Technical University of Munich

School of Computation, Information and Technology - Informatics

Master's Thesis

to attain the degree

Master of Science Computational Science and Engineering

Physics-Informed Deep Learning for Wave-Based Seismic Imaging

Submitted by

Kai Nierula

Technical University of Munich

School of Computation, Information and Technology - Informatics

Master's Thesis

to attain the degree

Master of Science Computational Science and Engineering

Physics-Informed Deep Learning for Wave-Based Seismic Imaging

-

Physikalisch informiertes Deep Learning für wellenbasierte seismische Bildgebung

Submitted by

Kai Nierula

Supervisor: Prof. Dr. Michael Bader

Advisors: Dr. Ban-Sok Shin,

Sebastian Wolf

Submission date: 30 June 2023

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

Munich, 30 June 2023 _____

Kai Nierula

Abstract

This thesis investigates using a physics-informed continuous conditional Generative Adversarial Network (CcGAN) for simulating seismic wave propagation. Seismic wave simulations are a key element in seismic imaging used for subsurface exploration and discovery. In recent years, physics-informed machine learning (ML) and deep learning (DL) methods have emerged as valuable additions or substitutes to classical numerical simulation of partial differential equations (PDEs), one of their main benefits being fast computing time after training. We extend Kadeethum, et al. [1]'s CcGAN designed to solve a time-dependent PDE on a 2D domain by adding a physics consistency-based loss. The time-depended PDE of interest in this thesis is the acoustic wave equation. A CcGAN allows for generalization across velocity distribution inputs and handling continuous conditional variables like time. To our knowledge, this is the first use of a physics-informed CcGAN.

We first compare a traditional CcGAN to a physics-informed one using uniform velocity distributions. Contrary to expectations, the traditional one outperformed the physics-informed one. Despite this, we applied the physics-informed CcGAN to a data set consisting of horizontally layered velocity distributions, hypothesizing that the advantages of using a physics-informed approach would become apparent with a more complex problem. However, the model showed mode collapse on the validation and test data sets, generating identical pressure wavefields regardless of the input velocity distributions. This mode collapse, a common issue in training traditional GANs, persisted despite employing a Wasserstein gradient-penalty GAN, which should have mitigated this problem.

Encouragingly, our model generated varied pressure wavefields on the training dataset. Additionally, it demonstrated the ability to handle wavefield progression based on input times, thereby enabling the possibility of querying wavefields at individual timesteps without the need for preceding ones. Unfortunately, continuous time-stepping defaulted back on timesteps used during training.

These findings underscore the need for additional research to address the unexpected physics-informed CcGAN behavior. Despite the challenges, the possibility of querying wavefields at arbitrary time steps highlights the potential of using DL methods to contribute to computational speed-ups in seismic simulation and imaging.

Acknowledgements

This thesis would not have been completed without the help of many people. First and foremost, I would like to thank Prof. Dr. Michael Bader for the opportunity to write this thesis under his supervision. I would like to thank Dr. Ban-Sok Shin for taking on the co-supervision, for lending his time and expertise, and for the warm welcome to the German Aerospace Center. I am very grateful for the support I received from Sebastian Wolf, who always offered his help an was quick to respond to any issues. Furthermore, I want to thank Dr. Linda Sauer Bredvik for taking the time to proofread my thesis and for helping me navigate the intricacies of the English language. Many thanks to my colleagues at the German Aerospace Center for the joyful distraction during breaks and the support during times of writing. I also want to thank Anne-Cathrine for proof-reading and motivating me. Last but not least, I would like to thank my parents. Their permanent support reaches far beyond this thesis.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Waves are an ubiquitous phenomenon that influences nearly every facet of our world. The most common aspect when defining waves is that they are a disturbance, i.e., a change from an equilibrium state [2]. They appear either as mechanical waves such as sound waves or electromagnetic waves such as light waves [3]. These examples already prove the importance of waves in our everyday life. In addition, they find extensive applications across many scientific and engineering disciplines.

Consider our modern information era, where waves play a crucial role in transporting information over long distances, e.g., using fiber-optics [4], to more close range applications such as wireless networking [5]. In medicine, wave phenomena are used to create detailed images of the body, helping healthcare providers make informed decisions on treatments [6]. Waves even provide insights into the very fabric of our universe from the quantum (e.g. wave-particle duality, where particles like photons exhibit wave-like properties [7]) to the cosmological scale (e.g., gravitational waves [8]). In geophysics, waves are indispensable for estimating subsurface structures [9], aiding in the discovery of resource deposits [10], understanding a planet's deep structure [11], and assessing potential natural hazards like earthquakes [12], amongst many other applications.

This process of estimating subsurface structures is referred to as seismic imaging [9] and often relies heavily on solving the wave equation that describes how waves propagate through a medium [13]. Over the years, many highly sophisticated numerical methods have been developed to solve the wave equation by discretizing the equation and employing iterative time-stepping schemes [14]. However, a key challenge is still their computational cost, especially in the 3D domain [15]. Additionally, numerically solving the wave equation creates challenges in terms of stability and meshing [14].

In recent years machine learning, and, in particular, deep learning techniques have shown promising results in their ability to simulate physical phenomena in a variety of scientific fields. In the field of fluid dynamics, machine and deep learning have been used for turbulence modeling around airfoils, significantly reducing the computational cost [16], [17]. In the realm of material science, machine learning has assisted in predicting properties of new materials, thus accelerating the discovery process [18]. In computational biology, deep learning has facilitated the understanding and prediction of protein structure and function [19]. In the context of geophysics, deep learning has shown potential in detecting and locating earthquakes [20], fast estimation of ground motion after earthquakes [21], volcano monitoring via automatic classification of volcano seismic events [22], and landslide susceptibility mapping for hazard assessment [23].

Many of these methods typically rely solely on their training data and therefore perform poorly outside of it, meaning that they do not generalize to other, but very similar problems [24]. As an extension to this purely data-driven approach, *physics-*

informed neural networks (PINNs), introduced by Raissi, et al. [25] insert existing physical knowledge into the neural network, e.g., using the underlying physical laws. The goal: ensuring physics-correct output and better generalization [25]. In the field of computational chemistry, physics-informed deep learning has been used to solve the time-independent electronic Schrödinger equation of a given atomic system to predict molecular properties by integrating a quantum wave-function ansatz [26]. In meteorology and climatology, physics-informed machine learning methods have been used for weather and climate modelling by incorporating physical knowledge of the atmosphere [27]. In geophysics, physics-informed neural networks have been used for fault prediction [28] and modelling of crustal deformation due to earthquakes [29], using the governing equations of rock mechanics.

These examples underscore the transformative potential of deep learning across many disciplines and in particular in geophysics. Given the significant computational cost of traditional methods in solving the wave equation, the primary motivation of this work is to explore an alternative approach using physics-informed deep learning to solve the wave equation.

1.2 Related work and contribution

There are two prevalent research directions on (physics-informed) machine learning and deep learning for seismic imaging. The first one is to simulate the wave equation, while the second focuses on predicting subsurface properties directly from measurements. In the context of seismic imaging, the former can be used in conjunction with classical numerical methods. In contrast, the second approach aims to completely replace traditional methods.

1.2.1 Wave equation solver

Here, we will shortly review a few examples of using a physics-informed deep learning approach to solve the wave equation. Karimpouli, et al. [30] solved the wave equation on a 1D domain with constant velocity using physics-informed deep learning. They relied on training data across the entire space and time domain. Moseley, et al. [31] solved the wave equation in a 2D domain, relying on numerical simulation for early wavefields, and showed that physics-informed deep learning can extrapolate from that point on, even in complex subsurface media. Rasht-Behesht, et al. [32] extended this 2D approach with more realistic boundary conditions, namely a reflective boundary at the top of the domain. While being only an extract of all studies done on solving the wave equation using physics-informed deep learning, they are all limited in that they solve the wave equation for a single subsurface only. Therefore, for other subsurfaces of interest, retraining is necessary. This is more compute-intensive than traditional numerical methods [31]. The advantage of these methods is that they are mesh-free and, once trained, arbitrary space-time points can be queried extremely fast [31].

1.2.2 Predicting subsurface properties

Using a deep learning approach to predict realistic subsurface properties from measurement data only is an active field of research. In addition to a wave equation solver, Rasht-Behesht, et al. [32] used a physics-informed deep learning approach to identify a subsurface that fits both the measurement data and the initial wavefields, both simulated using traditional numerical methods. Each distinct subsurface required its network, and the correct subsurface was found in an iterative manner during the training period. Song, et al. [33] moved the problem to the frequency domain. They first used a physics-informed network to reconstruct the wavefield in the frequency domain on a specific domain for a single source frequency. They relied on training data from a classical numerical simulation for this portion. Afterwards, a second physics-informed network was used to map the wavefield produced previously to a subsurface velocity. Wang, et al. [34] introduced a combination of two networks that do not use any physics-information or training data, but that can still generalize to other subsurfaces. The training process starts by giving an initial velocity distribution to the first network, which acts as a wave equation solver. The produced wavefields are then passed to a second network that tries to output the initial velocity distribution again. Using a complex, cyclic interplay, these two networks are trained so that the second network can be used to find subsurfaces that correspond to measurement data.

Direct subsurface imaging is advantageous in that the structural complexity is much simpler in nature than the wavefield's variations in space and time. This is the case as the subsurface is assumed to be constant in time for seismic imaging [32]. Still, as shown by these, this approach typically requires either simulated measurement data for a known subsurface or inversion results from a classical method if only measurement data is present.

1.2.3 Our contribution

In this study, we choose to focus on solving the wave equation instead of attempting to go directly from measurement data to estimating subsurface properties. Our reasoning is that having a (universal) wave-equation solver could be used with the already proven traditional numerical methods for seismic imaging. Moreover, to have a universal seismic imaging method would require obtaining a extremely large set of realistic velocity models that contain complex structures, such as salt bodies and faults [35].

The primary objective of our research is to explore the possibility of using a physics-informed deep learning method to generate pressure wavefields at **arbitrary** time steps, without the need to for a time-stepping scheme. Additionally, our deep learning method should have the ability to **generalize** to varying velocity distributions The ability to solve the wave equation more rapidly could significantly improve the inversion process, providing an advantageous edge in subsurface exploration.

This thesis is divided into six parts. **Chapter 2** introduces the topic of seismic imaging and deep learning, with a focus on physics-informed GANs. **Chapter 3** is concerned

with the methodology of this thesis. The wavefield data generation is explained and the neural network architecture, configuration, and training procedure detailed. **Chapter 4** demonstrates the results, split into two distinct parts. The first portion is a comparison between a traditional and a physics-informed neural network on a simple problem. The second is a detailed look at the neural network's performance on a more complex issue. In **Chapter 5**, the results and limitations of our approach are discussed. **Chapter 6** provides a conclusion for this thesis and gives an outlook for further research.

2. Theoretical background

2.1 Seismic imaging

The goal of seismic imaging is to estimate subsurface parameters from seismic data. Interpreting the distribution of these parameters then allows for an estimation of the geometry and lithology of subsurface layers [9]. Exemplary subsurface parameters are the spatial distribution of P- or S-wave velocities. P-waves (pressure waves) refer to waves traveling longitudinally, meaning that particles hit by this wave oscillate back and forth around their equilibrium position in the same direction as the wave propagates. They travel faster than S-waves (secondary or shear waves), which is why they are also called primary waves [36]. S-waves travel transversely, meaning that the particle motion is confined to the planes perpendicular to the direction of propagation [37].

To get an interpretable subsurface image, two main steps need to occur beforehand: seismic data acquisition and data processing. Seismic data acquisition is the process of generating seismic signals and the reception and storage of those signals after they have traveled through the interior of the body of interest. During the processing step an attempt is made to remove all effects on the signal that are not from the causative structure of interest or undesired in the context of the current research question [38]. An exemplary early processing measure is a gain correction, compensating for the reduction in signal amplitude due to wavefront divergence from geometric spreading [10]. Different research fields consider different signals to be undesirable, so while reflection seismics - a method commonly used in exploration of hydrocarbons - tries to remove non-geological ambient noise such as wind-driven surface gravity waves [38], there is an active research field that uses natural sources, e.g., to relate Antarctic sea ice extent to seismic activity [39]. Therefore, the amount and method of processing differs widely. An in-depth introduction to the topic of seismic data analysis is provided by Yilmaz [10].

2.1.1 Wave equation

The PDE describing the propagation of seismic waves, e.g. P- or S-waves, is the wave equation [11]. A few assumptions are usually done on the type of materials for which the PDE is introduced. These are acceptable in most cases when it comes to studying the subsurface and are the following [37]:

- Perfect elasticity, i.e., the material goes back to original shape after deformation,
- Isotropy, i.e., the material's properties are the same in all directions, and
- Neglection of body forces, i.e., gravity

With these assumptions, the wave equation is [11]:

$$\rho\frac{\partial^2 \boldsymbol{u}}{\partial t^2} - \nabla\lambda(\nabla\cdot\boldsymbol{u}) - \nabla\mu\cdot\left[\nabla\boldsymbol{u} - (\nabla\boldsymbol{u})^T\right] - (\lambda+2\mu)\nabla\nabla\cdot\boldsymbol{u} + \mu\nabla\times\nabla\times\boldsymbol{u} = s\,, \quad (2.1)$$

with ρ the density, $\boldsymbol{u}(\boldsymbol{x}, t)$ the displacement vector with components u, v, w, $s = s(\boldsymbol{x}, t)$ the source which is an applied force. λ and μ are the two elastic moduli for solids describing the stress-strain relation, also known as Lamé parameters, given by

$$\lambda = \frac{\sigma E}{(1+\sigma)(1-2\sigma)}, \qquad \mu = \frac{E}{2(1+\sigma)}, \tag{2.2}$$

where E is Young's modulus and σ is Poisson's ratio [40].

The second and third subtrahends in equation (2.1) involve gradients in the Lamé parameters and are non-zero whenever the material is inhomogeneous, making it difficult to solve efficiently. If the velocity of the seismic waves is only a function of depth - meaning the velocity distribution is horizontally layered - each material in these layers can be treated independently as homogeneous and the gradients of the Lamé parameters are 0. The solutions in the different media are then linked with an additional process. This is a common approach for solving the wave equation [11].

In this thesis, we only consider acoustic waves traveling in a 2D fluid medium that is horizontally layered or homogeneous. Therefore, the gradients of the Lamé parameter are 0 and only P-waves travel through the medium A further simplification to acoustic waves is commonly done to show that a certain wave-propagation method has promise, e.g., in Moseley, et al. [31], Rasht-Behesht, et al. [32]. Combining these further simplifications we get the well-known and easier formulation of the wave equation [41]:

$$\frac{1}{c^2}\frac{\partial^2 P}{\partial t^2} - \nabla^2 P = s\,, \tag{2.3}$$

where $P(\boldsymbol{x}, t)$ is the time-varying scalar pressure wavefield and per-layer constant speed of sound c.

2.1.2 Seismic inversion

There are multiple ways of getting from the recorded and - to some degree - processed data to an estimation of subsurface properties such as the P-wave velocity. One method with the most active research [42] is the *Full-Waveform Inversion* (FWI), which has the ability to provide high-resolution quantitative - not only structural - results [43]. In one of the main early works on this method, Tarantola [13] proposes a way to handle full-wavefield data for inversion. This is in contrast to other methods that rely on only part of the wavefield by not taking into account all different waveforms and/or only relying on the amplitude - not the phase - of the recorded wavefield [44]. The goal of FWI is to solve the minimization problem

$$E(m) = \min_m \left\{ \|\boldsymbol{d}_{\text{obs}} - \boldsymbol{d}_{\text{calc}}(m))\|_2^2 \right\}, \tag{2.4}$$

where $m = m(\boldsymbol{x})$ is a function of a subsurface property at a certain position $\boldsymbol{x} = (x, y, z)$, $\boldsymbol{d}_{\text{obs}}$ is a vector of sampled measurement data at receiver position $r(\boldsymbol{x})$. The calculated data $\boldsymbol{d}_{\text{calc}}$ sampled at certain receiver positions are a function of the model m. This calculation is based on the wave equation (2.3) [43], [45].

The actual process of FWI is quite difficult. It is a non-linear, ill-posed optimization problem that is solved iteratively [13], [46]. In each iteration step, the model m is updated to reduce the misfit function (2.4). As per Tarantola [13], each step in the iterative procedure encompasses a forward wave propagation simulation originating from the actual source, as well as a forward propagation (backward in time) of the residuals between the observed and simulated data at the receiver locations. Forward wave propagation refers to a simulation that solves the wave equation at successive timesteps, while "backward in time" refers to the technique of propagating the wavefield residuals from the receiver positions back towards the source, effectively using negative timesteps as if reversing time. Subsequently, the two wavefields are correlated at each spatial point, yielding a correction of the model. With this being only a very superficial description of the process, the interested reader is referred to Tarantola [13] to delve deeper into the topic.

This procedure requires both a good initial model and many computational resources for larger seismic experiments, which is why a more widespread use has only become possible recently, thanks to improved algorithms and progress in high-performance computing [44]. Efficient numerical modeling of the complete seismic wavefield remains a central challenge in FWI [46], as many wave propagation simulations solving wave equation (2.1) need to be done per iteration step.

