.

A simple model with bi-linear interpolation (E2E Vanilla (E2E-V)) for each component of Eq. (1.2) is used as a baseline. Each variant changes the baseline by exactly one aspect (see Table 4.1). This allows us to isolate the effect of each modification on the performance of the architecture. To draw in-depth conclusions from the evaluation, an initial analysis (experiment 1) serves to select the most promising architecture for a detailed evaluation. This way our conclusions will exhibit reduced dependency on the selected search space, thus yielding results that are representative of 'reasonable' tuning efforts. The selected variants are then trained using different strategies to form a more robust and faster learning algorithm (experiments 2–5). They are also evaluated on different media to account for a reliable generic application (cf. Subsection 4.1.2). As a second reference, the deep learning setup by Nguyen and Tsai [1] (Not E2E 3-level JNet (NE2E-JNet3)) is adjusted to our holistic setting. Thus, the effects of combining individual components into an end-to-end system can be identified.

For fair comparisons, the configuration of the individual variants must be similar, since different variants require different settings. Therefore, settings such as learning rate or batch size are uniquely determined for each variant. These configurations are set before training, control the learning process, and are called hyperparameters. For simplicity, this work is limited to a grid search. Here, all possible combinations for each variant are manually evaluated from a set of candidates and the best-performing hyperparameter combination is selected. A detailed explanation of the performance metrics can be found in Section 4.5.

4.1 Dataset and Methodology

For optimal results, the training horizon must be long enough to contain sufficient wave propagation patterns. Yet the number of iterations must remain small to maintain cross-media similarities. In particular, the dataset should not contain wave fields that significantly differ from the majority of the data caused by dispersion errors. Similar to Nguyen and Tsai (cf. Chapter 2 in [1]), we therefore chose to generate the dataset in the following way:

1. In line with Section 2.2 in [1], an initial wave field $\mathfrak{u}_0 = (u_0, \partial_t u_0) \in \mathfrak{F}_{u_0}$ is sampled from a Gaussian pulse,

$$\mathfrak{u}_0(x_1, x_2) = (u_0, p_0) = (e^{-\frac{(x+\tau)^2}{\sigma^2}},\ 0), \tag{4.1}$$

   with $x \in [-1,1]^2$ and $\frac{1}{\sigma^2} \sim \mathcal{N}(250, 10)$. $\tau \in [-0.5, 0.5]^2$ for $i \in \{1,2\}$ represents the displacement of the Gaussian pulse's center from its original position $(0,0)$.
2. Every $\mathfrak{u}_0 \in \mathfrak{F}_{u_0}$ is then propagated eight time steps $\Delta t^\star = 0.06$ (sec) by $\mathcal{F}_{\Delta t^\star}$. We define the solutions of the sufficiently accurate high-fidelity solver $\mathcal{F}_{\Delta t^\star}$ advancing the wave $\mathfrak{u}_n$ as the ground truth value of the wave at $\mathfrak{u}_{n+1}$.

The wave trajectories $u_{n+1} = \mathcal{F}_{\Delta t^\star} u_n$ provide the input and output data for the supervised learning algorithm, which aims to learn the solution map $\Psi^\theta_{\Delta t} : X \mapsto Y$:

$$\begin{aligned} X &:= \{(\nabla u_n, c^{-2}(u_n)_t, c)\} \\ Y &:= \{(\nabla u_{n+1}, c^{-2}(u_{n+1})_t)\}, \end{aligned} \tag{4.2}$$

where $\mathcal{D} = \{(x, y)\}$ with $x \in X$ and $y \in Y$, and $\mathfrak{u}_n \in \mathcal{M}$. Each trajectory is paired with a velocity profile, i.e., (u_0, c). For all experiments, we also adopt the fine grid settings of the spatial resolution ($\delta x = \frac{2}{128}$) and temporal resolution ($\delta t = \frac{1}{1280}$).

4.1.1 Dataset Split

The dataset is then split into three parts: a training set, a validation set for optimizing the hyperparameters, and a test set for the final evaluation. During testing, data points are sampled from $\mathcal{D}$, and the algorithms are applied for a single time step $\Delta t^\star$, as demonstrated in Eq. (1.2). During validation and testing, the variants are iteratively applied to an initial condition sampled from $\mathcal{D}$ for eight steps. Different sets of hyperparameters are used to train on the training set and evaluate on the validation set. The best-performing variant is then re-trained on the training and validation set and tested on the test set. It is important to note that the weights θ are not changed during testing. To ensure equal conditions, the training set is limited to 40,000 data points; the validation and test set are limited to 5,000 data points.

4.1.2 Velocity Profiles

Analogous to [1, 64], different synthetic geological structures are used to better understand the adaptability and robustness of our end-to-end model in a wide variety of scenarios. Therefore, we chose media with challenging and realistic conditions based on a real-world geophysical dataset as seen in Figure 4.1:

Marmousi profile. One commonly used synthetic benchmark profile is based on a real-world geophysical dataset from a North Sea oil field [65] and was first introduced in 1991 by the Institut Français du Pétrole (IFP). It has gained widespread adoption in geophysics, especially in the areas of seismic imaging and inversion. The medium can be characterized by its complex structure, including steeply dipping layers, faulting, and substantial lateral and vertical velocity variations.

BP profile. The British Petroleum (BP) released a well-established geophysical dataset in 2004, which includes synthetic media that mimic geological features found in the Gulf of Mexico, Caspian Sea, North Sea, and Trinidad [66]. These features represent salt bodies, sedimentary layers, and faulting, resulting in highly oscillating media with non-trivial variations.

For training, we employ a method similar to the one described in Section 2.2 of Nguyen's

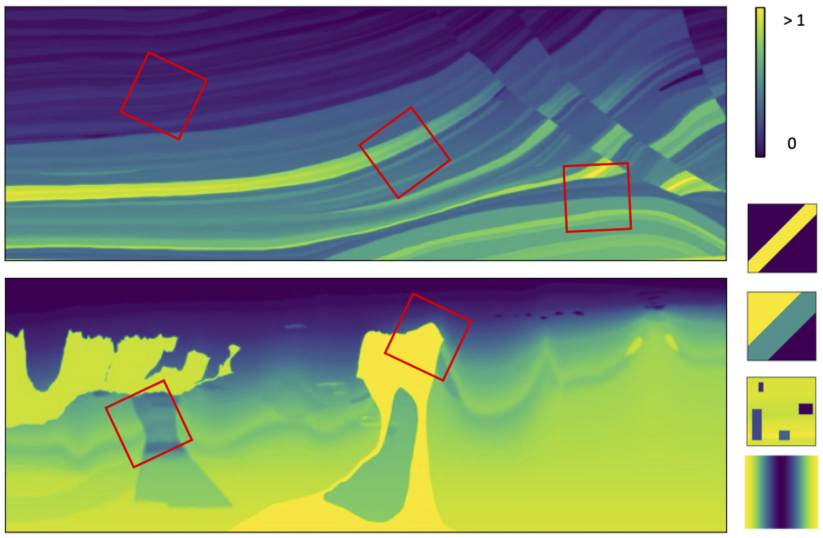

Figure 4.1: Velocity profiles. Randomly chosen subregions of the Marmousi (top) or BP (bottom) profile are shown in red squares. Velocity profiles on the right are used for testing. Brighter colors indicate higher velocity (see color bar at the top right corner). This figure is inspired by Figure 3 in [1].

2023 work [1]. Specifically, we use randomly chosen subregions of the Marmousi and BP profiles that are mapped onto the spatial grid $h\mathbb{Z}^2 \cap [-1,1]^2$. For testing, four manually modified velocity profiles are added to our testbed to examine unique behavior, such as rapid variations in velocity both laterally and vertically:

- Diagonal Ray: $c(x_1, x_2) = 3 - 1.5\ [|x_1 + x_2| > 0.3]$
- Three Layers: $c(x_1, x_2) = 2.5 - 0.7\ [|x_1 + x_2| > -0.4] * 2\ [|x_1 + x_2| > -0.6]$
- Wave Guide: $c(x_1, x_2) = 3 - 0.9\ \cos(\pi x_1)$
- Cracked Marmousi: $c(x_1, x_2) = c_{Marmousi}$ or 0.25 [if x_1 and x_2 in specific range]

Here, we draw samples from the Marmousi and BP profiles with a probability of 30% each, while the other velocity profiles are sampled with a probability of 10% each, respectively. It is important to mention that when using absorbing boundary conditions, these velocity profiles must be resized, as the pseudo-spectral method is only designed for periodic boundary conditions.

4.2 Experiments

4.2.1 Experiment 1: Architecture Preselection

The average training time of each variant, including hyperparameter tuning, is approximately 73 CPU core hours. Due to resource constraints, we therefore limit our study to one end-to-end variant. Based on this preliminary analysis, we selected the most promising approach from four state-of-the-art deep learning architectures for the upsampling component, along with two approaches for the downsampling component. Within the supervised learning framework, the models (see Section 4.3) are trained on dataset $\mathcal{D}$ and the most promising combination is selected based on their performance.

4.2.2 Experiment 2: Multi-Step Training

We train the baselines and the chosen end-to-end variant from experiment 1 on $\mathcal{D}^m$, described in Subsection 3.2.4, using an equal number of training points as in $\mathcal{D}$. The test set is consistent with $\mathcal{D}$ to enable comparison with other experiments.

4.2.3 Experiment 3: Weighted Multi-Step Training

The training and evaluation setup follow experiment 2, while the models are trained on $\mathcal{D}^{w,m}$, also described in Subsection 3.2.4.

4.2.4 Experiment 4: Time Reduction and Performance

A central objective of this thesis is to identify the optimal set of parameters that balance speed and sophistication. Related studies lack systematic empirical results of their grid spacing configurations. By finding the most suitable combination of parameters, we can improve efficiency without compromising on accuracy. Hence, we conduct a comprehensive grid search using dataset $\mathcal{D}$ to optimize the temporal and spatial grid size parameters of $\Psi^{\theta}_{\Delta t}$. In order to make some more interesting comparisons, we have also evaluated the most efficient model according to the previous experiments.

We have tried to select reasonable parameter ranges and satisfy the CFL condition; selecting parameters beyond this range introduces large dispersion errors in the wave field that accumulate over time. In this experiment, we perform nine grid searches (one for each grid spacing combination):

- $\Delta x \sim \{2^{-6}, 2^{-5}, 2^{-4}\}$: A smaller step size causes the algorithm to be more precise as details can be represented more accurately. This is particularly important when approximating phenomena with localized variations, sharp gradients, or discontinuities. However, this significantly increases the overall computational time on the wall-clock. A larger spatial step size can cause numerical dispersion errors or make the training algorithm unstable (especially for $\Delta x > \frac{c}{\Delta t\ \mathcal{C}}$ with the Courant number $\mathcal{C}$ (cf. Eq. (3.22))).
- $\Delta t \sim \{2^{-10}, 2^{-9}, 2^{-8}\}$: The second-order accuracy of the velocity Verlet algorithm can degrade if the time step is too large relative to the underlying dynamics of the system. Other than fulfilling the CFL condition, taking smaller time steps prevents energy fluctuations or drifts in the system's total energy.

4.2.5 Experiment 5: Enlarge Dataset

On the one hand, a larger dataset size helps to prevent overfitting and may improve performance. However, on the other hand, beyond a certain point, adding more data points might not significantly enhance accuracy. The increased computational costs and time spent processing the extra data may not yield a proportionate improvement. In this section, we aim to investigate the impact of dataset size on the end-to-end model's accuracy and training convergence. By employing random horizontal and vertical flipping with a 50% probability, we artificially augment the dataset by a factor of four, creating the augmented dataset $\mathcal{D}^4$.

