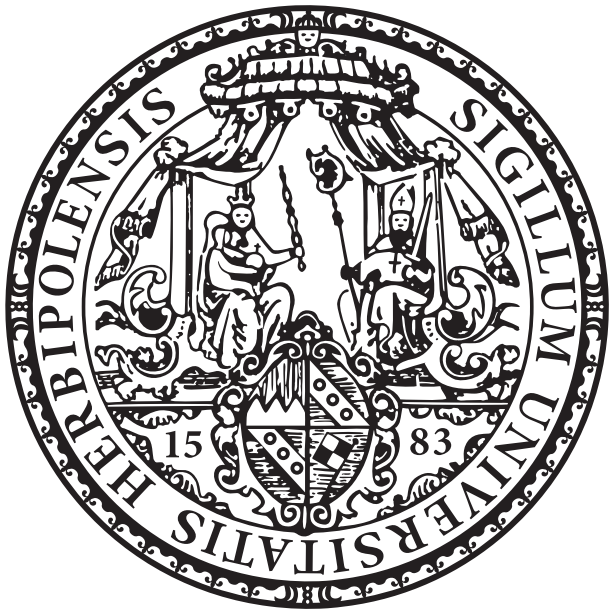
Enhancement of Numerical Wave Propagation Through Deep Learning

A Masterthesis submitted by

Miriam Schönleben

to the Institute of Mathematics
Julius-Maximilians-Universität Würzburg

for the degree of
Master of Science
in the subject of mathematics



Würzburg, August 2025

Supervisor: Prof. Dr. Christian Klingenberg

Acknowledgments

During the work on this thesis I received a lot of support and assistance.

I would like to express my gratitude to my supervisor, Prof. Dr. Christian Klingenberg (University of Wuerzburg), for his guidance, expertise and feedback throughout the course of this thesis.

I also extend my sincere thanks to the members of his research group for their feedback and support during the work on this thesis.

I also express my gratitude to Prof. Yen-Hsi Richard Tsai (University of Texas at Austin) for his expertise and valuable input.

Summary

The propagation of a wave field in heterogeneous media is a main component in many scientific and engineering areas. It is, for example, used to model seismic and acoustic waves and plays an important role in as medical imaging.
Therefore it is desired to compute the numerical solution of the wave equation fast and accurate. Traditional approaches use fine spatial and time discretizations to achieve a high accuracy to the true solution of a propagated wave field. This, however, results in high computational costs when propagating a wave field.

Recent studies [18] have introduced a supervised deep learning framework to enhance the numerical computation of wave propagation. This work builds upon the approach of Nguyen and Tsai [18] from 2023. Their framework uses a numerical fine solver to generate data for the training of a neural network, which is used to improve the accuracy of a coarse solver, together with the parareal algorithm [13]. By combining these two methods they achieve both, a high accuracy as well as a speedup through time parallelization.

This thesis generalizes the approach of Nguyen and Tsai [18] to three spatial dimensions and additionally to a slightly different partial differential equation, the wave equation with variable coefficients in two spatial dimensions. This thesis works as a proof of concept, that these generalizations can be computed with both, high accuracy and time efficiency.
To show this, we present the mentioned approach with focus on the supervised deep learning architecture. Furthermore, we present the results of a numerical study regarding the accuracy and efficiency of this framework for the mentioned generalizations.

Our results show that the approach can be generalized and yields good results for both considered partial differential equations. However, we conclude, that it is important to combine the neural network output with a parareal scheme to approximate the fine solution quite well. Furthermore our results indicate that the choice of the training data set is important. While the approach performs well for the three dimensional wave equation with a neural network trained on parareal-like training data, our results show, that a simpler set of training data is a better choice for the wave equation with variable coefficients in two spatial dimensions.

Zusammenfassung

Die Berechnung der Fortsetzung eines Wellenfeldes in heterogenen Medien ist ein wichtiger Bestandteil vieler wissenschaftlicher und technischer Bereiche, beispielsweise um seismische und akustische Wellen zu modellieren. Außerdem spielt sie eine wichtige Rolle in medizinischen Bildgebungsverfahren. Deshalb benötigt man Methoden, um die numerische Lösung der Wellengleichung schnell und akkurat zu berechnen. Traditionelle Verfahren nutzen eine feine räumliche und zeitliche Diskretisierung um eine hohe Genauigkeit im Vergleich zur wahren Lösung einer fortgesetzten Welle zu erreichen, was jedoch in hohem Rechenaufwand resultiert.

Neueste Forschungen [18] haben einen supervised Deep-Learning Ansatz präsentiert, der die numerische Lösung der Fortsetzung eines Wellenfeldes verbessert. Diese Thesis baut auf dem Ansatz von Nguyen und Tsai [18] aus dem Jahr 2023 auf, die einen feinen numerischen Löser nutzen um Trainingsdaten für ein neuronales Netz zu generieren, das die Genauigkeit eines groben Lösers verbessert. Danach wird dieses neuronale Netz mit einer zeitlichen Parallelisierung, dem Parareal Algorithmus [13], kombiniert, wodurch sie sowohl eine gute Genauigkeit als auch eine effizientere Berechnung erreichen.

Diese Thesis verallgemeinert den Ansatz von Nguyen und Tsai [18] auf drei Raumdimensionen, sowie auf eine leicht abgewandelte partielle Differentialgleichung, die Wellengleichung mit variablen Koeffizienten in zwei Raumdimensionen. Diese Arbeit dient als "Proof of Concept" dafür, dass diese Verallgemeinerungen in ähnlicher Weise sowohl numerisch genau als auch effizient gelöst werden können. Um dies zu zeigen, präsentieren wir den erwähnten Ansatz mit Fokus auf der Architektur des neuronalen Netzes. Weiterhin präsentieren wir die Ergebnisse einer numerischen Untersuchung bezüglich der Genauigkeit und Effizienz dieses Ansatzes für die betrachteten Verallgemeinerungen.

Aus unseren Ergebnissen schließen wir, dass der Ansatz verallgemeinert werden kann und gute Ergebnisse für die beiden betrachteten partiellen Differntialgleichungen liefert. Wir sehen jedoch, dass es wichtig ist die Ausgabe des neuronalen Netzes mit dem Parareal Algorithmus zu kombinieren um eine gute Approximation an die feine Lösung zu erhalten. Weiterhin zeigen unsere Resultate, dass die Wahl der Trainingsdatenmenge wichtig ist. Während der Ansatz für die drei-dimensionale Wellengleichung gute Ergebnisse liefert, wenn das neuronale Netz auf einer Menge von Daten trainiert wurde, die ähnliche Strukturen haben wie die Ausgaben des Parareal Algorithmus, zeigen unsere Resultate, dass eine einfachere Struktur der Trainingsdaten bessere Ergebnisse für die Wellengleichung mit variablen Koeffizienten liefert.

Contents

1 Introduction

Wave propagation in multiscale media in two and three dimensions occurs as an important part in a wide range of scientific areas with many applications in engineering and research. For example, wave propagation is used for modeling sound waves underwater in marine biology. It is also used to forecast the movement of atmospheric waves in meteorology. In addition, it often occurs as the forward problem when solving inverse problems, like in seismology to determine the origin of a seismic wave or in medical imaging to compute the image of an ultrasound.
Traditional numerical methods for wave propagation usually achieve higher accuracy the finer the spatial and temporal discretization is. Computing the numerical solution with such fine discretizations is computationally more expensive and therefore results in a trade-off between solution accuracy and computation time. The challenges of numerical methods for wave propagation in multiscale media are studied in detail in [23]. A better numerical computation of wave propagation leads directly to a better imaging or forecasting in the corresponding applications of wave propagation. Therefore, there are multiple attempts to obtain a framework, that numerically propagate a wave with both high accuracy and fast overall computation time.

1.1 Related Work

Many existing approaches to improve the efficiency of wave propagation reduce the complexity of the problem by focusing on special features of the global structure, such as the medium. Abdulle et al. [1] use a reduced basis method for the wave equation in heterogeneous media. This method computes a set of basis functions, that span a domain, which closely approximate the original, more complex one. Since these basis functions are medium-depended such models are designed to solve the wave equation on the same medium repeatedly with different initial conditions.
Other approaches uses a so called heterogeneous multiscale method, that combines simulations on microscales and macroscales for the different scales in the heterogeneous media. Engquist et al. showed that this method reduces the complexity significantly, compared to traditional techniques [6].

One main approach in the current research is to improve the accuracy and computation time of traditional numerical solvers by using time parallelization frameworks. Bernacki et al. present a discontinuous Galerkin method for parallelizing a velocity verlet time integra-

tion scheme for the heterogeneous wave equation in three dimensions [3]. In 2000, Lions et al. introduced an algorithm, the parareal algorithm, as a time discretized parallel computation for partial differential equations (PDE) [13]. However, this parareal algorithm can be instable for hyperbolic PDEs. Nguyen and Tsai proposed in 2020 a stabilized parareal scheme for the homogeneous wave equation [17].

Another approach for faster and more accurate wave propagation is to use different kinds of neural networks, such as physics-informed neural networks. This type of neural network respects the laws of physics by including the properties in their training process, to be precise they use a loss function, that accounts for the physical properties of a given PDE. In 2019, Raissi et al. introduced two types of physical-informed neural networks, one for problems with data-driven solution and one for problems with data-driven discovery of PDEs [19]. Another approach, introduced by Moseley et al. [15], applied the physics-informed neural networks specifically to the wave equation in heterogeneous media. Their approach uses a deep learning framework for the wave solution, while considering the wave equation itself and the boundary condition in the loss function.

In recent work, the combination of the parareal algorithm with a neural network was studied. In 2023, Ibrahim et al. trained a physics-informed neural network as the coarse solver in the parareal scheme to improve the efficiency of solving a PDE [8].
Meng et al. presented a parareal scheme by computing the output of a physical-informed neural network in parallel [14].

In 2022, Nguyen and Tsai [18] introduced a parareal scheme for the second order wave equation in two spatial dimensions on heterogeneous media. In this approach, they train a neural network in a supervised deep learning framework on data, generated by a fine solver, and apply this neural network as a correction operator to improve a given coarse solver. By doing so, they achieve an enhanced solver, which is then used together with the fine solver in a parareal scheme.

1.2 Key Contributions

This thesis works as a proof of concept that the approach, that combines a corrector neural network and the parareal algorithm, introduced by Nguyen and Tsai [18], could be generalized to the second order wave equation in three spatial dimensions. For this, we train a neural network as a corrector for a given coarse solver and apply it to compute the wave propagation with the considered approach. Afterwards, we compare the results with the numerical solution computed by the given fine solver.
Furthermore, we study, whether the approach could be generalized to other PDEs. For that, we consider the more complex wave equation with variable coefficients in two spatial dimensions. We train a neural network and apply it to this equation. Afterwards we compare

the results with the numerical solution of the fine solver. Furthermore, when evaluating our results, we also identify the type of training data, that gives the best results, especially in terms of accuracy and size of the training data set.

Our main contributions are:

(i) The generalization of the two dimensional approach for the wave equation introduced by Nguyen and Tsai [18] to three spatial dimensions, by applying the approach to this equation and presenting our numerical results.

(ii) The proof of concept, that the generalization of the approach works for equations different to the standard wave equation. This is shown by applying the approach to the two dimensional wave equation with variable coefficients and presenting our numerical results.

This thesis is structured as follows:
In Chapter 1, we motivate our research and give an overview of the current state of research and discuss other approaches for fast and accurate wave propagation. Chapter 2 gives an introduction to the mentioned second order wave equation in three spatial dimensions and to the wave equation with variable coefficients in two dimensions. In Chapter 3, we propose the general approach and introduce the concept of the energy representation of a wave field. In Chapter 4, we present the stable numerical solvers, which are problem dependent, in more detail and present the stability criterion, that we have used. In Chapter 5, we introduce important terminology in the field of neural networks and present the JNet architecture to the reader. Furthermore we describe how we generate our training data and how the neural network is trained. In Chapter 6, we present the parareal algorithm in more detail, especially by giving the algorithm in pseudo code. Chapter 7 provides a detailed presentation of the studied setups for the wave equation in three dimensions and the wave equation with variable coefficients in two dimensions. This is followed by the numerical results, that we have computed, and an interpretation thereof. Furthermore, we give an analysis about the type of training data, that results in the best performance, especially in terms of accuracy and size of the training data set. In Chapter 8 we give a short summary of this thesis and an outlook to possible future work.

2 Wave Equation Preliminaries

Wave propagation in multiscale media is a very important aspect of many scientific areas including seismology and meteorology and also plays a central role in applications such as medical imaging. It frequently occurs as the forward model in inverse problems. Due to this variety of research fields, there are many mathematical formulations to describe the movement of a wave in space and time. Some of these are more simplified models for the movement of a wave to reduce the computational complexity of wave propagation, whereas other models are more realistic, but also more complex, due to additional terms.
We present the standard wave equation in three spatial dimensions in Section 2.1. Furthermore, in Section 2.2, we present a variation of the wave equation with variable coefficients for two spatial dimensions, which accounts for the complexity of multiscale media by adding additional terms. Moreover, we briefly present the concept of periodic boundary conditions for two and three spatial dimensions in Section 2.3.

2.1 3D Wave Equation

The first equation, that we consider in this thesis, is the standard second order wave equation (2.1) in heterogeneous media, which was also studied by Nguyen and Tsai [18]. However, unlike Nguyen and Tsai, who focus on the two dimensional version of this equation, this work considers the three dimensional case, generalizing the applicability of the approach to more complex wave propagation scenarios.
This equation is given as a linear, hyperbolic second order partial differential equation. It describes how a wave moves and oscillates through a given medium. Therefore the function u describes a wave in space $x \in [-1,1]^3$ and in time $0 \leq t \leq T$. Accordingly, the wave equation is given as

$$\begin{aligned} u_{tt}(x,t) &= c^2(x)\Delta u(x,t) \\ u(x,0) &= u_0(x) \\ u_t(x,0) &= p_0(x) \end{aligned} \tag{2.1}$$

where u_t and u_{tt} denotes the first and second time derivatives of u.
In this equation $c(x) \in [-1,1]^3$ denotes the piecewise smooth wave speed in three spatial dimensions, whose exact values depend on the given heterogeneous medium. Furthermore Δu denotes the Laplace operator of u, which is defined as $\Delta u \coloneqq \sum_{i=1}^{3} \frac{\partial^2 u}{\partial^2 x_i}$ for $x = (x_1, x_2, x_3)$.

The wave equation in (2.1) is given as an initial value problem in time. Therefore, for it to be solvable with a unique solution, it is necessary to have two initial conditions with respect to time, because the PDE is of second order. These initial conditions are given in Equation (2.1) as $u(x,0) = u_0(x)$ and $u_t(x,0) = p_0(x)$, where u_0 and p_0 are functions only in the space variable x.
In our setups we consider the wave equation (2.1) with periodic boundary conditions for the spatial variable x, which are explained in detail in Section 2.3.

2.2 2D Wave Equation with Variable Coefficients

With the second equation, that we consider in this thesis, we generalize the studies from Nguyen and Tsai [18] and ours from equation (2.1) and consider a more complex, but also more physically accurate wave equation in heterogeneous media. This equation provides a more realistic simulation of the wave movement by adding a term, that accounts for the gradient of the wave speed. This term simulates the changes in the wave speed in the heterogeneous medium more accurately than the first equation (2.1).
We consider this second equation in the two dimensional case. Therefore, the wave is given as a function u in time $0 \leq t \leq T$ and in space $x \in [-1,1]^2$. Accordingly, the wave equation with variable coefficients is given as

$$\begin{aligned} u_{tt}(x,t) &= \nabla.(c^2(x)\nabla u(x,t)) \\ u(x,0) &= u_0(x) \\ u_t(x,0) &= p_0(x) \end{aligned} \tag{2.2}$$

As in the first Equation (2.1), u_t and u_{tt} denote the first and second time derivatives of u. Furthermore, the piecewise smooth wave speed is denoted by $c(x) \in [-1,1]^2$, where the exact values depend on the given heterogeneous medium. The nabla operator ∇ is defined as $\nabla = (\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2})$ for $x = (x_1, x_2)$.
The equation (2.2) is given as an initial value problem in time. Therefore it occurs with two initial conditions in time to be uniquely solvable, which are given as $u(x,0) = u_0(x)$ and $u_t(x,0) = p_0(x)$ in (2.2). Here are u_0 and p_0 functions only in the space variable x.
As in the first Equation (2.1), we use periodic boundary conditions for the space variable x.

2.3 Periodic Boundary Conditions

For equation (2.1) and (2.2) to be solvable with a unique solution, we need a boundary condition for the spatial variable x. In general, a boundary condition describes how the solution behaves on the edges of the computational domain. One type of boundary condition, which is often used, is the periodic one. This type of boundary condition models a periodicity on the edges of the spatial domain, meaning that the solution of a PDE repeats its behavior periodically for each spatial dimension.

In our setups, where we consider the spatial variable x only in the interval $[-1, 1]$ for each dimension, the periodic boundary conditions are given by the following equations, depending on whether the spatial domain is two or three dimensional.