2.2 Deep learning

Deep Learning (DL) is a subset of *Machine Learning* (ML) which in itself is an *Artificial Intelligence* (AI) method [47]. There is no unique and agreed upon definition of AI [48]. McCarthy [49] describes AI as “the science and engineering of making intelligent machines, especially intelligent computer programs [...], but AI does not have to confine itself to methods that are biologically observable.” Again, there is no general definition of intelligence, so we again refer to McCarthy [49], who defines intelligence as “the computational part of the ability to achieve goals in the world”. ML is defined by Bishop [50] as a discipline where computers are programmed to optimize a performance criterion using example data or past experience without being explicitly programmed. Goodfellow, et al. [47] define DL as a specific kind of machine learning where the underlying algorithms are inspired by the structure and function of the brain.

Conventional ML methods struggle to work on raw data and instead require manual engineering of a feature extractor to turn data into a representation with which the method can work [51]. A good example of this is given in Goodfellow, et al. [47]: a conventional ML method cannot directly infer medical recommendations from MRI scans, but can do so if given relevant information, e.g., the presence or absence of certain structures in the scan. Deep Learning methods on the other hand can be fed with raw data, as they have the ability to automatically discover the representations needed for the task at hand [51]. In this thesis, we will make use of a DL method.

There are three common forms of machine learning, be it deep or not. These are *supervised*, *unsupervised* and *reinforcement learning* [52]. Supervised learning is defined by the use of a labeled data set to classify or predict outcomes [53]. In unsupervised learning,

the model can analyze and cluster unlabeled data sets, identifying hidden patterns in the data set [53]. In reinforcement learning, the model learns through trial and error; it is rewarded for good outcomes and penalized for bad ones without there being a ground truth label [52]. The most common form is supervised learning [51], which is the one used in this thesis.

2.2.1 Introduction to deep learning procedure

Deep Learning is built upon *Artificial Neural Networks* (ANNs), in short *neural networks* (NNs) [54]. NNs can be seen as a universal approximator [55], meaning that they can approximate any continuous function with an arbitrary precision [56].

In their simplest form, NNs consist of an input, a hidden and an output layer. Each layer l contains many connected neurons, which are simple processing elements that work on inputs $\boldsymbol{u}$, part of the weighting parameter matrix $\boldsymbol{W}$ and biases $\boldsymbol{b}$ [57]. The elements W_{ij}^l in weight matrix $\boldsymbol{W}$ are used to connect the ith neuron in layer $l-1$ with the jth neuron in layer l. The weighting parameter controls the strength of the connection of two neurons. A bias term b_i is used as a shifting parameter. The output for the kth neuron in the lth layer (u_k^l) can be determined by a weighted sum of the inputs (outputs of the previous layer u^{l-1}) as follows [50]:

$$u_k^l = \phi\left(\sum_{j=1}^{k_l-1} w_{kj}^l u_j^{l-1} + b_k^l\right) , \tag{2.5}$$

where ϕ is a non-linear activation function. The reason for non-linear activation function is that in practice, data is generally not separable by linear functions (see Fig. 2.1). Therefore, non-linear activation functions are needed as otherwise even deep NNs will only produce output as linear function of inputs [58]. In the example below (see Fig. 2.1), this would mean that a NN would act as a function that when inputting a (x, y) - coordinate, outputs the color to which a dot at that position would belong to.

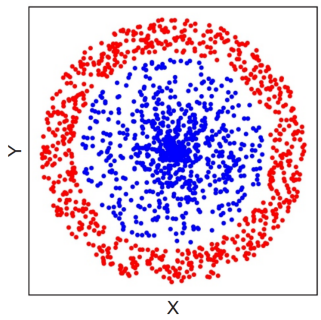

Figure 2.1: No linear function can separate the two colored data sets (assuming no further modifications to the representation).

The *deep* in deep learning refers to the use of multiple hidden layers, creating a depth in the network structure [47] (see Fig. 2.2). NNs in which all of the nodes in a layer are connected to all the output nodes in the next layer for all layers as in Fig. 2.2 are called *fully-connected neural networks* [59] or equivalently *multilayer perceptrons* (MLPs) [47].

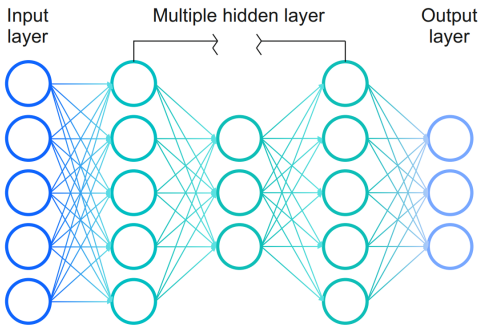

Figure 2.2: Deep neural network structure modified after Kavlakoglu [54]. Each circle corresponds to a neuron and carries equation (2.5). Different number of neurons per layer are allowed.

Our introduction to supervised learning is based on LeCun, et al. [51], if not noted otherwise. Our exemplary supervised learning task is to correctly identify objects shown in an image. This is a common goal in supervised learning. As a first step, a large data set is gathered or generated and each sample from the data set is labelled with its corresponding category, meaning the feature that is present in the image (e.g. a subsurface fault structure). The NN outputs a vector of scores for each image passed to it. Each entry in the score vector corresponds to one of the categories present in the data set. We want the NN to give the highest value to the entry that corresponds to the object present in the image. To get the NN to perform this task, a training phase for the NN is necessary. In this phase, an *objective function* J that measures the error between the output scores and the desired ones is used. Other terms for the objective function are *cost function* or *loss function* [47]. Based on the measured error, adjustable parameters - of which there can be hundreds of millions in modern NNs - are altered to minimize this error. These adjustable parameters are the *weights* $\boldsymbol{W}$ and *biases* $\boldsymbol{b}$. We will note both of them together as $\boldsymbol{\theta}$. The objective function is a high-dimensional function of $\boldsymbol{\theta}$: $J(\boldsymbol{\theta})$. By calculating the negative gradient vector $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ of this function with respect to (w.r.t.) the current adjustable parameters, one can find the direction of steepest descent, indicating the direction of a local minimum where the cost function's value is smaller. The weights and biases are then adjusted with a small step towards the local minimum. This method is known as gradient descent [60]:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \,, \tag{2.6}$$

where η is the learning rate, determining the size of the step taken toward the (local) minimum. This procedure is repeated during the training phase to optimize the NN. One should keep in mind that with gradient descent, one cannot guarantee that a global minimum can be found. This is often irrelevant in practice, as most local minima provide adequate results [61].

There is an additional difference between traditional optimization problems and optimization for NNs as Goodfellow, et al. [47] point out: In machine learning only the cost function $J(\boldsymbol{\theta})$ over the finite data set can be minimized but not over the true

underlying data distribution as this is unknown. The finite data set that is used during training is called *training data set*. The hope is that by minimizing the cost function in the training phase, minimization also occurs w.r.t the underlying data distribution. This procedure is prone to overfitting, meaning that the NN memorizes the training data set instead of generalizing to the underlying data distribution. To mitigate the issue of overfitting, a stopping criterion based on the performance of the NN on the *validation data set* is introduced. Due to the early stopping of the training, gradients might still be large - contrary to traditional optimization [47]. There is a third and final phase: the test phase with the *testing data set*. All the data sets stem from the same data distribution and the samples are assumed to be independently and identically distributed (i.i.d) [62]. In the validation phase, the network's behavior is checked during training, but no weight optimization is done. This is to determine if the network is overfitting, as stated above, and to gain an intermediate result on non-training data. The same validation data set is therefore seen multiple times by the network. In the test phase, the NN's performance on data that has never been seen before is evaluated after the completion of the training. This can be understood as a check for the generalization ability of the NN to new data [47].

The optimization process contains two main concepts in the gradient calculation: *stochastic gradient descent* (SGD) and *backpropagation* [51]. SGD is a variation of the traditional gradient descent algorithm from (2.6). The explanation of SGD is based on Ruder [60]. SGD usually refers to *mini-batch* gradient descent, an algorithm that is typically chosen in training. Instead of computing the gradient of the cost function w.r.t. $\boldsymbol{\theta}$ for all training data at once and then updating the weights and biases, an update is performed for every mini-batch of n training samples:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{x}^{(i:i+n)}, \boldsymbol{y}^{(i:i+n)}) \,, \tag{2.7}$$

where $\boldsymbol{x}^{(i:i+n)}$ are n samples with their corresponding labels $\boldsymbol{y}^{(i:i+n)}$. A mini-batch is therefore just a pick of n samples from the overall training data set. The number of samples in a mini-batch is commonly referred to as *batch size* [63]. Mini-batch gradient descent brings a few advantages with it: With traditional gradient descent, we need to fit the whole data set into memory which is not possible for larger data sets. In contrast, with SGD, only n training samples of a mini-batch need to be in memory, where n can be chosen accordingly. Additionally, SGD's gradient calculations have high variance, as the true shape of the cost function of the whole training set is only approximated with the limited amount of samples in a mini-batch. The high variance and the resulting fluctuations of the gradient allow SGD to evade non-optimal local minima or saddle points to get to potentially better local minima. However, it might be necessary to decrease the learning rate η to decrease overshooting while converging to the lowest point of a local minimum.

Backpropagation is a simple, computationally inexpensive method to calculate the gradient of the cost function $J(\boldsymbol{\theta})$ w.r.t. all $\boldsymbol{\theta}$ needed for the SGD [47]. Backpropagation works on the concept of a *computational graph*, which is a way of rewriting a mathematical

expressions into multiple intermediate calculations and connecting these to show how intermediate results are passed on [64]. We will stick to the example given in Olah [64] to explain both the computational graph, as well as the backpropagation algorithm, with additional information from other sources cited accordingly. Consider the expression

$$e = (a + b) \cdot (b + 1) \tag{2.8}$$

with intermediate variables $c = a + b$ and $d = b + 1$. This gives every function output its own variable and e can be then calculated as $e = c \cdot d$. The computational graph for the calculation of e can be seen in Fig. 2.3.

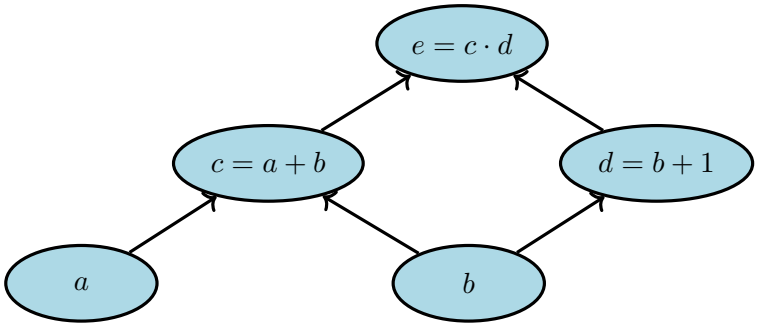

Figure 2.3: Computational graph of the equation $e = c \cdot d$, where $c = a + b$ and $d = b + 1$.

The graph is created by putting each operation and the inputs into nodes. The edges of the graph are represented as arrows, indicating that one node's value is used as an input for another node. The inputs a and b provide initial information that flows upwards towards the final node with the value e. A neural network can be understood as a more complex computational graph than our example. The neural network structure in Fig. 2.2 visualizes this concept, where the equivalent procedure of information flow from input through hidden layers towards the output layer is called forward propagation [47]. The flow of information through the graph is further visualized in Fig. 2.4 by setting the input variables to $a = 2$ and $b = 1$.

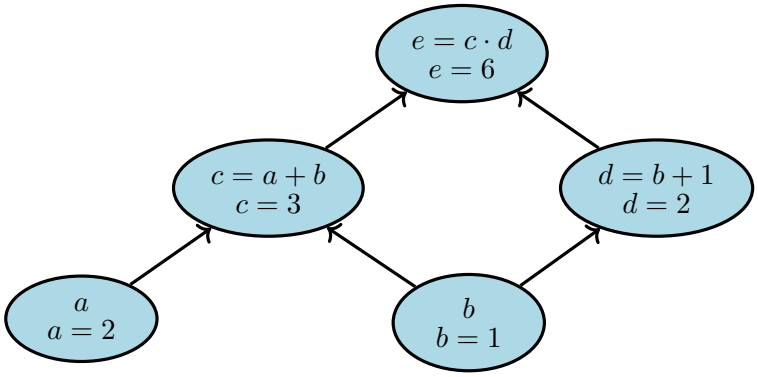

Figure 2.4: Computational graph of the equation $e = c \cdot d$ with exemplary input values.

The key to understanding derivatives in the context of computational graphs is understanding derivatives on the edges. These are the partial derivatives of one node w.r.t. the input node(s), e.g. $\frac{\partial c}{\partial a}$, see Fig. 2.5.

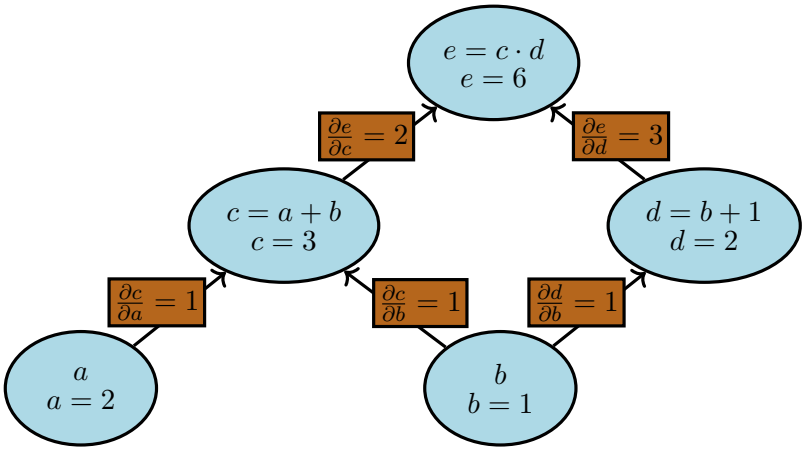

Figure 2.5: Derivatives on edges.

As stated in the beginning of the explanation of the backpropagation method, in the case of NNs, we are interested in the partial derivatives of the cost function $J(\boldsymbol{\theta})$ w.r.t. all $\boldsymbol{\theta}$. Translating our example equation (2.8) and the related computational graph to the case of a NN would mean that the value of the cost function $J(\boldsymbol{\theta})$ for specific $\boldsymbol{\theta}$'s would correspond to e. In this context, calculating all the partial derivatives of e w.r.t. all the other nodes is equivalent to the calculation of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. The general rule of calculating the partial derivatives w.r.t. another node is to sum over all possible paths from one node to the other, multiplying the partial derivatives on each edge of a specific path. This is the equivalent of the multivariate chain rule. For example, for $\frac{\partial e}{\partial b}$ we get

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c}\frac{\partial c}{\partial b} + \frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3\,. \tag{2.9}$$

In practice, with large computational graphs - or in our case large NNs - repeating this chain rule for all partial derivatives of interest is inefficient mainly for two reasons: First, due to repeated calculation of some partial derivatives [47]. In our case an example would be $\frac{\partial e}{\partial d}$ which is needed for both $\frac{\partial e}{\partial a}$ and $\frac{\partial e}{\partial b}$ (see Fig. 2.5). Second, the need to sum up over many different paths to get the derivative w.r.t. a single value. Both issues accumulate over large neural networks. This issue can be overcome with a method called *reverse-mode differentiation* which, in the context of NNs, is backpropagation. In it, the gradient information flows backwards through the network. It is still based upon the chain rule but allows for a layer-wise calculation of the gradients of the output w.r.t. all the previous nodes, making it an efficient method for the calculation of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Fig. 2.6 gives an idea of the process based on our exemplary mathematical expression.

The interested reader is referred to Part II of Goodfellow, et al. [47] for both an outlook on variations from the standard training procedure, with adaptations of the optimization process, and another slightly different introduction to the topic of backpropagation.

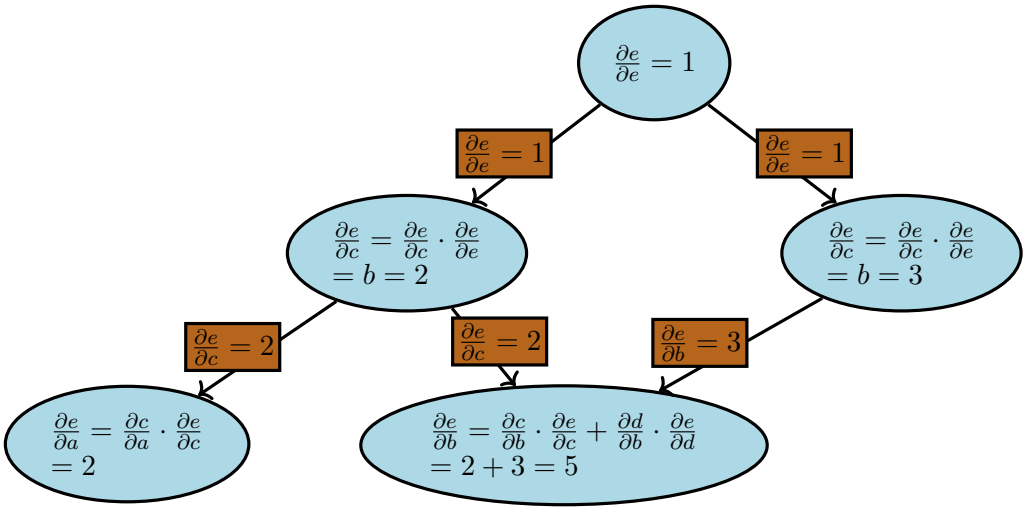

Figure 2.6: Reverse-mode differentiation or backpropagation. The change of arrow direction indicates that the gradient information flows from the output back to all previous nodes.