We adhere to the same setup as in previous experiments for consistency. For fair comparisons, we additionally evaluate the model trained on the original dataset $\mathcal{D}$ with an increased number of training epochs. Hence, the total number of training instances (epochs $\times$ dataset size) remains equal for both the original and augmented dataset. This analysis offers insights into the optimal dataset size.

4.2.6 Experiment 6: Parareal Optimization

To the best of our knowledge, there has been a gap in the systematic analysis of time-efficient optimization strategies for end-to-end wave propagators. Among numerous optimization strategies, the parallel-in-time method Parareal has shown good results in related studies [1, 14]. Consequently, we explore improvements to our variants using the Parareal scheme on two datasets:

A. Fine-tuning. One tactic is to develop a training algorithm by integrating the pre-trained end-to-end variants with the Parareal scheme. The novel Parareal refinement dataset is denoted as $\mathcal{D}^p_{\text{refine}}$; it is composed as follows: To ensure fair comparisons with other experiments, the training dataset must encompass the same amount of data points as $\mathcal{D}$, i.e., $\|\mathcal{D}^p_{\text{refine}}\| = 40{,}000$. This is why the variants are first trained on a random subset containing half of the original dataset $\mathcal{D}$. Next, we randomly select another subset that constitutes an eighth of $\mathcal{D}$. For each data point within this chosen subset, $\Psi^{\theta}_{\Delta t^\star}$ is applied according to the Parareal scheme in Eq. (3.42) and Eq. (3.43) with $K = 4$. The losses from all Parareal iterations for each batch are summed to reduce errors across iterations. The batch size is divided by four to account for the same amount of data points in the computation of the loss.

B. Comprehensive Training. Rather than employing a pre-trained model, models are solely trained through Parareal iterations. The new dataset is denoted as $\mathcal{D}^p_{\text{train}}$, while also $\|\mathcal{D}^p_{\text{train}}\| = 40{,}000$. In this case, we apply the same Parareal training procedure as above, starting with a random sample that constitutes a quarter of the original dataset $\mathcal{D}$.

4.3 Model Variants

The evaluation encompasses four end-to-end (E2E) variants and two benchmark models. Table 4.1 summarizes all models with their respective formula and dataset. $\mathcal{G}_{\Delta t^\star}$'s parameters for all models were set to $\Delta x = 2/64$ and $\Delta t = 1/600$, and training datasets are consistent across all experiments.

4.3.1 End-to-end Models

Bilinear interpolation with a factor of two serves as the downsampling component, while a JNet-based method is employed for upsampling. This allows us to isolate the effect of architectural changes on performance. We explore four JNet-based architectures that were chosen due to their recent success in related super-resolution studies [67, 68, 69]. It is important to note that the model configurations such as the optimizer and dropout rate were adopted from the original papers. This guarantees that we are assessing the models under the optimal conditions defined by their respective authors. For the decoder phase, a bilinear interpolation layer followed by regular convolutional layers is added to

Table 4.1: A summary of all model variants with their respective formula and training dataset.

name	formula	dataset
E2E-V	$\Psi^{0,0}_{\Delta t^\star} := \mathcal{I}^0 G_{\Delta t^\star} \mathcal{R}^0$	-
NE2E-JNet3	$\mathcal{I}^{\theta}_{\Delta t^\star}(G_{\Delta t^\star} \mathcal{R}^0)$	$\mathcal{D}$
E2E-JNet3	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}$
E2E-CNN-UNet3	$\Psi^{\theta_1,\theta_2}_{\Delta t^\star} := \mathcal{I}^{\theta_1}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^{\theta_2}_{\Delta t^\star}$	$\mathcal{D}$
E2E-JNet3	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}^4$
E2E-JNet3	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}^m$
E2E-JNet3	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}^{w,m}$
E2E-JNet3	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}^p_{\text{refine}}$
E2E-JNet3	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}^p_{\text{train}}$
E2E-JNet5	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}$
E2E-Tira	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}$
E2E-Trans	$\Psi^{\theta,0}_{\Delta t^\star} := \mathcal{I}^{\theta}_{\Delta t^\star} G_{\Delta t^\star} \mathcal{R}^0$	$\mathcal{D}$

achieve the JNet structure[1].

3-level and 5-level JNet employs a fully-convolutional network with skip connections. We investigate two different sizes: The E2E 3-level JNet (E2E-JNet3) comprises three ResNet blocks, while the E2E 5-level JNet (E2E-JNet5) consists of five ResNet blocks in the encoding and decoding phase. A major benefit is that this model structure has fewer parameters than others with similar performance on popular datasets [71, 72]; thus, it can be trained from few images and has lower computational requirements. A schematic of the E2E-JNet3 can be seen in Figure 4.4.

Tiramisu JNet (E2E-Tira) [73] features dense blocks in addition to the ResNet structure. Each layer of a dense block receives the feature maps of all preceding layers as an input. These connections are achieved through the concatenation of feature maps along the channel dimension. This design encourages feature reuse, mitigates the vanishing-gradient problem, and allows for training deeper networks.

Transformer JNet (E2E-Trans) [74] combines the advantages of U-Net and Transformer layers. Transformers are a type of neural network component that were introduced by [75] in 2017 to process sequential data by capturing long-range dependencies. They

[1]This method has an advantage over transposed convolution because of checkerboard artifacts [70]. It addresses a common issue where reconstructed feature maps exhibit patterns of repeating squares or grid-like structures. To further increase the resolution, this process can be repeated or the interpolation factor can be modified.

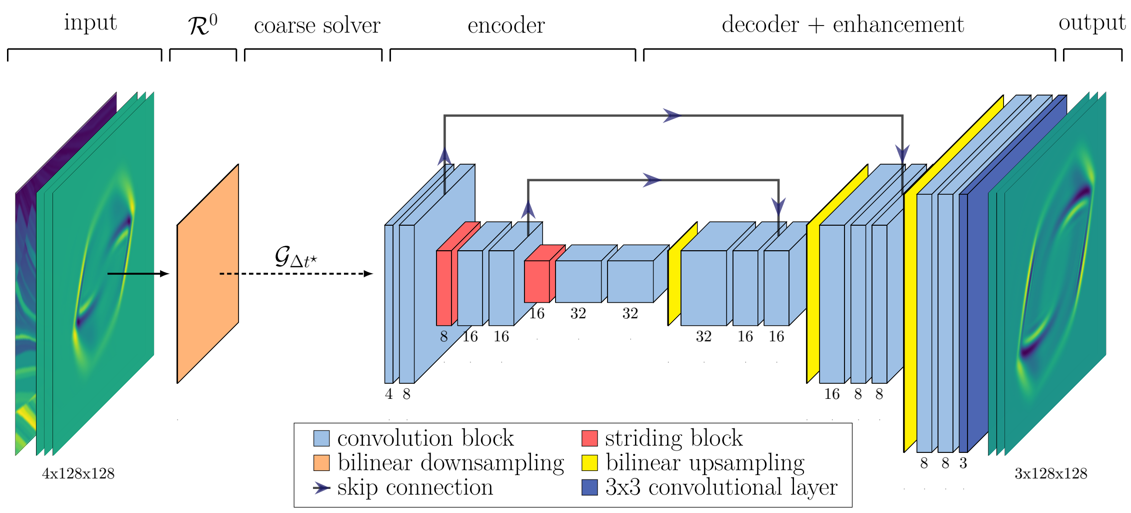

Figure 4.4: Detailed schematic of the end-to-end 3-level JNet (E2E-JNet3). The architecture of the neural network, which includes the encoder, decoder, and enhancement components, is adapted from Figure 1 in [73]. Each convolutional block (blue) encompasses a 3x3 convolutional layer (groups = 3, padding = 1), followed by a batch normalization and a ReLU activation function. To reduce the resolution, the 3rd and 6th convolutional block (red) employs a stride of two. Bilinear interpolation with a factor of two is used for downsampling in $\mathcal{R}^0$ and upsampling in the decoder part. The last block only contains the 3x3 convolutional layer. Connectivity within the network is depicted by arrows, with the dashed arrow specifically indicating a single application of $G_{\Delta t^\star}$.

utilize a self-attention mechanism to weigh the relevance of different parts of an input sequence. This design allows the end-to-end model to capture both local and global contextual information efficiently, making it suitable for wave propagation tasks.

E2E CNN 3-level UNet (E2E-CNN-UNet3) employs a more sophisticated technique for the downsampling component, as opposed to using bilinear interpolation. Resembling [7], a stack of convolutional layers and a striding layer are used. The upsampling component consists of a 3-level JNet block, enabling comparisons with both the benchmark and the end-to-end configurations. A major issue in our general model setup involves the potential loss of high-frequency components during the encoding phase of the JNet structure. To address this issue, we added a skip connection in $\mathcal{R}^{\theta_1}$ before the striding layer and connected it to the second last convolutional layer in $\mathcal{I}^{\theta_2}$. Therefore, detailed features can skip the coarse grid representation and missing information may be recovered.

4.3.2 Benchmarks

The benchmark models are evaluated on the same datasets as the variants, while

E2E-V demonstrates the performance of $\mathcal{G}_{\Delta t^\star}$ without using deep learning. Bilinear interpolation is employed for both the upsampling and downsampling components, which are end-to-end integrated with $\mathcal{G}_{\Delta t^\star}$.

NE2E-JNet3 is taken from Nguyen and Tsai's work [1]. This model does not solve the wave equation end-to-end. In contrast, results of $\mathcal{G}_{\Delta t^\star}$ are used to separately train the E2E-JNet3 upsampling component, while the training labels are provided by the dataset $\mathcal{D}$. This setup is not suitable for any form of multi-step training, i.e., experiment 2 and 3 are excluded. We further modified the solver to have absorbing boundary conditions to maintain comparable results.

4.3.3 Training Specifications

The variants share the following common setup: Networks employing the ReLU activation function are used for numerical solutions to the 2D wave equation. For better comparability, the variants are further standardized in some aspects: The learning rate decays by an exponential factor denoted as gamma = 0.97 after every epoch, i.e., $\mathrm{lr}_{n+1} = \mathrm{lr}_n * \mathrm{gamma}^n$, where n is the number of epochs. The purpose of learning rate decay is to find a good balance between fast convergence at the beginning of training and fine-tuning towards the end, as described for example in [76]. The initial weights for all networks were drawn from a Kaiming uniform distribution. Training stopped after twenty epochs, or if there was no improvement on the validation set for more than five epochs. This method is called early stopping and reduces the risk of overfitting (cf. [38]). The adjustment of other hyperparameters is done separately for each variant and is described in the next section.

4.4 Hyperparameter Search

While there are other methods to search for good hyperparameters more precisely [77], a grid search has some advantages for this evaluation setting: it is easy to implement, covers the search space well, and provides an initial estimate of the optimal combination in an efficient way. We limit our fine-tuning efforts to the hyperparameters below, since optimizing the dropout rate, loss function, or optimizer, as well as adding a learning rate scheduler [78], did not yield notable improvements.