In the two dimensional case, i.e. $x = (x_1, x_2) \in [-1, 1]^2$, the periodic boundary conditions are given for every point in time $0 \leq t \leq T$ by

$$\begin{aligned} u((-1, x_2), t) &= u((1, x_2), t) \\ u((x_1, -1), t) &= u((x_1, 1), t) \end{aligned} \tag{2.3}$$

For $x = (x_1, x_2, x_3) \in [-1, 1]^3$ are the periodic boundary conditions in three spatial dimensions given for every time t in $0 \leq t \leq T$ by

$$\begin{aligned} u((-1, x_2, x_3), t) &= u((1, x_2, x_3), t) \\ u((x_1, -1, x_3), t) &= u((x_1, 1, x_3), t) \\ u((x_1, x_2, -1), t) &= u((x_1, x_2, 1), t) \end{aligned} \tag{2.4}$$

This type of boundary condition is used because of its simplifying nature for the computation, but it does not represent the behavior of a wave on the edges of the domain in a realistic way. However, the behavior of the wave movement is modeled realistically on the inside of the computational domain. Therefore, we can compute the propagation of a wave field realistically by using the periodic boundary conditions.

3 General Approach

In this chapter we follow Section 1.1 and parts of Section 1.3 of Nguyen and Tsai [18] and introduce the supervised deep learning approach. We first derive a parareal scheme for wave propagation and then stabilize it by enhancing a given coarse solver with a neural network. Secondly, we introduce a representation of a wave field as energy components, which are used as a way to reduce computation errors. Furthermore, we provide formulas to compute the energy components of a wave field, when given the physical components, and vice versa.

3.1 The Supervised Deep Learning Approach

In our approach we assume that we have given two numerical solvers for the wave equation (2.1) or (2.2), respectively. Both solvers propagate a given wave field $\boldsymbol{u} \coloneqq (u, u_t)$ from time t to time $t + \Delta t^*$ on an also given, heterogeneous wave speed profile, which is denoted by c. We use these two solvers in a supervised deep learning framework, where the trained neural network enhances the propagation of the coarse solver $\mathcal{G}_{\Delta t^*}$. The fine solver $\mathcal{F}_{\Delta t^*}$ serves to generate the training data for this neural network.

First, we have given a coarse solver $\mathcal{G}_{\Delta t^*}$. This solver operates on a coarse grid with spatial step size Δx. Therefore $\mathcal{G}_{\Delta t^*}$ has low computational costs, but may not represent the wave field accurately. We require $\mathcal{G}_{\Delta t^*}$ to be time-reversible and stable.
Second, we have given a fine solver $\mathcal{F}_{\Delta t^*}$, which operates on a grid with a higher resolution than the coarse solver $\mathcal{G}_{\Delta t^*}$. We denote the spatial step size of the fine grid with δx. This solver yields a more accurate approximation to the true solution of the wave field, but has also higher computational costs than the coarse solver due to the fact that the given PDE has to be solved on a higher number of grid points. We require $\mathcal{F}_{\Delta t^*}$ to be stable and to have a sufficiently high accuracy for wave propagation in the considered class of velocity profiles.
Both solvers depend on the given wave field $\boldsymbol{u}$ and on the wave speed profile c, but for brevity of notation we write $\mathcal{G}_{\Delta t^*}(\boldsymbol{u})$ instead of $\mathcal{G}_{\Delta t^*}(\boldsymbol{u}, c)$ and for the fine solver analogously, if c is clear from the context. In Chapter 4, our choice of numerical solvers $\mathcal{G}_{\Delta t^*}$ and $\mathcal{F}_{\Delta t^*}$ is presented in detail. We note that, while both solvers propagate the wave field from time t to $t + \Delta t^*$, the fine solver repeats its stepping scheme more times than the coarse solver to propagate the wave field for one time step of size Δt^* due to the stability requirement for both solvers.

Since we consider two grids, one for the fine solver with a finer resolution and one for the coarse solver with a coarser resolution, we further introduce a restriction operator $\mathcal{R}$ and a prolongation operator $\mathcal{I}$. The restriction operator $\mathcal{R}$ maps functions defined on the fine grid to ones defined on the coarse grid. $\mathcal{R}$ can for example be given as a projection from the fine to the coarse grid. The prolongation operator $\mathcal{I}$ on the other hand maps functions defined on the coarse grid to ones defined on the fine grid. $\mathcal{I}$ could for example be given as an interpolation operator or as in this approach as a neural network. By defining these two operators, we are able to compare both solvers on the same grid.
It is important to note, that $\mathcal{I}\mathcal{R}\boldsymbol{u} \neq \boldsymbol{u}$ in general.

For this approach, we consider the given wave field $\boldsymbol{u}$ on the fine grid. We aim to propagate $\boldsymbol{u}$ for N time steps Δt^* from time $t = 0$ to a final time $t = T$ with accuracy comparable to that of the fine solver. We aim to obtain an approach with low computational costs, to be precise, lower computational costs than the fine solver $\mathcal{F}_{\Delta t^*}$. The number N of time steps and thus the size of one time step $\Delta t^* \coloneqq \lceil \frac{T}{N} \rceil$ is usually determined by the hardware, specifically by how many time steps can be computed in parallel.
We use the notation $\boldsymbol{u}_n$ for the wave field $\boldsymbol{u}$ at time step n, i.e. at time $t = n \cdot \Delta t^*$. Therefore we aim to approximate

$$\boldsymbol{u}_{n+1} \coloneqq \mathcal{F}_{\Delta t^*}(\boldsymbol{u}_n) \tag{3.1}$$

for all $n = 0, \ldots, N-1$. By adding zero and reorganizing this formula we obtain

$$\boldsymbol{u}_{n+1} = \mathcal{F}_{\Delta t^*}(\mathcal{I}\mathcal{R}\boldsymbol{u}_n) + \mathcal{F}_{\Delta t^*}(\boldsymbol{u}_n) - \mathcal{F}_{\Delta t^*}(\mathcal{I}\mathcal{R}\boldsymbol{u}_n) \tag{3.2}$$

The first step of the approach is to reduce the computational cost by deriving a parareal scheme. For this, we replace $\mathcal{F}_{\Delta t^*}(\mathcal{I}\mathcal{R}\boldsymbol{u}_n)$ with a cheaper approximation $\mathcal{I}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n)$ and furthermore introduce a fixed point iteration as a self improving loop to equation (3.2). Thus we obtain a parareal scheme where $k = 0, \ldots, K-1$ denotes the parareal iteration

$$\begin{aligned} \boldsymbol{u}_{n+1}^0 &= \mathcal{I}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n^0) \\ \boldsymbol{u}_{n+1}^{k+1} &= \mathcal{I}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n^{k+1}) + \mathcal{F}_{\Delta t^*}(\boldsymbol{u}_n^k) - \mathcal{I}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n^k) \end{aligned} \tag{3.3}$$

For each parareal iteration k, the wave field $\boldsymbol{u}_0^k$ is given by the initial conditions of the PDE (2.1) or (2.2), respectively. The parareal scheme enables us to reduce the overall computation time further, by allowing the fine solver to be computed for each time step Δt^* in parallel. Chapter 6 presents the general parareal scheme in detail, including pseudo-code for its implementation.

The second step of the approach is the stabilization of this parareal scheme. Since the scheme needs $\mathcal{I}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n)$ to be a close approximation of $\mathcal{F}_{\Delta t^*}(\mathcal{I}\mathcal{R}\boldsymbol{u}_n)$ to be stable, we improve $\mathcal{I}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n)$ by replacing $\mathcal{I}$ with a neural network, that approximates $\mathcal{F}_{\Delta t^*}(\mathcal{I}\mathcal{G}_{\Delta t^*}^{-1}(\boldsymbol{u}))$. To further improve the approach, we introduce the energy representation of a wave field in

Chapter 3.2, where we define the functions Λ_h and $\Lambda_h^\dagger$, where h denotes the spatial step size of the considered grid, i.e. $h = \Delta x$ for the coarse grid and $h = \delta x$ for the fine grid. The function Λ_h computes the energy components of a wave field on the respective grid and $\Lambda_h^\dagger$ denotes the pseudo inverse function on the respective grid. Since the energy representation performed well in similar work [3], [18], we use these two functions to define the enhanced solver on the fine grid as

$$\tilde{\mathcal{G}}_{\Delta t^*}(\boldsymbol{u}, c) \coloneqq \Lambda_{\delta x}^\dagger \mathcal{H}_{\Delta t^*}^\theta(\Lambda_{\Delta x}\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}, c)) \tag{3.4}$$

where $\mathcal{H}_{\Delta t^*}^\theta$ denotes the function of a neural network with the trainable parameters θ. This neural network is trained on input data of the form $((c, \Lambda_{\Delta x}\mathcal{G}_{\Delta t^*}\mathcal{R}\boldsymbol{u}), \Lambda_{\delta x}\mathcal{F}_{\Delta t^*}\boldsymbol{u})$ for the time step Δt^*. For a shorter notation we write $\tilde{\mathcal{G}}_{\Delta t^*}(\boldsymbol{u})$ instead of $\tilde{\mathcal{G}}_{\Delta t^*}(\boldsymbol{u}, c)$, when c is clearly determined by the context. We remark that $\tilde{\mathcal{G}}_{\Delta t^*}$ propagate a wave field $\boldsymbol{u}$ for one time step Δt^* on the fine grid. Chapter 5 presents the design of the neural network and the generation of the training data in detail.
Using the enhanced solver $\tilde{\mathcal{G}}_{\Delta t^*}$ in (3.3) we derive the following parareal scheme for the wave propagation

$$\begin{aligned} \boldsymbol{u}_{n+1}^0 &= \tilde{\mathcal{G}}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n^0) \\ \boldsymbol{u}_{n+1}^{k+1} &= \tilde{\mathcal{G}}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n^{k+1}) + \mathcal{F}_{\Delta t^*}(\boldsymbol{u}_n^k) - \tilde{\mathcal{G}}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}_n^k) \end{aligned} \tag{3.5}$$

where the wave fields $\boldsymbol{u}_0^k$ are given through the initial conditions of the PDE for all parareal iterations $k = 0, \ldots, K-1$.

3.2 Energy Representation

To enhance the general idea further, we consider the energy semi-norm for a given wave field $\boldsymbol{u} := (u, u_t)$, where u_t denotes the first time derivative $\frac{\partial}{\partial t}u$, and the corresponding wave speed profile c. This semi-norm is given by

$$\mathbb{E}[\boldsymbol{u}(x,t)] \coloneqq \frac{1}{2} \int\limits_{[-1,1]^d} |\nabla u(x,t)|^2 + c^{-2}|u_t(x,t)|^2 dx \tag{3.6}$$

where $x \in [-1,1]^d$ denotes the spatial variable in $d = 2$ or $d = 3$ spatial dimensions, depending on the setup. Furthermore, t denotes the time variable.
Due to the well-posedness of the wave equation in this semi-norm, we can choose it as a metric for the comparison of wave fields. Since the wave equation is well posed in this semi norm, it is a suitable metric to compare wave fields. Using this semi-norm we obtain a stable metric for wave propagation, in the sense that small perturbations in the wave field results only in small differences in the corresponding energy of the wave field.

We define the energy components of a wave field $\boldsymbol{u}$ as $(\nabla u, c^{-1}u_t)$ and the corresponding

function Λ, which maps a wave field to its energy components, as

$$\Lambda \colon \boldsymbol{u} \mapsto (\nabla u, c^{-1} u_t) \tag{3.7}$$

We observe that the representation of $\boldsymbol{u}$ as its energy components allow us to compute the energy $\mathbb{E}[\boldsymbol{u}]$ of $\boldsymbol{u}$ directly.

Furthermore, $\Lambda^\dagger$ denotes the pseudo inverse of Λ. In our approach we compute a wave field $\boldsymbol{v} = \mathcal{H}^w_{\Delta t^*}(\Lambda_{\Delta x} \mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u}))$ through the neural network. Therefore we need to compute $\Lambda^\dagger(\boldsymbol{v})$, where $\boldsymbol{v}$ is given as its energy components, i.e. $\boldsymbol{v} = (\nabla v, c^{-1} v_t)$. Moreover, the corresponding velocity profile c is given. Following Appendix A.1. in [18], the definition of $\Lambda^\dagger$ is then given in the Fourier domain as

$$\mathrm{fft}(v) = \begin{cases} -i(\xi \cdot \mathrm{fft}(\nabla v))|\xi|^2 & \text{for } |\xi| \neq 0 \\ C_0 & \text{for } |\xi| = 0 \end{cases} \tag{3.8}$$

where ξ denotes the frequency variable and $\cdot$ denotes the scalar product.

The constant C_0 should theoretically be the integral of the wave field $\boldsymbol{v}$. Due to $\boldsymbol{v}$ being only an approximation of the true wave field, C_0 can subsequently only be defined as an approximation as well. To be precise, C_0 is given as

$$C_0 \coloneqq \sum_{i=0}^{r} \sum_{j=0}^{r} \sum_{l=0}^{r} v(x_i, x_j, x_l) \tag{3.9}$$

for the three dimensional setups and as

$$C_0 \coloneqq \sum_{i=0}^{r} \sum_{j=0}^{r} v(x_i, x_j) \tag{3.10}$$

for the two dimensional setups. Here denotes r the number of grid points of the fine grid in each spatial dimension and for $i, j, l \in \{0, \dots, r\}$, x_i denotes the i-th grid point in the first spatial dimension, i.e. $x_i = x_0 + i\delta_t$, and both x_j and x_l are defined analogously for the second and third spatial dimension.

Since we employ a spatial discretization for the numerical solvers of the wave equation (2.1) and (2.2), it is reasonable to also consider the discretized version of the energy semi-norm. For this reason, we introduce the operator ∇_h, which denotes the discretized version of the nabla operator with discretization step size h. As a result we obtain the following discretized energy semi-norm.

$$\mathbb{E}_h[\boldsymbol{u}(x,t)] \coloneqq \sum_{x \in (h\mathbb{Z})^d \cap [-1,1]^d} \Big(\|\nabla_h u(x,t)\|_2^2 + |c^{-1}(x) u_t(x,t)|^2 \Big) h^2 \tag{3.11}$$

where $d = 2$ or $d = 3$ denotes the number of spatial dimensions in the setup.

We further define a discretized version of Λ and $\Lambda^\dagger$. For this we use the discretized nabla operator ∇_h instead of ∇ and thus derive the discretized Λ operator Λ_h as

$$\Lambda_h \colon \boldsymbol{u} \mapsto (\nabla_h u, c^{-1} u_t) \tag{3.12}$$

We note, that $\Lambda(\boldsymbol{u})$ could be computed directly by numerically computing the discretized gradient of u.
We define the disrectized version $\Lambda_h^\dagger$ of $\Lambda^\dagger$ by discretizing the output of $\Lambda^\dagger$ on the grid with spatial step size h.

4 Numerical Solver

This chapter specifies the choice of the fine solver $\mathcal{F}_{\Delta t^*}$ and the coarse solver $\mathcal{G}_{\Delta t^*}$ used in our setups. As described in Chapter 3, the approach requires the numerical solvers to be stable and, additionally, the coarse solver to be time-reversible. Therefore the most suitable numerical method depends on the given PDE.
Regarding the setup with the wave equation (2.1), the central difference scheme in space combined with the Velocity Verlet time integration method is well suited. This second order method was introduced by Verlet [24]. The resulting stepping scheme $S_{h,k}$ is derived in Chapter 4.1, where h denotes the spatial discretization step and k denotes the time step of the stepping scheme.
Regarding the also considered wave equation with variable coefficients (2.2), a slightly adapted variant of the stepping scheme is suitable. This second stepping scheme is derived in Chapter 4.2 and denoted by $\tilde{S}_{h,k}$, where h and k denotes the spatial step and the time step of the stepping scheme.
The spatial steps are given by the resolution of the fine and coarse grid. Due to the stability requirement, we need to determine the time step for the stepping schemes $S_{h,k}$ and $\tilde{S}_{h,k}$ for both solvers and define the solvers by applying the stepping schemes repeatedly. This is described in detail in Chapter 4.3.