2.2.2 Modern network architectures

There are many different neural network architectures that deviate from the fully connected neural network as seen in Fig. 2.2. Many of them perform better at certain tasks than fully connected NNs. An example would be *Long Short-Term Memory Recurrent Neural Networks* (LSTM-RNNs) for dynamic classification, where signals from previous timesteps are fed back into the network [65]. They have circular connections between higher- and lower-layer neurons and optionally self-feedback connections, therefore deviating from the layer-wise, static fully connected neural networks. They have been proven to perform well for tasks related to memorizing data for longer time, such as speech and handwriting recognition or machine translation. Another example would be graph neural networks (GNNs) [66]. As the name suggests, they operate on graph data structures, an example being the computational graphs described earlier. Graphs can be used as a denotation of a large number of systems. These appear in many different areas, such as social networks (interaction/connection between participants) or molecule interaction where one might want to predict whether it will bind to a receptor implicated in a disease [67].

In this thesis, we will also deviate from the fully-connected NN architecture. The change consists of two parts. First, instead of using a single neural network to produce output, an adversarial modeling framework containing two NNs that have a special kind of interplay is used, called *Generative Adversarial Nets* (GANs) [68]. Second, instead of using two fully connected neural networks for the NNs in a GAN, *Convolutional Neural Networks* (CNNs) are used [69].

2.2.2.1 Generative adversarial nets

We will be brief in our introduction to the topic of GANs here and will go more in-depth on changes done to the traditional GAN - such as different objective functions - in Chapter 3.2.1. Goodfellow, et al. [68] introduced GANs, which consist of a generative model - the

generator G - designed to replicate the training data distribution, and a discriminative model - the discriminator D - aimed at determining the likelihood of a sample originating from the training data rather than from G. This allows GANs to excel in applications in image/video processing, i.e., image-to-image translation, image manipulation or increasing the resolution of an image/video [70].

The following introduction is adapted from Goodfellow, et al. [68]. The training methodology for G is centered on increasing the odds of D committing errors. To learn a generator distribution p_g over training data $\boldsymbol{x}$, the generator builds a mapping function from a prior noise distribution $p_{\boldsymbol{z}}(\boldsymbol{z})$ (e.g., a vector filled with uniform random values) to data space as $G(\boldsymbol{z};\boldsymbol{\theta}_g)$, where $\boldsymbol{\theta}_g$ are the generator's weights and biases. The discriminator $D(\boldsymbol{x};\boldsymbol{\theta}_d)$ - where $\boldsymbol{\theta}_d$ are the discriminator's weights and biases- outputs a single scalar representing the probability that $\boldsymbol{x}$ came from the training data rather than p_g. The discriminator can be understood as a binary classifier, trying to assign probability 1 to data samples from the training set and 0 to generated data samples. G and D are trained simultaneously, with the goal of adjusting the parameters $\boldsymbol{\theta}_g$ and $\boldsymbol{\theta}_d$ so that G minimizes $\log(1 - D(G(\boldsymbol{z}))$ and D maximizes the probability of correctly detecting if its input is from the training data or an output of the generator. This structure aligns with a two-player, min-max game

$$\min_G \max_D J(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}\,(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]\,, \tag{2.10}$$

with objective function $J(D,G)$. Within the realm of arbitrary functions G and D, a singular solution is attainable, whereby G successfully mimics the distribution of the training data, while D is uniformly 0.5 everywhere. For a more extensive introduction to classical GANs, their advantages and shortcomings, see Goodfellow [71].

In its original formulation, GANs are an unsupervised learning method as they do not work on labeled data. As we are interested in producing pressure wavefields that correspond to certain subsurface velocity distribution, we will make use of an extension of GANs, namely *conditional GANs* (cGANs). These were introduced by Mirza, et al. [72], who conditioned both the generator as well as the discriminator on extra information $\boldsymbol{y}$ (e.g., class labels encoded in a one-hot encoded vector). With this, cGANs are becomeing a supervised learning method. The conditioning works by feeding $\boldsymbol{y}$ into both networks as additional input next to the prior noise distribution $p_{\boldsymbol{z}}(\boldsymbol{z})$. The exact implementation of the input mechanism for $\boldsymbol{y}$ can vary, but the objective function of the two-player min-max game stays the same as in Eq. (2.10), with the only change being that the generator and discriminator now have an additional input $\boldsymbol{y}$:

$$\min_G \max_D J(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}\,(\boldsymbol{x})}[\log D(\boldsymbol{x}|\boldsymbol{y})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{y})))]\,. \tag{2.11}$$

As previously implied, GAN can support various network architectures - not only fully-connected NN (e.g., LSTM-RNNs [73] or CNNs [74]).

2.2.2.2 Convolutional neural networks

In this thesis, CNNs are used for both the generator and discriminator, which is why we will introduce the basic concepts of CNNs here. The introduction to CNNs is based on Goodfellow, et al. [47], if not noted otherwise.

CNNs are a specialized type of neural network engineered for processing data with a grid-like structure. This encompasses data such as time series, which can be seen as a one-dimensional grid with samples taken in regular timesteps, and image data, which can be considered as a two-dimensional grid of pixels. As the name suggests, the prevalent mathematical operation in a CNN is a convolution. In CNNs, convolutions replace general matrix multiplications in at least one of the layers. Before explaining the mathematical underpinnings of convolutions, let us first understand one of the main issues that arises with handling images with fully-connected NNs. Assume a 3-channel (C) image with height (H) and width (W) 10, denoted as $(10, 10, 3)$ with (H,W,C). We want to feed that image to just 5 neurons in a single layer. In a fully-connected NN, each neuron would be connected to each pixel in the image, resulting in 300 weights per neuron, so overall 1500 weights in the specific layer (see Fig. 2.7).

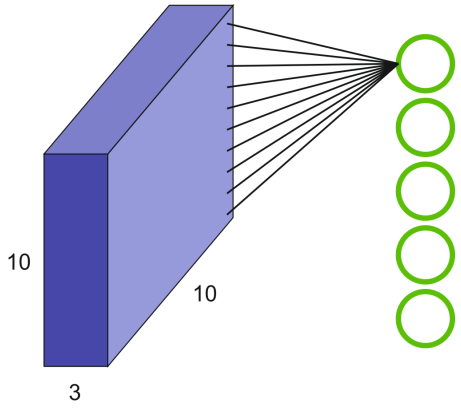

Figure 2.7: Example of how one neuron (green) is connected to one column in one channel of the image (blue), inspired by lecture notes from Nießner [75]. Overall, 1500 connections (black lines) would be present if all pixels were connected to all neurons.

This becomes rapidly impractical when increasing both the image size as well as the amount of neurons in just the one layer, where an image of size $(512, 512, 3)$ feeds into 1000 neurons in a layer - which are reasonable values - resulting in more than 700 million weights just for the one layer. Before explaining how CNNs provide a remedy to this issue, let us consider what convolutions are. In the most general form, convolution is a mathematical operation on two functions with real-valued arguments. In the continuous case it is defined as

$$s(t) = \int x(a)k(t-a)da\,, \tag{2.12}$$

where a practical example would be to think of $k(a)$ as a weighting function for time-dependent measurement data $x(t)$, where a is the age of the measurement. The weighting function $k(a)$ is reflected about the y-axis and shifted before being put into the integral.

The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * k)(t)\,. \tag{2.13}$$

In convolutional network terminology, the initial argument (in the above example x) to the convolution is usually referred to as *input*, and the second argument (in our example k) as *kernel*. The result of the convolution is often called a *feature map*.

When working with computers data is discretized, so that we also require a discrete convolution:

$$s(t) = (x * k)(t) = \sum_{a=-\infty}^{\infty} x(a)k(t-a)\,. \tag{2.14}$$

In deep learning applications, the input is usually a (multidimensional) array of data and the kernel a (multidimensional) array of weights. Therefore, it is fair to assume that functions $x(t)$ and $k(a)$ are zero everywhere except in the finite set of points that are covered by the aforementioned arrays. This means that, in actual implementation, we do not need to calculate infinite sum $\sum_{a=-\infty}^{\infty}$ ..., only requiring a summation over a finite number of array elements. In the case of a two-dimensional image X as input, we might be interested in doing the convolution with a two-dimensional kernel K, resulting in the discrete convolution in two-dimensions:

$$S(i,j) = (X * K)(i,j) = \sum_{m}\sum_{n} X(m,n)K(i-m, j-n)\,. \tag{2.15}$$

Many neural network libraries do not implement convolutions, but rather *cross-correlations*. The operation is exactly the same but the kernel is not flipped before its use. In practical deep learning application, it does not matter if the kernel is flipped or not, as the learning algorithm will learn the appropriate weights at the appropriate place in the kernel. In most cases, the libraries therefore do not differentiate when it comes to naming the two and use the term "convolution" for both cross-correlation and normal convolutions. We will follow this convention of calling both operations convolution.

Fig. 2.8 sketches how a kernel works on a RGB image. In order to calculate the dimensions of the feature map, the following formula was used [76]:

$$\left(\frac{\text{width}(X) - \text{width}(K)}{S_i} + 1, \frac{\text{height}(X) - \text{height}(K)}{S_j} + 1\right)\,, \tag{2.16}$$

where S is the *stride*, indicating how many pixels the filter slides before its impact on the input is calculated again. The subscripts indicate that this calculation needs to be done in both image dimensions. In our example of the RGB image, the depth of the kernel equals the number of channels in the input image, reducing the depth of the feature map to 1.

To calculate the impact of the kernel - the weights - and a specific chunk of the image, the dot product can be used after flattening both three dimensional matrices to a one-dimensional vector. As with fully-connected NNs, there is also a bias term that is constant for the convolution. Therefore, a value s_i in the feature map can be calculated

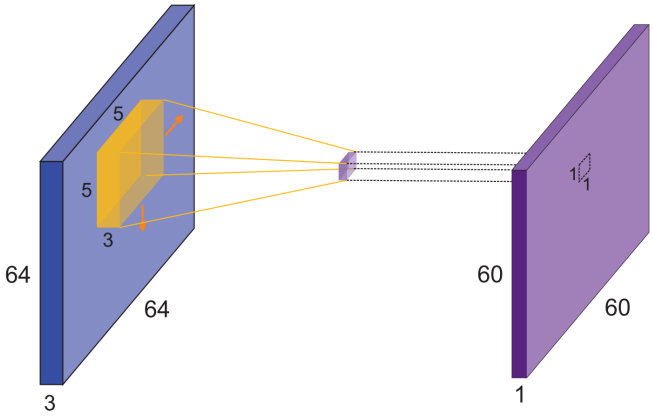

Figure 2.8: Example how a kernel (light orange) turns an image (blue) into feature map (purple), inspired by lecture notes from Nießner [75]. The orange arrows indicate the sliding of the kernel across the image. A stride of 1 was assumed for the calculation of the dimensions of the feature map.

from the flattened kernel $\boldsymbol{k}$ and $\boldsymbol{x}_i$ - the flattened i-th chunk of the input image X - and the bias b [76]:

$$s_i = \boldsymbol{k}^T \boldsymbol{x_i} + b\,. \tag{2.17}$$

Understanding the basics of how a convolution works, let us now consider the advantages of CNNs compared to fully connected NN, namely *sparse interaction*, *parameter sharing* and *equivariant representation* of CNNs. Sparse interaction solves the issue of having an interaction between each input and output unit as sketched in Fig. 2.7 and previously explained. With a convolutional, one output node is only connected to part of the input image. The spatial extent of this connectivity is called receptive field. It can also be tracked through multiple convolutions, see Fig. 2.9. This means that while direct connections are very sparse, deeper layers are indirectly connected to all or most of the input image.

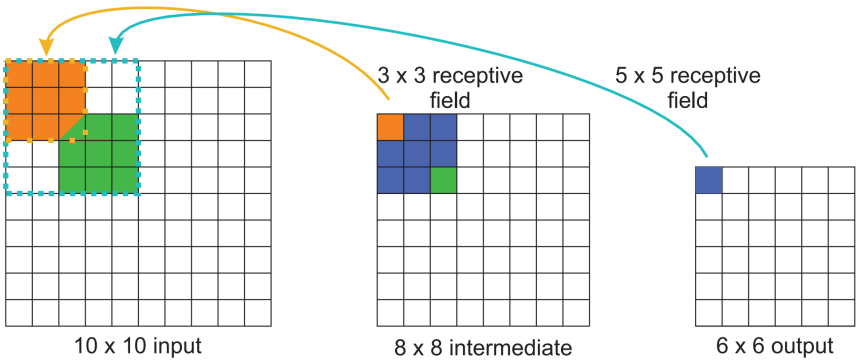

Figure 2.9: Getting the receptive field (dotted squares) of one pixel in the intermediate layer and output with regard to the input, assuming a $(3, 3)$ kernel. The different colors indicate different image-kernel interactions. Inspired by lecture notes from Nießner [75].

Parameter sharing - also referred to as *tied weights* - allows the model to use the same parameter multiple times across different functions, contrary to fully-connected neural networks where each weight is used only once. In a CNN, each member of the kernel - a weight value - is used at every position of the input so that only a single set of parameters is learned instead of a set of for every location. In practice, this does not impact the runtime of forward propagation, but further reduces memory requirements.

The particular form of parameter sharing causes the convolutional to be equivariant to translation. This property means that the output mirrors the way the input changes, specifically when the input is shifted or translated. An example would be shifting an image pixel to the right before applying the convolution, leading to an equivalent outcome as if the convolution was applied first and the shift second. This property is advantageous in time-series data and image processing, as it creates a timeline or 2-D map that reflects when or where particular features appear in the input. This is useful in scenarios like detecting edges across an image where we know that the same parameters - so the same kernel - can achieve this. There are circumstances, however, where parameter sharing might not be desired, such as processing images focused on a person's face, where different features need to be extracted at different locations. There are other kinds of transformations, like rotation of an image, that require other mechanisms so that convolution is able to handle these.

There are two other main components - next to the convolution - that make up a convolutional layer that is used in a CNN. As the convolution is a linear transformation, we need again to pass the results of it through a non-linear activation function. These intermediate values are then fed into a pooling function that can be understood as a feature selector, with the convolution being a feature extractor in this sense. The pooling also makes the representation approximately invariant to small translations of the input. This invariance can be a useful property when we are more concerned with whether a feature exists rather than its exact location. A common example for a pooling function is *max-pooling*, where the maximum value within a neighbourhood - e.g., a square in two dimensions - is picked. This further reduces the output size, improving the computational efficiency of the network.

When it comes to training deep convolutional GANs (DCGANs), the block of convolutional layer, activation and pooling layer is often interlaced with a normalization layer and/or a dropout layer [77]. Usually *batch normalization* is used as a normalization layer, stabilizing learning by transforming each training mini-batch to have zero mean and unit variance [78]. This procedure is still not to be confused with normalizing values to a specific range, e.g., $[0, 1]$, but rather a form of standardization [78]. *Dropout* acts as a regularization in deep NNs to prevent the network from overfitting and was initially developed for fully-connected NNs [79]. With it, a certain amount of random neurons in each layer and their connections are dropped during training. Therefore, multiple different thinned networks are trained and then averaged in validation/testing, where dropout is inactive. The approach can be translated to CNNs, where entire channels of intermediate feature maps are dropped [80], [81].

2.2.3 Physics-informed deep learning

While classical numerical methods based on the discretization of PDEs have shown great progress in simulating multi-physics problems, it is still challenging to seamlessly incorporate noisy data into existing algorithms and to generate the necessary meshes. Additionally, solving inverse problems with only partially known physics - such as missing boundary conditions - is often prohibitively expensive [82]. Moreover, maintaining the used programs with often more than $100,000$ lines of code adds another layer of complexity. With the increase of available multi-fidelity observation data, data-driven methods such as machine learning have shown to be a promising alternative to classical numerical method due to their ability to explore massive design spaces, identify multi-dimensional correlations, and manage ill-posed problems [82]. This is especially true for deep learning approaches [83]. Additionally - depending on the complexity of the problem - a data-driven approach can outperform the classical method in terms of computational speed after training, while retaining a high or equal quality of the solution [84].