A total of 30 grid searches are performed for three runs (18 for experiment 4; six for experiment 1; two for experiment 5 and 6; one for experiment 2 and 3):

- Learning Rate $\sim \{10^{-3}, 10^{-4}\}$: Setting the hyperparameter α of Eq. (3.29) is crucial for convergence and accuracy. On the one hand, a smaller learning rate leads to slower convergence but may achieve better accuracy with the risk of getting stuck in suboptimal local optima. On the other hand, a larger learning rate may speed up convergence but risk overstepping the optimal solution.
- Weight Decay $\sim \{10^{-2}, 10^{-3}\}$: A smaller weight decay value reduces the penalty on large weights, which leads to a more flexible model that could potentially cause a reduced generalization performance on unseen data. Conversely, a larger weight decay value enforces stronger regularization, promoting model sparsity and reducing overfitting. However, this may result in underfitting the training data if the constraint is overly strict.
- Batch Size $\sim \{2^6, 2^8\}$: A common strategy is to set the largest batch size that fits within the GPU memory constraints. However, [79] suggests that large batch sizes may lead to convergence to sharp minima, therefore reducing generalization capabilities. Smaller batch sizes, on the other hand, can enhance generalization and memory efficiency but may slow down training. Conversely, larger batch sizes can expedite training through GPU parallelization while potentially sacrificing generalization.

4.5 Metrics

Quantitative analysis is conducted by examining the energy components, while visual representations show the wave fields in the energy semi-norm (Eq. (3.2)). For a wave field w and a reference solution v in energy components, the quantitative evaluation metrics used in each experiment are:

- Energy Mean Squared Error (MSE):

$$\text{energy MSE } (w, v) = \frac{1}{n} \sum (E_{\delta x}[w] - E_{\delta x}[v])^2 \tag{4.3}$$

  The MSE is a metric that quantifies the average squared difference between predicted and actual wavefield values. The index δx refers to the grid spacing in the discretized energy semi-norm, while n denotes the number of pixels in $E_{\delta x}[w]$ and $E_{\delta x}[v]$.

- Relative Energy Mean Squared Error (cf. Chapter 3 in [1]):

$$\text{relative energy MSE } (w, v) = \frac{1}{n} \sum \left\| \frac{E_{\delta x}[w - v]}{E_{\delta x}[v]} \right\|^2 \tag{4.4}$$

Relative errors provide a fair comparison by taking into account the differences in the magnitudes of the true values. This ensures that the comparison focuses on the relative performance of the methods, rather than being influenced by the scales of the true values.

- Average Computation Time:
  The average runtime (measured in seconds) and training time (measured in hours) of a model are critical indicators of its efficiency and practical applicability in real-world scenarios. Note that variations in hardware specifications and programming languages may lead to different outcomes. Individual details are elaborated upon in Chapter 5. The goal of this study is not to provide state-of-the-art results, but rather to conduct a fair comparison of different wave propagators. Therefore, these numbers are only meant as a vague guideline for the reader.

To compare the results on the validation set, each experiment utilizes the energy MSE with the fine solver's solution as the reference. To emulate a practical scenario, the model consecutively processes its prior solution over eight time intervals, starting from the initial condition, i.e., $\sum_{k=1}^{8} \text{MSE}\left((\Psi_{\Delta t^\star}^{\theta})^k \mathbf{u}, \mathcal{F}_{\Delta t^\star}^{k} \mathbf{u}\right)$. It should be noted that calculations from $\mathcal{G}_{\Delta t^\star}$ are integrated into the NE2E-JNet3 calculations to produce comparable test outcomes.

5 Discussion

Each of the total 90 runs required an average of 72.8 GPU core hours on one NVIDIA A100 Tensor Core GPU[1] to complete, while the E2E-JNet3 was trained almost 41% faster and the E2E-Tira three times slower than the average. This sums up to a total runtime on a single GPU of just over 6,552 hours.

The best trial on the test set was achieved by E2E-Tira with an energy MSE of 0.0109, which is well below the 0.0462 from the previously published model, NE2E-JNet3, by Nguyen and Tsai [1]. Our most efficient variant is E2E-JNet3 trained on $\mathcal{D}^{w,m}$ with an energy MSE of 0.0169, which is close to the results of more extensive models such as the E2E-Tira and E2E-Trans, but is more than five times faster. Our best result for the fully convolutional model, E2E-CNN-UNet3, is 0.0539, which is above the average 0.0250 for the standard E2E-JNet3. The best Parareal-based result is 0.0322 on the test set for the E2E-JNet3 which was fine-tuned using $\mathcal{D}^{p}_{\text{refine}}$.

This chapter presents the results based on the experimental setup of Chapter 4. It provides the analysis of different variants, including performance improvements through the end-to-end structure (5.1.1), multi-step loss strategies (5.1.2), deep learning architectures (5.1.3), numerical solver configurations (5.1.4), and the impact of other modifications such as adding data augmentation, changing the downsampling component, and applying Parareal (5.1.5). Additionally, we offer an analysis on hyperparameters and training procedures, as detailed in Section 5.2. Welch's t-test with a significance level of $p = 0.05$ was used to examine the interrelationships of some characteristics and hyperparameters.

5.1 Comparison of the Variants

A summary of the grid search results of the 12 investigated variants is shown in Figure 5.1. The results represent the distribution of 30 validation set performances (excluding experiment 4) over the whole search space. Any conclusions drawn from them are therefore specific to our choice of search ranges. To make fair comparisons, the training time

[1]40 GB of Ram and 1410 MHz of GPU Clock Speed

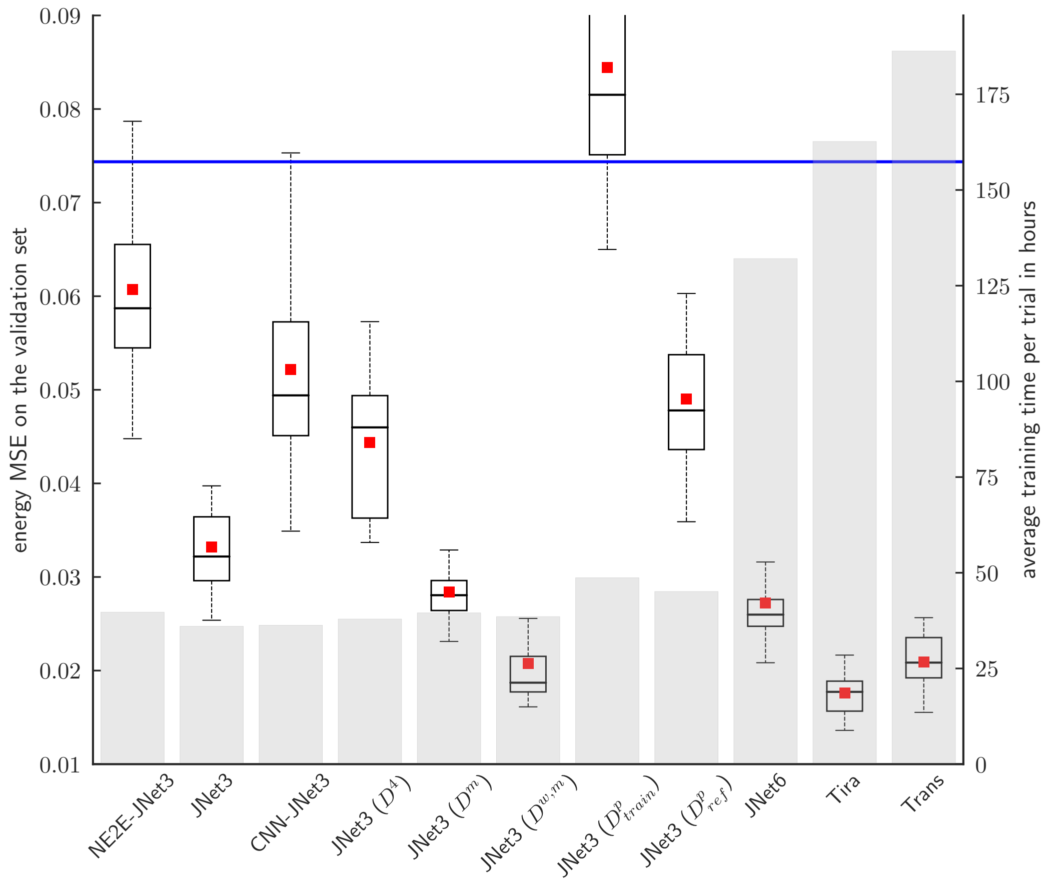

Figure 5.1: Total performance of all hyperparameter search trials for the wave propagation variants on the validation set. The boxes represent the range between the 25th and 75th percentile of values, while the whiskers indicate 1.5 times the interquartile range. The blue line illustrates the result of the E2E-V implementation and serves as a baseline. The red dot shows the mean and the black line marks the median of the data. The grey histograms in the background present the average training time of the respective variant in hours on a single GPU. The variant names are abbreviated for reasons of brevity.

estimate for NE2E-JNet3 includes the coarse solver's computations for each data point.

5.1.1 Evaluation of the End-to-end Structure

The first important observation based on Figure 5.1 is that integrating NE2E-JNet3 into a single, end-to-end system (i.e., E2E-JNet3), improved the average accuracy on the validation set by more than 46%. This effect is even more significant[2] on the test set as

[2]Exemplary significance test: The energy MSEs of 10 runs on the test set of the NE2E-JNet3 and E2E-JNet3 are taken. We conclude that the mean results differ significantly with a p-value of ca. $p \approx 0.000012$.

we can observe a reduction in the energy MSE by ca. 53%. Apart from the E2E-JNet3 ($\mathcal{D}^{p}_{train}$), which seems to have an unstable training progress, the ability to include the loss of both the coarse solver and the downsampling layer appears to be critical for the wave propagation task. In fact, having separated parts also caused a higher standard deviation and outliers skewed towards higher values, since the mean is well above the median for the NE2E-JNet3 compared to the E2E-JNet3.

Another related but unsurprising observation is a significant decrease in training time. This is because the time needed to advance a wave for one time step $\Delta t^{\star}$ is reduced by ca. 3% over the fragmented architecture given our parameter setup in Section 4.1. We assume that a single model instance is usually faster in most programming languages as the mathematical operations are optimized inside a machine learning framework. Since all modern applications depend on some form of framework, it is beneficial to integrate these modifications into the architecture to minimize computational effort.

5.1.2 (Weighted) Multi-Step Training

Introducing a multi-step training loss significantly enhanced the benefits of an end-to-end architecture even further (cf. E2E-JNet3 ($\mathcal{D}^{w,m}$) in Figure 5.1). Now, the computation graph can dynamically expand based on the number of wave advancements defined during training. Since computing the loss sequentially does not increase the number of parameters, only a marginal increase in training time was observed due to a more expensive backpropagation.

Figure 5.2 depicts how the average relative energy MSE increased over time for different variants on the test set. All end-to-end models had a much lower relative energy MSE increase particularly for the first three time steps. Hence, we conclude that connecting wave states to incorporate temporal propagation dynamics in the training data appears to be especially important for the early stages of wave advancements.