4.1 Velocity Verlet Method

Given the second order PDE (2.1), we need to choose our fine and coarse solver such that the resulting wave solution is a close approximation of the true wave solution. Therefore, in our setup, we use a combination of the standard central difference scheme (CDS) and the Velocity Verlet time integration method [24] as the stepping scheme for the fine and coarse solver.
In a first step we use the CDS to find a good approximation Q_h for the second order derivative Δu on a spatial grid with step size h. In one spatial dimension the CDS of first order is given as

$$\nabla u(x,t) = \frac{u(x+\frac{h}{2},t) - u(x-\frac{h}{2},t)}{h} \tag{4.1}$$

Applying this formula twice and considering that u is a three dimensional function in our setups, we get the following approximation Q_h for Δu. For simplicity we introduce the following

notations: Let r denote the number of grid points in each spatial dimension, depending on the grid resolution. Furthermore, for $i, j, l \in \{0, \dots, r\}$, x_i is defined as the i-th grid coordinate in the first spatial dimension, i.e. $x_i \coloneqq x_0 + i \cdot h$, and x_j, x_l are defined analogously on the second and third spatial dimension. Due to the periodic boundary conditions we identify $x_0 = x_r$ for each spatial dimension. At last we introduce the notation $u_{i,j,l}(t) \coloneqq u(x_i, x_j, x_l, t)$. Using this, we derive

$$\begin{aligned}\Delta u_{i,j,l}(t) \approx \frac{1}{h^2}\Big(&u_{i+1,j,l}(t) + u_{i-1,j,l}(t) + u_{i,j+1,l}(t) + u_{i,j-1,l}(t) \\ &+ u_{i,j,l+1}(t) + u_{i,j,l-1}(t) - 6 \cdot u_{i,j,l}(t)\Big)\end{aligned} \tag{4.2}$$

In a second step we use the Velocity Verlet method for a numerical integration in time with discretization step size k. The Velocity Verlet time integration scheme is a second order method for PDEs of the form $u_{tt} = A(x)$ for a value $A(x)$, which depends on the space variable x. The method consists of updating the approximations for the values of u and u_t for all time steps. Therefore we need to derive a suitable updating scheme. We know that the first three terms of the Taylor expansion provide a good approximation for $u(t + k)$, i.e.

$$u(t + k) \approx u(t) + u_t(t) \cdot k + \frac{1}{2} \cdot k^2 \cdot u_{tt}(t) \tag{4.3}$$

An approximation for u_{tt} is given by the wave equation (2.1) in combination with formula (4.2), i.e.

$$u_{tt}(t) \approx c^2(x) \cdot Q_h(t) \tag{4.4}$$

In the following we use v as an approximation for u_t given by the CDS of first order. Therefore v is given by

$$v(t + k) = \frac{u(t + 2k) - u(t)}{2k} \tag{4.5}$$

Thus, we have found an updating step for $u(t + k)$, i.e.

$$u(t + k) = u(t) + v(t) \cdot k + \frac{1}{2}k^2 \cdot c^2(x) \cdot Q_h(t) \tag{4.6}$$

Applying (4.6) twice on (4.5), we also get the updating scheme for $v(t + k)$

$$v(t + k) = v(t) + \frac{1}{2}k \cdot c^2(x) \cdot \Big(Q_h(t) + Q_h(t + k)\Big) \tag{4.7}$$

Thus, we have derived the stepping scheme $S_{h,k}$ by using the updating steps (4.6) and (4.7), where the initial time steps $u(0)$ and $v(0)$ are given by the initial conditions in time of the PDE (2.1). Using this stepping scheme in the approach, the coarse solver is time-reversible as desired.

4.2 Varied Velocity Verlet Method

For the setups with the wave equation with variable coefficients (2.2) we only need to adapt the above stepping scheme in the computation of the approximation of the second time derivative $u_{tt}(t)$. Let $Lu(t)$ denote the approximation of $u_{tt}(t)$ corresponding to the wave equation with variable coefficients (2.2). In two spatial dimensions, by applying the product rule we get. that (2.2) is equivalent to

$$\begin{aligned} u_{tt}(x,y,t) = 2 \cdot \frac{\partial}{\partial x} c(x,y) \cdot c(x,y) \cdot \frac{\partial}{\partial x} u(x,y,t) + c^2(x,y) \cdot \frac{\partial}{\partial x}\frac{\partial}{\partial x} u(x,y,t) \\ + 2 \cdot \frac{\partial}{\partial y} c(x,y) \cdot c(x,y) \cdot \frac{\partial}{\partial y} u(x,y,t) + c^2(x,y) \cdot \frac{\partial}{\partial y}\frac{\partial}{\partial y} u(x,y,t) \end{aligned} \tag{4.8}$$

We extend the previously introduced notation $u_{i,j}(t) \coloneqq u(t, x_i, y_j)$ to $c_{i,j} \coloneqq c(x_i, y_j)$, where x_i and y_j denotes the grid points in the two spatial dimensions for $i, j \in \{0, \dots, r\}$. Due to the periodic boundary condition we identify $x_0 = x_r$ and $y_0 = y_r$. Applying the one dimensional CDS with spatial step size h for each term of (4.8), we get the approximation Lu as a function in time as

$$\begin{aligned} Lu \coloneqq \frac{1}{2h^2}\Big(& c_{i+1,j}c_{i,j}u_{i+1,j} - c_{i+1,j}c_{i,j}u_{i-1,j} - c_{i-1,j}c_{i,j}u_{i+1,j} + c_{i-1,j}c_{i,j}u_{i-1,j} \\ & + c_{i,j+1}c_{i,j}u_{i,j+1} - c_{i,j+1}c_{i,j}u_{i,j-1} - c_{i,j-1}c_{i,j}u_{i,j+1} + c_{i,j-1}c_{i,j}u_{i,j-1} \\ & + \frac{1}{2}\Big(c_{i,j}^2 u_{i+2,j} + c_{i,j}^2 u_{i-2,j} - 4c_{i,j}^2 u_{i,j} + c_{i,j}^2 u_{i,j+2} + c_{i,j}^2 u_{i,j-2} \Big) \Big) \end{aligned} \tag{4.9}$$

In a second step we can apply the Velocity Verlet time integration scheme as above. Since the Velocity Verlet scheme is independent of the number of spatial dimensions, we can derive the following updating steps as in the first stepping scheme.

$$u(t+k) = u(t) + k \cdot v(t) + \frac{1}{2}k^2 \cdot Lu(t) \tag{4.10}$$

$$v(t+k) = v(t) + \frac{1}{2}k \cdot \Big(Lu(t) + Lu(t+k) \Big) \tag{4.11}$$

Thus, we have derived the stepping scheme $\tilde{S}_{h,k}$ by the updating steps (4.10) and (4.11), where the start values for $u(0)$ and $v(0)$ are given by the initial values in time of the PDE (2.2). Using this stepping scheme, the coarse solver is time-reversible as desired.

4.3 Numerical stability

As previously mentioned, the approach requires the numerical stability of the solvers. Numerical stability intuitively means that small perturbations in the initial data cause only small changes in the numerical solution. One necessary condition for the stability of a solver is the Courant-Friedrichs-Lewy (CFL) condition, which was introduced in 1928 by Courant, Friedrichs and Lewy [12]. This criterion describes a relation between the time step k, the spatial step h of a numerical solver and the wave speed c. One time step $c \cdot k$, multiplied with the given wave speed, must be bounded by the spatial step h. To be precise, the CFL

condition states

$$\mathcal{C} \coloneqq \frac{c \cdot k}{h} \leq C_{max} = 1 \tag{4.12}$$

where $\mathcal{C}$ is called the Courant number. This condition describes that the physical wave speed c must not exceed the speed $\frac{h}{k}$ at which the numerical scheme can propagate information. To satisfy the CFL condition, the simplest way is to rely on the fastest wave speed c in the considered wave speed media.

As described in Chapter 3, the spatial discretization step h is given through the grid resolution as Δx for the coarse solver and as δx for the fine solver. Due to the stability requirement for $\mathcal{G}_{\Delta t^*}$ and $\mathcal{F}_{\Delta t^*}$ we have to choose the time step k of our stepping schemes $S_{h,k}$ and $\tilde{S}_{h,k}$ such that the CFL condition is satisfied. We denote the obtained time steps with Δt for the solver on the coarse grid and δt for the solver on the fine grid.
Since $\mathcal{F}_{\Delta t^*}$ and $\mathcal{G}_{\Delta t^*}$ are defined to propagate a wave field for a time step Δt^*, we define

$$\mathcal{F}_{\Delta t^*} \coloneqq (S_{\delta x, \delta t})^m \tag{4.13}$$

$$\mathcal{G}_{\Delta t^*} \coloneqq (S_{\Delta x, \Delta t})^M \tag{4.14}$$

for the setups with equation (2.1), where $m \coloneqq \lceil \frac{\Delta t^*}{\delta t} \rceil$ and $M \coloneqq \lceil \frac{\Delta t^*}{\Delta t} \rceil$ denotes the number of times the stepping scheme is repeated until the time step Δt^* is reached. For the setups with equation (2.2) we define $\mathcal{F}_{\Delta t^*}$ and $\mathcal{G}_{\Delta t^*}$ as

$$\mathcal{F}_{\Delta t^*} \coloneqq (\tilde{S}_{\delta x, \delta t})^m \tag{4.15}$$

$$\mathcal{G}_{\Delta t^*} \coloneqq (\tilde{S}_{\Delta x, \Delta t})^M \tag{4.16}$$

with m and M defined as above. Using these definitions for the fine and coarse solver we provide stable and time-reversible numerical solver for the wave equation (2.1) and (2.2).

5 Neural Network

In this chapter, we introduce the important concepts from the domain of neural networks that are relevant for this thesis. Section 5.1 explains our deep learning framework and especially introduces our used neural network architecture. In Section 5.2, we present our approach for generating the training data we used in two and in three spatial dimensions. In Section 5.3, we explain our training algorithm with our choice of the loss function and the optimizer. In the last section of this chapter, we present our choice of metric to measure the error of the neural network output compared to the fine solver.

5.1 Neural Network Architecture

As described in Chapter 3, we consider a neural network to enhance the propagation of a wave field. This neural network serves as a correction operator for the coarse solver. In general, a neural network (NN) is a computational model to predict the unknown output for given input data. A neural network consist of multiple interconnected nodes, so called neurons and is designed to artificially model the neurons in the brain.

5.1.1 Supervised Learning

In our approach, we use a supervised deep learning framework for the considered neural network. For this learning strategy, we have a set of training data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, on which the model is trained. x_i denotes an input data point and y_i denotes the desired output for this input point. Supervised learning aims for the neural network to learn a relationship between the input data and their corresponding labeled output. Furthermore, it should learn the underlying structure in the data to generalize this relationship to be able to predict the output for unseen input data accurately.

In our approach, the data points (x_i, y_i) are given in the form $((c, \Lambda_{\Delta x}\mathcal{G}_{\Delta t^*}\mathcal{R}\boldsymbol{u}), \Lambda_{\delta x}\mathcal{F}_{\Delta t^*}\boldsymbol{u})$. This means that the input of the NN is a tensor $(c, \Lambda_{\Delta x}\mathcal{G}_{\Delta t^*}\mathcal{R}\boldsymbol{u})$, whose size depends on whether the setup is a two or three dimensional one. For the three dimensional setups, this tensor is of size $w \times h \times d \times 5$ and in the two dimensional setups, it is of size $w \times h \times 4$. Here, w denotes the width, h the height and, in the three dimensional cases, d the depth of the given input wave field. In this thesis they are given by the number of grid points of the coarse grid in the corresponding dimension. The number 5 in the three dimensional case and 4 in the two dimensional case denotes the number of channels of the input, which corresponds

to the wave speed profile c and the energy components of a wave field on the coarse grid. The corresponding label is given as a tensor $(\Lambda_{\delta x}\mathcal{F}_{\Delta t^*}\boldsymbol{u}))$ of size $\tilde{w} \times \tilde{h} \times \tilde{d} \times 4$ in the three dimensional setups and of size $\tilde{w} \times \tilde{h} \times 3$ in the two dimensional ones, where $\tilde{w}$ denotes the width, $\tilde{h}$ the height and, for the three dimensional setups, $\tilde{d}$ the depth of the output wave field, i.e. the number of grid points of the fine grid in the corresponding spatial direction. The numbers 3 and 4 denote the number of channels of the output. It corresponds to the size of the energy components of the output wave field. However, we note that in contrast to the input tensor, the output tensor does not contain the velocity profile c.

The aim of the supervised learning paradigm in this thesis can be formulated as approximating an unknown function

$$\begin{aligned}&\mathcal{H}^{\theta}_{\Delta t^*} \colon \mathbb{R}^{w\times h\times d\times 5} \to \mathbb{R}^{\tilde{w}\times\tilde{h}\times\tilde{d}\times 4}\\&\mathcal{H}^{\theta}_{\Delta t^*}((c, \Lambda_{\Delta x}\mathcal{G}_{\Delta t^*}\mathcal{R}\boldsymbol{u})) = \Lambda_{\delta x}\mathcal{F}_{\Delta t^*}\boldsymbol{u}\end{aligned}$$

in the three dimensional case and as approximating

$$\begin{aligned}&\tilde{\mathcal{H}}^{\theta}_{\Delta t^*} \colon \mathbb{R}^{w\times h\times d\times 4} \to \mathbb{R}^{\tilde{w}\times\tilde{h}\times d\times 3}\\&\tilde{\mathcal{H}}^{\theta}_{\Delta t^*}((c, \Lambda_{\Delta x}\mathcal{G}_{\Delta t^*}\mathcal{R}\boldsymbol{u})) = \Lambda_{\delta x}\mathcal{F}_{\Delta t^*}\boldsymbol{u}\end{aligned}$$

in the two dimensional case. In both functions, θ denotes the set of trainable parameters corresponding to the neural network.

5.1.2 Feedforward Neural Network

In the following, *in* denotes the size of the input tensor x and *out* denotes the size of the output tensor y. A Feedforward Neural Network (FNN) is a neural network, where the neurons are organized in a sequence of multiple layers. Let $L + 2$ denotes the number of layers. A FNN contains one input layer, one output layer and L hidden layers. These layers are arranged as shown in Figure 5.1. The FNN computes a function $\mathcal{H}^{\theta} \colon \mathbb{R}^{in} \to \mathbb{R}^{out}$ by processing the input x_l of each layer $l = 0, \ldots, L + 1$ by a function h_l. If not defined otherwise, we consider the Layer l to be a fully connected layer, for which the function h_l is given by

$$x^{l+1} = h_l(x^l) = \sigma(W^l x^l + B^l)$$

where x^{l+1} denotes the output of layer l and the input of layer $l + 1$, except for $l = L + 1$, where x^{l+1} denotes the neural network output. This type of layer requires the input tensor x^l to be a vector, i.e. $in \in \mathbb{N}$.

Each such fully connected layer l depends on trainable parameters, which are denoted by W^l and B^l.
W^l denotes the weight matrix of layer l, where each entry $w_{i,j}$ of this matrix represents how much the j-th neuron in layer l impacts the i-th neuron of layer $l + 1$.

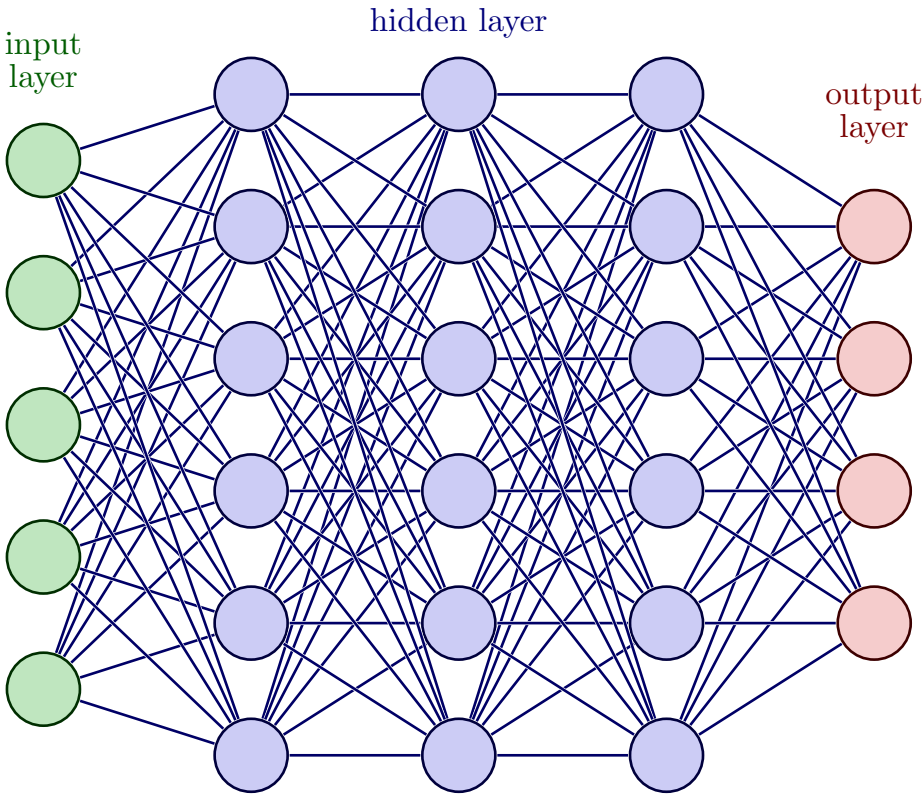


Fig. 5.1: The layers of a FNN with the input layer on the left, the output layer on the right and three hidden layers in between. This figure is a modification of [16].

The weights W^l in our considered neural network are initialized with the standard PyTorch initialization, which is given as the Kaiming Initialization with a uniform distribution. This initialization method was introduced by He et al. in 2015 [7].

B^l denotes the bias vector of the layer l. These trainable parameter are added to the weighted input of a layer, before the activation function σ is applied. These parameters allow the neural network to better learn to fit the data.
In our approach the biases are initialized by the standard pytorch initialization, which is a randomly initialization with a uniform distribution.