Nonetheless, even with the large amount of observational data, labeled data required for supervised machine learning is scarce in relation to the complexity of certain physical, biological or engineering systems due to the cost of deploying and maintain sensors [25], [85]. Therefore, for regimes with little to no data, conclusions need to be drawn from the available partial information [25]. This leads us to the main drawbacks of using purely data-driven models: They may fit quite satisfactorily fit observations, but predictions may be physically inconsistent or implausible, leading to poor generalization performance [82]. The solution to this issue is to add prior knowledge or constraints to the machine learning method, leading us to physics-informed machine learning. Karniadakis, et al. [82] defined *physics-informed machine learning* as "the process by which prior knowledge stemming from our observational, empirical, physical or mathematical understanding of the world can be leveraged to improve the performance of a learning algorithm". A common class of deep learning algorithms that achieves this was introduced by Raissi, et al. [25], namely *physics informed neural networks* (PINNs). These are trained to solve supervised learning tasks while respecting general nonlinear partial differential equations that describe the physical system of interest. PINNs do so by considering both measurement data as well as information from the PDEs in the objective function of a neural network parameterized by θ [82]:

$$\mathcal{J}(\boldsymbol{\theta}) = \alpha_{\text{data}}\mathcal{L}_{\text{data}} + \alpha_{\text{physics}}\mathcal{L}_{\text{physics}}\,, \tag{2.18}$$

with weighting factors α for the losses $\mathcal{L}$ stemming from the purely data-driven, supervised method and from checking the adherence with the physical underpinnings of the system that we want to model.

$$\mathcal{L}_{\text{data}} = \frac{1}{N_u}\sum_{i=1}^{N_u} \left\| u\left(t_i, x_i\right) - \Lambda\left(t_i, x_i, \boldsymbol{\theta}\right)\right\|^2 \tag{2.19}$$

is the supervised training loss, where $u(t, x)$ represents the solution of a space and time-

dependent system and Λ is the prediction of the neural network. $u(t, x)$ can either come from measurement data or from simulation and acts as a ground truth. The *physical consistency-based losses* are described by

$$\mathcal{L}_{\text{physics}} = \frac{1}{N_\Lambda} \sum_{j=1}^{N_\Lambda} \left\| \mathcal{N} \left[\Lambda \left(t_j, x_j; \boldsymbol{\theta} \right); \lambda \right] \right\|^2, \tag{2.20}$$

where $\mathcal{N}$ is an underlying differential operator parameterized by λ that describes the physical system [31]. With this formulation, two groups of problems can be tackled: First is the forward problem, i.e., obtaining the behavior of $u(t, x)$ based on fixed model parameters λ. The second is the inverse problem, where we are interested in finding parameters λ that best describe observed data [25]. In many cases, adding the physics-based loss also accelerates training in addition to favoring physically consistent solutions and better generalization. Additionally, it is even possible to train the network without any measurement data, by relying solely on physical consistency-based losses [82].

In order to differentiate from the fully-connected neural networks on which the introduction of PINNs by Raissi, et al. [25] is based, we will refer to our approach as *physics-based deep learning* (PIDL). This is to indicate that the underlying network architecture used in the thesis is a GAN. The exact implementation of the physical consistency-based losses is detailed in Chapter 3.2.2.

Equipped with this knowledge, we will introduce the methods used to develop a NN that can generate pressure wavefields in Chapter 3.

3. Methodology

We employ a supervised NN to generate varied pressure wavefields depending on the underlying velocity distribution. Therefore, we first have to get a labeled data set or generate a new one for training, validation and testing purpose. The procedure is detailed in Chapter 3.1. Afterwards, we will explain the exact network architecture, objective function and training procedure in Chapter 3.2.

3.1 Wavefield data generation

For our study, there was no existing data set fulfilling our requirements of multiple spatial P-wave velocity distributions with constant domain size and the corresponding pressure wave fields in close time succession generated from the same source at the same position. With P-waves being the only wave type present in an acoustic medium, we refer to P-wave velocity distributions simply as velocity distribution. SpecFem2D [86], a solver for - among other things - 2D simulations of acoustic seismic wave propagation, as well as FWI, was used to simulate the wave propagation. It is based on a *spectral element method* (SEM), which is a variant of traditional *finite element methods* (FEM) that use higher order basis functions. We will not dive into the numerics of it, as we are using SpecFem2D purely as a tool to generate wave fields for later use in the deep learning algorithm. The interested reader is therefore referred to Hafeez, et al. [87] for an overview on SEMs.

3.1.1 Simulation domain and parameters

We solve the acoustic wave equation (2.3) on a square domain of interest with width x and height z of 1000 m with the goal of simulating the pressure wavefield. This wavefield is a response to an initial acoustic pressure source of Gaussian shape with a dominant frequency of 10 Hz in the middle of the domain at (x, z)-position $(500, 500)$, with $z = 0$ being at the bottom of the domain. *Perfectly matched layers* (PMLs) [88], [89] are employed at the left, right and bottom boundary to get absorbing boundary conditions. These are used to minimize - in the best case eliminate - reflections from these boundaries in order to truncate a larger domain to the domain of interest [90]. A *free surface* boundary is used at the top, mimicking the behaviour at the interface between a half-space in contact with vacuum [91]. The domain of interest is split into 40 square spectral elements of size 25 m in both directions respectively, resulting in a total of 1600 elements. The absorbing boundaries are 3-spectral-element-thick. SpecFem2D automatically checks if the shape and amount of the elements allows for an accurate simulation. We passed both checks; SpecFem2D even suggested using less elements to speed up the computation. Nevertheless, we adhered to the amount of elements to have a high resolution image that would not introduce too many artifacts when interpolating to a different grid size (see Chapter 3.1.2). A fourth-order 6-stage low storage Runge-Kutta time-stepping scheme

with a time step size of 0.001 s is chosen to simulate 1 s. Fig. 3.1 shows an overview of the domain .

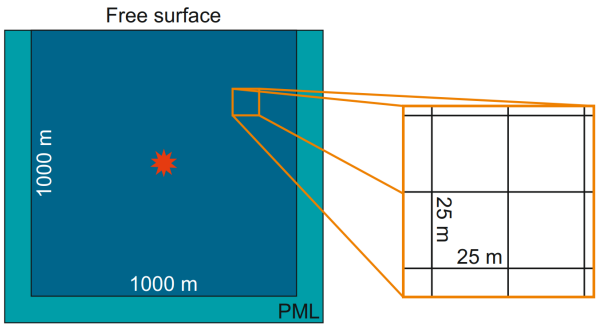

Figure 3.1: Sketch of the simulation domain.

Simulation parameters are passed to SpecFem2D via the *source*, *interface*, and *parameter file*. The source file contains information about the source location, type, and dominant frequency and is constant throughout all simulations. The interface file requires the amount of interfaces, the interface vertices and the amount of spectral elements in vertical direction. The parameter file is the main input file, where the parameters of interest for us can be split into the constant ones and the ones that are adjusted for different simulations. The constant parameters are the time step size, the simulation period, the amount of spectral elements for the PML boundaries, amount of spectral elements in x direction, and the output step-size of 0.01 s for the wavefield data. For the adjustable category, we first have the amount of material layers, where a layer is the space between two interfaces from the interface file. Second are the associated material velocities and corresponding densities. Finally, we have to define which spectral elements belong to which layer.

In order to generate a large data set covering multiple layers and multiple different velocity distributions, we automate the process of changing all the parameters using bash scripts to change the interface and parameter file. We generate two distinct data sets. The first being a simple one, consisting of 15 different uniform velocity distributions. This data set will be used to compare the performance between a physics-informed NN and a non-physics-informed NN. The second data set is more complex, though we still only consider horizontal layering and three types of velocity layering - uniform, tow- and three-layer. Layer thicknesses were calculated randomly in a loop, with a minimum layer thickness of 150 m, and the upper limit depending on the amount of layers that still need to be put in the domain. The thickness of the upmost layer - the one at the surface - was chosen to fill the domain. The amount of spectral elements per layer is proportionally distributed. A similar procedure is used to distribute the density and velocity values to the layers. We chose a minimum density value of 800 $\mathrm{kg/m^3}$ and a maximum of 2500 $\mathrm{kg/m^3}$ with corresponding minimum velocity of 1200 m/s and maximum of 2700 m/s. A minimum difference of 180 $\mathrm{kg/m^3}$ between density values was chosen. We use a minimum layer thickness as well as a minimum step size between density values (from which the velocity values result), to allow for a visual inspection of the results. If

layers are too thin, we would not notice them and if the difference in velocities is too small, almost indiscernible reflections would occur.

When modifying the velocity values in the parameter file, it is important to also change the corresponding density values. Even though we are only interested in velocities as an input to our NN, SpecFem2D will produce artifact-heavy wavefields if the densities are held constant for velocities with larger differences. To solve that problem, we employ a linear interpolation between pairs of (density, velocity) values for which the simulation produced artifact-free wavefields to calculate velocity values from certain velocities. This fixed the issue, but it is important to note that the density values were not chose to represent a specific material. The generated density values and accompanying velocity values are randomly given to a certain layer in the parameter file.

3.1.2 Data processing

There are four main data processing steps that need to be done before feeding data to the NN. First, a velocity distribution image needs to be generated. We were unable to retrieve the underlying velocity distribution from SpecFem2D directly, which is why we wrote a Python program that reads all the necessary values from the interface and parameter file and creates a velocity distribution image of size $(128, 128)$. Second, the pressure wavefields outputted by SpecFem2D contain PMLs that we do not need as part of the input to the NN. Moreover, we require the image size to be $(128, 128)$, whereas SpecFem2Ds output size - including the PMLs - is $(173, 185)$ in z- and x-direction, respectively. Therefore, the pressure wavefields are interpolated to an equidistant 2D grid that excludes the PMLs and has a size of $(128, 128)$. Third, early simulation time steps before 0.18 s were removed from the data set to remove wavefields that are mainly dominated by source physics. Close to the source in space and time deformations might not be elastic and/or not small [11]. Therefore, we empirically remove any time steps for which the simplifications of the acoustic wave equation (2.3) that we use in our physics consistency-based loss are not valid. Afterwards, time steps in an interval of 0.7 s are chosen as time steps of interest on which the NN is trained on. Additionally, we save four surrounding time steps for each time step of interest, e.g. [0.23, 0.24, 0.26, 0.27] for 0.25. We require these to calculate the partial derivative of the wavefield w.r.t. the time step of interest for the physics consistency-based loss. Fourth, all input and output variables of the network - in our case velocity and the pressure values - are normalized to the interval $[0, 1] \in \mathbb{R}$, as this is the range of values on which the NN can perform.

After processing, the accumulated training, validation, and testing data set contains 12 time steps of interest with related pressure wavefields for each velocity distribution. For the more complex data set with horizontal layering, we have 200 examples of uniform velocity distributions, 300 for the two-layer case, and 400 different three-layer velocity distributions. We chose an uneven split to give more examples of the more intricate velocity distributions to the NN. Overall, the more complex NN data set contains $(200 + 300 + 400) \cdot 12 = 10,800$ wavefields, compared to the $15 \cdot 12 = 300$ wavefields for the data set with only uniform velocity distributions. Fig. 3.2 shows examples for the normalized velocity distributions

and corresponding pressure wavefields. In addition to the above data sets that play a direct role in the learning phase, 48 overall wavefields surrounding all the time steps of interest are needed per velocity distribution for the calculation of the physics consistency-based loss (see Chapter 3.2.2).

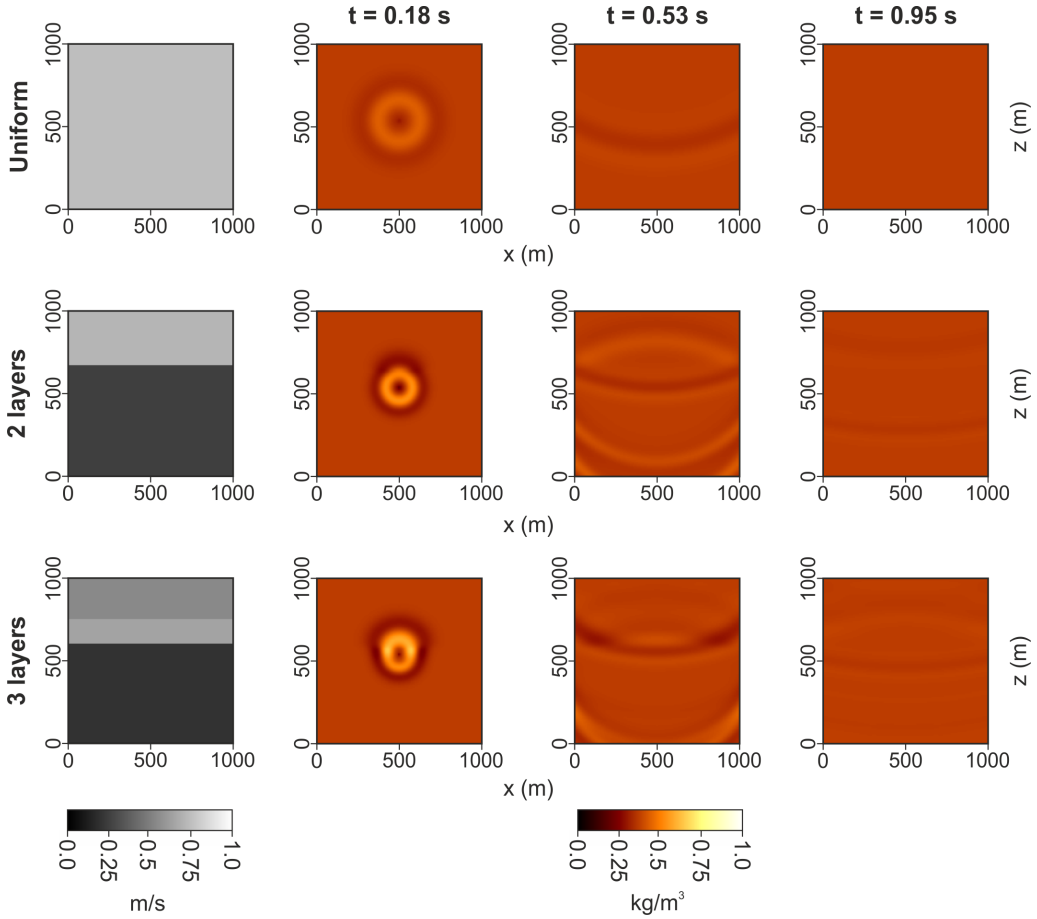

Figure 3.2: Processed and normalized examples of velocity distributions (first column) and corresponding pressure wavefields at exemplary time steps.

3.2 Network building and training

The deep learning approach is taken from Kadeethum, et al. [1] and adapted only minimally. In their work, Kadeethum, et al. [1] use a *continuous conditional generative adversarial network* (CcGAN) to solve the time-dependent PDE of the transient response of the coupled poroelastic process on a 2D domain, where heterogeneous permeability fields are used as input and pressure or displacement fields over time are outputted. Their approach generalizes well to other permeability fields and allows for a continuous time-stepping without the necessity to compute output of preceding time steps. Given our goal of inputting a velocity distribution and a specific time step and obtaining a pressure wavefield as output, their approach served as a logical foundation for our work. It is important to note, that Kadeethum, et al. [1] do not use any physical consistency-based losses.

CcGAN combines the image-to-image translation cGAN from Isola, et al. [74] with Ding, et al. [92]'s extension to cGANs with continuous variables as the condition. An earlier work from Kadeethum, et al. [93] on solving the steady-state solution of the same coupled poroelastic process provides the reasoning for many of the decisions for the network architecture and training process. The code to build and train the NNs has been published under the "CC0 1.0 Universal" license at `https://codeocean.com/capsule/2052868/tree/v1` and was used as a basis for our program. It is built upon PyTorch, an open-source deep learning framework compatible with Python [94].

3.2.1 Continuous conditional generative adversarial network

We split the introduction to the CcGAN into two parts. First, the exact architecture of the network is detailed, e.g., the different layers. Moreover, the scheme to input the time information is presented. Second, changes to the classical GAN's objective function are explained in Chapter 3.2.1.2.

3.2.1.1 Architecture

The CcGAN consists of a generator built from a *U-Net* and a *patch-based critic*. Fig. 3.3 shows a sketch of the architecture as well as the inputs and outputs of the generator and critic.

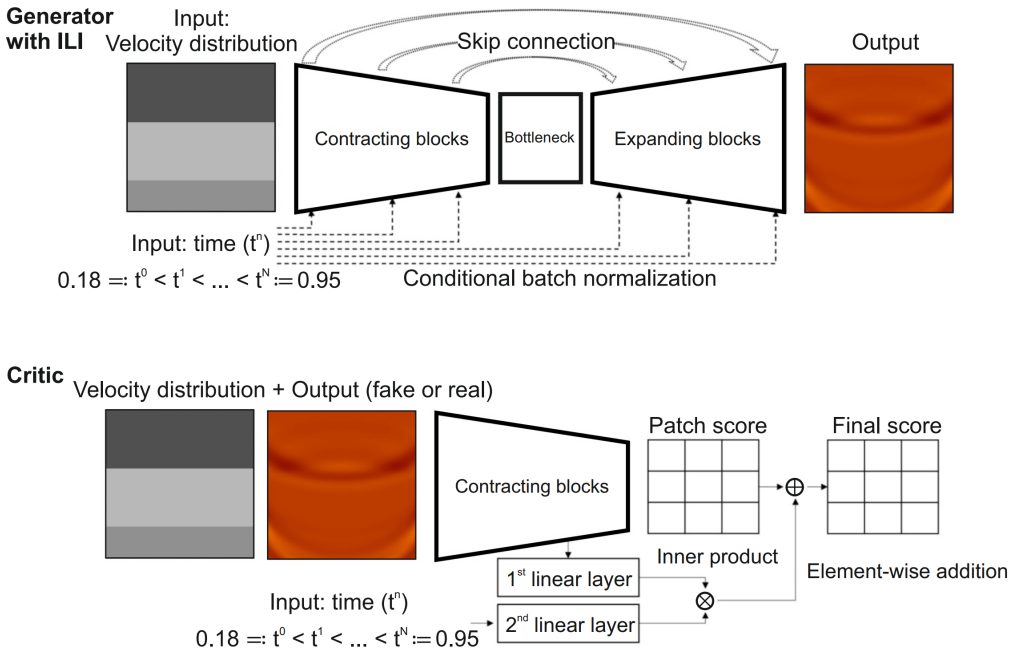

Figure 3.3: Sketch of CcGAN architecture and time-inputting mechanism, modified after Kadeethum, et al. [1].