However, randomly sampling the number of time steps taken per run can in principle cause high performance fluctuations on the validation set when the model is only partially trained. Therefore, if the model takes many sequential steps in the early stages of training, the errors will accumulate rapidly and the gradients will be inaccurate. By taking fewer steps through sampling from a normal distribution that is being shifted along the x-axis (cf. Subsection 3.2.4), we avoid this problem by reducing the solution space. As training progresses, the wave propagation task becomes increasingly complex since the model's inputs are highly dependent on previous iterations. In this stage, a stable and accurate

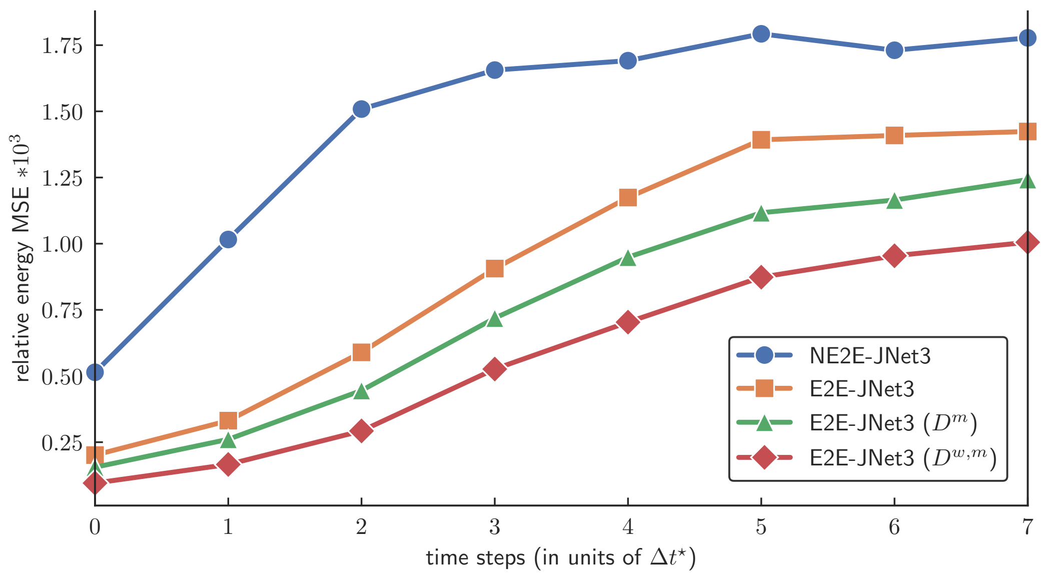

Figure 5.2: Comparing the NE2E-JNet3 model and three end-to-end JNet variants that differ in their training algorithms. Initial conditions and velocity profiles are sampled from $\mathcal{D}$ and the relative energy MSE results of 10 runs are averaged. As expected, all models show a bounded growth as the waves vanish due to absorbing boundary conditions.

neural network can then guarantee that the training signals remain in the wave solution manifold $\mathcal{M}$ (cf. Eq. (1.3)). We hypothesize that these behaviors will generalize to similar supervised learning problems related to PDEs, providing an efficient enhancement to the training algorithm.

5.1.3 Upsampling Architecture

An overview of the time complexity and test set performances of each upsampling architecture can be found in Table 5.1. We have tried to choose reasonable parameters that include the best solver settings and are still big enough to allow for an efficient computation.

As expected, the larger networks, namely the Tiramisu and Transformer JNet models, performed slightly better compared to the 3-level JNet architecture. Intensifying the dimensionality reduction of the model allows a more nuanced representation of features within the deeper structure of the network. However, for the ResNet architecture (E2E-JNet5), more weights did not lead to performance improvements on the test set despite the much higher computation expenses (more than 3.7 times compared to E2E-JNet3). Although the architecture demonstrated lower error rates on the validation set, it did

Table 5.1: Time Complexity and average test energy MSE of all upsampling variants using a batch size of 64. An initial condition was advanced eight times with $\Delta t^{\star} = 0.06$ and the mean of ten runs was taken. For the fine solver, we chose $\delta x = 2/128$, $\delta t = 1/1280$, while the coarse solver's parameters for all models were set to $\Delta x = 2/64$ and $\Delta t = 1/600$. Our results are specific to implementations in PyTorch and three NVIDIA A100 Tensor Core GPUs as described above. [80] provides more detailed information on the environment setup.

variant	number of parameters	GPU time (sec)	test energy MSE
$\mathcal{F}_{\Delta t^{\star}}$	-	57.96749	-
E2E-V	-	2.40421	0.07437
NE2E-JNet3 [1]	40,008	2.97328	0.05370
E2E-JNet3	40,008	2.88331	0.02496
E2E-JNet5	640,776	10.84893	0.02379
E2E-Tira	123,427	13.57449	0.01274
E2E-Trans	936,816	15.67633	0.01743

not generalize as well as other expensive models on unseen velocity profiles. For example, the Transformer JNet implementation, which has an even greater number of trainable parameters than the E2E-JNet5, also showed superior generalization capabilities as evidenced by a more than 26% reduction in energy MSE on the test set. Consequently, we theorize that the design of the ResNet may be insufficient for capturing more advanced patterns like high-fidelity wave components, potentially leading to a plateaued loss or overfitting.

Highly-connected layers with an optimized feature propagation and gradient flow (E2E-Tira) significantly improved the performance and even yielded the best overall results on both the validation and test sets. In contrast, the ability to weigh the importance of input features when making approximations (E2E-Trans) appears to be less critical for wave propagation. The argument against using these variants is that they increase the number of parameters, thereby increasing the time complexity by a factor of ca. 4.7 for E2E-Tira, and ca. 5.4 for E2E-Trans, compared to the E2E-JNet3. Given that E2E-JNet3 ($\mathcal{D}^{w,m}$) had only a slightly worse average energy MSE on the test set (0.01784), we generally advise against using the expensive models in our setup due to a very high computational complexity.

5.1.4 Optimal Numerical Settings

For a batch size of 64, E2E-JNet3 was more than 20 times faster than the $\mathcal{F}_{\Delta t^\star}$, and increased the accuracy by approximately 66% compared to E2E-V, the model that does not contain a neural network (cf. Table 5.1). In Figure 5.3, we can see that choosing appropriate coarse solver parameters was crucial for the performance, and, in most cases, suboptimal settings cannot be corrected by a neural network. The heat map serves as an initial structured analysis. To comprehensively evaluate the interplay between variables, we require more samples and a finer set of numerical settings.

Spatial setting. Nevertheless, given our observations so far, numerical spatial grid spacing (Δx) is the most important parameter for both models. A related observation is that there is a sweet-spot for the grid spacing at the center of the chosen candidates. In this region, performance was optimal, and the standard deviation was minimal. Consequently, while searching for a good spatial value, a detailed parameter tuning might be worthwhile to perform as the most computationally expensive option did not yield the lowest energy MSE.

Temporal setting. A small (but statistically insignificant) average performance improvement was observed for taking smaller time steps (Δt) given that the spatial grid spacing is set correctly. To be precise, the best overall performance in experiment 4 was achieved by E2E-JNet3 ($\mathcal{D}^{w,m}$) using the smallest time step setting ($\delta t = 2^{-10}$). It is obvious that applying the Verlet time integrator more frequently can better capture abrupt changes in the wave speed, which might be smoothed over with larger steps. Similarly, the authors of [1] conclude that their enhanced solver offers greater accuracy with a smaller $\Delta t^\star$. Hence, we recommend to prioritize finer time stepping for both $\Delta t^\star$ and Δt even if it compromises speed. Nonetheless, it is important to mention that very short time increments can introduce numerical dispersion errors as seen in the rightmost columns of each heatmap in Figure 5.3.

5.1.5 Further Modifications

Data Augmentation. One unexpected result of this study is that data augmentation (see experiment 5) affected the test performance and training time negatively in a significant way. Randomly flipping the input image (E2E-JNet3 ($\mathcal{D}^4$)) appears to be hurtful for a stable training progress; training E2E-JNet3 for a longer period slightly improved the mean performance on the validation set, but led to very similar results on the test set.

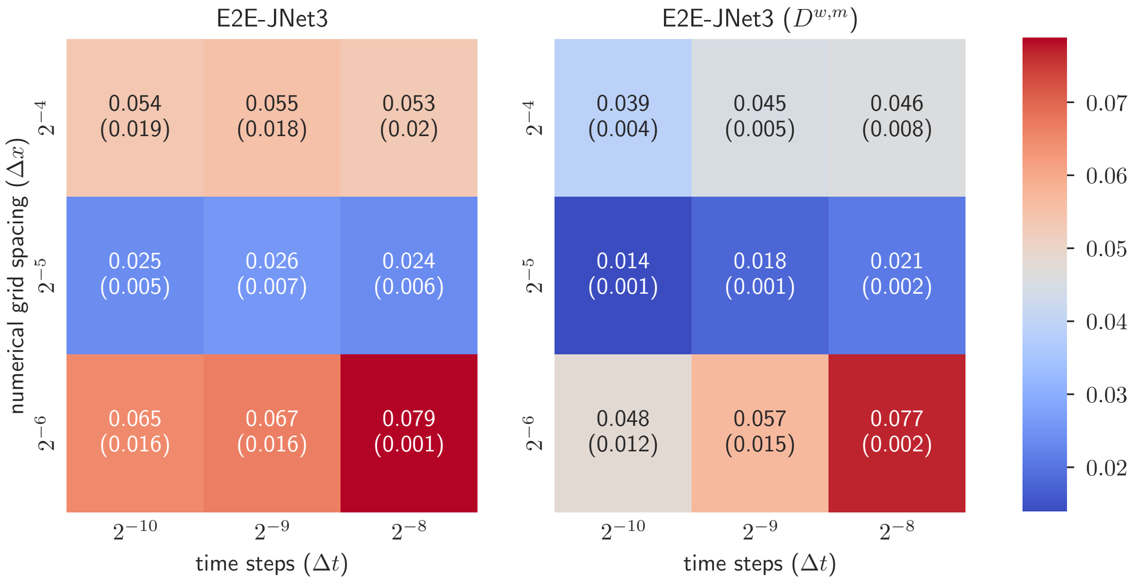

Figure 5.3: Total energy MSE performance for all combinations of coarse solver settings on the validation set (see experiment 4). The rows and columns of these matrices represent the different grid spacing and time-stepping parameters, respectively. The color encodes the performance as measured by the energy MSE, while low (blue) is better than high (red). Values in brackets indicate the standard deviation between the hyperparameter trials and, thus, the reliability of the accuracy. A hyperparameter search was done for each parameter combination, concluding to 72 trials in total for each variant.

Fully Convolutional Downsampling. Introducing a more sophisticated downsampling component with a skip connection from $\mathcal{R}^{\theta_1}$ to $\mathcal{I}^{\theta_2}$ led to a significantly lower accuracy. Altering the input too much before applying $\mathcal{G}_{\Delta t^\star}$ caused strong oscillations on the validation set. We hypothesize that the gradients of the downsampling component were not accurate enough to ensure a stable learning process. Given that this variant also slightly increased the computational cost, we generally advise against using it in our setup.

Parareal. Contrary to the findings in [1], the Parareal scheme in our setup did not enhance the training features when evaluated on $\mathcal{D}$. While a significant average performance improvement was observed for the refined Parareal model ($\mathcal{D}^p_{refine}$) compared to the fragmented baseline (NE2E-JNet3), decomposing and solving time intervals concurrently through Parareal had a negative effect when using our end-to-end structure.