Additionally, we use an activation function σ to compute the layer output x^{l+1}. After summing together the weighted inputs and the biases of a layer, we apply this function. It is used to enhance the layers output, by introducing non-linearity, which results in the capability of the neural network to model more complex pattern in the data.
One commonly used activation function is the Rectified Linear Unit (ReLu) activation function. It is defined as

$$\mathrm{ReLU}(x) = \max\{0, x\} \tag{5.1}$$

This function maps all negative values to zero and keeps positive values unchanged. Since this function is very simple, it is a common choice as the activation function. In our neural

network every layer uses the ReLU function as its activation function σ, since it has performed well in similar work [18].
Note that the output layer normally uses the identity function $id(x) = x$ as activation function, to allow negative output values.

5.1.3 Convolutional Neural Network

Another important layer type is the convolutional layer. It is defined for a multidimensional input tensor x. For example, this type of layer is applied to the input tensor of our neural network in the three dimensional setups, which is of size $\mathbb{R}^{w \times h \times d \times 5}$. A convolutional layer l uses a kernel K_d^l and a convolution between this kernel and the layer input x^l, which is defined as

$$K_d^l * x^l(i,j) \coloneqq \sum_m \sum_n K_d^l(m,n) \cdot x^l(i+m, j+n) \tag{5.2}$$

for a two dimensional convolution and as

$$K_d^l * x^l(h,i,j) \coloneqq \sum_m \sum_n \sum_p K_d^l(m,n,p) \cdot x^l(h+m, i+n, j+p) \tag{5.3}$$

for a three dimensional convolution. Then, the layer function of a convolutional layer in both cases is given by

$$x_d^{l+1} = \sigma(\sum_c K_d^l * x_c^l + B_d^l) \tag{5.4}$$

where c, d denote the channel numbers of the input and output of the convolutional layer.

In contrast to the fully connected layers, the weights of the convolutional layers are given by the kernel K_d^l and the bias B_d^l is given as a scalar value which is applied to every spatial location. The activation function of a convolutional layer is defined in the same way as for the fully connected layers.

A convolutional layer often occurs with a striding parameter. This parameter is used to down sample the input of a layer. It denotes the fraction of the number of entries on which the kernel is applied. For example a striding factor of 2 would result in considering only every other entry of the input when applying the kernel.

Another parameter, which is used in the context of a convolutional layer, is the padding parameter. Padding is applied to a convolutional layer before the actual convolution to preserve the spatial dimension when managing boundary conditions. One common padding strategy is to add zero entries around the input before applying the convolution, where the padding parameter denotes the number of rows and columns which are added around the input.

A FNN where most of the layers are convolutional ones, is called a Convolutional Neural Network (CNN). CNNs are designed specifically for imaging, originally in two dimension, but nowadays CNNs are are also applied for three dimensional imaging.
One main advantage of CNNs is that they have less parameters than a normal NN. Therefore it is possible to train deep architectures efficiently.

In a CNN, a convolutional layer is normally combined with other layers, such as pooling layers. A pooling layer is used to down sample the size of the input. It summarizes local regions by also maintaining important features. In contrast to the fully connected and convolutional layers, the pooling layer does not have any trainable parameters.
Two commonly used pooling strategies are the max-pooling and the average pooling. The first one projects a local region to the maximum value in it. The average pooling on the other hand computes the mean of the values in a local region. In this thesis we use the average pooling layers for the convolution blocks in the considered neural network.

For a FNN or a CNN, the set θ of trainable parameter is defined as the set of all weights and biases of the used layers. These trainable parameters are optimized during the training progress, which is described in detail in Chapter 5.3.

5.1.4 The JNet Architecture

The UNet architecture as a type of CNN was introduced by Ronneberger et al. [20], originally for biomedical imaging. A UNet consist of a sequence of convolutional blocks. These blocks are organized in a downsampling path, that encodes the input to a so called feature space, followed by an upsampling path, that decodes the values to the final output.
Following the neural network architecture presented by Nguyen and Tsai in chapter 2.1 [18], we consider a JNet as our neural network choice. A JNet is a special UNet, where the downsampling path contains less blocks of layers than the upsampling path. Therefore the output of a JNet has a higher resolution than the original input. The exact architecture of the JNet can be found in Chapter 2.1 in [18]. For the two dimensional setups the design of the chosen neural network is shown in Figure 5.2. The JNet architecture for the three dimesional setups is similar, just with a three dimensional convolution in the convolutional blocks and trilinear downsampling and upsampling layers.
We further introduce some skip connections to the JNet. These connections adds the output $\tilde{y}$ of a block of layers of the downsampling path to the corresponding level on the upsampling path, where we say that to blocks are on the same level, when their output has the same resolution. Thus, the output y of a block of layers on the upsampling path is given as

$$y = h(y) + \tilde{y}$$

where h denotes the concatenation of all functions of the layers between these two blocks.

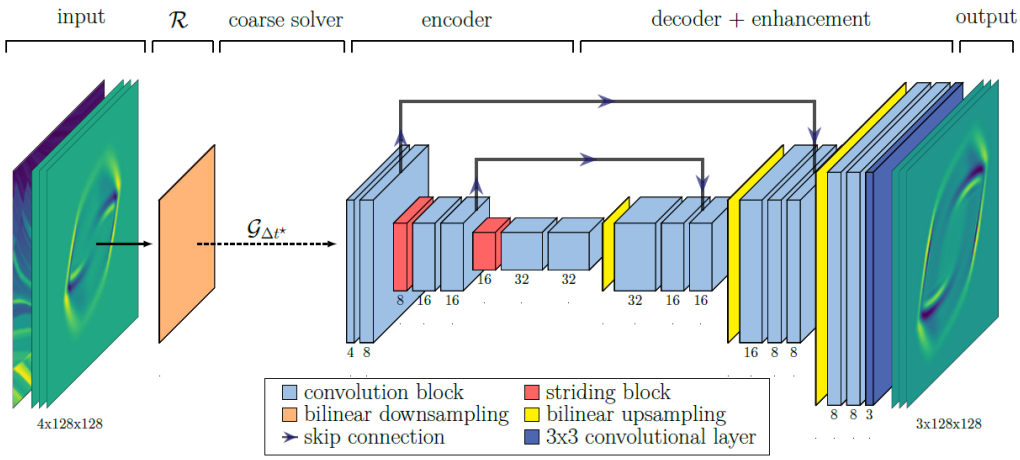


Fig. 5.2: The architecture of the chosen JNet with the encoding downsampling path and the decoding upsampling path. This figure is taken from Figure 4.4 in [9].

Using this technique we can preserve information from the downsampling layers, which could otherwise be lost during the downsampling process.

5.2 Generating Training Data

As described in Chapter 5.1, the inputs of the neural network are a wave field in form of its energy components and the corresponding velocity profile, both on the coarse grid. To be more precise, the input tensor of the neural network has 4 channels in the two dimensional case and 5 channels in the three dimensional one. These channels are for the spatial derivatives of u, for u_t and for the velocity profile c. The output of the neural network are the energy components of the input wave field propagated by one time step Δt^* on the fine grid. Therefore, the output tensor has 3 channels in the two dimensional case and 4 channels in the three dimensional one. These channels are for the spatial derivatives of u and for u_t.

5.2.1 Generating Velocity Profiles

For a high accuracy of the trained neural network we need sufficiently representative training examples for our considered class of wave speed.
We consider two wave speed profiles for our training data. The first one is the Marmousi profile [5], which was introduced by Bourgeois et al. This profile is a complex two dimensional wave speed model based on the geology in parts of Angola and was designed for generating synthetic seismic data. The second considered wave speed profile is the BP profile [4] which was introduced in 2004 by Billette and Brandsberg-Dahl. This profile is also a complex two dimensional wave speed model. It is based on the geology of the Gulf of Mexico and is designed for generating realistic and synthetic seismic data.

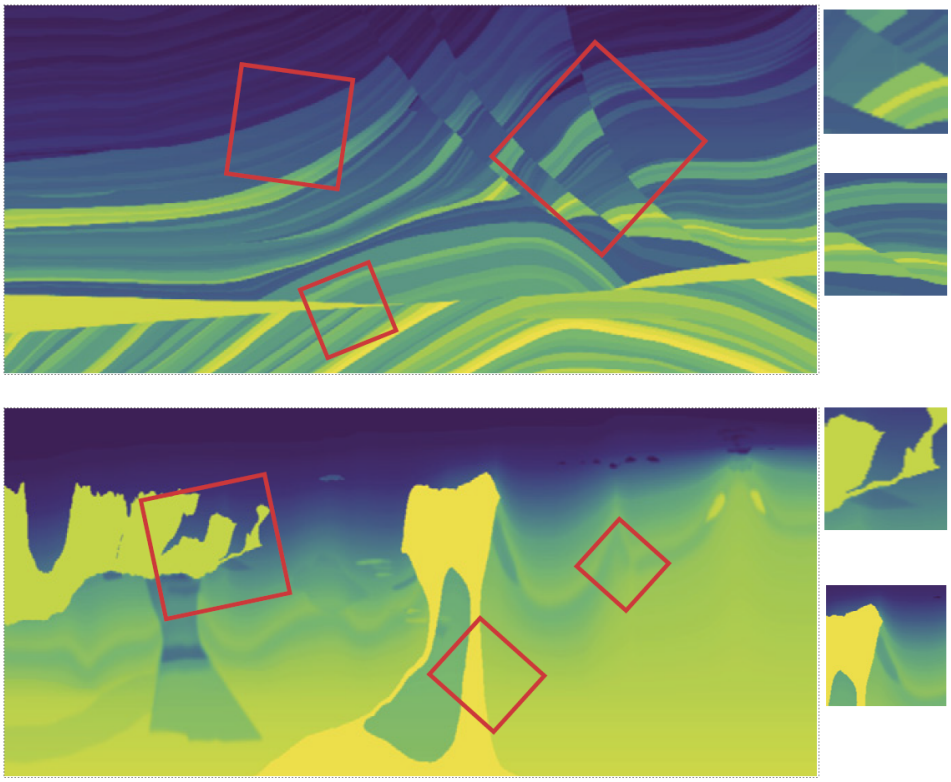


Fig. 5.3: This figure shows the process of sampling two dimensional velocity profiles from the Marmousi profile (top) and BP profile (bottom). The red squares mark randomly selected subregions. On the right side are some examples of velocity profiles, that are generated with this method. This figure is a modification of Figure 3. and Figure 4. in [18].

For our two dimensional setups we crop segments of both profiles in the size of our fine grid resolution. For this, we follow Section 2.2 in [18] and choose randomly selected and rotated subregions from the Marmousi and BP profile and rescale them by an integer to make sure that the coarse solver is stable with the used Δt and Δx. We furthermore map these subregions onto the spatial grid $h\mathbb{Z}^2 \cap [-1,1]^2$, where h denotes the discretization step size of the fine grid. This process is shown in Figure 5.3. For the three dimensional setups we start with randomly cropping two dimensional subregions as described above. Afterwards we extend these wave speed profiles to three spatial dimensions by replicating them along the third dimension. This process is shown in Figure 5.4.

5.2.2 Generating Standard Training Data

We first generate a data set $\mathcal{D}_0$ which contains pairs of an initial wave field $\boldsymbol{u}_0 = (u_0, \partial_t u_0)$ and its corresponding velocity profile c for every cropped wave speed profile, where c is cropped as two or three dimensional velocity profile following the explanation above, where the dimension depends on the setup. These initial wave fields are sampled from Gaussian pulses of the following forms.

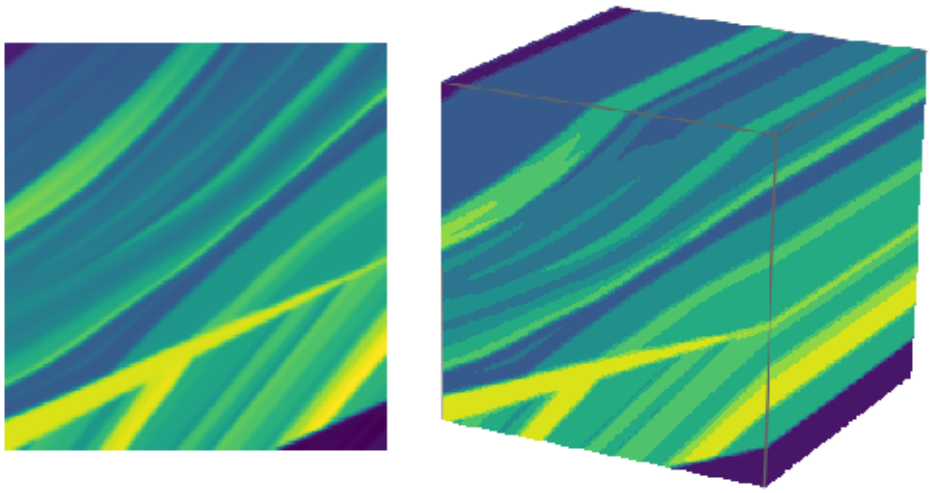


Fig. 5.4: The process of generating three dimensional velocity profiles: On the left is a randomly cropped two dimensional segment of the marmousi profile and on the right is the corresponding extended three dimensional velocity profile

For the two dimensional setups $\boldsymbol{u}_0$ is given by

$$\begin{aligned} u_0(x,y) &= \exp\left(-\frac{x^2+y^2}{\sigma^2}\right) \\ \partial_t u_0(x,y) &= 0 \end{aligned} \tag{5.5}$$

where $(x,y) \in \delta x\mathbb{Z}^2 \cap [-1,1]^2$ and δx denotes the spatial discretization step size of the fine grid. Similarly $\boldsymbol{u}_0$ in the three dimensional setups is given as

$$\begin{aligned} u_0(x,y,z) &= \exp\left(-\frac{x^2+y^2+z^2}{\sigma^2}\right) \\ \partial_t u_0(x,y,z) &= 0 \end{aligned} \tag{5.6}$$

where $(x,y,z) \in \delta x\mathbb{Z}^3 \cap [-1,1]^3$ and δx denotes the spatial discretization step size of the fine grid. In both cases, $\frac{1}{\sigma^2}$ follows a normal distribution, given by $\frac{1}{\sigma^2} \sim \mathcal{N}(250, 10)$
The data set $\mathcal{D}_0$ contains all pairs $(c, \boldsymbol{u}_0)$.

In a second step, we generate a set $\mathcal{D}(\Delta t^*)$ of standard training data. For this we propagate each wave field $\boldsymbol{u}_0$ in $\mathcal{D}_0$ with the given fine solver $\mathcal{F}_{\Delta t^*}$ for N time steps of size Δt^*, i.e.

$$\boldsymbol{u}_{n+1} = \mathcal{F}_{\Delta t^*}\boldsymbol{u}_n$$

for $n = 0, 1, \ldots, N-1$.

As previously mentioned, the input of the neural network are the energy components of the wave field propagated by the coarse solver $\mathcal{G}_{\Delta t^*}$ on the coarse grid and the target output of the neural network are the energy components of the wave field propagated by the fine

solver $\mathcal{F}_{\Delta t^*}$ on the fine grid. Thus we collect the pairs

$$((c, \Lambda_{\Delta x} \mathcal{G}_{\Delta t^*} \mathcal{R} \boldsymbol{u}_n), \Lambda_{\delta x} \mathcal{F}_{\Delta t^*} \boldsymbol{u}_n)$$

for all $n = 0, 1, \ldots, N-1$ in the data set $\mathcal{D}(\Delta t^*)$.

5.2.3 Generating Parareal-Like Training Data

As described in Chapter 3, we combine the neural network with the parareal algorithm for the considered approach. Consequently, the input data of the neural network is of a parareal-like structure. To account for this structure in the actual data, we generate the training data set using the parareal algorithm. In doing so, we get a higher similarity between the training data and the actual data, on which the neural network is applied, compared to the training data set $\mathcal{D}(\Delta t^*)$

First we generate $\mathcal{D}_0$ as described above. Using this data set we develop a more complex data set $\mathcal{D}_0^p(\Delta t^*)$, which contains parareal-like initial wave fields. To be precise, for all samples $(c, \boldsymbol{u}_0)$ from $\mathcal{D}_0$ and for $k \in \{0, 1, 2, 3, 4\}$ and $n \in \{0, 1, \ldots, N\}$ we compute $\boldsymbol{u}_n^k$ by the parareal algorithm with a parareal time step Δt^*. This algorithm is presented in detail in Chapter 6. Afterwards all pairs $(c, \boldsymbol{u}_n^k)$ are collected in $\mathcal{D}_0^p(\Delta t^*)$.

Afterwards we generate a set $\mathcal{D}^p(\Delta t^*)$ of parareal-like training data. For this we propagate every wave field $\boldsymbol{u}$ in $\mathcal{D}_0^p(\Delta t^*)$ by one time step Δt^* by applying the coarse solver $\mathcal{G}_{\Delta t^*}$ to generate the input data and by one time step Δt^* by applying the fine solver $\mathcal{F}_{\Delta t^*}$ to obtain the corresponding label. Finally, we collect the pairs

$$((c, \Lambda_{\Delta x} \mathcal{G}_{\Delta t^*} \mathcal{R} \boldsymbol{u}), \Lambda_{\delta x} \mathcal{F}_{\Delta t^*} \boldsymbol{u})$$

for all $\boldsymbol{u}$ in $\mathcal{D}_0^p(\Delta t^*)$ in the data set $\mathcal{D}^p(\Delta t^*)$.