The input to the U-net generator is the one-channel normalized velocity distribution and the output is a generated pressure wavefield with values in the interval $[0, 1] \in \mathbb{R}$. The U-Net architecture was first introduced by Ronneberger, et al. [95] for image segmentation tasks. It consists of an encoder/contracting path that takes the velocity distribution

as the input and a decoder/expanding path that reconstructs the wavefield. Both are built upon blocks of convolutional layers. Additionally, cropped feature maps from the contracting path are fed and concatenated to a corresponding intermediate feature map in the block of convolutional layers in the expanding path. The cropping is required as border pixels can get lost when doing convolutions [95]. These concatenations are also referred to as *skip-connections* and can help the recovery of spatial information - often the location of certain features in the image - in the decoder path. This information is lost in the pooling operation, as mentioned in Chapter 2.2.2 [96].

A critic is a special kind of discriminator, for which we will explain the differences in Chapter 3.2.1.2. For now, it is enough to think of it as a traditional discriminator that takes the velocity distribution, as well as either the output of the generator or a real pressure wavefield as an input. The output score is a matrix, not a single value as would be the case for non-patch based critic [74]. This patch-based approach means that patches that cover only part of the image are evaluated and given a score, instead of evaluating a whole image at once. Each value in the critic's output matrix has a receptive field that corresponds to a patch in the input image to the critic. This helps with modeling high-frequency structures, as the attention of the critic is focused on local image patches. The matrix of scores outputted by the critic is averaged at the end. The critic consists of only contracting blocks that are very similar to the decoder part of the generator.

Modifications are needed to the classical U-Net to input the necessary time information. Kadeethum, et al. [1] treat time as a continuous variable and show that the *improved label input* (ILI) from Ding, et al. [92] performs very well in their research. However, their way of inputting the time information differs from Ding, et al. [92] and relies on *conditional batch normalization* (CBN) from Vries, et al. [97] to input the temporal term to all layers inside the generator. Instead of using embedding layers as in Ding, et al. [92] and Vries, et al. [97], a fully-connected NN is used in conjunction with a batch normalization layer. The weights of the fully-connected NN are also learned during the training. To pass the time information to the critic, its patch score is element-wise added to an inner product resulting from the time and the output of the contracting blocks being passed through linear layers.

A detailed look at the generator and critic is given in Fig. A.1, with exemplary image dimensions to understand the impact of different operation. The encoder part of the generator starts with a convolutional layer with a 1×1 kernel, stride 1, no padding and no activation function to map the input channels to a larger hidden layer size. The output of this convolutional layer is passed to the first skip-connection. This first convolutional layer is followed by six contracting blocks consisting of two blocks of each a convolutional layer with kernel size 3×3, stride 1 and a padding of 1; the CBN; a dropout layer with probability 0.5 of dropping a channel; and Leaky Rectified Linear Unit (LeakyReLU, see Fig. 3.4) with a slope of 0.2 in the negative domain as the activation function, defined as

$$f_{\text{LeakyReLU}}(x) = \begin{cases} x & \text{if } x > 0\,, \\ 0.2x & \text{otherwise}\,. \end{cases} \tag{3.1}$$

This combination is followed by a max-pooling operation with a 2×2 kernel and a stride of 2 that halves the image height and width. Except for the sixth block, the output of the max-pooling is also where the other skip-connections begin. The decoder consists of six expanding blocks. Each of them starts with a 2D bilinear upsampling to double the image height and width. This is followed by a first convolutional layer with a 2×2 kernel, stride 1 and no padding. The cropped image from the skip-connection is concatenated to the resulting feature map. The input image to the skip-connection needs to be cropped, as the output of the first convolutional layer is 1 smaller in both width and height than the image on the encoder side. The concatenated image is then passed to a 3×3 convolutional layer with stride 1 and a padding of 1; a CBN; a dropout layer with probability 0.5; and a normal Rectified Linear Unit (ReLU, see Fig. 3.4) activation function

$$f_{\text{ReLU}}(x) = \begin{cases} x & \text{if } x > 0\,, \\ 0 & \text{otherwise}\,. \end{cases} \tag{3.2}$$

This process is repeated once more, exchanging the 3×3 convolutional layer with a convolutional layer with kernel size 2×2, stride 1 and a padding of 1 which increases the image size by 1 compared to the convolutional layers before. After the sixth expanding block, a convolutional layer with a 1×1 kernel, stride 1, and no padding is used to reduce the amount of channels to 1. The output of that last convolution is passed through a sigmoid activation function (see Fig. 3.4) to map the output to the $[0, 1] \in \mathbb{R}$ interval, defined as

$$f_{\text{sigmoid}}(x) = \frac{1}{1 + e^{-x}}\,. \tag{3.3}$$

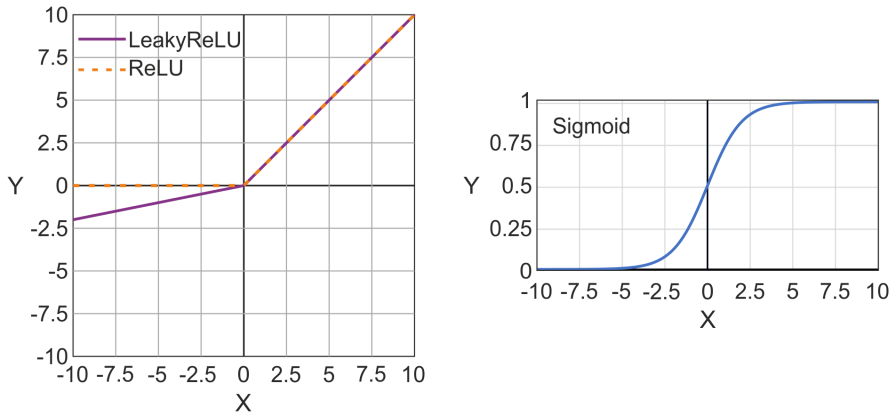

Figure 3.4: Activation functions used in the CcGAN.

To understand the impact of the operations on the size of the input, feature maps, and output, see Table 3.1. The amount of channels for the feature maps are only an example; we will use multiple different ones later. The table also highlights that dropout was only used in the first three contracting blocks.

	Input size [𝔹, C, H, W]	Output size [𝔹, C, H, W]	CBN	Dropout
1st convolutional layer	[𝔹, 1, 128, 128]	[𝔹, 32, 128, 128]		
1st contracting block	[𝔹, 32, 128, 128]	[𝔹, 64, 64, 64]	✓	✓
2nd contracting block	[𝔹, 64, 64, 64]	[𝔹, 128, 32, 32]	✓	✓
3rd contracting block	[𝔹, 128, 32, 32]	[𝔹, 256, 16, 16]	✓	✓
4th contracting block	[𝔹, 256, 16, 16]	[𝔹, 512, 8, 8]	✓	
5th contracting block	[𝔹, 512, 8, 8]	[𝔹, 1024, 4, 4]	✓	
6th contracting block	[𝔹, 1024, 4, 4]	[𝔹, 2048, 2, 2]	✓	
1st expanding block	[𝔹, 2048, 2, 2]	[𝔹, 1024, 4, 4]	✓	
2nd expanding block	[𝔹, 1024, 4, 4]	[𝔹, 512, 8, 8]	✓	
3rd expanding block	[𝔹, 512, 8, 8]	[𝔹, 256, 16, 16]	✓	
4th expanding block	[𝔹, 256, 16, 16]	[𝔹, 128, 32, 32]	✓	
5th expanding block	[𝔹, 128, 32, 32]	[𝔹, 64, 64, 64]	✓	
6th expanding block	[𝔹, 64, 64, 64]	[𝔹, 32, 128, 128]	✓	
2nd convolutional layer	[𝔹, 32, 128, 128]	[𝔹, 1, 128, 128]		

Table 3.1: Generator's input and output sizes for each block, where 𝔹 is the batch size

The contracting path in the critic is constructed very similarly to the encoder part in the generator. It also starts with a convolutional layer with 1×1 kernel, stride 1, and no padding to increase the channel size. Four contracting block follow that are the same as in the generator with the difference being that the aforementioned layer normalization is used. After four contracting block, another convolutional layer with 1×1 kernel, stride 1 and no padding is used to reduce the channel size back to 1. There is no need for an activation function when using a WGAN-gp. In contrast to Kadeethum, et al. [1], we use dropout layers in the critic, as this has shown better results. To understand the impact of the operations on the size of the input, feature maps, and output of the critic, see Table 3.2. Here, the amount of channels in the feature map is correct for all later experiments.

	Input size [𝔹, C, H, W]	Output size [𝔹, C, H, W]	Layer normalization	Dropout
1st convolutional layer	[𝔹, 2, 128, 128]	[𝔹, 8, 128, 128]		
1st contracting block	[𝔹, 8, 128, 128]	[𝔹, 16, 64, 64]	✓	✓
2nd contracting block	[𝔹, 16, 64, 64]	[𝔹, 32, 32, 32]	✓	✓
3rd contracting block	[𝔹, 32, 32, 32]	[𝔹, 64, 16, 16]	✓	✓
4th contracting block	[𝔹, 64, 16, 16]	[𝔹, 128, 8, 8]	✓	✓
2nd convolutional layer	[𝔹, 128, 8, 8]	[𝔹, 1, 8, 8]		

Table 3.2: Patch-based critic's input and output sizes for each block, where 𝔹 is the batch size.

3.2.1.2 Objective function

We exchanged the traditional GAN from Chapter 2.2.2 with a *Wasserstein GAN with gradient penalty* (WGAN-gp). These were introduced by Gulrajani, et al. [98] as an improved version of *Wasserstein GANs* (WGANs). WGANs were proposed by Arjovsky,

et al. [99] to stabilize the training of GANs/cGANs. The difference between a WGAN and a normal GAN is in the objective functions. This difference in objective function also means a different purpose of the discriminator in a classical GAN/cGAN and of the critic from a WGAN, explaining the change in naming.

Arjovsky, et al. [99] found that an optimal discriminator in a traditional GAN will provide adequate information upon which the generator can improve. In many cases, though, the generator is not yet good enough to produce images that are close enough to the ground truth distribution. In these cases, the gradient for the generator diminishes and it will not learn anything. From the original GAN cost equation (2.10), this means:

$$-\nabla_{\boldsymbol{\theta}_G} \left[\log(1 - D(G(\boldsymbol{z})))\right] \to 0\,. \tag{3.4}$$

Even with an alternative cost function for the generator

$$\left[\log(D(G(\boldsymbol{z})))\right]\,, \tag{3.5}$$

from the original paper by Goodfellow, et al. [68], which was proposed to solve the issue of vanishing gradients, training remains unstable due to large variances of gradients [99]. Instead, Arjovsky, et al. [99] propose using the *Wasserstein-1* W_1 distance, which is also commonly refereed to as *Kantorovich–Rubinstein* distance [100]. It is defined as

$$W_1\left(\mathbb{P}_r, \mathbb{P}_g\right) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]\,, \tag{3.6}$$

with real data distribution $\mathbb{P}_r$ and generated data distribution $\mathbb{P}_g$. $\Pi\left(\mathbb{P}_r, \mathbb{P}_g\right)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively $\mathbb{P}_r$ and $\mathbb{P}_g$. $\gamma(x, y)$ describes how much "mass" needs to be transported from x to y to transform $\mathbb{P}_r$ into $\mathbb{P}_g$. Π contains all the different transport plans. The W_1 distance is the cost - mass times transport distance - of the optimal transport plan, meaning the minimum work that needs to be done to transform one data distribution into the other. The advantage of this approach is that the W_1 distance is continuous and almost differentiable everywhere (see Fig. 3.5) [99].

However, equation (3.6) is highly intractable due to the infimum [99]. A work-around is using the duality formula for the Kantorovich–Rubinstein distance

$$W_1\left(\mathbb{P}_r, \mathbb{P}_g\right) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim\mathbb{P}_r}[f(x)] - \mathbb{E}_{x\sim\mathbb{P}_g}[f(x)]\,, \tag{3.7}$$

where the supremum is over all the 1-Lipschitz functions $f : X \to \mathbb{R}$ [100]. Replacing $\|f\|_L \leq 1$ for $\|f\|_L \leq K$ (K-Lipschitz for some constant K), we end up with a distance $K \cdot W_1\left(\mathbb{P}_r, \mathbb{P}_g\right)$ [99].

With parameterized family of functions $\{f_w\}$, $w \in \mathcal{W}$ that are K-Lipschitz for some K, we can consider solving the WGAN objective function

$$\max_{w\in\mathcal{W}} \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r} [f_w(\boldsymbol{x})] - \mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g} [f_w(\tilde{\boldsymbol{x}})]\,, \tag{3.8}$$

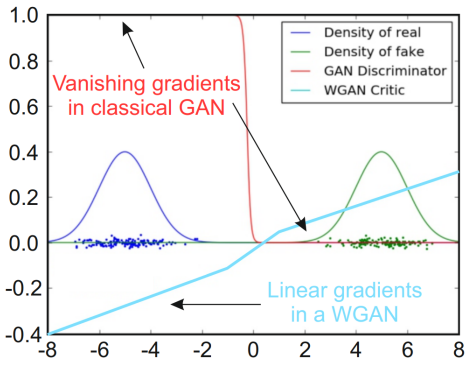

Figure 3.5: Optimal discriminator and critic when learning to differentiate two Gaussians. The discriminator of a classical min-max GAN saturates and results in vanishing gradients. In contrast, WGAN critic provides linear gradients on all parts of the space. Modified after Arjovsky, et al. [99].

where $\tilde{\boldsymbol{x}}$ is the output of the generator $G_{\boldsymbol{\theta}}$. f_w is approximated by a NN - the critic - parameterized with weights w lying in a compact space $\mathcal{W}$ [99]. In contrast to the discriminator in a classical GAN, which works as a classifier, using the W_1 distance gives a meaningful loss metric. When the critic is adequately trained, the generator loss is an estimate of the W_1 distance up to a constant factor K. Therefore, a reduction in generator loss correlates to the quality of the generated samples, which is not the case with classical GANs [99]. To emphasize this difference, the counterpart of the generator in a WGAN is named critic [99].

A challenge that comes with using the W_1 distance is enforcing the Lipschitz constraint by having the weights w of the critic laying in a compact space W. In the original WGAN paper, Arjovsky, et al. [99] clipped the weights of the critic to be in a compact space $[-c, c]$. They already noted that this is not an optimal way of enforcing the Lipschitz constraint, as it can severely limit the capability of the WGAN to model complex functions. Gulrajani, et al. [98] solve that issue by introducing a *gradient penalty* term. They note that a differentiable function is 1-Lipschtiz if and only if it has gradients with norm at most 1 everywhere. Therefore, they constrain the gradient norm of the critic's output w.r.t. its input. They do so using a soft version of the constraint by employing a penalty on the gradient norm for random samples $\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}$. The new objective function of the WGAN-gp is then

$$\mathcal{J} = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}} \sim \mathbb{P}_g} [f_w(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \mathbb{P}_r} [f_w(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda_{\text{gp}} \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}} \left[(\|\nabla_{\hat{\boldsymbol{x}}} f_w(\hat{\boldsymbol{x}})\|_2 - 1)^2 \right]}_{\text{Gradient penalty}}, \tag{3.9}$$

where λ_{gp} is a weighing parameter. $\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}$ is sampled uniformly along straight lines between pairs of points $\boldsymbol{x}$ sampled from real data distribution $\mathbb{P}_r$ and $\tilde{\boldsymbol{x}}$ from generated data distribution $\mathbb{P}_g$:

$$\hat{\boldsymbol{x}} = \epsilon \tilde{\boldsymbol{x}} + (1 - \epsilon) \boldsymbol{x}\,, \tag{3.10}$$

where ϵ is a random number in $[0, 1] \in \mathbb{R}$.

It is important to note that using the gradient penalty does not allow for the usage of batch normalization in the critic, as the norm of the critic's gradient w.r.t. each input is independently penalized. Batch normalization introduced a correlation between samples as it uses batch-level metrics - such as the average batch value - to normalize all samples in a batch (see end of Chapter 2.2.2). This would invalidate the training objective [99]. Looking at their code, Kadeethum, et al. [93] - and in conclusion Kadeethum, et al. [1] - seemingly did not take that into consideration and still used batch normalization layers in their critic, nevertheless producing high-quality results. Nonetheless, we deviated from Kadeethum, et al. [1]'s approach by following the recommendation of Gulrajani, et al. [98] to instead use a layer normalization procedure. With layer normalization, the normalization is done along the feature dimension instead of across the batch dimension, so that no correlation between samples in a batch is introduced [101]. In our case, the feature dimension are the channels of the intermediate feature maps.

The objective function of the WGAN-gp is further enhanced by an ℓ_1 loss to help with low-frequency features [74]. To summarize, the critic's cost function for one image is calculated as

$$\mathcal{J}_{\text{critic}} = [f_w(\tilde{\boldsymbol{x}}) - f_w(\boldsymbol{x})] + \lambda_{\text{gp}} \left[\|\nabla_{\hat{\boldsymbol{x}}} f_w(\hat{\boldsymbol{x}})\|_2 - 1\right]^2 . \tag{3.11}$$

The generator's cost function is calculated as

$$\mathcal{J}_{\text{G}} = f_w(\tilde{\boldsymbol{x}}) + \lambda_{\ell_1} \|\tilde{\boldsymbol{x}} - \boldsymbol{x}\|_1 , \tag{3.12}$$

where λ_{ℓ_1} is a *hyperparameter* that is searched for in the hyperparameter-tuning phase (see Chapter 3.2.3). Hyperparameters control the learning algorithm and are set by the user. That is in contrast to the internal parameters of the network such as the weights and biases [47].