However, applying E2E-JNet3 ($\mathcal{D}^p_{refine}$) within the Parareal scheme showed better results than E2E-JNet3 ($\mathcal{D}$) with Parareal. Figure 5.4 indicates that the accuracy was especially improved after a few time steps. As this training method further improved the stability

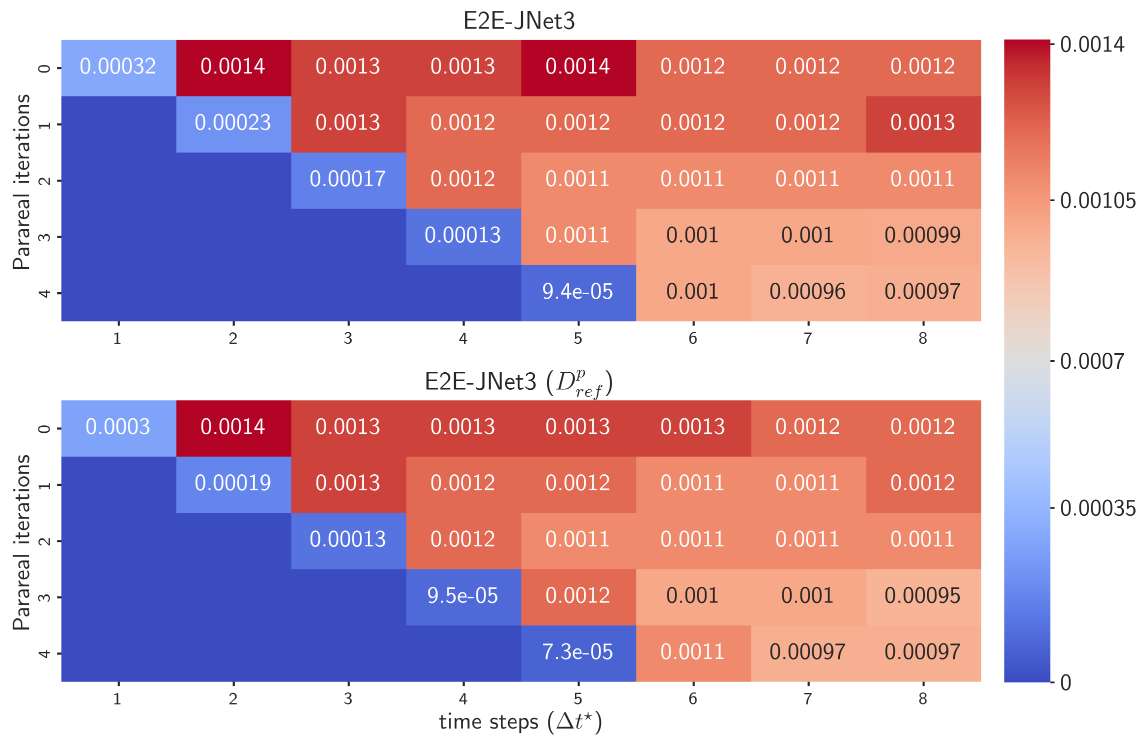

Figure 5.4: Energy MSE comparison between E2E-JNet3 ($\mathcal{D}$) and E2E-JNet3 ($\mathcal{D}^p_{refine}$), averaged over 10 runs, while the initial condition and the wave speed profile were sampled from $\mathcal{D}$. The color encodes the performance as measured by the MSE, while low (blue) is better than high (red). The evaluation uses the same algorithm as described in Section 3.3. Furthermore, we limited the number of Parareal iterations to four to remain consistent with experiment 6.

of Parareal iterations without sacrificing speed, it appears to be an efficient enhancement to our wave propagator.

5.2 Training and Impact of Hyperparameters

Due to early stopping, high fluctuations in the training duration of different runs of the same variant were observed. In some cases, the algorithm terminated after less than 10 iterations, while it sometimes iterated the full 20 epochs with the same parameter combination. However, the trials for one variant had a very similar accuracy, which suggests that the models are robust against different weight initializations and initial conditions. Similarly, the same hyperparameter combination repeatedly showed superior performance across nearly all trials for each variant, underlining the importance of a hyperparameter analysis.

To analyze the impact of hyperparameters in detail, Hutter et al.'s functional Analysis of Variance (fANOVA) method aims to evaluate their effects by efficiently handling dimension marginalization using regression trees [10]. This process enables the prediction of marginal error for a singular hyperparameter by averaging the others. To calculate the average performance for any isolated slice of the hyperparameter space, a regression tree is trained and its predictions along the relevant dimensions are aggregated. Specifically, a random regression forest[3] of 100 trees is utilized, and the mean performance is assessed. The resulting marginals are employed to break down the variance into additive components via the fANOVA technique [81] to assess the overall significance and interactions of the hyperparameters.

Learning Rate. As expected, understanding how to properly configure the learning rate is the most important setting to achieving optimal accuracy. The top half of Figure 5.5 indicates that speeding up convergence through a higher learning rate resulted in a significantly better performance for all variants, while reducing training time. A related but surprising observation is that a larger learning rate did not increase the variance, which can be explained by the smooth loss landscape of ResNets (cf. [82]).

The results in the bottom half of Figure 5.5 show that the learning rate significantly outweighs other hyperparameters in terms of their importance, consistently accounting for over one-third of the variance. It also highlights that the variance attributed to the learning rate is much bigger than the variance due to interactions between the learning rate and weight decay (a component of the 'higher order' share). They both control the training progress, but the results suggest that tuning the learning rate without considering 'higher-order' effects might be sufficient for good results.

Batch Size. Especially for the E2E-JNet3 model, batching the data into smaller parts, and, therefore updating the weights more frequently, significantly increased the performance. We hypothesize that a low batch size, similar to a larger learning rate, allows stable training as the gradient estimates do not seem to be noisy despite fewer samples. However, we find that quadrupling the batch size decreased the training time by more than 15% on average due to parallelization. Hence, we conclude that there is little practical value in setting the batch size optimally. For accelerated convergence, it might be worthwhile to only fine-tune the learning rate, while maximizing the batch size within the constraints of the available GPU memory.

[3]A random forest is a machine learning model that generates a collection of decision trees and uses their combined predictions for the final output. It introduces randomness in both selecting samples for training individual trees and also choosing features for splitting nodes.

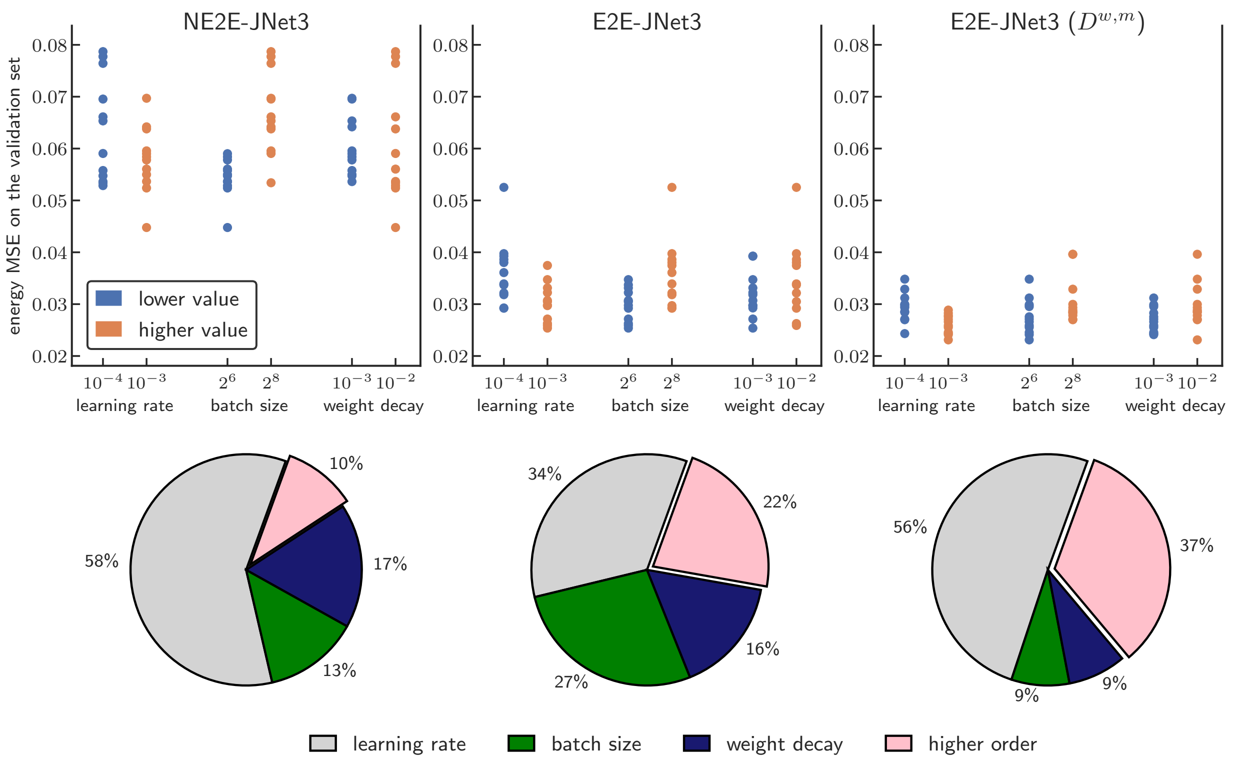

Figure 5.5: Energy MSE results of all hyperparameter search trials for different values of learning rate, batch size, and weight decay (columns inside a graph) and importance measured by variance using the fANOVA framework for three variants (pie charts). A dot in the scatterplot (top half) indicates a single result on the validation set grouped by hyperparameters, while lower hyperparameter values are shown in blue, and higher values in orange. The pie charts (bottom half) show which fraction of the variance of the test set performance can be attributed to each of the hyperparameters. The proportion of variance resulting from interactions between multiple parameters is labeled as 'higher order'.

Weight decay. Surprisingly, weight decay only slightly affected performance and variance. This may be the result of an adjusted learning rate and suggests that tuning weight decay does not offer substantial benefits when training ResNets with AdamW for wave simulations.

Interaction of Hyperparameters. Another important observation based on Figure 5.5 is that within an end-to-end system, higher-order interactions become more important. Expanding the computation graph through adding a multi-step loss function (E2E-JNet3 ($\mathcal{D}^{w,m}$)) increased the importance of these interactions even further. Hence, this tendency shows the growing relevance of hyperparameter-tuning for more connected wave propagators.

6 Conclusion

In this thesis, we presented a deep learning model that improves the method proposed by Nguyen and Tsai [1], offering fast, accurate and scalable solutions to the 2D wave equation across complex, multiscale media. We also reported the results of a large-scale study on different variants that investigate the efficacy of these improvements, and provided insights into parameter settings.

We concluded that the lightweight end-to-end 3-level JNet (E2E-JNet3) performed reasonably well given its low computation cost. In particular, all end-to-end variants without training modifications outperformed the modular framework of Nguyen and Tsai [1]. The method was further enhanced by introducing a weighted, multi-step training scheme ($\mathcal{D}^{w,m}$) to learn time-dependent wave dynamics. Incorporating these modifications into the training dataset is advantageous, as they do not add complexity to the model or substantially extend the training duration.

None of the investigated other modifications significantly improved performance. Surprisingly, augmenting the data, or training an additional neural network for the downsampling component led to a significantly lower accuracy. We speculated that both adaptations challenge stable gradient computations, making these variants less suitable for our setup. Contrary to previous research, our application of the Parareal scheme negatively affected the end-to-end structure when evaluated on sequential time steps. However, evaluating the Parareal-based model with Parareal iterations showed significant performance improvements over E2E-JNet3.

As expected, certain expensive upsampling architectures, such as intensify the interconnections between feature and gradient flows (Tiramisu JNet), significantly increased the accuracy. The Tiramisu variant is more attractive for extensive wave simulation than the ability to weigh the importance of input features (Transformer JNet) because it has a lower number of parameters. However, both models demand a high computational cost and are mostly impractical in modern engineering workflows.

The analysis of numerical parameter interaction revealed a clear structure. As the most resource-intensive choice did not consistently result in the lowest accuracy, thorough fine-tuning is essential. This implies that relying solely on neural networks may not suffice to rectify suboptimal settings. The same applies to adjusting the learning rate, which is the most crucial hyperparameter, while the batch size and weight decay were found to be unimportant in our setting. Surprisingly though, adjusting all hyperparameters was increasingly crucial for our advanced end-to-end models with larger backpropagation graphs due to higher-order interactions.