5.3 Training

5.3.1 Optimization Problem

The aim of the training process is to minimize the optimization problem

$$\min_{\mathcal{H}_{\Delta t^*}^{\theta} \in NN} \mathcal{L}(\theta, D) \tag{5.7}$$

where D denotes the used training data set and NN denotes the considered class of neural network, in our setups NN is the class of 3-Level JNets, also denoted as JNet(3,1). This class contains JNets with 3 blocks of layers on the downsampling path and $3+1$ blocks of layers on the upsampling path. $\mathcal{L}$ denotes the loss function, which measures the difference between the input data and their corresponding output label.

In other words, given a training data set D, the aim of the training process is to compute the set of the trainable parameters θ of the NN such that the value of the loss function is minimized.

A very commonly used loss function is the mean squared error (MSE), which is given by

$$\mathcal{L}(\theta, D) \coloneqq \frac{1}{|D|} \sum_{((c,x),y)\in D} \left\| \mathcal{H}^{\theta}_{\Delta t^*}((c,x)) - y \right\|^2 \tag{5.8}$$

where $\mathcal{H}^{\theta}_{\Delta t^*}((c,x))$ is the predicted output of the neural network and y denotes the desired output label for an input value $((c,x))$. In our setups we also use the MSE loss as the loss function in our training process.

Due to our choice of the loss function, the optimization problem (5.7) in our setups is therefore given by

$$\min_{\mathcal{H}^{\theta}_{\Delta t^*}\in \mathrm{JNet}(3,1)} \mathcal{L}(\theta, D) \coloneqq \min_{\mathcal{H}^{\theta}_{\Delta t^*}\in \mathrm{JNet}(3,1)} \frac{1}{|D|} \sum_{((c,x),y)\in D} \left\| \mathcal{H}^{\theta}_{\Delta t^*}((c,x)) - y \right\|^2 \tag{5.9}$$

5.3.2 Optimization Algorithm

We train the JNet with a variant of the stochastic gradient descent algorithm, described below. The stochastic gradient descent algorithm is an iterative method, designed to find the global minimum of a convex function. In contrast to the normal gradient descent algorithm, which consider the gradient of the loss for all points in the training data set, the stochastic gradient descent algorithm consider the gradient of the loss for smaller subsets of the training data set, the so called mini-batches with a batch size b. The algorithm computes the gradient of the batch loss, denoted by $J(\theta)$. This is calculated as follows

$$\nabla J(\theta) = \frac{1}{b} \nabla_\theta \mathcal{L}(\theta, B) \tag{5.10}$$

where $\mathcal{L}(\theta, B) \coloneqq \sum_{i=1}^{b} \nabla_\theta \mathcal{L}(\theta, (x_i, y_i))$ for the mini batch $B = \{(x_1, y_1), \dots (x_b, y_b)\}$.

We have given a learning rate η, which describes the step size of the algorithm. Then the parameters are updated by the following step

$$\theta^{s+1} = \theta^s - \eta \nabla J(\theta) \tag{5.11}$$

where s denotes the iteration of the stochastic gradient descent algorithm.

The variant of the stochastic gradient descent algorithm, which we use for the training of the JNet is the so called Adam optimizer [11]. This algorithm is based on adaptive estimates,

meaning that the algorithm uses momentum to accelerate the training process by using past gradients. It further allows us to use an adaptive learning rate.

5.4 Numerical Analysis

To measure the quality of our trained neural network we consider the relative energy MSE error as metric, which is given by

$$e(\sigma, \mathcal{T}) \coloneqq \frac{1}{|\mathcal{T}|} \sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{T}} \left\| \frac{\mathbb{E}_{\delta x}[\boldsymbol{u}-\boldsymbol{v}]}{\mathbb{E}_{\delta x}[v]} \right\|^2 \tag{5.12}$$

where $\mathcal{T}$ denotes a set of test data, generated by a gaussian pulse (5.5) or (5.6), respectively. For each pair $(\boldsymbol{u}, \boldsymbol{v}) \in \mathcal{T}$, $\boldsymbol{u}$ denotes the propagated wave field computed by the enhanced solver $\tilde{\mathcal{G}}_{\Delta t^*}$ and $\boldsymbol{v}$ denotes the corresponding labeled output computed by the fine solver $\mathcal{F}_{\Delta t^*}$. This error measures the accuracy of the neural network output by comparing it with the labeled fine solution. The error is computed for several pairs to achieve an average of the error. We measure this error in relation to the fine solution.

In terms of our trained neural network, $\boldsymbol{v}$ is given as the energy components of the fine solution and $\boldsymbol{u}$ is given as the neural network output, therefore equation (5.12) is in our setups given as follows

$$e(\sigma, \mathcal{T}) \coloneqq \frac{1}{|\mathcal{T}|} \sum_{\boldsymbol{w}\in\mathcal{T}} \left\| \frac{\mathbb{E}_{\delta x}[\mathcal{H}^{\omega}_{\Delta t^*}(c, \mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{w})) - \Lambda_{\delta x}\mathcal{F}_{\Delta t^*}(\boldsymbol{w})]}{\mathbb{E}_{\delta x}[\Lambda_{\delta x}\mathcal{F}_{\Delta t^*}(\boldsymbol{w})]} \right\|^2 \tag{5.13}$$

where $\boldsymbol{w}$ denotes a test wave field in the test set $\mathcal{T}$, initialized by a gaussian pulse (5.5) or (5.6), respectively, depending on the two or three dimensional case.

6 Parareal Scheme

6.1 General Scheme

For a given fine solver $\mathcal{F}$ and a given coarse solver $\tilde{\mathcal{G}}$, both operating on the same grid, the general idea of the parareal scheme introduced by [13] is to compute a good approximation of the fine solution and therefore a good approximation of the wave solution u by simultaneously reducing the necessary overall computation time.
For this, we split the time interval $[0, T]$ into N equally distributed time intervals, each of length Δt^*. In practice the number N of time intervals, and thus the time step $\Delta t^* = \lceil \frac{T}{N} \rceil$, is typically chosen based on the number of parallel computations the hardware is able to perform.

We can now compute an approximation for u_n, where u_n denotes the wave solution u at time $t = n \cdot \Delta t^*$, for each time step $n = 0, \ldots, N-1$ using the given coarse solver $\tilde{\mathcal{G}}$. Therefore we get initial parareal guesses as

$$u_{n+1}^0 \coloneqq \tilde{\mathcal{G}}(u_n^0) \tag{6.1}$$

where the start value u_0^0 is given through the initial condition of the considered PDE. Afterwards we can update the approximations for each u_n, in a self improving loop, the so called parareal updates

$$u_0^{k+1} = u_0^k \tag{6.2}$$

$$u_{n+1}^{k+1} = \tilde{\mathcal{G}}(u_n^{k+1}) + \mathcal{F}(u_n^k) - \tilde{\mathcal{G}}(u_n^k) \tag{6.3}$$

where $k = 0, \ldots, K-1$ denotes the parareal iteration. The first term $\tilde{\mathcal{G}}(u_n^{k+1})$ serves as a new approximation for u_{n+1}, using the cheaper coarse solver $\tilde{\mathcal{G}}$. The second term $\mathcal{F}(u_n^k) - \tilde{\mathcal{G}}(u_n^k)$ serves as a correction term for the high frequencies, which are missing from the last iteration, due to the lower accuracy of the coarse solver in the previous parareal iteration.
We recognize that the second term only relies on approximations from the previous parareal iteration. Therefore we can compute $\mathcal{F}(u_n^k)$ and consequently also $\mathcal{F}(u_n^k) - \tilde{\mathcal{G}}(u_n^k)$ for all $n \in \{0, \ldots, N-1\}$ in parallel. Thus using the parareal scheme in (6.1) and (6.3) we can reduce the overall computation time for $K \ll N$ by computing the fine solver in parallel but also achieve a high accuracy similar to the accuracy of the fine solver. The convergence of the parareal scheme is studied a lot in recent work [2], [21]. We note that the accuracy of the

parareal scheme can only be as good as the accuracy of the fine solver and that the parareal scheme needs less iterations to achieve a high accuracy if the given coarse solver $\tilde{\mathcal{G}}$ provides a solution close to the solution from the fine solver $\mathcal{F}$.

6.2 Used Fine and Coarse Solver

In our setup we use the fine solver $\mathcal{F}_{\Delta t^*}$ defined in Chapter 4 as the fine solver of the parareal scheme and the enhanced solver $\tilde{\mathcal{G}}_{\Delta t^*}$ defined in Chapter 3 as the coarse solver of the parareal scheme. As described beforehand the enhanced solver is computed by applying the neural network to the energy components of the coarse solution $\mathcal{G}_{\Delta t^*}(\mathcal{R}\boldsymbol{u})$. Note that in contrast to the originally assumed fine and coarse solver, which operate on a fine and a coarse grid, the enhanced solver $\tilde{\mathcal{G}}_{\Delta t^*}$ is defined on the fine grid like the fine solver $\mathcal{F}_{\Delta t^*}$. We apply the parareal scheme to our approach since it performed well in similar work [18], [14].
One important change from the general parareal scheme is that our computed function $\boldsymbol{u} = (u, u_t)$ is given by its physical components and therefore by a tuple of two components, instead of just one function, as described in Section 6.1.
Therefore the initial parareal guesses (6.1) and the parareal updates from (6.3) have to compute both, the function u and its first time derivative u_t.

6.3 Pseudo-Code

The following pseudo code contains the general parareal scheme with further improvements. We save the solution of the coarse solver $\mathcal{G}$ for the parareal iteration k in a variable g^k and therefore we can reuse it in the parareal iteration $k+1$.
The algorithm receives the coarse and fine solver for the general scheme, the initial wave field $\boldsymbol{u}_0$, the number of parareal iterations K and the number of time steps N as input.

Algorithm 1 Parareal($\mathcal{G}, \mathcal{F}, u_0, K, N$)

$u_0^0 = u_0$
for $n = 0, \dots, N-1$ **do**
  Compute $g_n^0 = \mathcal{G}(u_n^0)$
  Compute $u_{n+1}^0 = g_n^0$
end for
for $k = 0, \dots, K-1$ **do**
  for $n = 0, \dots, N-1$ **do**
    Compute $f_n^{k-1} = \mathcal{F}(u_n^{k-1})$ in parallel
  end for
  $u_0^{k+1} = u_0^k$
  for $n = 0, \dots, N-1$ **do**
    Compute $g_n^{k+1} = \mathcal{G}(u_n^{k+1})$
    Update $u_{n+1}^{k+1} = g_n^{k+1} + f_n^k - g_n^k$
  end for
end for

7 Results

The first key contribution of our work is to numerically show that the considered approach, introduced by Nguyen and Tsai [18], can be generalized from two spatial dimensions to the standard wave equation (2.1) for three spatial dimensions. To achieve this we describe some numerical comparisons between the fine solution and the approximation of it, computed by the neural network combined with the parareal scheme. The results of this comparison are shown in Chapter 7.1.
The second key contribution of our work is the generalization of the studied approach to a broader range of PDEs by applying the approach to the wave equation with variable coefficients (2.2). We compare the solution of the fine solver and the solution that is computed by the neural network combined with the parareal algorithm, numerically. The results of these comparisons are shown in Chapter 7.2. Furthermore we study what type of training data set is the best choice for the wave equation with variable coefficients.
All of the following computations were performed on an HP Proliant DL380 Gen9 server equipped with 160 GB of RAM, an Intel Xeon E5-2687W v4 CPU (12 cores, 3.0 GHz) and an Nvidia Tesla P100 GPU (16GB).

7.1 Numerical Results for the 3D Wave Equation

We have trained a JNet(3,1) for the three dimensional wave equation (2.1), which denotes a JNet with 3 convolutional downsampling blocks on the downsampling path and 3+1 convolutional blocks of layers and trilinear interpolation layers on the upsampling path. The convolutional blocks uses a three dimensional convolution, since we are studying a three dimensional setup.
The JNet is trained on the data set $\mathcal{D}^p(\Delta t^*)$ with 4000 samples of parareal-like training data that are generated as described in Chapter 5.2. We choose the parareal step Δt^* to be 0.06 seconds, a stepping size of $\delta x = \frac{2.0}{128.0}$ for the fine grid and a stepping size of $\Delta x = \frac{2.0}{64.0}$ for the coarse grid. To generate the training data, we propagate each wave field for 10 time steps Δt^*, therefore the final time T is 0.6 seconds.
According to the CFL-Condition, the time steps for the numerical solvers are chosen as $\delta t = (\frac{2.0}{128.0})/20.0$ seconds and $\Delta t = \frac{1.0}{600.0}$. The training process of the neural network took around 16 hours of computation time for 10 training epochs.

The results of this NN approach for a random generated velocity profile, shown in Figure 7.1

on a grid with 128 grid points for each spatial dimension are shown in Figure 7.2 (a) - (d).

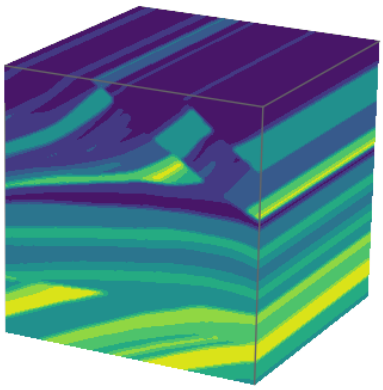
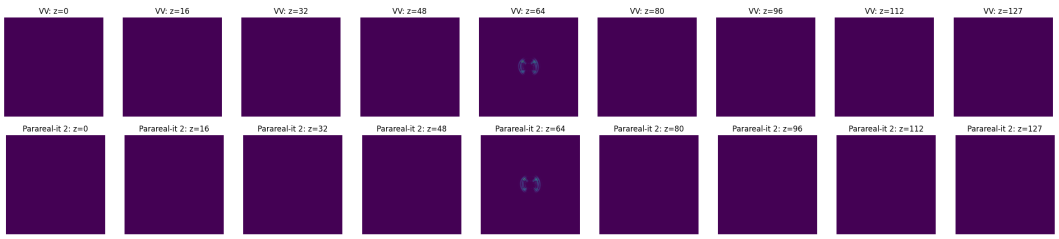


Fig. 7.1: The randomly cropped velocity profile for the comparison of the fine solution and the numerical solution, computed with the neural network and two parareal iterations, for the three dimensional wave equation, that is shown in Figure 7.2.

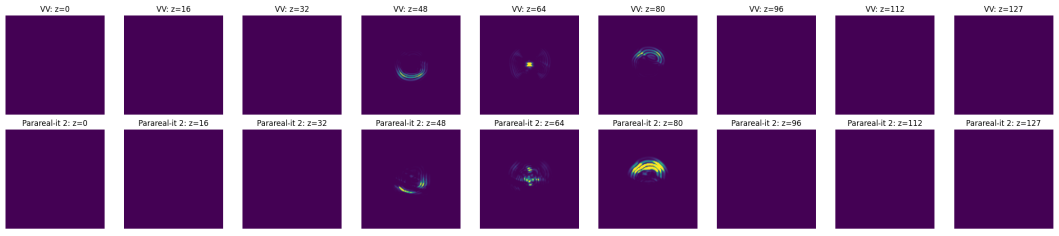
Each of these pictures shows the solution of the fine solver $\mathcal{F}_{\Delta t^*}$ in the first row and the output of the JNet(3,1) combined with 2 parareal iterations in the second row. Each picture shows the wave field at one time step, ranging from $t = \Delta t^* = 0.06$ seconds in Figure 7.2 (a) to $t = 5 \cdot \Delta t^* = 0.3$ seconds in Figure 7.2 (d), where the pictures from left to right shows different xy-slices of the three dimensional domain.
The computation of the fine solution for ten time steps of size Δt^* needs 28.0 seconds, whereas the neural network only needs 21.9 seconds to compute its output.

We can see in Figure 7.2 that the output of the neural network does approximate the fine solution quite well for the first time steps. We further note that the neural network misses some parts of the wave field for $z \leq 48$ for the later time steps. This could be due to the small size of the training data with only 4000 samples. We assume that the neural network can achieve a higher accuracy, also for later time steps, by training the neural network with a larger set of parareal-like training data.

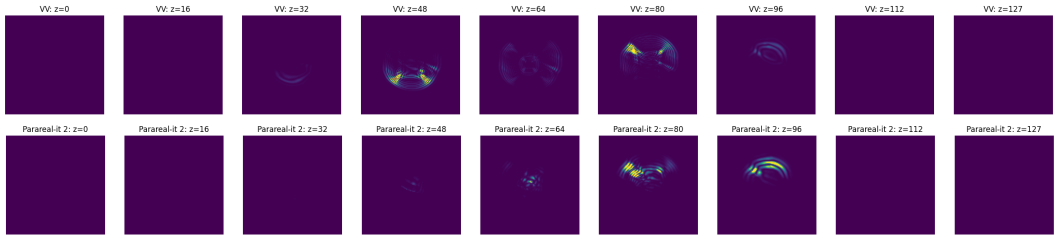
Figure 7.3 visualizes the accuracy of the neural network together with up to two parareal iterations compared to the fine solver measured by the relative energy MSE error. The values were computed by solving the wave equation in three dimensions for ten different velocity profiles with the fine solver and with the neural network combined with the parareal scheme. Figure 7.3 shows that the relative energy MSE error is large for the neural network without the parareal scheme. However, we also see, that two parareal iterations significantly improve the accuracy of the neural network output. Therefore we conclude that the approach, combining the neural network and the parareal algorithm, can be generalized to three dimensions. Note that the relative energy MSE error increases significantly in the later time steps. We state as an open question for future work, whether the same holds, when using a multi step loss, which measures the loss not only for one time step, but for multiple time steps and consider this loss in the training process.