3.2.2 Physical consistency-based losses

We follow Rasht-Behesht, et al. [32]'s reasoning for the physical consistency-based losses. They use a PINN to solve the wave equation on a domain with a free surface on top and absorbing boundaries everywhere else. We add the physical consistency-based losses to the cost function of the generator similar to Yang, et al. [102], as we want the generator to produce physics-consistent wavefields.

In the introduction to physical consistency-based losses in Chapter 2.2.3, we put all the physics-related losses into equation (2.20). For our approach, we can split these into two parts: Checking the output of the generator for adherence to the boundary conditions and to the acoustic wave PDE (2.3). Rasht-Behesht, et al. [32] set the source term s in the wave equation to $s \equiv 0$ and instead enforce external forces through a perturbation of the initial field acting at some time. This is without loss of generality, but helps with calculating the adherence to the PDE, as we can simply check if

$$\frac{1}{c^2} \frac{\partial^2 P_G}{\partial t^2} - \nabla^2 P_G = 0 , \tag{3.13}$$

for each point on the 128×128 grids for P_G generated by the generator and c from the underlying velocity distribution. Additionally, we denormalize the pressure and velocity values. Rasht-Behesht, et al. [32] argued that PINNs are able to enforce absorbing boundary conditions by default without the need to prescribe them in the training. Following their approach, the only boundary condition that we check for is at the free surface, where pressure is fixed to zero

$$P_G(x, t, z = 1000\,\mathrm{m}) = 0\,. \tag{3.14}$$

Rasht-Behesht, et al. [32] do the adherence check with regards to the PDE by using the automatic differentiation capabilities of the neural network libraries to get the required partial derivatives in the wave equation. We cannot use that method for two reasons. First, in order to solve

$$\nabla^2 P_G = \frac{\partial^2}{\partial x^2} P_G + \frac{\partial^2}{\partial z^2} P_G\,, \tag{3.15}$$

we need the second partial derivatives of P_G w.r.t. x, z, and t. As we do not input specific x and z values to the NN - but rather use an image - we cannot use the automatic differentiation as we do not have x and z on the underlying computational graph needed for the differentiation. This differentiation process is very similar to the backpropagation in Chapter 2.2.1 [103]. Second, when trying to solve $\frac{\partial^2 P_G}{\partial t^2}$ using automatic differentiation a different problem occurs. The differentiation works, but because we input t multiple times to the generator using the CBN, multiple values for $\frac{\partial^2 P_G}{\partial t^2}$ are returned, which we cannot interpret. To address these issue, we use a fourth-order accurate central finite difference approximation for the second derivatives in space and time with uniform grid spacing [104]. Calculating $\frac{\partial^2}{\partial x^2} P_G$ and $\frac{\partial^2}{\partial z^2} P_G$ for all pixels in the image except for the two rows and columns close to the boundaries can be done pixel-wise using

$$\frac{\partial^2}{\partial x^2} P_{G_{i,j}} \approx \frac{-\frac{1}{12} P_{G_{i-2,j}} + \frac{4}{3} P_{G_{i-1,j}} - \frac{5}{2} P_{G_{i,j}} + \frac{4}{3} P_{G_{i+1,j}} - \frac{1}{12} P_{G_{i+2,j}}}{dx^2} + O\left(dx^4\right) \tag{3.16}$$

and

$$\frac{\partial^2}{\partial z^2} P_{G_{i,j}} \approx \frac{-\frac{1}{12} P_{G_{i,j-2}} + \frac{4}{3} P_{G_{i,j-1}} - \frac{5}{2} P_{G_{i,j}} + \frac{4}{3} P_{G_{i,j+1}} - \frac{1}{12} P_{G_{i,j+2}}}{dz^2} + O\left(dz^4\right)\,, \tag{3.17}$$

where $P_{G_{i,j}}$ indicate the pressure value at a certain pixel (i, j), and $P_{G_{i-1,j}}$ the one to left and $P_{G_{i,j+1}}$ the one under it. The other subscripts follow accordingly. dx and dz represent the uniform spacing between two adjacent pixels, calculated from the overall domain size and the amount of pixels to be $\approx 7.87\,\mathrm{m}$. We cannot compute the second derivatives for the pixels that are in the two rows or columns close to the boundary, as we would need values outside of the domain. Therefore, we get two images of size 126×126 that represent $\frac{\partial^2}{\partial x^2} P_G$ and $\frac{\partial^2}{\partial z^2} P_G$, respectively. The same central finite difference approximation is used to calculate the second derivative of P_G w.r.t. t. However, there is an added difficulty, as this requires full wavefields from two previous and two later wavefields. We circumvent

that issue by using the additional wavefields that we saved during the data processing (see Chapter 3.1). We therefore surround the generated wavefield by the real wavefields from that dataset to calculate

$$\frac{\partial^2}{\partial t^2} P_{g_t} \approx \frac{-\frac{1}{12} P_{g_{t-2}} + \frac{4}{3} P_{g_{t-1}} - \frac{5}{2} P_{g_t} + \frac{4}{3} P_{g_{t+1}} - \frac{1}{12} P_{g_{t+2}}}{dt^2} + O\left(dt^4\right) , \tag{3.18}$$

with P_{g_t} the current pressure wavefield, $P_{g_{t-1}}$ the preceding one, and $P_{g_{t+1}}$ the one directly following with a step size in time of $dt = 0.1\,\mathrm{s}$. The other subscripts follow accordingly. The whole pressure wavefield P_G can be treated at once instead of pixel-wise. The resulting 128×128 image representing $\frac{\partial^2}{\partial t^2} P_G$ is cropped to 126×126 so that we can check the pixel-wise adherence to the PDE for all pixels that are not in a two-pixel thick area next to the boundary.

Finally, the two physical consistency-based losses are added to the generator cost function (3.12) to get

$$\mathcal{J}_G = f_w(\tilde{\boldsymbol{x}}) + \lambda_{\ell_1} \|\tilde{\boldsymbol{x}} - \boldsymbol{x}\|_1 + \lambda_{\mathrm{PDE}} \mathcal{L}_{\mathrm{PDE}} + \lambda_{\mathrm{b}} \mathcal{L}_{\mathrm{b}} \,, \tag{3.19}$$

with the PDE-related loss (see Fig. 3.6 for a visualization of the process to calculate it):

$$\mathcal{L}_{\mathrm{PDE}} = \frac{1}{N_{\mathrm{p}}} \sum_{n_p=1}^{N_{\mathrm{p}}} \left(\frac{1}{c^2} \frac{\partial^2 P_G}{\partial t^2} - \nabla^2 P_G - 0 \right)^2 \tag{3.20}$$

and the boundary-related loss

$$\mathcal{L}_{\mathrm{b}} = \frac{1}{N_{\mathrm{b}}} \sum_{n_b=1}^{N_{\mathrm{b}}} \left(P_G(x, t, z = 1000\,\mathrm{m}) - 0 \right)^2 , \tag{3.21}$$

where the running subscripts n_p and n_b are left out of the equation for readability purposes. n_p refers to the pixels (i, j) on which $\mathcal{L}_{\mathrm{PDE}}$ is calculated, of which there are $N_{\mathrm{p}} = 126 \cdot 126 = 15,876$. Additionally, λ should be thought of as discretized, not continuous anymore, as we working on individual pixels. n_b refers to the pixels depicting the free surface of which there are $N_{\mathrm{b}} = 128$. λ_{PDE} and λ_{n} are again hyperparameters that are searched for in the hyperparameter-tuning phase (see Chapter 3.2.3).

3.2.3 Training procedure

We use PyTorch as the deep learning framework of choice and use a graphics card (GPU) to accelerate the training. We use the same seed for the random variables for all *runs*. With runs we denote a complete training and validation procedure. The same seed is used to ensure that the same data are always in the different data sets. This does not mean that the training is completely reproducible even for the same hyperparameters, as non-deterministic algorithms, e.g., the 2D convolution on a GPU, are used during training to increase the performance of the underlying calculations. It is possible to require many of the non-deterministic algorithms to use a deterministic version, but some algorithms

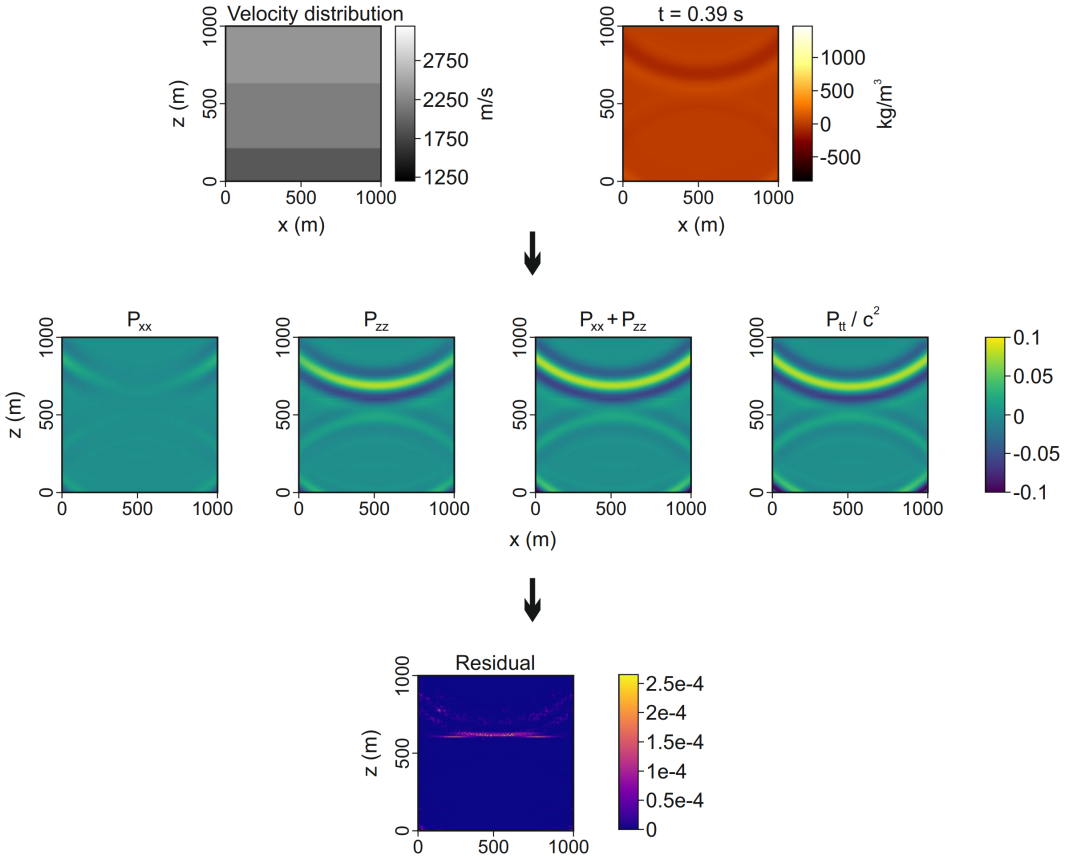

Figure 3.6: Visualization of the parts of the $\mathcal{L}_{\text{PDE}}$ loss for one specific wavefield, where the subscripts in the titles of the middle row denote the second derivatives w.r.t. x, z, and t, respectively. Residual in the lowest plot is the result of $\left(\frac{1}{c^2}\frac{\partial^2 P_G}{\partial t^2} - \nabla^2 P_G - 0\right)^2$.

do not allow for that [105]. In our case the backpropagation through the 2D bilinear upsampling layer prohibited using a deterministic implementation.

We divide the data set into three parts: the training set, the validation set, and the testing set. We allocate 80% of the data for training purposes, as this constitutes the majority of the data from which our model learns. The remaining data is divided equally into validation and test sets, each representing 10% of the total data. This is a commonly used split in deep learning [47]. The samples in the training data set are shuffled randomly, ensuring that the network does not learn on data that is in chronological order. We do this, as we want our network to produce single pressure wavefields at certain time steps without the need to have early wavefields.

Next, we initialize the model's weights with values drawn from a normal distribution with a mean of 0.0 and a standard deviation of 0.02. The biases of the model are initialized to a constant 0. These values follow Kadeethum, et al. [1]'s recommendation. Although the choice of initialization in deep learning architectures can significantly impact the model's performance, our usage of normalization layers alleviates much of this concern [78], [101].

Instead of using the traditional SGD introduced in Chapter 2.2.1, we use a more advanced learning rate optimization algorithm called *Adam* [106]. It is an adaptive learning rate method that changes the learning rate of the gradient descent depending on exponential moving averages of the gradients as well as the squared gradients. The moving average of the gradients is an estimate of the first moment (the mean) and the moving averages of the squared gradients is an estimate of the second raw moment (the uncentered variance) of the gradient. With the current gradient matrix $\boldsymbol{G}^k = \nabla_{\boldsymbol{\theta}} \mathcal{J}^k$ calculated using backpropgataion, $\boldsymbol{m}^k$ the current vector-valued first moment value, and $\boldsymbol{v}^k$ the current vector-valued second moment value, the next moment's values can be calculated using

$$\boldsymbol{m}^{k+1} = \beta_1 \boldsymbol{m}^k + (1 - \beta_1)\boldsymbol{G} \tag{3.22}$$

$$\boldsymbol{v}^{k+1} = \beta_2 \boldsymbol{v}^k + (1 - \beta_2)\boldsymbol{G} \odot \boldsymbol{G}\,, \tag{3.23}$$

where hyperparameters β_1, $\beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages, and $\odot$ indicates that the matrices are element-wise multiplied. $\boldsymbol{m}$ and $\boldsymbol{v}$ are usually initialized to 0, therefore being biased towards 0. To counteract that bias, we introduce the bias-corrected moments

$$\hat{\boldsymbol{m}}^{k+1} = \frac{\boldsymbol{m}^{k+1}}{1 - \beta_1^{k+1}} \tag{3.24}$$

$$\hat{\boldsymbol{v}}^{k+1} = \frac{\boldsymbol{v}^{k+1}}{1 - \beta_2^{k+1}} \tag{3.25}$$

The updated weights $\boldsymbol{\theta}^{k+1}$ are then calculated via

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta} - \eta \frac{\hat{\boldsymbol{m}}^{k+1}}{\sqrt{\hat{\boldsymbol{v}}^{k+1}} + \epsilon}\,, \tag{3.26}$$

with η the an initial learning rate and ϵ a small value to prevent division by 0. Adam is fairly robust to the choice of values for the decay rates and initial learning rate, especially compared to SGD [107]. No single best optimization algorithm has emerged so far, but the robustness of adaptive methods have made them the preferred choice of use [47].

We employ Optuna for finding the best hyperparameter for our model. Optuna is an automatic hyperparameter optimization software framework for machine learning [108]. It works by doing multiple *trials* - meaning multiple runs - with different hyperparameters. It enables efficient hyperparameter optimization using sampling algorithms, from the more traditional *grid search* and *random search* to the default *Tree-structured Parzan Estimator* (TPE) [109], which we use in this thesis. In the grid search approach, hyperparameters are selected from an exhaustive set of combinations that lie on a high-dimensional grid, ensuring thorough but computationally heavy exploration. Random search arbitrarily chooses hyperparameters in a certain range, reducing the computational load and allowing for wider and more efficient exploration, particularly in high-dimensional spaces [110]. TPE is a more sophisticated approach, as it models the objective function as a mixture of two Gaussian processes and iteratively updates these processes to make informed decisions

about where to smaple next in the hyperparameter space. This method balances the exploration-exploitation trade-off and can deliver superior results in less time compared to grid search or random search [108]. The hyperparameters and the ranges in which we search for optimal values can be found in Table 3.3.

Hyperparamter	Type	Min	Max
Generator learning rate	Float	$5 \cdot 10^{-5}$	$8 \cdot 10^{-4}$
Generator hidden channel amount	Integer	32	50
Discriminator learning rate	Float	$5 \cdot 10^{-5}$	$1 \cdot 10^{-3}$
β_1	Float	0.6	0.99
β_2	Float	0.9	0.999
Batch size larger data set	Categorical	16	32
Batch size smaller data set	Categorical	4	8
CBN hidden layer size	Integer	50	100
CBN output size	Integer	6	12
λ_{gp}	Integer	5	15
λ_{ℓ_1}	Integer	5	250
λ_{PDE}	Integer	50	400
λ_{b}	Integer	0	15

Table 3.3: Hyperparameters used in the Optuna optimization process with their type, minimum, and maximum values. When optimizing hyperparameters for a non-physics-informed NN, we simply set $\lambda_{\text{PDE}} = \lambda_{\text{b}} = 0$.

Following Kadeethum, et al. [93], the metric that we use to measure the performance of our network is the *relative root-mean-square error* (RMSE) between the sampled real wavefield P_r and generated wavefields P_G, defined as

$$\sqrt{\frac{\frac{1}{N}\sum_{n=1}^{N}(P_G - P_r)^2}{\frac{1}{N}\sum_{n=1}^{N}{P_r}^2}}, \tag{3.27}$$

where $N = 128 \cdot 128 = 16,384$ is the total amount of pixels in the pressure wavefield image. Again, we left out the subscript n referring to all pixels (i, j) for readability purpose.