Developing stable neural network-based wave models is challenging due to their undefined complexities. Our study intends to bridge expert intuition with data, guiding the selection of architectures, training methods and fine-tuning parameters for these systems. In future work, we aim to investigate more advanced modifications of the unified, end-to-end wave propagator. One possibility is to enhance the Parareal algorithm with multi-step training, a strategy that has demonstrated efficacy in our current analysis and may stabilize the iterative refinements in our end-to-end setup further. Moreover, deploying a pre-trained deep learning restriction component may reduce the strongly oscillating loss in early stages of training. Further reduction in resolution will also provide valuable insights into the speed benefits of the end-to-end approach.

References

[1] Nguyen, H. and Tsai, R. ‘Numerical Wave Propagation Aided by Deep Learning’. In: *Journal of Computational Physics* vol. 475(1) (2023).

[2] Moseley, B., Markham, A., and Nissen-Meyer, T. ‘Solving the Wave Equation with Physics-Informed Deep Learning’. ArXiv eprint at abs/2006.11894. (2020).

[3] Ovadia, O., Kahana, A., Turkel, E., and Dekel, S. ‘Beyond the Courant-Friedrichs-Lewy Condition: Numerical Methods for the Wave Problem using Deep Learning’. In: *Journal of Computational Physics* vol. 442(1) (2021).

[4] Siahkoohi, A., Louboutin, M., and Herrmann, F. J. ‘Neural Network Augmented Wave-equation Simulation’. ArXiv eprint at abs/1910.00925. (2019).

[5] Meng, X., Li, Z., Zhang, D., and Karniadakis, G. E. ‘PPINN: Parareal Physics-Informed Neural Network for Time-Dependent PDEs’. In: *Computer Methods in Applied Mechanics and Engineering* vol. 370(1) (2020).

[6] Raissi, M., Perdikaris, P., and Karniadakis, G.E. ‘Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations’. In: *Journal of Computational Physics* vol. 378(1) (2019), pp. 686–707.

[7] Rizzuti, G., Siahkoohi, A., and Herrmann, F. J. ‘Learned Iterative Solvers for the Helmholtz Equation’. In: *81st EAGE Conference and Exhibition 2019*. Vol. 2019 (1). European Association of Geoscientists and Engineers, (2019), pp. 1–5.

[8] Kochkov, D., Smith, J., Alieva, A., Wang, Q., Brenner, M., and Hoyer, S. ‘Machine Learning–Accelerated Computational Fluid Dynamics’. In: *Proceedings of the National Academy of Sciences* vol. 118(21) (2021), pp. 89–97.

[9] Lions, J.-L., Maday, Y., and Turinici, G. ‘A “Parareal” in Time Discretization of Pde’s’. In: *Comptes Rendus de l’Académie des Sciences. Série I. Mathématique* vol. 332(7) (2001), pp. 34–76.

[10] Hutter, F., Hoos, H., and Leyton-Brown, K. ‘An Efficient Approach for Assessing Hyperparameter Importance’. In: *Proceedings of the 31st International Conference on Machine Learning*. Vol. 32 (1). Proceedings of Machine Learning Research. PMLR, (2014), pp. 754–762.

[11] Krishnapriyan, A., Gholami, A., Zhe, S., Kirby, R., and Mahoney, M. W. 'Characterizing Possible Failure Modes in Physics-Informed Neural Networks'. In: *Advances in Neural Information Processing Systems* vol. 34(1) (2021).

[12] Ibrahim, A. Q., Götschel, S., and Ruprecht, D. 'Parareal with a Physics-Informed Neural Network as Coarse Propagator'. In: *Euro-Par 2023: Parallel Processing.* Springer Nature Switzerland, (2023), pp. 649–663. ISBN: 978-3-031-39698-4.

[13] Nguyen, H. 'Parallel-in-time Methods for Wave Propagation in Heterogeneous Media'. PhD Thesis. The University of Texas at Austin, (2020).

[14] Nguyen, H. and Tsai, R. 'A Stable Parareal-Like Method for the Second Order Wave Equation'. In: *Journal of Computational Physics* vol. 405(1) (2020).

[15] Evans, L. C. *Partial Differential Equations.* 1st ed. Vol. 19, Graduate Studies in Mathematics. American Mathematical Society, (1998). ISBN: 0821807722.

[16] Rocha, D. and Sava, P. 'Elastic Least-Squares Reverse Time Migration Using the Energy Norm'. In: *Geophysics* vol. 83(3) (2018), pp. 5MJ–Z13.

[17] Rocha, D., Sava, P., Shragge, J., and Witten, B. '3D passive Wavefield Imaging using the Energy Norm'. In: *Geophysics* vol. 84(2) (2019), pp. 1MA–Z11.

[18] Rocha, D., Tanushev, N., and Sava, P. 'Acoustic Wavefield Imaging using the Energy Norm'. In: *Geophysics* vol. 81(4) (2016), pp. 1JA–Z38.

[19] Berenger, J.-P. 'A perfectly matched layer for the absorption of electromagnetic waves'. In: *Journal of Computational Physics* vol. 114(2) (1994), pp. 185–200.

[20] Engquist, B. and Majda, A. 'Absorbing Boundary Conditions for Numerical Simulation of Waves'. In: *Proceedings of the National Academy of Sciences* vol. 74(5) (1977), pp. 1765–1766.

[21] Higdon, R. L. 'Absorbing Boundary Conditions for Difference Approximations to the Multi-Dimensional Wave Equation'. In: *Mathematics of Computation* vol. 47(176) (1986), pp. 437–459.

[22] Verlet, L. 'Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules'. In: *Physical Review Journals Archive* vol. 159(1) (1967), pp. 98–103.

[23] Runge, C. 'Ueber die Numerische Aufloesung von Differentialgleichungen'. In: *Mathematische Annalen* vol. 46(1) (1895), pp. 167–178.

[24] Kutta, W. 'Beitrag zur Naeherungsweisen Integration Totaler Differentialgleichungen'. In: *Zeitschrift für Mathematik und Physik* vol. 46(1) (1901), pp. 435–453.

[25] Courant, R., Friedrichs, K., and Lewy, H. ‘Über die Partiellen Differenzengleichungen der Mathematischen Physik’. In: *Mathematische Annalen* vol. 100(1) (1928), pp. 32–74.

[26] Isaacson, E. and Keller, H. B. *Analysis of Numerical Methods*. Revised Edition. Dover Publications, (1994). ISBN: 978-0486680293.

[27] Fotiadis, S., Pignatelli, E., Valencia, M. L., Cantwell, C., Storkey, A., and Bharath, A. A. ‘Comparing Recurrent and Convolutional Neural Networks for Predicting Wave Propagation’. ArXiv eprint at abs/2002.08981. (2020).

[28] Sorteberg, W. E., Garasto, S., Pouplin, A. S., Cantwell, C. D., and Bharath, A. A. ‘Approximating the Solution to Wave Propagation using Deep Neural Networks’. ArXiv eprint at abs/1812.01609. (2018).

[29] Deoa, I. K. and Jaimanb, R. ‘Predicting Waves in Fluids with Deep Neural Network’. In: *Physics of Fluids* vol. 34(6) (2022), pp. 67–108.

[30] Raghu, M. and Schmidt, E. ‘A Survey of Deep Learning for Scientific Discovery’. ArXiv eprint at abs/2003.11755. (2020).

[31] O’Shea, K. and Nash, R. ‘An Introduction to Convolutional Neural Networks’. ArXiv eprint at abs/1511.08458. (2015).

[32] Alzubaidi, L., Zhang, J., and Humaidi, A. J. ‘Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions’. In: *Journal of Big Data* vol. 8(1) (2021), pp. 1–74.

[33] Tsai, R. ‘Mathematics in Deep Learning’. Lecture Series at The University of Texas at Austin, Fall 2021, Department of Mathematics. (2021).

[34] Neutelings, I. ‘Neural Network Latex Drawings’. https://tikz.net/neural_networks/. (2020).

[35] He, K., Zhang, X., Ren, S., and Sun, J. ‘Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification’. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE Computer Society, (2015), pp. 1026–1034.

[36] Hanin, B. and Rolnick, D. ‘How to Start Training: The Effect of Initialization and Architecture’. In: *NIPS’18: Proceedings or the 32nd International Conference on Neural Information Processing Systems* (2018), pp. 569–579.

[37] Sharma, S., Sharma, S., and Athaiya, A. ‘Activation Functions in Neural Networks’. In: *International Journal of Engineering Applied Sciences and Technology* vol. 4(12) (2020), pp. 310–316.

[38] Krähenbühl, P. ‘Neural Networks’. Lecture Series at The University of Texas at Austin, Fall 2021, Department of Computer Science; http://www.philkr.net/. (2021).

[39] Kingma, D. P. and Ba, J. ‘Adam: A Method for Stochastic Optimization’. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* ArXiv eprint at abs/1412.6980. (2014).

[40] Ruder, S. ‘An Overview of Gradient Descent Optimization Algorithms’. ArXiv eprint at abs/1609.04747. (2016).

[41] Hecht-Nielsen, R. In: *Neural Networks for Perception.* Academic Press, (1992). Chap. III.3 - Theory of the Backpropagation Neural Network, pp. 65–93. ISBN: 978-0-12-741252-8.

[42] Strutz, D. ‘Illustrating (Convolutional) Neural Networks in LaTeX with TikZ’. https://davidstutz.de/illustrating-convolutional-neural-networks-in-latex-with-tikz/. (2020).

[43] Sellami, A. and Hwang, H. ‘A Robust Deep Convolutional Neural Network with Batch-Weighted Loss for Heartbeat Classification’. In: *Expert Systems with Applications* vol. 122(1) (2019), pp. 75–84.

[44] Phan, H., Krawczyk-Becker, M., Gerkmann, T., and Mertins, A. ‘DNN and CNN with weighted and Multi-task Loss Function for Audio Event Detection’. ArXiv eprint at abs/1708.03211. (2017).

[45] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. ‘Dropout: A Simple Way to Prevent Neural Networks from Overfitting’. In: *Journal of Machine Learning Research* vol. 15(56) (2014), pp. 1929–1958.

[46] Ioffe, S. and Szegedy, C. ‘Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift’. In: *International Conference on Machine Learning.* ArXiv eprint at abs/1502.03167. (2015).

[47] Yang, S., Xiao, W., Zhang, M., Guo, S., Zhao, J., and Shen, F. ‘Image Data Augmentation for Deep Learning: A Survey’. ArXiv eprint at abs/2204.08610. 2022.

[48] He, J. and Ward R. Tsai R. ‘Deep Residual Learning for Image Recognition’. In: *Research in the Mathematical Sciences* vol. 10(1) (2023), p. 13.

[49] Loshchilov, I. and Hutter, F. ‘Fixing Weight Decay Regularization in Adam’. ICLR 2018 Conference Submission, ArXiv eprint at abs/1711.05101. (2018).

[50] Barron, A. R. ‘Approximation and Estimation Bounds for Artificial Neural Networks’. In: *Machine Learning* vol. 14(1) (1994), pp. 115–133.

[51] Barron, A. R. ‘Neural Net Approximation’. In: *Proceedings 7th Yale Workshop on Adaptive and Learning Systems*. Center for Systems Science, (1992), pp. 69–72.

[52] Yarotsky, D. ‘Error Bounds for Approximations with Deep ReLU Networks’. In: *Neural Networks* vol. 94(1) (2017), pp. 103–114.

[53] Adams, R. and Fournier, J. *Sobolev Spaces*. 2nd ed. Academic Press, (2003). ISBN: 9780120441433.