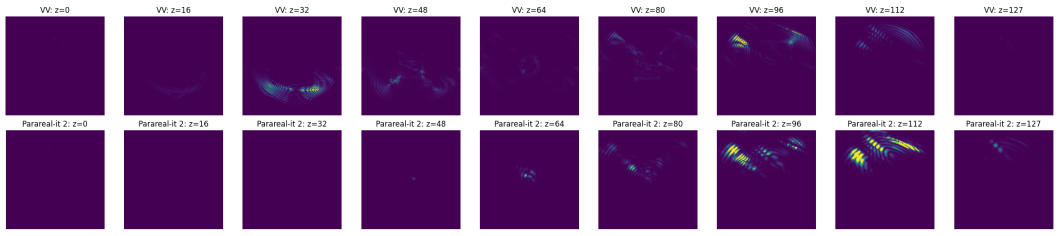
(a) $t = \Delta t^* = 0.06$



(b) $t = 2 \cdot \Delta t^* = 0.12$

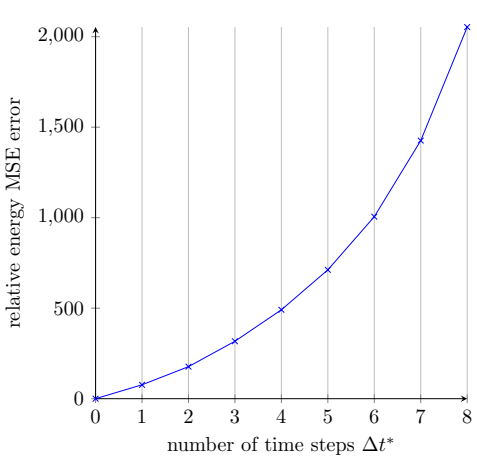


(c) $t = 3 \cdot \Delta t^* = 0.18$

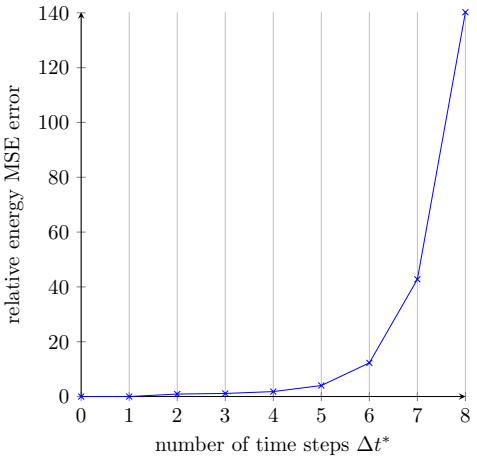


(d) $t = 5 \cdot \Delta t^* = 0.3$

Fig. 7.2: Comparison of the fine solution (first row) and the numerical solution computed by the neural network, trained on 4000 samples of parareal-like training data, combined with two parareal iterations (second row) for the three dimensional wave equation (2.1) at different times t.



(a) The relative energy MSE error of the neural network, trained with parareal-like training data, without any parareal iteration for 8 time steps, computed as average of ten randomly cropped velocity profiles



(b) The relative energy MSE error of the neural network, trained with parareal-like training data, combined with two parareal iteration for 8 time steps, computed as average of ten randomly cropped velocity profiles

Fig. 7.3: Comparison of the relative energy MSE error for a neural network, trained on parareal-like training data, for the three dimensional wave equation

7.2 Numerical Results for the 2D Wave Equation With Variable Coefficients

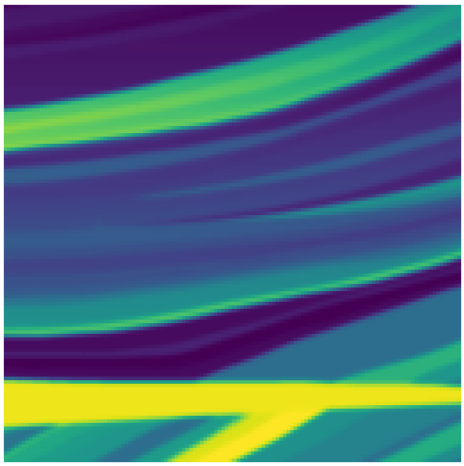
7.2.1 Standard Training Data

For the two dimensional wave equation (2.2) with variable coefficients we have trained a JNet(3,1) with 20000 data points of the standard training data $\mathcal{D}(\Delta t^*)$ generated as described in Chapter 5.2. The JNet uses two dimensional convolutions in its convolutional blocks and the upsampling path uses bilinear interpolation.

The NN is trained for 10 time steps of size $\Delta t^* = 0.06$ seconds and therefore has a final time $T = 0.6$. We choose the step size of the fine solver as $\delta x = \frac{2.0}{128.0}$ and $\delta t = (\frac{2.0}{128.0})/20.0$ seconds. The coarse solver is defined with $\Delta x = \frac{2.0}{64.0}$ and $\Delta t = \frac{1.0}{600.0}$ seconds. With these time and spatial step sizes the fine and coarse solver are stable, according to the CFL Condition as described in Chapter 4.

We have trained the neural network for 10 training epochs, which took about 1 hour of computation time. The results of this neural network combined with two parareal iterations are shown in Figure 7.5 for a randomly cropped velocity profile shown in Figure 7.4 on a grid with 128 grid points in each spatial dimension.
The Figure 7.5 shows the numerical solution computed with the fine solver in the first row, and the solution of the neural network combined with the parareal algorithm in the second row. The different time steps are shown from left to right, starting with $t = \Delta t^*$ on the left

(a) Velocity profile for the computations of the fine solver and the neural network, trained with standard training data, for the two dimensional wave equation with variable coefficients corresponding to the results shown in Figure 7.5.



(b) Velocity profile for the computations of the fine solver and the neural nework, trained with parareal-like data, for the two dimensional wave equation with variable coefficients corresponding to the results shown in Figure 7.7.

Fig. 7.4: The randomly cropped velocity profiles for the comparisons regarding the two dimensional wave equation with variable coefficients.

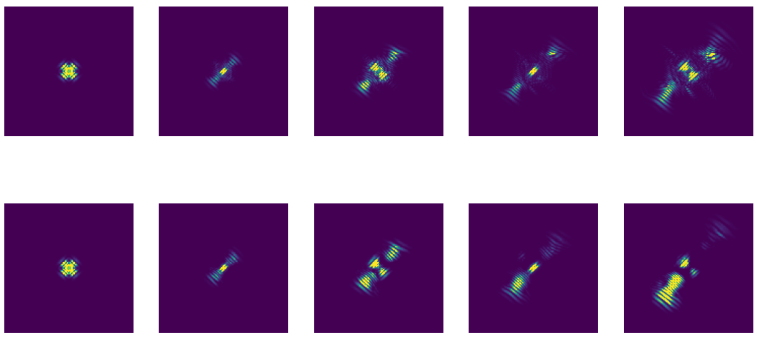
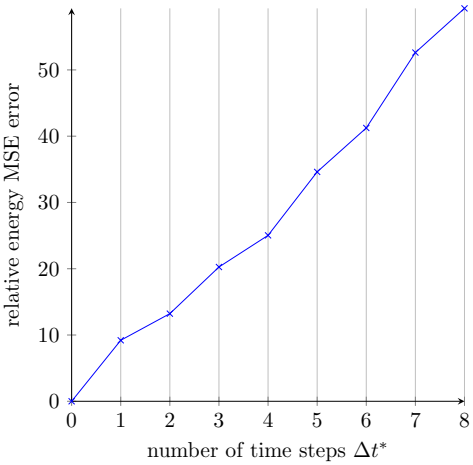


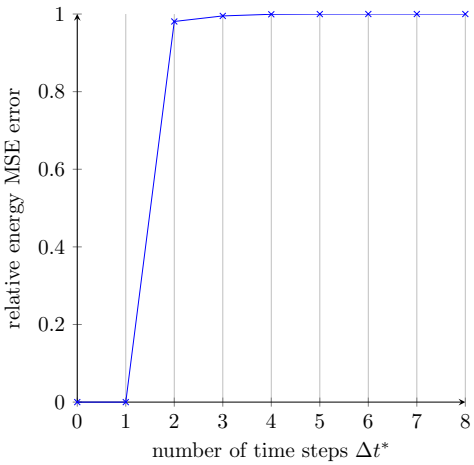
Fig. 7.5: Comparison of the fine solution (first row) and the solution computed with the neural network combined with the parareal algorithm (second row) for the wave equation with variable coefficients for a NN trained on 20 000 data points of standard training data. The results are shown for different time steps starting from $t = \Delta t^* = 0.06$ seconds on the left and ending with $t = 5 \cdot \Delta t^* = 0.3$ seconds on the right.

and ending with $t = 5 \cdot \Delta t^*$ on the right.

The Figure 7.5 shows that the NN approach does approximate the fine solution quite well, as desired. We see that for example at time $t = 4 \cdot \Delta t^* = 0.24$ seconds the fine solver does show more details of the wave field in the upper right corner, which are missing in the second row. The accuracy of the neural network to the fine solution is measured with the relative energy MSE error. The results of this comparison are shown in Figure 7.6.



(a) The relative energy MSE error of the neural network, trained with standard training data, without any parareal iteration for 8 time steps, computed as average of ten randomly cropped velocity profiles



(b) The relative energy MSE error of the neural network, trained with standard training data, combined with two parareal iteration for 8 time steps, computed as average of ten randomly cropped velocity profiles

Fig. 7.6: Comparison of the relative energy MSE error for a neural network, trained on standard training data, for the two dimensional wave equation with variable coefficients.

As Figure 7.6 shows, the relative energy MSE error is large for the neural network without the parareal scheme as shown in Figure 7.6 (a). In comparison the relative energy MSE error is far smaller after computing two parareal iterations, shown in Figure 7.6 (b). Furthermore, we remark that the small relative energy MSE error for the first two time steps is 0. This is due to computing two parareal iterations. For these two time steps, the parareal update is computed only through the fine solver, since the terms of the coarse solver cancel each other out, and thus, the relative energy MSE error results in 0 for these time steps.

7.2.2 Parareal-Like Training Data

In a second experiment for the wave equation (2.2) with variable coefficients, a JNet(3,1) is trained with a set of 5000 data points of parareal-like training data $\mathcal{D}^p(\Delta t^*)$, that are generated as described in Chapter 5.2. This JNet uses two dimensional convolutions in its convolutional blocks and also bilinear interpolation for the upsampling path.

This NN is trained for 10 time steps of size $\Delta t^* = 0.06$ seconds. Therefore the final time T is

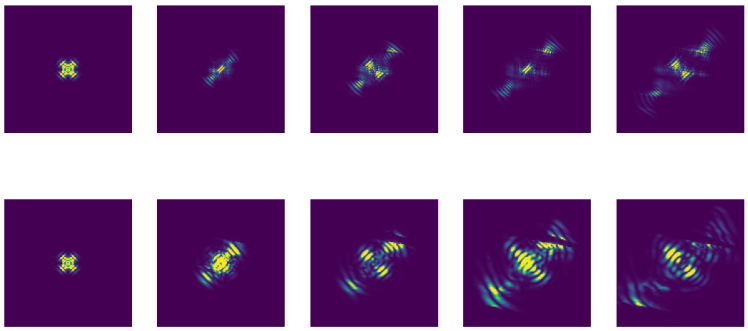


Fig. 7.7: Comparison of the fine solution (first row) and the solution computed with the neural network combined with the parareal algorithm (second row) for the wave equation with variable coefficients for a NN trained on 5000 data points of parareal-like training data. The results are shown for different time steps starting from $t = \Delta t^* = 0.06$ seconds on the left and ending with $t = 5 \cdot \Delta t^* = 0.3$ seconds on the right.

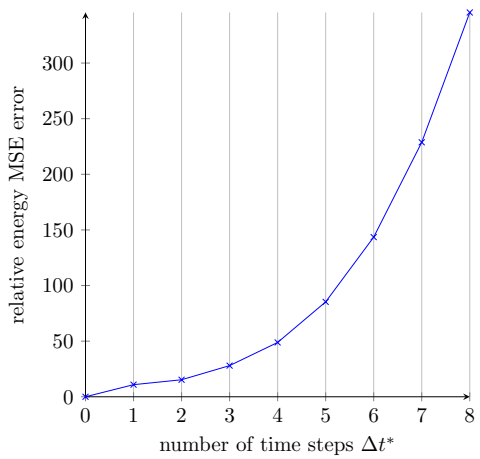
given as $T = 0.6$ seconds. The fine solver has a spatial step size $\delta x = \frac{2.0}{128.0}$ whereas the coarse solver uses the step size $\Delta x = \frac{2.0}{64.0}$. According to the CFL condition presented in Chapter 4, we choose the time step for the fine solver as $\delta t = (\frac{2.0}{128.0})/20.0$ seconds and for the coarse solver as $\Delta t = \frac{1.0}{600.0}$.

The training process for this neural network took around 25 hours of computation time for 10 training epochs. In Figure 7.7, we compare the fine solution with the solution computed by this neural network combined with two parareal iterations for a randomly cropped velocity profile, shown in Figure 7.4, on a grid with 128 grid points in each spatial dimension.

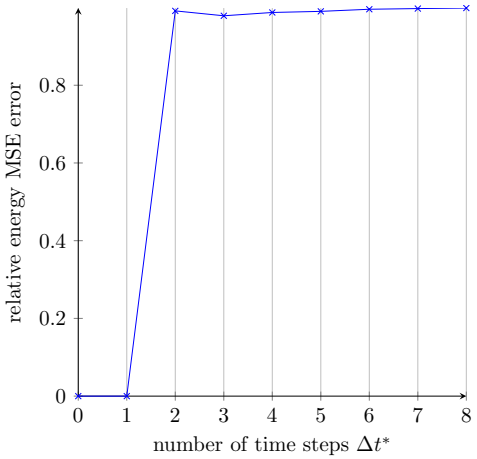
The Figure 7.7 shows the numerical solution, computed with the fine solver $\mathcal{F}_{\Delta t^*}$, in the first row and the solution, computed with the neural network combined with two parareal iterations, in the second row. The pictures show different time steps from left to right, starting with $t = \Delta t^* = 0.06$ seconds on the left and ending with $t = 5 \cdot \Delta t^* = 0.3$ seconds on the right.
We see that the neural network approximates the fine solution, but as one can see, there is a discrepancy between the fine solution and the numerical solution, computed by the neural network in combination with the parareal scheme. This discrepancy is shown in more detail in the relative energy MSE error in Figure 7.8.

This Figure 7.8 shows that the relative energy MSE error is large for the neural network without the parareal scheme as shown in Figure 7.8 (a). In comparison the relative energy MSE error is far smaller after computing two parareal iterations, shown in Figure 7.8 (b). Furthermore, we remark that the small relative energy MSE error for the first two time steps is 0. As before, this is due to computing two parareal iterations.

In comparison to Figure 7.6 we see that the relative energy MSE error for the neural network, trained on the parareal-like data set, is larger than the relative energy MSE error for



(a) The relative energy MSE error of the neural network, trained with parareal training data, without any parareal iteration for 8 time steps, computed as average of ten randomly cropped velocity profiles.



(b) The relative energy MSE error of the neural network, trained with parareal training data, combined with two parareal iteration for 8 time steps, computed as average of ten randomly cropped velocity profiles.

Fig. 7.8: Comparison of the relative energy MSE error for a neural network, trained on parareal-like training data, for the two dimensional wave equation with variable coefficients.

the neural network trained on standard training data. Comparing these results and taking the computation time of the training process for both neural network into concern, we conclude that the standard training data seems to be a better choice for the two dimensional wave equation with variable coefficients. The main advantage of the standard training data is that we can train the neural network a lot faster than with the parareal training data, even if we use a significantly larger amount of training samples.

8 Conclusion

Nguyen and Tsai [18] show, that a supervised deep learning framework can be used to enhance the accuracy of a given coarse solver and therefore improve the overall computation time for wave propagation in heterogeneous media while also approximating the true solution quite well.
This work provides a proof of concept that the neural network approach from Nguyen and Tsai [18] can be generalized to three spatial dimensions and furthermore to a broader range of PDEs. This is shown by considering the three dimensional wave equation (2.1) and a slightly different equation, the two dimensional wave equation with variable coefficients (2.2). This proof of concept is given by presenting the approach with focus on the deep learning architecture and additionally presenting the results of a numerical study regarding the accuracy and efficiency of this framework for the mentioned generalizations.