For both the smaller data set with only uniform velocity distributions as well as for the larger data set with examples of horizontal layering, we train the network for 400 *epochs*, where an epoch is a complete pass through the entire dataset [63]. We stop the training of the network early if either the average RMSE of the training samples in an epoch (we will refer to that as training RMSE) or the average RMSE of the validation samples in an epoch (validation RMSE) surpasses a certain threshold. Additionally, if the validation RMSE does not improve for 270 epochs, training also stops. The first check is to catch networks that do not learn properly. The other two are done at later epochs to stop training when overfitting occurs, which is indicated by the validation RMSE not improving any more and rising after reaching a minimum [111].

4. Results

4.1 Physics-informed vs. purely data-driven

In a first step, we want to check how our physics-informed network compares to the non-physics-informed counterpart. Both network architectures are exactly the same, the only difference is that λ_{PDE} and λ_{b} are set to 0 in the non-physics-informed cost function of the generator. We use the smaller data set of 15 uniform velocity distributions and do 100 Optuna-trials for both the physics- and the non-physics-informed network to find the hyperparameters that minimize the validation RMSE. We restrict ourselves to the small data set to reduce the time training and validating takes. We will refer to both together as learning time. Relatively small batch sizes of 4 or 8 are tried, where 4 is the batch size used in Kadeethum, et al. [93], but 8 provides a faster learning time of $\approx 40\,\mathrm{min}$ compared to $\approx 60\,\mathrm{min}$. The runtime varied with the hyperparameters, but were similar for both physics- and non-physics-informed training. The 100 trials were performed on the two GeForce RTX 3080 10GB GPUs simultaneously and overall took $\approx 84\,\mathrm{hours}$. The network occupied between $\approx 2\,\mathrm{GB}$ and $\approx 4\,\mathrm{GB}$ on a GPU's memory, depending on the number of hidden channels in the generator and the CBN's hidden layer & output size - where more layers and larger output size means a bigger network. All wavefields surrounding the time steps of interest are also put on the GPU for fast access and occupy $\approx 55\,\mathrm{MB}$. This is less then the $\approx 3\,\mathrm{GB}$ and $\approx 7\,\mathrm{GB}$ after deducing background operations when querying the memory usage using `nvidia-smi` in the terminal. This is because some unused memory can be held by the caching allocator and some context needs to be created on GPU [112].

4.1.1 Learning dynamics and hyperparameters

Out of 100 trials, the 12 with the lowest epoch-averaged validation RMSE are picked. We usually stick with epoch-averaged values instead of per step (equal to per mini-batch) metrics, as we are generally interested in the performance on the whole training or validation data set. From the 12 best trials, 6 each are from the physics-informed and non-physics-informed approach. These are again equally split into runs with a batch size of 4 or 8. Out of 12 trials, the two best are chosen for further examination, one from the physics-informed and one from the non-physics-informed approach. Fig. 4.1 shows the progression of the lowest validation RMSE through the epochs. After a fast initial drop, the RMSEs stay constant until around epoch 150 for a batch size of 4 and epoch 250 for a batch size of 8 before they start to improve. The best hyperparameters for the physics-informed approach were found in optimization trial 51, where the network parameters $\boldsymbol{\theta}$ that minimize the validation RMSE were calculated in epoch 364. The best hyperparameters for the non-physics-informed network were found in trial 26, with the best network parameters $\boldsymbol{\theta}$ determined in epoch 263 (see Table 4.1).

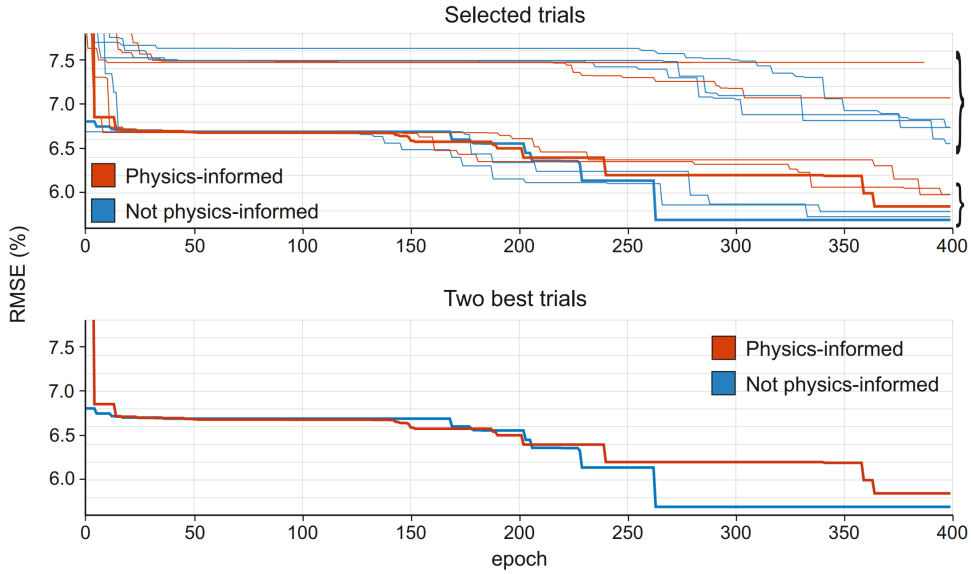

Figure 4.1: The upper plot shows 12 of the best trials out of 100. Trials with a batch size of 8 all end in the upper curly bracket and trials with a batch size of 4 all end in the lower curly bracket. The best trials for the physics-informed and non-physics-informed approach are highlighted in the lower plot.

Hyperparameter	Physics-informed	Non-physics-informed
Generator learning rate	$3.6107 \cdot 10^{-4}$	$2.4762 \cdot 10^{-4}$
Generator hidden channel number	35	33
Discriminator learning rate	$6.1314 \cdot 10^{-4}$	$7.4802 \cdot 10^{-4}$
β_1	0.95404	0.82513
β_2	0.90882	0.93580
Batch size	4	4
CBN hidden layer size	74	55
CBN output size	6	7
λ_{gp}	13	11
λ_{ℓ_1}	217	70
λ_{PDE}	318	0
λ_{b}	0	0

Table 4.1: Hyperparameters used in the trials resulting in the lowest validation RMSE

We verify that the best RMSEs are dependent on the batch size and that this is not due to an error in the calculation of the RMSE for different batch sizes. We do so by using two additional loss metrics, the well-known *mean squared error* (MSE) $\frac{1}{N}\sum_{n=1}^{N}\left[(\boldsymbol{P}_g - \boldsymbol{P}_r)^2\right]$ and the *structural similarity index measure* (SSIM). The Structural Similarity Index (SSIM) is a method for measuring the similarity between two images, which focuses on the preservation of structural information by comparing local patterns of pixel intensities that have been normalized for luminance and contrast, thereby providing a more perceptually relevant assessment of image quality [113]. We do the verification on non-averaged metrics on the training data set of the best performing trials with a batch size of 4. We add the best performing run with a batch size of 8 from the non-physics-informed hyperparameter

search, which is from trial 4. All of these coincide with the respective best performing runs based on the validation RMSE. Instead of relying on the highly fluctuating epoch-averaged validation RMSE, we use training metrics as these converge to lower values over time, making it easier to compare the different runs. We can then see that the best performing run with a batch size of 8 performs worse than the run with a batch size of 4, validating our finding that the RMSEs are batch-size dependent. Interestingly, the training metrics of the best physics-informed trial with a batch size of 4 is lower than the best non-physics-informed trial with a batch size of 8 (see Fig. 4.2). This is in contrast to the minimum validation RMSE, where the trials with smaller batch sizes outperform the ones with larger batch sizes in all cases - independent of the whether or not they are physics-informed.

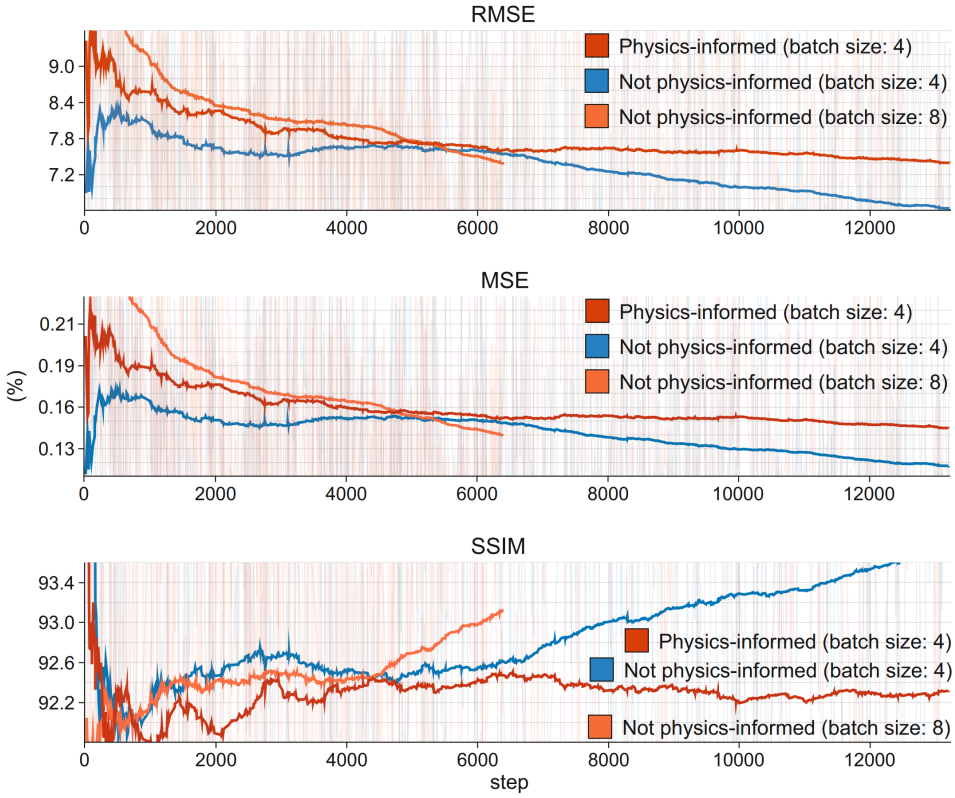

Figure 4.2: Comparison of alternative loss metrics. Notice the x-axis in steps, not epochs. The graphs are extremely smoothed as the variance between steps is very high. The orange function ends earlier, as a batch size of 8 leads to less steps required per epoch to get trough the whole training data set.

We now take a closer look at the network's learning behavior for the two best trials. It is important to note that the cost values are the current values of the high-dimensional cost function, not a representation of the function itself. This current value representation is also called *learning curve* of the network - in our case, for both the generator as well as the critic [114]. The generator's learning curve as well as the unweighted, i.e., without taking into account the λ's, parts that make up the generator's cost, namely the ℓ_1-related loss, $\mathcal{L}_{\text{PDE}}$, and $\mathcal{L}_{\text{b}}$ are shown in Fig. 4.3. In the following, we will refer to the current value of the ℓ_1-related loss as *L1 loss*, of $\mathcal{L}_{\text{PDE}}$ as *PDE loss* and of $\mathcal{L}_{\text{b}}$ as *boundary loss*.

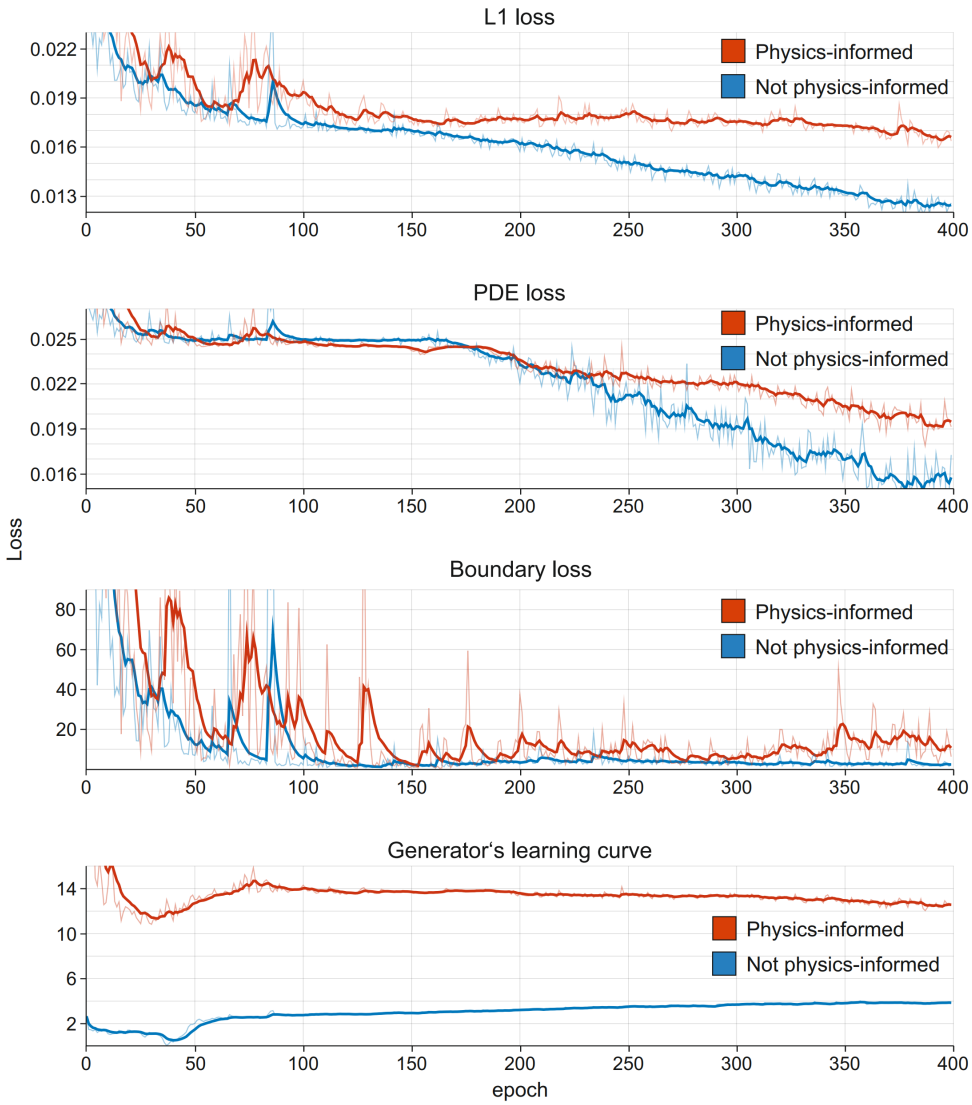

Figure 4.3: Generator's learning curve (lowest plot) and behavior of the unweighted summands of the generators's cost function.

One can see that the non-physics-informed network outperforms the physics-informed network for all parts of the generator's cost. For the non-physics-informed run, the L1 loss reduces rapidly for about 50 epochs. It is then relatively stagnant until approximately epoch 160, with the exception of a sudden spike in loss at around epoch 80. After epoch 160, the loss decreases again, but slower than in the beginning. The PDE loss behaves very similar to the L1 loss, reducing around epoch 160 after an initial drop followed by a constant loss from epoch 50 to 160. A similar spike can be seen at around epoch 80. The boundary loss also shows the initial, fast drop to lower values. The same pronounced spike as in the L1 loss at around epoch 80 can be seen as well. Afterwards, the loss stays at a constant low value.

For the physics-informed run, the L1 loss stagnates at around epoch 125, after an initial reduction accompanied by two bigger fluctuations. The L1 loss goes down only slightly starting around epoch 350. In contrast to the non-physics-informed run, the PDE loss behaves differently than the L1 loss, showing a clear reduction in loss starting around

epoch 180, though it never reaches as low values as the non-physics-informed run. The boundary loss is characterized by large fluctuations throughout and generally slightly higher values after the initial drop. Additionally, the loss seems to go up again at around epoch 300. The weighting λ_b is 0 for the best physics-informed run, which might explain the high fluctuations for the boundary loss. In the non-physics-informed run, the L1 loss also leads to a lower boundary-related loss. In the physics-informed run, the L1 loss is almost constant for most of the epochs later then 125, meaning that the network does not learn to enforce it and consequently the boundary-related losses are also neglected.

The two learning curves of the generator cannot be compared, as the weighting of the different parts of the cost function defines the scaling of it. In the traditional WGAN-setting, the generator's current cost value corresponds to the quality of the output. This is not the case with our new cost function, as the non-physics-informed generator's learning curve's value increases almost throughout, even though the metrics on the training images improve. The increase in cost seems counterintuitive, as the different parts of the cost function shown in Fig. 4.3 have a decreasing tendency. This increase is due to the unweighted part in our generator's cost function that stems from the traditional WGAN-generator cost function - we will refer it as traditional WGAN-loss (see equation 3.19 and Fig. 4.4). The addition of the different loss parts - as well as their weighting - therefore plays a big role in the shape of the generator's learning curve. Moreover, these additions mean that the generator's learning curve no longer gives an indication of the output's quality.

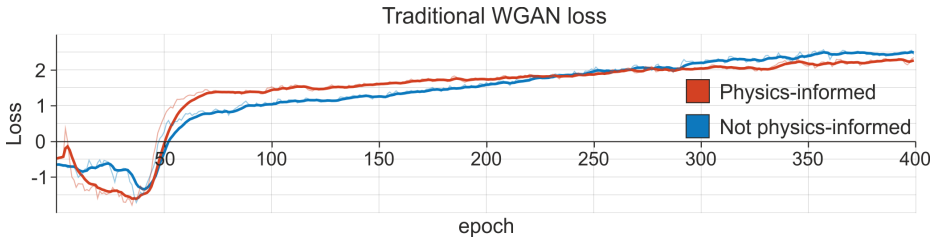

Figure 4.4: Part of the generator's cost that stems from the traditional WGAN's generator's cost function.