[54] Cucker, F. and Zhou, D.-X. ‘Learning Theory: An Approximation Theory Viewpoint’. In: 24th ed. Part of Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2007, pp. 1–4. ISBN: 978-0521865593.

[55] Smale, S. and Zhou, D.-X. ‘Estimating the Approximation Error in Learning Theory’. In: *Analysis and Applications* vol. 1(1) (2003), pp. 1–25.

[56] Zhou, D.-X. ‘Universality of Deep Convolutional Neural Networks’. In: *Applied and Computational Harmonic Analysis* vol. 48(2) (2018), pp. 787–794.

[57] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. ‘Striving for Simplicity: The All Convolutional Net’. 3rd International Conference on Learning Representations (ICLR), ArXiv eprint at abs/1412.6806. (2015).

[58] He, K., Zhang, X., Ren, S., and Sun, J. ‘Deep Residual Learning for Image Recognition’. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ArXiv eprint at abs/1512.03385. (2015), pp. 770–778.

[59] Veit, A., Wilber, M., and Belongie, S. ‘Residual Networks Behave Like Ensembles of Relatively Shallow Networks’. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. ArXiv eprint at abs/1605.06431. (2016), pp. 550–558.

[60] Long, J., Shelhamer, E., and Darrell, T. ‘Fully Convolutional Networks for Semantic Segmentation’. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Proceedings*. (2015), pp. 3431–3440.

[61] Ariel, G., Nguyen, H., and Tsai, R. ‘θ – Parareal Schemes’. ArXiv eprint at abs/1704.06882. (2018).

[62] Bal, G. ‘On the Convergence and the Stability of the Parareal Algorithm to Solve Partial Differential Equations’. In: *Domain Decomposition Methods in Science and Engineering*. Vol. 40. 1. Springer, Berlin, Heidelberg, (2005), pp. 425–432.

[63] Gander, M. and Vandewalle, S. ‘Analysis of the Parareal Time-Parallel Time-Integration Method’. In: *SIAM J. Scientific Computing* vol. 29(2) (2007), pp. 556–578.

[64] Sun, H. and Demanet, L. In: *SEG Technical Program Expanded Abstracts 2018, Proceedings.* Society of Exploration Geophysicists, (2018). Chap. Low-Frequency Extrapolation with Deep Learning, pp. 2011–2015.

[65] Brougois, A., Bourget, M., Lailly, P., Poulet, M., Ricarte, P., and Versteeg, R. 'Marmousi, model and data'. In: *Conference: EAEG Workshop - Practical Aspects of Seismic Data Inversion.* (1990).

[66] Billette, F. and Brandsberg-Dahl, S. 'The 2004 BP Velocity Benchmark'. In: *European Association of Geoscientists Engineers.* 67th EAGE Conference Exhibition, (2005).

[67] Yang, W., Zhang, X., Tian, Y., Wang, W., Xue, J.-H., and Liao, Q. 'Deep Learning for Single Image Super-Resolution: A Brief Review'. In: *IEEE Transactions on Multimedia* vol. 21(12) (2019), pp. 3106–3121.

[68] Lu, Z., Li, J., Liu, H., Huang, C., Zhang, L., and Zeng, T. 'Transformer for Single Image Super-Resolution'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops.* IEEE Computer Society, (2022), pp. 456–465.

[69] Galliani, S., Lanaras, C., Marmanis, D., Baltsavias, E., and Schindler, K. 'Learned Spectral Super-Resolution'. ArXiv eprint at abs/1703.09470. (2017).

[70] Odena, A., Dumoulin, V., and Olah, C. 'Deconvolution and Checkerboard Artifacts'. In: *Distill* vol. 1(10) (2016), pp. e3.

[71] He, K., Zhang, X., Ren, S., and Sun, J. 'Deep Residual Learning for Image Recognition'. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Proceedings* (2015), pp. 770–778.

[72] He, K., Zhang, X., Ren, S., and Sun, J. 'Identity Mappings in Deep Residual Networks'. In: *Computer Vision – ECCV 2016, Proceedings.* Springer International Publishing, (2016), pp. 630–645.

[73] Jégou, S., Drozdzal, M., Vazquez, D., Romero, A., and Bengio, Y. 'The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation'. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW).* IEEE Computer Society, Proceedings, (2017), pp. 1175–1183.

[74] Petit, O., Thome, N., Rambour, C., and Soler, L. 'U-Net Transformer: Self and Cross Attention for Medical Image Segmentation'. In: *Machine Learning in Medical Imaging.* Springer International Publishing, (2021), pp. 267–276. ISBN: 978-3-030-87589-3.

[75] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. 'Attention Is All You Need'. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems* vol. 30(1) (2017), 6000–6010.

[76] You, K., Long M., Jordan M. I., and Wang J. 'Learning Stages: Phenomenon, Root Cause, Mechanism Hypothesis, and Implications'. ArXiv eprint at abs/1908.01878. (2019).

[77] Lautenschlager, F., Becker, M., Kobs, K., Steininger, M., Davidson, P., Krause, A., and Hotho, A. 'OpenLUR: Off-the-shelf air pollution modeling with open features and machine learning'. In: *Atmospheric Environment* vol. 233(1) (2020).

[78] Lewkowycz, A. 'How to Decay your Learning Rate'. ArXiv eprint at abs/2103.12682, Under review as a conference paper to ICLR 2022. (2021).

[79] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. 'On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima'. In: *5th International Conference on Learning Representations (ICLR)* (2017). ArXiv eprint at abs/1609.04836.

[80] Kaiser, L. 'GitHub Repository with Code'. github.com/utkaiser/masterthesis_notebooks. (2023).

[81] Hooker, G. 'Generalized Functional ANOVA Diagnostics for High-Dimensional Functions of Dependent Variables'. In: *Journal of Computational and Graphical Statistics* vol. 16(3) (2007), pp. 709–732.

[82] Li, H., Xu, Z., Taylor, G., and Goldstein, T. 'Visualizing the Loss Landscape of Neural Nets'. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Curran Associates Inc., (2018), pp. 6391–6401.

[83] Quarteroni, A., Sacco, R., and Saleri, F. *Numerical Mathematics (Texts in Applied Mathematics 37)*. 2nd ed. Springer, (2006). ISBN: 978-3540346586.

[84] Nussbaumer, H. J. 'The Fast Fourier Transform'. In: *Fast Fourier Transform and Convolution Algorithms*. Springer Berlin Heidelberg, (982), pp. 0–111. ISBN: 978-3642818974.

[85] William, H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C: The Art of Scientific Computing*. 3rd ed. Cambridge University Press, (1992). ISBN: 0521880688.

Acronyms

$\mathcal{F}_{\Delta t^{\star}}$ fine solver. 3, 10, 11, 12, 13, 14, 21, 29, 32, 47, 48

$\mathcal{G}_{\Delta t^{\star}}$ coarse solver. 2, 10, 11, 12, 14, 29, 31, 37, 40, 42, 49

CFL Courant, Friedrichs, and Lewy. 14, 15, 35, 36

CNN Convolutional Neural Network. 4, 5, 24, 25, 26, 27

DFT Discrete Fourier Transform. 63, 64

E2E-Trans Transformer JNet. 38, 43, 47

E2E-Tira Tiramisu JNet. 38, 43, 47

E2E-JNet5 E2E 5-level JNet. 38, 46, 47

E2E-CNN-UNet3 E2E CNN 3-level UNet. 38, 39, 43

E2E-JNet3 E2E 3-level JNet. 38, 39, 40, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53

E2E-V E2E Vanilla. 31, 38, 40, 44, 47, 48

fANOVA functional Analysis of Variance. 51, 52

MSE Mean Squared Error. 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 52

NE2E-JNet3 Not E2E 3-level JNet. 32, 38, 40, 42, 43, 44, 45, 46, 47, 49

PDE Partial Differential Equation. 4, 5, 6, 12, 14, 15, 46

PINN Physics-Informed Neural Network. 4, 5

RK4 Runge-Kutta of forth-order method. 13

Appendix

The exact implementations, e.g., the specifications of both solvers and how we handled boundaries numerically, can be found in a GitHub repository [80]. The following equations are taken from [83].

A Fourier Transform and Fast Fourier Transform

The Fourier Transform and the Fast Fourier Transform are mathematical techniques used to transform a signal between the time or spatial domain and frequency domain. For a continuous signal in the time domain, $x(t)$, the Fourier Transform is expressed as:

$$\mathcal{X}(f) = \int_{-\infty}^{\infty} x(t) e^{j2\pi ft} dt. \tag{6.1}$$

In this definition, $\mathcal{X}(f)$ represents the signal's frequency domain counterpart, while f denotes frequency, and j is the imaginary unit. The inverse process, i.e. transforming a frequency domain signal back to its time domain form, is given by:

$$x(t) = \int_{-\infty}^{\infty} \mathcal{X}(f) e^{j2\pi ft} df. \tag{6.2}$$

The Fast Fourier Transform is a highly efficient method for calculating the Discrete Fourier Transform (DFT) and its reverse. For a discrete signal in the time domain $x[n]$, where $n = 0, 1, \ldots, N-1$, the DFT is defined as:

$$\mathcal{X}[k] = \sum_{n=0}^{N-1} x[n] e^{j2\pi \frac{kn}{N}}. \tag{6.3}$$

Here, $\mathcal{X}[k]$ denotes the signal's representation in the frequency domain, while k is indicative of discrete frequency intervals, and N is the total number of samples. The Fast Fourier Transform reduces the computational complexity of DFT from $O(N^2)$ to $O(N \log(N))$, making it practical for a wide range of applications. It does this primarily through a divide-and-conquer approach known as the Cooley-Tukey algorithm, which recursively decomposes the DFT into smaller components. For detailed information, refer to [84].

B Taylor Approximation

A Taylor series approximates a function to a certain degree though a series of derivatives at a specific point. For a function $f(x)$ and a point a, the Taylor series up to the n-th term is given by:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ \cdots \ + \frac{f^n(a)}{n!}(x-a)^n. \tag{6.4}$$

The accuracy of this approximation increases with the number of terms used.

C Central Differencing

Central differencing is a numerical method used to approximate partial derivatives. For the 2D case, the central difference of a function u at a point $(x_{i,j}) := (x_i, x_j) = (ih, jh)$ with a small interval h is given by:

$$\nabla_h u(x_{i,j}) = (\frac{u(x_{i+1,j}) - u(x_{i-1,j})}{2h}, \frac{u(x_{i,j+1}) - u(x_{i,j-1})}{2h}). \tag{6.5}$$

This method calculates the gradient of u at $(x_{i,j})$ by averaging the differences in both directions, providing a more precise approximation for multi-dimensional functions than one-sided differencing methods.

D Bilinear Interpolation

To estimate the value of a function at a point (x, y) within a rectangular grid, we use bilinear interpolation based on [85]. It is widely used in image scaling as it provides smooth transitions between data points while being computationally efficient. However, for our purposes, it is not optimal because we aim to approximate functions with high curvature or rapid changes, whereas bilinear interpolation assumes linear variation between data points.

Mathematically, we perform linear interpolation first in one direction (e.g., horizontally) and then in the other direction (e.g., vertically). Assume we have a rectangular grid with points (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2), and their corresponding function values f_{11},

f_{12}, f_{21}, and f_{22}. We can now perform two linear interpolations along the x-axis

$$\begin{aligned} f(x, y_1) &= \frac{x_2 - x}{x_2 - x_1} f_{11} + \frac{x - x_1}{x_2 - x_1} f_{21}, \\ f(x, y_2) &= \frac{x_2 - x}{x_2 - x_1} f_{12} + \frac{x - x_1}{x_2 - x_1} f_{22}, \end{aligned} \tag{6.6}$$

and continue by interpolating in the y-direction

$$f(x, y) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2). \tag{6.7}$$

E Pseudocode

This section serves as an introduction on how to use the code base in [80]. The mathematics behind complex algorithms, such as the numerical solvers, are explained in Chapter 3. Please refer to [80] for specific implementations in Python. Here, we explain how to use these solvers to propagate a wave (Subsection E.1), generate training data (Subsection E.2), and train the end-to-end models (Subsection E.3). In Algorithm 1, we show how to apply the Parareal scheme.