We compare the numerical solution, computed by a neural network for the three dimensional wave equation with the numerical solution computed with the fine solver. This neural network is trained with 4000 samples of parareal-like training data. We conclude, that this neural network output can be used in a parareal scheme to improve the accuarcy further and that it results in a quite accurate approximation of the fine solution after two parareal iterations. Therefore, we conclude that the approach, introduced by Nguyen and Tsai [18], can be generalized to three spatial dimensions and provides an accurate and efficient framework to compute the wave propagation. However we remark, that our numerical studies show, that the combination of the neural network with the parareal scheme is necessary to achieve high accuracy.
An open question for future work is, whether it is possible for the neural network to achieve a higher accuracy to the fine solution without the parareal scheme, when it is trained on another set of training data, either larger in size or generated using a different approach. Achieving this can also improve the approximation through the parareal scheme, since a smaller difference between the fine solver and the enhanced solver results in a faster convergence of the parareal scheme.

Furthermore, we considered the generalization of the approach to a slightly different PDE, the two dimensional wave equation with variable coefficients. Our comparisons show that the neural network trained with 20000 samples of standard training data performs much better than the neural network trained on 5000 samples of parareal-like training data. However, both neural networks must be combined with the parareal scheme to approximate the fine

solver well enough.

We conclude, that the approach from Nguyen and Tsai [18] can be generalized to PDEs slightly different from the standard wave equation. However, we conclude, that the choice of the training data set is important for the efficiency of the training process. In our case the training with standard training data was both, much faster in training time and in the accuracy of the resulting neural network. An open question for future work is, whether the approach can be generalized further to other PDEs.

Acronyms

$\mathcal{F}_{\Delta t^*}$	fine solver
$\mathcal{G}_{\Delta t^*}$	coarse solver
$\tilde{\mathcal{G}}_{\Delta t^*}$	enhanced solver
CDS	Central Difference Scheme
CFL	Courant–Friedrichs–Lewy condition
CNN	Convolutional Neural Network
FNN	Feedforward Neural Network
MSE	Mean Squared Error
NN	Neural Network
PDE	Partial Differential Equation

Bibliography

[1] A. Abdulle, Y. Bai, and T. Pouchon. Reduced basis numerical homogenization method for the multiscale wave equation. *Numerical Mathematics and Advanced Applications - ENUMATH 2013*, pages 397–405, 2014.

[2] G. Bal. On the convergence and the stability of the parareal algorithm to solve partial differential equations. In T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick, R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, and Jinchao Xu, editors, *Domain Decomposition Methods in Science and Engineering*, pages 425–432, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[3] M. Bernacki, S. Lanteri, and S. Piperno. Time-domain parallel simulation of hetereogeneous wave propagation on unstructured grids using explicit, nondiffusive, discontinuous galerkin methods. *Journal of Computational Acoustics*, 14:57–81, 2006.

[4] F.J. Billette and S. Brandsberg-Dahl. The 2004 bp velocity benchmark. 2005.

[5] A. Brougois, M. Bourget, P. Lailly, M. Poulet, P. Ricarte, and R. Versteeg. Marmousi, model and data. 1990.

[6] B. Engquist, H. Holst, and O. Runborg. Multiscale methods for wave propagation in heterogeneous media over long time. In *Numerical Analysis of Multiscale Computations*, pages 167–186. Springer International Publishing, Berlin, Heidelberg, 2011.

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

[8] A. Ibrahim, S. Götschek, and D. Ruprecht. Parareal with a physics-informed neural network as coarse propagator. In *Euro-Par 2023: Parallel Processing*, pages 649–663. Springer International Publishing, 2023.

[9] L. Kaiser. Fast, accurate and scalable numerical wave propagation: Enhancement by deep learning. 2023.

[10] L. Kaiser. Github repository with code. `github.com/utkaiser/masterthesis\_masterthesis\_notebooks`, 2023.

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

[12] H. Lewy, K. Friedrichs, and R. Courant. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100:32–74, 1928.

[13] J.-L. Lions, Y. Maday, and G. Turinici. A "parareal" in time discretization of pde's. *C.R. Math. Acad. Sci. 332, 661-668*, 2001.

[14] X. Meng, Z. Li, D. Zhang, and G. Karniadakis. Ppinn: Parareal physics-informed neural network for time-dependent pdes. *Computer Methods in Applied Mechanics and Engineering*, 371, 2020.

[15] B. Moseley, A. Markham, and T. Nissen-Meyer. Solving the wave equation with pyhsics-informed deep learning. 2020.

[16] I. Neutelings. Neural networks. `https://tikz.net/neural\_networks/`, 2021. Accessed: 2025-08-02.

[17] H. Nguyen and R. Tsai. A stable parareal-like method for the second order wave equation. *Journal of Computational Physics*, 405, 2020.

[18] H. Nguyen and R. Tsai. Numerical wave propagation aided by deep learning. *Journal of Computational Physics*, 475, 2023.

[19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[20] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[21] D. Ruprecht. Wave propagation characteristics of parareal. *Computing and Visualization in Science*, 19:1–17, 2018.

[22] M. Schönleben. Code repository. `github.com/schoenlebenM/Masterthesis`, 2025.

[23] B. Verfürth. Numerical multiscale methods forwaves in high-contrast media. *Jahresbericht der Deutschen Mathematiker-Vereinigung (2023)*, 2023.

[24] L. Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

Appendix

This section presents the code for the three dimensional wave equation and the numerical solver for the two dimensional wave equation. This code is modified from [18] and [10]. The full implementation can be found in [22].

A1 Generate Velocity Profile

```
from scipy.io import loadmat
from skimage.filters import gaussian
from skimage.transform import resize
import numpy as np
import scipy.ndimage


def generate_velocity_crops( resolution = 128, output_dir = 'data/
    crop100_test.npz',  output_dir2 = 'data/crop100_test2.npz',
    num_crops = 10):
    '''
    :param resolution: (int) resolution of the velocity profil,
        usually 128
    :param output_dir: (string) path of the output file for 3D
        velocity profiles, ending with ".npz"
    :param output_dir2: (string) path of the output file for 2D
        velocity profiles, ending with ".npz"
    :param num_crops: (int) number of crops created from each image
    :return: generates and saves the 2D- and 3D-velocity crops in an .
        npz-file
    '''

    # load images
    datamat = loadmat('data/marm1nonsmooth.mat')  # Marmousi velocity
        image
    fullmarm = gaussian(datamat['marm1larg'], 4)  # smoothing the
        image
    databp = loadmat('data/bp2004.mat')  # BP velocity image
    fullbp = gaussian(databp['V'], 4)/1000  # smoothing the image and
        different order of magnitude
```

```
    # randomly crop and save images at "output_dir"
    twoD_profils = generate_velocity_profile_crop(v_images = [fullmarm
        , fullbp], m = resolution, num_times = num_crops)
    wavespeed_list = to_3D(twoD_profils, m = resolution)

    #save the velocity profile in compressed .npz files
    np.savez_compressed(output_dir, wavespeedlist=wavespeed_list)
    np.savez_compressed(output_dir2, wavespeedlist=twoD_profils)



def generate_velocity_profile_crop(v_images, m, num_times):
    '''
    :param v_images: (tensor) full-size velocity profile that needs to
        be cropped
    :param m: (int) resolution, usually 128 (*1, *2 or *3)
    :param num_times: (int) number of crops
    :return: sample (num_times*(number of profils in v_images))-many 2
       d-velocity profiles by cropping randomly rotated and scaled
       images
    '''

    #create list for velocity profiles
    wavespeed_list = []

    #for Marmousi and BP profiles
    for img in v_images:
        #crop num_times many velocity profiles
        for j in range(num_times):
            scale = (
                0.08 + 0.04 * np.random.rand()
            )  # chose this scaling because performed well
            angle = np.random.randint(4) * 22.5  # in degrees
            M = int(m / scale)  # how much we crop before resizing to
               m
            npimg = scipy.ndimage.rotate(img, angle, cval=1.0, order
               =4, mode="wrap")  # bilinear interp and rotation

            h, w = npimg.shape

            # crop but make sure it is not blank image
            while True:
                xTopLeft = np.random.randint(max(1, w - M))
                yTopLeft = np.random.randint(max(1, h - M))
                newim = npimg[yTopLeft : yTopLeft + M, xTopLeft :
                   xTopLeft + M]

                # make sure it is not blank image
```

```
                if ( newim.std() > 0.005 and newim.mean() < 3.8 and
                    not np.all(newim == 0)):
                     npimg = 1.0 * newim
                     break

            wavespeed_list.append(resize(npimg, (m, m), order=4))
    return wavespeed_list



def to_3D(twoD_profils, m = 128):
    '''
    :param twoD_profils: (np.array) 2d-velocity profils
    :param m: (int) resolution of the velocity profil, usually 128
    :return: (np.array) list with 3d-velocity-profils
    '''

    #create list for 3D velocity profiles
    wavespeed_list = []

    for img in twoD_profils:
        img_exp = np.expand_dims(img, axis=0)
        wavespeed = np.tile(img_exp, (m, 1, 1))
        wavespeed_list.append(np.copy(wavespeed))

    return wavespeed_list
```

A2 Generate Standard Training Data

```
import numpy as np
import torch

from utils_visualize import  visualize_one_velocity_crop,
   visualize_wavefield
from utils_generate_veloctiy import generate_velocity_crops,
   crop_center
from utils_energy import WaveSol_from_EnergyComponent_tensor
from utils_numericalSolver import initial_condition_gaussian,
   one_iteration_velocity_verlet_energy,
   one_iteration_velocity_verlet

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def generate_data_end_to_end( input_path = "data/crop_test.npz",
   output_path = "data/datagen_test.npz", n_snaps = 10, res = 128,
                               n_it = 10, f_delta_x =  2. / 128., mode
```

```
                                = "energy_comp", visualize = True,
                                save_img = False):

    '''
    :param input_path: (string) velocity profile data path
    :param output_path: (string) wave field data path
    :param n_snaps: (int) amount of snapshots / dt_star steps to take
    :param res: (int) resolution of wave field output
    :param n_it: (int) number of generated wave fields
    :param f_delta_x: (float) grid time stepping of fine solver
    :param mode: (string) either "physical_components" or "energy_comp
        "
    :param visualize: (boolean) if true, than image will be plotted
    :param save_img: (boolean) if true, the plotted image is saved in
        a file

    :return: saves generated wave propagation iterations in file
    '''

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

    # setup tensors to store wave energy components and velocity
        profile
    Ux, Uy, Uz, Utc = np.zeros([n_it, n_snaps + 1, res, res,res]), np.
        zeros([n_it, n_snaps + 1, res, res, res]), \
                      np.zeros([n_it, n_snaps + 1, res, res, res]), np
                          .zeros([n_it, n_snaps + 1, res, res, res])
    V = np.zeros([n_it, n_snaps+1, res, res,res])

    # training
    for it in range(n_it):              #for each velocity-profil do
        ...
         if it >= len(velocities):
             print("less velocity profils croped than iteration in
                generating_data")
             vel = velocities[0]
         else:
             vel = velocities[it]

         # computing initial wavefield using 3d gaussian pulse (switch
            to pytorch tensor if needed)
         if mode == "physical_components":
             u0, ut = initial_condition_gaussian(torch.from_numpy(vel),
                  mode="physical_components", res_padded=res)
         else: # mode == "energy_comp"
             u_energy, sum = initial_condition_gaussian(torch.
```

```
        from_numpy(vel), mode="energy_comp", res_padded=res,)

# create and save velocity crop
vel_crop = crop_center(vel, res, 2)

V[it] = np.repeat(vel[np.newaxis, :, :, :], n_snaps + 1, axis
   =0)  # save velocity image (n_snaps + 1) times in V

if visualize == True:
# visualize velocity profile used for iterations
    visualize_one_velocity_crop(vel_crop, it)

# integrate dt_star (step size) n_snaps times
for s in range(n_snaps+1):      # for each time step dt_star
   do...

    # visualize Wavefield (and if necessary get wave field
       from energy components beforehand)
    if visualize == True:
        if mode == "physical_components":
            if s>0:
                u0, ut = u0.numpy(), ut.numpy()
            visualize_wavefield(u0, ut, vel, f_delta_x, it, s,
                xy = True, xz = True, yz = True, save =
               save_img)
        else:  # mode == "energy_comp"
            # change energy components to wave field
               representation
            u_elapse, ut_elapse =
               WaveSol_from_EnergyComponent_tensor(u_energy
               [:,0], u_energy[:,1], u_energy[:,2], u_energy
               [:,3],
                                    torch.from_numpy(vel),
                                       f_delta_x, sum)
            u_elapse, ut_elapse = u_elapse.numpy(), ut_elapse.
               numpy()
            visualize_wavefield(u_elapse, ut_elapse, vel,
               f_delta_x, it, s, save = save_img)


    # save current wavefield in output-vector
    if mode == "physical_components":

        Ux[it, s], Uy[it, s], Uz[it, s], Utc[it, s] = u0[0],
           u0[1], u0[2], ut
    else:
        Ux[it, s], Uy[it, s], Uz[it, s], Utc[it, s] = u_energy
           [0,0], u_energy[0,1], u_energy[0,2], u_energy[0,3]
```

```
            # integration step (done for all iterations only not for
                last one)
            if s < n_snaps + 1:
                # apply the velocity verlet solver
                if mode == "physical_components":
                    u0, ut = one_iteration_velocity_verlet(torch.
                        from_numpy(u0), torch.from_numpy(ut), torch.
                        from_numpy(vel), mode = "physical_components")
                else:
                    u_energy = one_iteration_velocity_verlet_energy(
                        torch.cat([u_energy, torch.from_numpy(vel).
                        unsqueeze(dim=0).unsqueeze(dim=0)], dim=1),
                        sum)

    # save tensors in a compressed file, accessible through key-value
        queries
    np.savez_compressed(output_path, vel=V, Ux=Ux, Uy=Uy, Uz=Uz, Utc=
        Utc)
    print("Data generated and saved")
```

A3 Generate Parareal-Like Training Data

```
import numpy as np
import torch

from utils_visualize import  visualize_one_velocity_crop,
    visualize_wavefield
from utils_generate_veloctiy import generate_velocity_crops,
    crop_center
from utils_energy import WaveSol_from_EnergyComponent_tensor
from utils_numericalSolver import initial_condition_gaussian,
    one_iteration_velocity_verlet_energy,
    one_iteration_velocity_verlet

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


def generate_parareal_data_end_to_end( input_path = "data/crop_test.
    npz", output_path = "data/parareal_data.npz", n_crop= 10, res =
    128,
                                       n_snaps = 10, n_parareal = 4,
                                          f_delta_x =  2. / 128.,
                                          mode = "energy_comp",
                                          visualize = True, save_img
                                          = False):
```

```
'''
:param input_path: (string) velocity profile data path
:param output_path: (string) wave field data path
:param n_crop: (int) amount of different wave fields
:param res: (int) resolution of wave field output
:param n_snaps: (int) amount of snapshots / dt_star steps to take
:param n_parareal: (int) number of parareal iterations
:param f_delta_x: (float) grid time stepping of fine solver
:param mode: (string) either "physical_components" or "energy_comp
    "
:param visualize: (boolean) if true, than image will be plotted
:param save_img: (boolean) if true, the plotted image is saved in
    a file

:return: saves generated wave propagation iterations in file
'''

#step 1) generate velocity crops
# load velocity model created in function above `
    generate_velocity_crops()`
velocities = np.load(input_path)['wavespeedlist']

# setup tensors to store wave energy components and velocity
    profile
Ux, Uy, Uz, Utc = np.zeros([n_crop, n_parareal +1, n_snaps +1, res
    , res,res]), np.zeros([n_crop, n_parareal+1, n_snaps + 1, res,
     res, res]), \
                  np.zeros([n_crop, n_parareal+1, n_snaps + 1, res
                      , res, res]), np.zeros([n_crop, n_parareal
                      +1, n_snaps + 1, res, res, res])
V = np.zeros([n_crop, n_parareal +1, n_snaps+1, res, res,res])

# sample training data
for it in range(n_crop):              #for each velocity-profil do
    ...
    print(f'start generating data for velocity crop {it +1}')
    # sample velocity instance
    if it >= len(velocities):
        print("less velocity profils croped than iteration in
            generating_data")
        vel = velocities[0]
    else:
        vel = velocities[it]


    #step 2) generate initial wave field via gaussian pulse
    # computing initial wavefield using 3d gaussian pulse (switch
```

```
    to pytorch tensor if needed)
if mode == "physical_components":
    u0, ut = initial_condition_gaussian(torch.from_numpy(vel),
        mode="physical_components", res_padded=res)
else: # mode == "energy_comp"
    u_energy, sum = initial_condition_gaussian(torch.
       from_numpy(vel), mode="energy_comp", res_padded=res,)

# create and save velocity crop
vel_crop = crop_center(vel, res, 2)
V[it] = np.tile(vel[np.newaxis, np.newaxis, :, :, :], (
    n_parareal + 1, n_snaps + 1, 1, 1, 1))
if visualize == True:
# visualize velocity profile used for iterations
    visualize_one_velocity_crop(vel_crop, it)


# integrate n_snaps time step dt_star (step size)
big_tensor = parareal_algo(torch.cat([u_energy, torch.
    from_numpy(vel).unsqueeze(dim=0).unsqueeze(dim=0)], dim=1)
    ,
                                    n_parareal, n_snaps)
Ux[it], Uy[it], Uz[it], Utc[it] = big_tensor[:, :, 0],
    big_tensor[:, :, 1], big_tensor[:, :, 2], big_tensor[:, :,
     3]



# visualize Wavefield (and if necessary get wave field from
    energy components beforehand)
if visualize == True:
    if mode == "physical_components":
        if s>0:
            u0, ut = u0.numpy(), ut.numpy()
        visualize_wavefield(u0, ut, vel, f_delta_x, it, s, xy
            = True, xz = True, yz = True, save = save_img)
    else:  # mode == "energy_comp"
        # change energy components to wave field
            representation
        u_elapse, ut_elapse =
            WaveSol_from_EnergyComponent_tensor(u_energy[:,0],
             u_energy[:,1], u_energy[:,2], u_energy[:,3],
                                  torch.from_numpy(vel),
                                     f_delta_x, sum)
        u_elapse, ut_elapse = u_elapse.numpy(), ut_elapse.
            numpy()
        visualize_wavefield(u_elapse, ut_elapse, vel,
            f_delta_x, it, s, save = save_img)
```

```
    # save tensors in file, accessible through key-value queries
    np.savez_compressed(output_path, vel=V, Ux=Ux, Uy=Uy, Uz=Uz, Utc=
        Utc)
    print("Data generated and saved")
```