E.1 Application of the Numerical Solvers

```
from utils_use_numerical_solver import get_velocity_model, pseudo_spectral_tensor,
    velocity_verlet_tensor, init_pulse_gaussian, WaveEnergyField_tensor
import matplotlib.pyplot as plt
import torch

def visualize_numerical_solver_periodic(
        vel_data_path = "data/crop_test.npz",
        method = "pseudo-spectral",
        dx = 2./128.,
        dt = 1/600.,
        dt_star = .06
):
    '''
    Parameters
    ----------
    vel_data_path : (string) path to velocity profile crops
    method : (string) "pseudo-spectral" or "velocity-verlet"
    dx : (float) spatial step size numerical solver
    dt : (float) temporal step size numerical solver
    dt_star : (float) time interval the solver is applied once

    Returns
    -------
    (void) visualizes 8 advancements of timestep dt_star
    with periodic boundary conditions
    '''

    vel = torch.from_numpy(get_velocity_model(vel_data_path))

    # computing initial condition using gaussian pulse
    u, ut = init_pulse_gaussian(width=7000, padding=128, x_1=0, x_2=0)
    u, ut = torch.from_numpy(u), torch.from_numpy(ut)

    for s in range(8):
        # run one iteration of the RK4 pseudo-spectral
          / velocity Verlet method for time dt_star and increments dx, dt
        if method == "pseudo-spectral":
            u, ut = pseudo_spectral_tensor(u, ut, vel, dx, dt, dt_star)
        else:  # method == "velocity_verlet"
            u, ut = velocity_verlet_tensor(u, ut, vel, dx, dt, dt_star)

        # change representation to energy semi-norm
        w = WaveEnergyField_tensor(u, ut, vel, dx)

        # visualize results
        plt.imshow(w)
        plt.title(f"wave field for iteration {s}")
        plt.show()
```

E.2 Generate Training Data

Generate Velocity Crops

```
from utils_generating_data import generate_velocity_profile_crop
from skimage.filters import gaussian
from scipy.io import loadmat

def generate_velocity_crops(
        resolution = 128,
        output_dir = 'data/crop_test.npz',
        num_crops = 1,
):
    '''
    Parameters
    ----------
    resolution : (int) target resolution of crop
    output_dir : (string) output file path, format is ".npz"
    num_crops :  (int) number of crops per image

    Returns
    -------
    (void) saves the velocity crops in an .npz-file
    '''

    # load images
    data_mat = loadmat('data/marm1nonsmooth.mat')  # Marmousi velocity image
    modified_marm = gaussian(data_mat['marm1larg'],4)  # smoothing the image
    data_bp = loadmat('data/bp2004.mat')  # BP velocity image
    # smoothing the image and different order of magnitude
    modified_bp = gaussian(data_bp['V'],4)/1000

    # randomly crop and save images at "output_dir"
    generate_velocity_profile_crop(
        v_images = [modified_marm,modified_bp],
        m = resolution,
        output_path = output_dir,
        num_times = num_crops
    )
```

Create Training Dataset

```
from matplotlib import pyplot as plt
import numpy as np
import torch
from utils_generating_data import crop_center, initial_condition_gaussian,
    one_iteration_pseudo_spectral_tensor
from wave_component_function import WaveSol_from_EnergyComponent_tensor,
    WaveEnergyComponentField_end_to_end, WaveEnergyField

def generate_data_end_to_end(
        input_path = "data/crop_test.npz",
        output_path = "data/datagen_test.npz",
        boundary_condition = "periodic",
        n_snaps = 10,
        res = 128,
        n_it = 10,
        f_delta_x =  2. / 128.
):
    '''
    Parameters
    ----------
    input_path : (string) velocity profile data path
    output_path: (string) generated wave field data path
    boundary_condition : (string) either "periodic" or "absorbing"
    n_snaps : (int) amount of snapshots / dt_star steps to take
    res : (int) resolution of wave field output
    n_it : (int) amount of different wave propagation series
    f_delta_x : (float) grid time stepping of fine solver

    Returns
    -------
    (void) saves generated wave propagation iterations in file
    '''

    # load velocity model created in function above ‘generate_velocity_crops()‘
    velocities = np.load(input_path)['wavespeedlist']

    # set up tensors to store wave energy components and velocity profile
    # goal: save tensor for each iteration and snapshot
    Ux, Uy, Utc = np.zeros([n_it, n_snaps + 1, res, res]), \
                  np.zeros([n_it, n_snaps + 1, res, res]), \
                  np.zeros([n_it, n_snaps + 1, res, res])
    V = np.zeros([n_it, n_snaps+1, res, res])

    # data generation
    for it in range(n_it):
        # sample velocity instance
        if it >= len(velocities): vel = velocities[0]
        else: vel = velocities[it]

        # computing initial condition using gaussian pulse u_energy[b] relates to
        #  the partial derivatives, while b denotes the batch size (used later)
        u_energy = initial_condition_gaussian(
            torch.from_numpy(vel),
            mode="energy_comp",
            res_padded=res,  # needed to account for absorbing boundaries
        )

        # create and save velocity crop
        # crop center of the image if absorbing boundaries
        vel_crop = crop_center(vel, res, 2)
        # save velocity image (n_snaps + 1) times in V
        V[it] = np.repeat(vel[np.newaxis, :, :], n_snaps + 1, axis=0)

        # integrate dt_star (step size) for n_snaps times
        for s in range(n_snaps+1):
            # change energy components to wave field representation
            u_elapse, ut_elapse = WaveSol_from_EnergyComponent_tensor(
                u_energy[:,0], u_energy[:,1], u_energy[:,2],
                torch.from_numpy(vel),
                f_delta_x,
                torch.sum(torch.sum(torch.sum(u_energy[:,0])))
            )

            if boundary_condition == "absorbing":
                # crop and save current snapshot in tensors
                u_elapse_crop = crop_center(u_elapse.squeeze(), res, 2)
                ut_elapse_crop = crop_center(ut_elapse.squeeze(), res, 2)
                Ux[it, s], Uy[it, s], Utc[it, s] = \
                    WaveEnergyComponentField_end_to_end(u_elapse_crop, \
                    ut_elapse_crop, vel_crop, f_delta_x)

            else:  # boundary_condition == "periodic"
                # save current snapshot in tensors
                Ux[it, s], Uy[it, s], Utc[it, s] = \
                    u_energy[0,0], u_energy[0,1], u_energy[0,2]

            # itegration step (done for all iterations but for the last one)
            if s < n_snaps + 1:
                # apply the fine solver
                u_energy = one_iteration_pseudo_spectral_tensor(torch.cat([u_energy, \
                    torch.from_numpy(vel).unsqueeze(dim=0).unsqueeze(dim=0)], dim=1))

# save tensors in file, accessible through key-value queries (dictionary)
np.savez(output_path, vel=V, Ux=Ux, Uy=Uy, Utc=Utc)
```

E.3 Train End-to-End Model

```
import torch
from utils_training_model import save_model, get_params,
    fetch_data_end_to_end, Model_end_to_end
import random
import numpy as np

def train_model(
        model_name = "test",
        lr = .001,
        batch_size = 1,
        n_epochs = 10,
        downsampling_model = "Interpolation",
        upsampling_model = "UNet3",
        data_paths = "data/datagen_test.npz",
        val_paths = "data/datagen_test2.npz"
):
    '''
    Parameters
    ----------
    model_name : (string) name of model
                 used as name for output-file containing model parameters
    lr : (float) learning rate of model
    batch_size : (int) batch size
    n_epochs : (int) amount of epochs model is trained
    downsampling_model : (string) name of downsampling model
    upsampling_model : (string) name of upsampling model
    res_scaler : (int) scale the model downsamples the input
    model_res : (int) resolution of model
    data_paths : (string) training data path
    val_paths : (string) validation data path

    Returns
    -------
    (void) trained model parameters in a ".pt"-file
    '''

    # model setup
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    param_dict = get_params()  # contains all model specifications
    model = Model_end_to_end(param_dict, \
        downsampling_model, upsampling_model).double()
```

```
model = torch.nn.DataParallel(model).to(device)  # multi-GPU use

# data setup
train_loader, val_loader, _ = fetch_data_end_to_end( \
    [data_paths], batch_size, [val_paths])

# deep learning setup (optimizer and loss function)
optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
loss_f = torch.nn.MSELoss()

for epoch in range(n_epochs):

    # training
    model.train()  # allow modification of weights, track gradients
    train_loss_list = []  # list to store loss values of training

    for i, data in enumerate(train_loader):
        loss_list = []  # tmp for backpropagating multiple losses
        n_snaps = data[0].shape[1]  # number of snapshots
        data = data[0].to(device)  # use GPUs if available

        # choose n_snaps start indices randomly
        for input_idx in random.choices(range(n_snaps - 2), k=n_snaps):
            # detach because if not, computation graph too far back
            input_tensor = data[:, input_idx].detach()

            # one-step loss function (this loop is used to show that we can
            # use something else for "input_idx+2" to get a multi-step loss
            for label_idx in range(input_idx + 1, input_idx + 2):
                output = model(input_tensor)  # apply end-to-end model
                loss_list.append(loss_f(output, data[:, label_idx, :3]))
                # save current result to use for next iteration if multi-step-loss
                input_tensor = torch.cat((output, \
                    input_tensor[:, 3].unsqueeze(dim=1)), dim=1)

        # optimizer stepping
        optimizer.zero_grad()
        sum(loss_list).backward()
        optimizer.step()
        # save loss to later print out
        train_loss_list.append( \
            np.array([l.cpu().detach().numpy() for l in loss_list]).mean())
    # validation
    model.eval()  # disallow modification of weights
    with torch.no_grad():  # do not track gradients
        val_loss_list = []  # initialize list to save losses

        for i, data in enumerate(val_loader):
            n_snaps = data[0].shape[1]  # number of snapshots
            data = data[0].to(device)  # use GPUs if available
            input_tensor = data[:, 0]
            vel = input_tensor[:, 3].unsqueeze(dim=1)

            # advance a wave field for (n_snaps - 1) time steps
            for label_idx in range(1, n_snaps):
                label = data[:, label_idx, :3]
                output = model(input_tensor)  # apply end-to-end model
                # get and save loss (this could be optimized by using a metric)
                val_loss_list.append(loss_f(output, label).item())
                # save current result to use for next iteration if multi-step-loss
                input_tensor = torch.cat((output, vel), dim=1)

    print(f'epoch %d, train loss: %.5f, test loss: %.5f'
          %(epoch + 1, np.array(train_loss_list).mean(),
         np.array(val_loss_list).mean()))

save_model(model, model_name, "results/") # save model parameters as a ".pt"-file
```

Declaration of Authorship

I hereby confirm that my thesis entitled ‘Fast, Accurate, and Scalable Numerical Wave Propagation: Enhancement by Deep Learning’ is the result of my own work. I did not receive any help or support from commercial consultants. All sources and / or materials applied are listed and specified in the thesis. Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

Wuerzburg, 12.01.2024

[illegible]

Luis Kaiser