A4 Training the Neural Network

```
import numpy as np
import torch
import random
import os
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "max_split_size_mb:64"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.backends.cudnn.deterministic = True


from utils_numericalSolver import one_iteration_velocity_verlet_energy
from utils_model import Model_end_to_end, get_params,
    fetch_data_end_to_end, save_model


def train_model( model_name = "test", learningrate = .001, batch_size
    = 1, n_epochs = 10, downsampling_model = "Interpolation",
                 upsampling_model = "UNet3", data_paths = "data/
                     datagen.npz", datatype = 'normal'):
    '''
    :param model_name: (string) name of model, used as name for output
        -file containing model parameters
    :param learningrate: (float) learning rate of model
    :param batch_size: (int) batch size
    :param n_epochs: (int) number of iterations model sees all data; i
        .e. amount of epochs model is trained
    :param downsampling_model: (string) name of downsampling model
    :param upsampling_model: (string) name of upsampling model
    :param data_paths: (string) data path of the trainingsdata
    :param datatype: (string) type of the training data, either '
        normal' or 'parareal'
    :return: trained model parameters in ".pt"-file
    '''

    # model setup
    param_dict = get_params()
    model = Model_end_to_end(param_dict, downsampling_model,
        upsampling_model).double()
    if device == "cuda":
        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, datatype = datatype)

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

#for every training epoch
for epoch in range(n_epochs):
    # trainingsmode
    model.train()

    train_loss_list = []  # initialize list to store loss values
        of training
    for i, data in enumerate(train_loader):  # iterate over data
        points in train_loader
        loss_list = []  # create tmp loss list for back
            propagating multiple losses at once
        if datatype =='normal':
            n_snaps = data[0].shape[1]  # number of snapshots
                defined by data input
        else:
            n_snaps = len(train_loader)
        data = data[0].to(device)  # use GPUs if available for
            faster training

        if datatype =='normal':
            for input_idx in random.choices(range(n_snaps - 2), k=
                n_snaps):
                input_tensor = data[:, input_idx].detach()  #
                    detach because if not computation graph would
                    go too far back
                for label_idx in range(input_idx + 1, input_idx +
                    2):
                    label = data[:, label_idx, :4].detach() #
                        corresponding label is given as the fine
                        solver, i.e. the next time step in data
                    output = model(input_tensor)  # apply end-to-
                        end model
                    loss_list.append(loss_f(output, label))  #
                        save loss

        else: #datatype = 'parareal'
            input_tensor = data[0].detach()
```

```
            input_tensor = input_tensor.unsqueeze(dim=0)
            label = data[0].detach()
            #compute label as the wave field propagated by the
               fine solver
            fine_tensor = one_iteration_velocity_verlet_energy(
               label, torch.sum(label[0, :, :, :].clone()),
               batchdimension=False)
            label = fine_tensor.unsqueeze(dim=0)
            # train the model with input_tensor
            output = model(input_tensor)  # apply end-to-end model
            output = output.float()
            label = label.float()
            loss_list.append(loss_f(output, label))  # save loss

        # 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())

    # save parameter for each training epoch as a backup
    print(f"Training erfolgreich, Epoche {epoch + 1}")
    save_model(model, model_name, f't={epoch + 1}', 'results/')
    print(f"parameter saved, Epoche {epoch + 1}")



    # validation
    model.eval()
    with torch.no_grad():

        # initialize list to save losses
        val_loss_list = []
        for i, data in enumerate(val_loader):  # iterate over data
            points in val_loader
            if datatype == 'normal':
                n_snaps = data[0].shape[1]  # number of snapshots
                   defined by data input
            else:
                n_snaps = len(train_loader)
            data = data[0].to(device)  # use GPUs if available for
                faster training

            if datatype == 'normal':
```

```
                    input_tensor = data[:, 0].detach()  # detach
                        because if not computation graph would go too
                        far back
                    vel = input_tensor[:, 4].unsqueeze(dim=1)  #
                        access velocity profile in input_tensor
                    for label_idx in range(1, n_snaps):  # advance a
                        wave field for (n_snaps - 1) time steps
                        label = data[:, label_idx, :4]
                        output = model(input_tensor.to(device))  #
                            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())
                        input_tensor = torch.cat((output, vel), dim=1)

                else:  # datatype = 'parareal'
                    input_tensor = data[0].detach()
                    input_tensor = input_tensor.unsqueeze(dim=0)
                    #compute the corresponding label with the fine
                        solver
                    label = data[0].detach()
                    fine_tensor = one_iteration_velocity_verlet_energy
                        (label, torch.sum(label[0, :, :, :].clone()),
                        batchdimension=False)
                    label = fine_tensor.unsqueeze(dim=0)
                    #evaluate the neural network with the input_tensor
                        and the corresponding label
                    output = model(input_tensor.to(device))  # apply
                        end-to-end model

                    # get and save loss
                    val_loss_list.append(loss_f(output, label).item())

        print(f'epoch %d, train loss: %.8f, test loss: %.8f'
              %(epoch + 1, np.array(train_loss_list).mean(), np.array(
                  val_loss_list).mean()))

    # save final model parameters after all training epochs as a ".pt
        "-file
    save_model(model, model_name, 'final', 'results/')
```

A5 Applying the Parareal Scheme

```
import sys
import torch
import time
```

```
sys.path.append("..")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

from utils_numericalSolver import one_iteration_velocity_verlet_energy


def parareal_scheme(model, u_0, n_parareal, n_snapshots,
    boundary_conditions, t=True):
    '''
    :param model: (pytorch.model) trained model
    :param u_0: (torch tensor) energycomponent of initial wavefield
    :param n_parareal: (int) number of parareal iterations
    :param n_snapshots: (int) number of timesteps delta_t_star
    :param boundary_conditions: (string) 'periodic' or 'absorbing'
    :param t: (boolean): if true, time of parareal computation is
        measured
    :return: the parareal computation, for all pararal iterations and
        time steps
    '''

    u_n = u_0.clone()
    vel = u_0[:, 4].clone().unsqueeze(dim=0)
    batch_size, channel, width, height, depth = u_n.shape
    if boundary_conditions == "absorbing":
        width, height, depth = width * 2, height * 2, depth * 2

        # create tensor to save results during iterations
    big_tensor = torch.zeros([n_parareal + 1, n_snapshots, batch_size,
        channel - 1, width, height, depth])
    big_tensor[0, 0] = u_0[:, :4].clone()

    # initial guess, first iteration without parareal
    start_time = time.time()
    print(f"Performing initial guess.")
    for n in range(n_snapshots - 1):
        u_n1 = model(u_n)
        big_tensor[0, n + 1] = u_n1
        u_n = torch.cat((u_n1, vel.to(device)), dim=1)
    end_time = time.time()
    run_time_NN = end_time - start_time

    # parareal iterations: k = 1, 2, ...
    for k in range(1, n_parareal + 1):
        print(f"Performing Parareal iteration {k}.")

        big_tensor[k, 0] = u_0[0, :4].clone()
        if t == True:
```

```
            parareal_terms, time_parallel = get_parareal_terms(model,
                big_tensor[k].clone(), n_snapshots, vel.clone(), t=
                True)
            run_time_NN = run_time_NN + time_parallel
        else:
            parareal_terms = get_parareal_terms(model, big_tensor[k].
                clone(), n_snapshots, vel.clone(), t=False)

        new_big_tensor = torch.zeros([n_snapshots, batch_size, channel
            - 1, width, height, depth])
        new_big_tensor[0] = u_0[:, :4].clone()
        start_time = time.time()
        for n in range(n_snapshots - 1):
            u_n_k1 = torch.cat((new_big_tensor[n], vel), dim=1)
            u_n1_k1 = model(u_n_k1).to("cpu") + parareal_terms[n]
            new_big_tensor[n + 1] = u_n1_k1
        end_time = time.time()
        run_time_NN = run_time_NN + end_time - start_time

        if k < n_parareal:
            big_tensor[k + 1] = new_big_tensor.clone()

    if t == True:
        return big_tensor, run_time_NN
    else:
        return big_tensor


def get_parareal_terms(model, big_pseudo_tensor, n_snapshots, vel, t=
    True):
    '''
    :param model: (pytorch.Model) trained end-to-end model to advance
        a wave front
    :param big_pseudo_tensor: (pytorch tensor) tensor containing
        previous solution (high resolution due to pseudo-spectral
        cropping)
    :param n_snapshots: (int) number of iterations (number of
        iterations with length dt_star), i.e. number of timesteps
    :param vel: (pytorch tensor) velocity profile
    :param t: (boolean) if true, compute time for pararaeal iteration
    :return: get Parareal terms that can be computed in parallel
    '''

    with torch.no_grad():
        parareal_terms = torch.zeros(big_pseudo_tensor.shape)
        for s in range(n_snapshots):
            if s == 1:
                start_time = time.time()
```

```
            parareal_terms[s] = compute_parareal_term(model, torch.cat
                ([big_pseudo_tensor[s], vel], dim=1))
            if s == 1:
                end_time = time.time()
                run_time_para = end_time - start_time
    if t == True:
        return parareal_terms, run_time_para
    else:
        return parareal_terms


def compute_parareal_term(model, u_n_k):
    '''
    :param model: (pytorch.Model) end-to-end model to advance a wave
        front
    :param u_n_k: (pytorch tensor) current wave field
    :return: difference between Parareal terms of right-hand side of
        main Parareal equation
    '''

    res_fine_solver = one_iteration_velocity_verlet_energy(u_n_k, sum=
        torch.sum(u_n_k[:, 0]))  # = F(u_n^k)
    res_model = model(u_n_k)  # = G(u_n^k)
    return res_fine_solver.to(device) - res_model.to(device)  # = F(
        u_n^k) - G(u_n^k)
```

A6 Numerical Solver for 3D Wave Equation

```
import torch



def velocity_verlet_tensor(u0, ut0, vel, dx, dt, delta_t_star,
    batchdimension = False, boundary_c="periodic"):
    '''
    :param u0: (pytorch tensor) physical wave component, displacement
        of wave
    :param ut0: (pytorch tensor) physical wave component derived by t,
         velocity of wave
    :param vel: (pytorch tensor) velocity profile dependent on x_1 and
         x_2 and x_3
    :param dx: (float) time step in both dimensions / grid spacing
    :param dt: (float) temporal step size
    :param delta_t_star: (float) time step a solver propagates a wave
        and solvers are compared
    :param batchdimension: (boolean) True, if batch is added as a
        dimension to u
```

```
:param boundary_c: (string) choice of boundary condition, "
   periodic" or "absorbing"
:return: propagate wavefield using velocity Verlet in time and the
    second order discrete Laplacian in space
'''

def _periLaplacian_tensor(v, dx, batchdimension = False):
    '''
    :param v: (pytorch Tensor) Wave Solution u for time t
    :param dx:  (float) grid spacing in x_1, x_2 and x_3 dimension
    :param batchdimension: (boolean) tells, if u contains a
       dimension for batches, If true, change number from 0 to 1
    :return: compute discrete Laplacian with periodic boundary
       condition
    '''
    number = 0
    if batchdimension == True:
        number = 1


    Lv = (torch.roll(v, 1, dims=1 + number) - 2 * v + torch.roll(v
       , -1, dims=1 + number)) / (dx**2) + \
         (torch.roll(v, 1, dims=0 + number) - 2 * v + torch.roll(v
            , -1, dims=0 + number)) / (dx**2) + \
         (torch.roll(v, 1, dims=2 + number) - 2 * v + torch.roll(v
            , -1, dims=2 + number)) / (dx**2)
    return Lv

Nt = round(abs(delta_t_star / dt))
c2 = torch.mul(vel, vel)                    # = c^2
u, ut = u0, ut0

if boundary_c == "periodic":
    for i in range(Nt):
        #compute Velocity Verlet Stepping Scheme

        ddxou = _periLaplacian_tensor(u, dx, batchdimension =
           batchdimension)
        u = u + dt * ut + 0.5 * dt**2 * torch.mul(c2, ddxou)
        ddxu = _periLaplacian_tensor(u, dx, batchdimension=
           batchdimension)
        ut = ut + 0.5 * dt * torch.mul(c2, ddxou + ddxu)
    return u, ut

else:
    raise NotImplementedError("this boundary condition is not
       implemented")
```

A6 Numerical Solver for 2D Wave Equation

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


def varied_velocity_verlet_tensor(u0, ut0, vel, dx, dt, delta_t_star,
    boundary_c="periodic", batchdimension = False):
    '''
        :param u0: (pytorch tensor) physical wave component,
           displacement of wave
        :param ut0: (pytorch tensor) physical wave component derived by
            t, velocity of wave
        :param vel: (pytorch tensor) velocity profile dependent on x_1
           and x_2 and x_3
        :param dx: (float) time step in both dimensions / grid spacing
        :param dt: (float) temporal step size
        :param delta_t_star: (float) time step a solver propagates a
           wave and solvers are compared
        :param boundary_c: (string) choice of boundary condition, "
           periodic"
        :param batchdimension: (boolean) True, if batch is added as a
           dimension to u

        :return: propagate wavefield using velocity Verlet in time and
           the second order discrete Laplacian in space
        '''


    def _varied_periLaplacian_tensor(v, vel, dx, batchdimension =
        False):
        '''
        :param v: (pytorch tensor) Wave Solution u for time t
        :param vel: (pytorch tensor) velocity profile
        :param dx:  (float) grid spacing in x_1, x_2 and x_3 dimension
        :param batchdimension: (boolean) tells, if u contains a
           dimension for batches, If true, change number from 0 to 1
        :return: compute discrete Laplacian with periodic boundary
           condition
        '''

        number = 0
        if batchdimension == True:
            number = 1

        c2 = torch.mul(vel, vel)
        additional_sum = torch.mul(torch.roll(vel, 1, dims=1 + number)
```

```
        , torch.roll(v, 1, dims=1 + number)) \
                        - torch.mul(torch.roll(vel, 1, dims=1 +
                            number), torch.roll(v, -1, dims=1 +
                            number))\
                        - torch.mul(torch.roll(vel, -1, dims=1 +
                            number), torch.roll(v, 1, dims=1 + number
                            ))\
                        + torch.mul(torch.roll(vel, -1, dims=1 +
                            number), torch.roll(v, -1, dims=1 +
                            number))\
                        + torch.mul(torch.roll(vel, 1, dims=0 +
                            number), torch.roll(v, 1, dims=0 + number
                            ))\
                        - torch.mul(torch.roll(vel, 1, dims=0 +
                            number), torch.roll(v, -1, dims=0 +
                            number))\
                        - torch.mul(torch.roll(vel, -1, dims=0 +
                            number), torch.roll(v, 1, dims=0 + number
                            ))\
                        + torch.mul(torch.roll(vel, -1, dims=0 +
                            number), torch.roll(v, -1, dims=0 +
                            number))
    sum = torch.mul(vel, additional_sum)
    Lu_sum= torch.roll(v, 2, dims=1 + number) - 2 * v + torch.roll
        (v, -2, dims=1 + number) \
             + torch.roll(v, 2, dims=0 + number) - 2 * v + torch.
                 roll(v, -2, dims=0 + number)
    Lu = 0.5 * torch.mul(c2, Lu_sum)
    Lv = (0.5 * (sum +Lu)) / (dx**2)
    return Lv

Nt = round(abs(delta_t_star / dt))
u, ut = u0, ut0

if boundary_c == "periodic":
    for i in range(Nt):
        # Compute stepping scheme for Varied Velocity Verlet
        ddxou = _varied_periLaplacian_tensor(u, vel, dx,
            batchdimension = batchdimension)
        u = u + dt * ut + 0.5 * dt**2 * ddxou
        ddxu = _varied_periLaplacian_tensor(u, vel, dx,
            batchdimension=batchdimension)
        ut = ut + 0.5 * dt * (ddxou + ddxu)
    return u, ut

else:
    raise NotImplementedError("this boundary condition is not
        implemented")
```

Versicherung zur Selbstständigen Leistungserbringung

Ich versichere, dass ich die vorstehende schriftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

M. Schönleben

Würzburg, 06.08.2025, Miriam Schönleben