The critic's learning curve is presented in Fig 4.5, showing that the cost converges towards 0 very quickly. This a wanted behavior, as the WGAN approach works on the assumption of an optimal critic. This rapid drop to almost 0 at around epoch 50 correlates to the change in behavior in the generator's cost function at the same epoch.

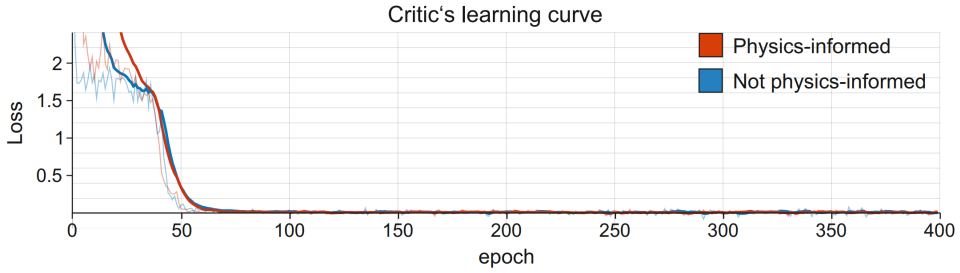

Figure 4.5: Critic's learning curve.

The training and validation RMSE can be seen in Fig. 4.6. The behavior of the training RMSEs is very similar to the PDE-related loss. It is almost constant with a value $\approx 7.5\,\%$ from epoch 50 to 160 for both the physics-informed and the non-physics-informed networks. From epoch 160 onward, the non-physics-informed network's training RMSE decreases steadily to a minimum of $\approx 5.3\,\%$ at the end of the training. In contrast, the physics-informed network's training RMSE stays almost constant until around epoch 300, where it reaches a value of $\approx 7.4\,\%$, from where on it decreases to a minimum of $\approx 6.4\,\%$ at the end of the training. The validation RMSE behaves similarly to the training RMSE with its initial reduction to a constant RMSE until around epoch 200, though in validation this constant value is $\approx 7.0\,\%$. Following this phase, the graph transitions into a period denoted by high fluctuations. To make these fluctuations more visible, we show only the non-smoothed RMSE graphs in Fig. 4.6. From epoch 200 to 350, the values oscillate around what could be visualized as a convex parabola, akin to an upside-down bowl. Some of the values in this phase go down so far that we find the minimum validation RMSE of $5.7\,\%$ at epoch 263 for the non-physics-informed run. For the last 50 epochs, the RMSE oscillates around $\approx 7.0\,\%$. Here we also find the lowest validation RMSE of $5.8\,\%$ at epoch 364 for the physics-informed run.

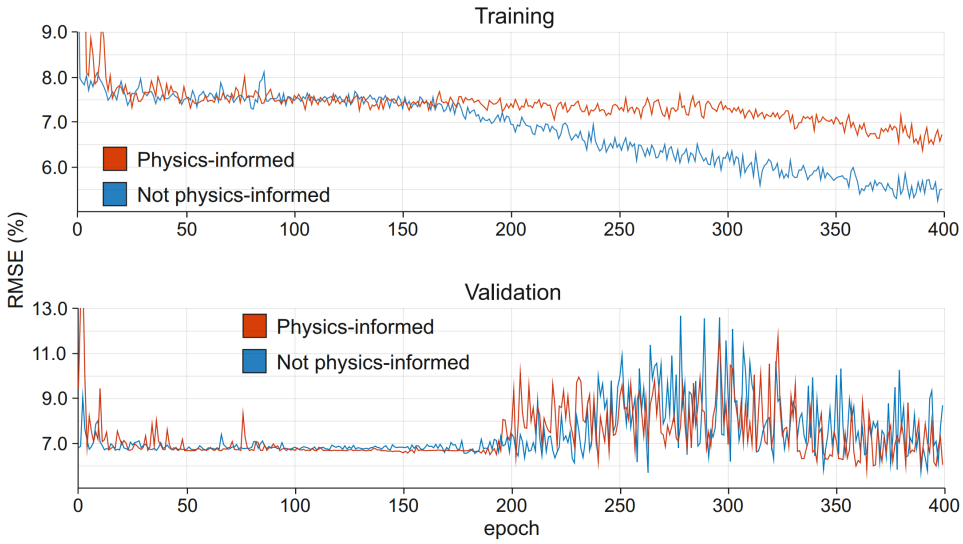

Figure 4.6: Non-smoothed training and validation RMSEs for the two best runs.

4.1.2 Visual inspection of generated pressure wavefields

We will look at examples of generated pressure wavefields to see how they correspond to the calculated RMSEs. For both physics-informed and non-physic-informed runs, we will present examples corresponding to velocity distributions from the validation data set at the epochs where the best RMSE was calculated. Additionally, generated wavefields corresponding to velocity distributions from the training data set at the end of training are shown. A few remarks to the figures: Unfortunately, we made a mistake in the saving of the figures, which is why we did not have access to all batches of training or validation data and do not have the time step for which the wavefields are generated. In addition,

we only saved the pressure wavefields in gray-scale, instead of using the same color as in Chapter 3. Nonetheless, as we are purely interested in the visual quality of the wavefield images right now, these issues are not important. Also, we do not label the axis to make it easier to focus on the actual wavefield images.

On the validation data set, we see that the non-physics-informed generator is able to produce wavefields that resemble the true wavefields at time steps before it is affected by the boundary. While the position, shape, and size are good, the amplitudes are slightly off. As soon as the wavefields come into contact with the boundary, the shape of the wavefields starts smearing, therefore differing from the true one. Reflected waves from the free-surface boundary are never drawn by the generator (see Fig. 4.7). The same behavior can still be observed on the training data set at the end of training, with slightly better performance for wavefields close to the boundaries. Even on the training data set, no reflected waves are generated (see Fig. 4.8).

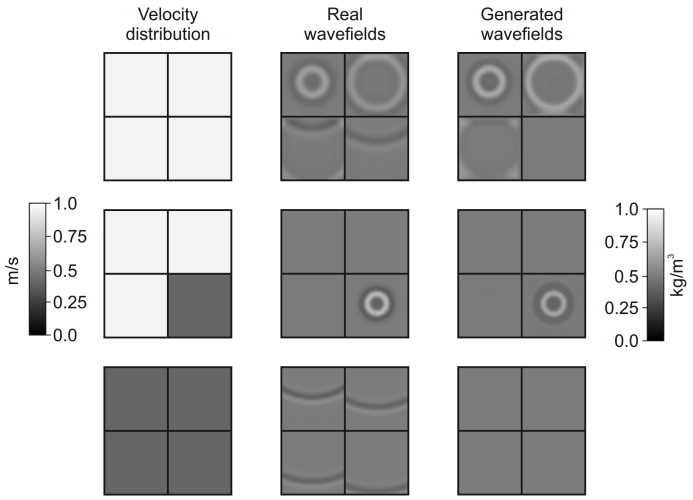

Figure 4.7: Validation examples from the epoch with the lowest validation RMSE of all non-physics-informed runs.

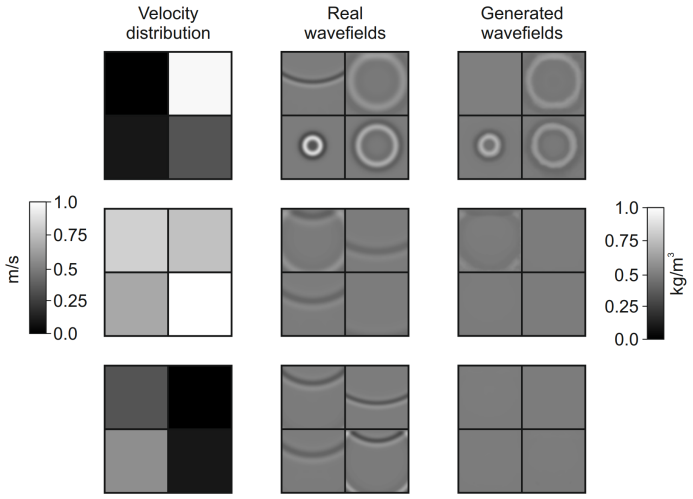

Figure 4.8: Training examples at the end of the best non-physics-informed run based on lowest validation RMSE.

The physics-informed generator struggles to reproduce wavefields on the validation data set. Though limited by the available examples, we can still see that it has even more problems generating wavefields at later time steps. Not only are no reflected waves visible from the free surface, but also no wavefields touching the boundaries can be seen. The only wavefield shown has a circular shape, but it differs in its diameter from the true one. The amplitude is also off significantly. We can also see a kind of focal point in the middle of the generated wavefield with high amplitude which is not present in the real wavefield (see Fig. 4.9). On the training data set, the visual quality of the generated wavefields does not improve significantly. We still observe that the generated wavefields are not matching the real ones in size and amplitude. Again, no wavefields are drawn for later time steps at which the wavefield is impacted or reflected from the boundaries (see Fig. 4.10).

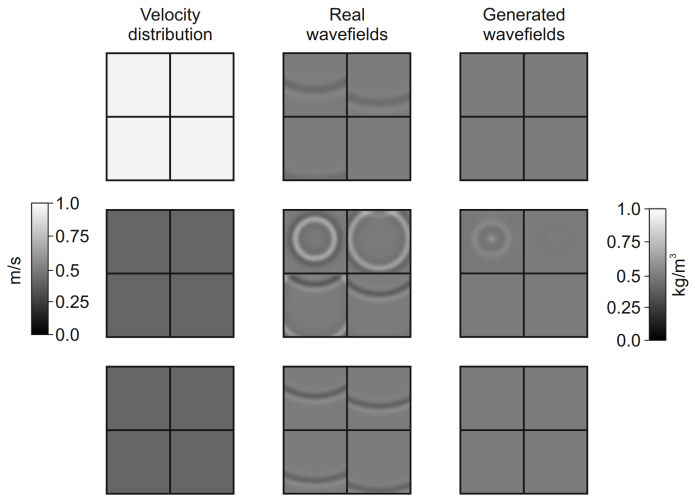

Figure 4.9: Validation examples from the epoch with the lowest validation RMSE of all physics-informed runs.

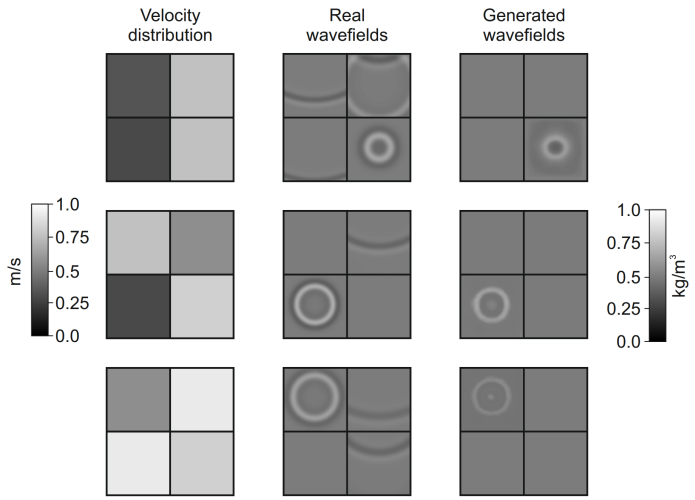

Figure 4.10: Training examples at the end of the best physics-informed run based on lowest validation RMSE.

In conclusion, our findings that the metrics of the non-physics-informed network outperform those of the physics-informed networks are confirmed by the superior visual quality of the wavefield images in both testing and validation. We will do a more thorough inspection of generated wavefield images for the layered velocity distributions in Chapter 4.2. We did not check the networks performance on the testing dataset. The main reason is that we would have needed to save all the networks during the hyperparameter search to select the relevant examples at a later stage. This would have occupied prohibitively large storage in the system we used. Assuming a middle ground scenario where the network takes up 3 GB, we would have needed $2\,\text{checkpoints} \cdot 2\,\text{GPUs} \cdot 100\,\text{trials} \cdot 3\,\text{GB} = 1200\,\text{GB}$, as we would have needed to save the network producing the lowest validation RMSE as well as the final network. This amount of storage was not available on the computer we used. We therefore checked the performance on the test data set in the layered velocity distribution portion (see Chapter 4.2), where we were able to save the best and final network for all trials done.

4.2 Layered velocity distribution

Even though the physics-informed network performed worse on the data set with only uniform velocity distributions, we anticipated that adding physics information to the generator would be helpful in a setting where the velocity distributions and therefore the pressure wavefields are more complex. We therefore decided to continue with the physics-informed approach to test its performance on the more complex and larger data set with layered velocity distributions. We did not do a comparison between physics- and non-physics-informed networks on the larger data set due to restrictions in computer capabilities. We used a GeForce RTX 4080 16GB GPU, which is generally more capable than the previously used GeForce RTX 3080s, but only a single unit was available. In the smaller data set, we see that a smaller batch size improves the validation RMSE but also means an increase in learning time. In order to get reasonable learning times we let Optuna try 16 or 32 as batch size, resulting in learning times of ≈ 16 hours, independent of the two batch sizes. It should be noted that we tried smaller batch sizes and saw a significant slow-down compared to the above chosen batch sizes. Overall the 14 trials performed took ≈ 171 hours or ≈ 7 days to complete.

The network occupies the same amount of GPU memory as before. All wavefields surrounding the time steps of interest now occupy ≈ 3.5 GB due to the increased data-set size. Overall, after deducing background processes, between ≈ 7.5 GB and ≈ 12 GB of the GPU's memory are used during the learning process, explaining our need for a different graphics card with more memory than the GeForce RTX 3080 with 10 GB of available memory.

4.2.1 Learning dynamics and hyperparameters

The progression of the lowest validation RMSE of the 14 trials is shown in Fig. 4.11. We again see a dependency of the validation RMSE on the batch size, with the additional

finding that most of the trials with a batch size of 32 are stopped very early due to high training RMSEs. There is only a single trial that reaches epoch 400, though its validation RMSE only improves from 5.089 % to 5.088 % after epoch 6. We focus our examination on the two best performing trials 0 and 3 (see lower plot in Fig. 4.11). Trial 0 reaches the lowest validation RMSE of all trials at epoch 106 and is stopped at epoch 376. Trial 3 has its lowest validation RMSE in epoch 90 and is also stopped early at epoch 360. Both of them are stopped early as they fulfilled the criterion of stopping based on the validation RMSE not improving for 270 epochs. The hyperparamters that lead to network parameters $\boldsymbol{\theta}$ that minimize the validation RMSE are presented in Tab. 4.2.

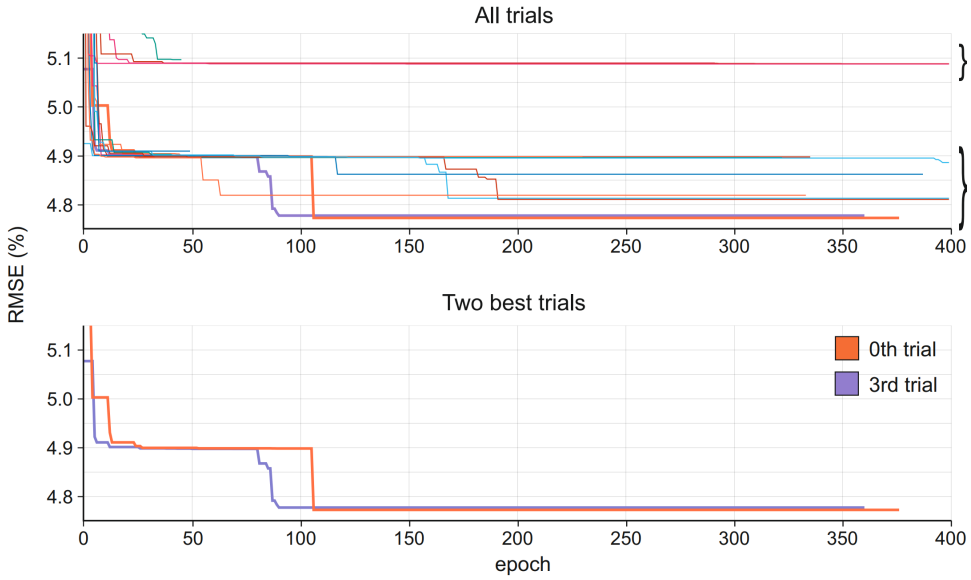

Figure 4.11: The upper plot shows all 14 trials. Trials with a batch size of 32 all end in the upper curly bracket and trials with a batch size of 16 all end in the lower curly bracket. The two best trials are highlighted in the lower plot.

Hyperparameter	trial 0	trial 3
Generator learning rate	$5.4078 \cdot 10^{-4}$	$3.6146 \cdot 10^{-4}$
Generator hidden channel number	38	44
Discriminator learning rate	$2.3158 \cdot 10^{-4}$	$8.2316 \cdot 10^{-4}$
β_1	0.81743	0.80751
β_2	0.94956	0.92293
Batch size	16	16
CBN hidden layer size	70	82
CBN output size	7	7
λ_{gp}	12	15
λ_{ℓ_1}	180	246
λ_{PDE}	253	110
λ_{b}	14	2

Table 4.2: Hyperparameters used in the two trials resulting in the lowest validation RMSE

The generator's learning curve as well as the unweighted parts that make up the generator's cost, namely the traditional WGAN-loss, the ℓ_1-related loss, $\mathcal{L}_{\text{PDE}}$, and $\mathcal{L}_{\text{b}}$ are shown in Fig. 4.12.

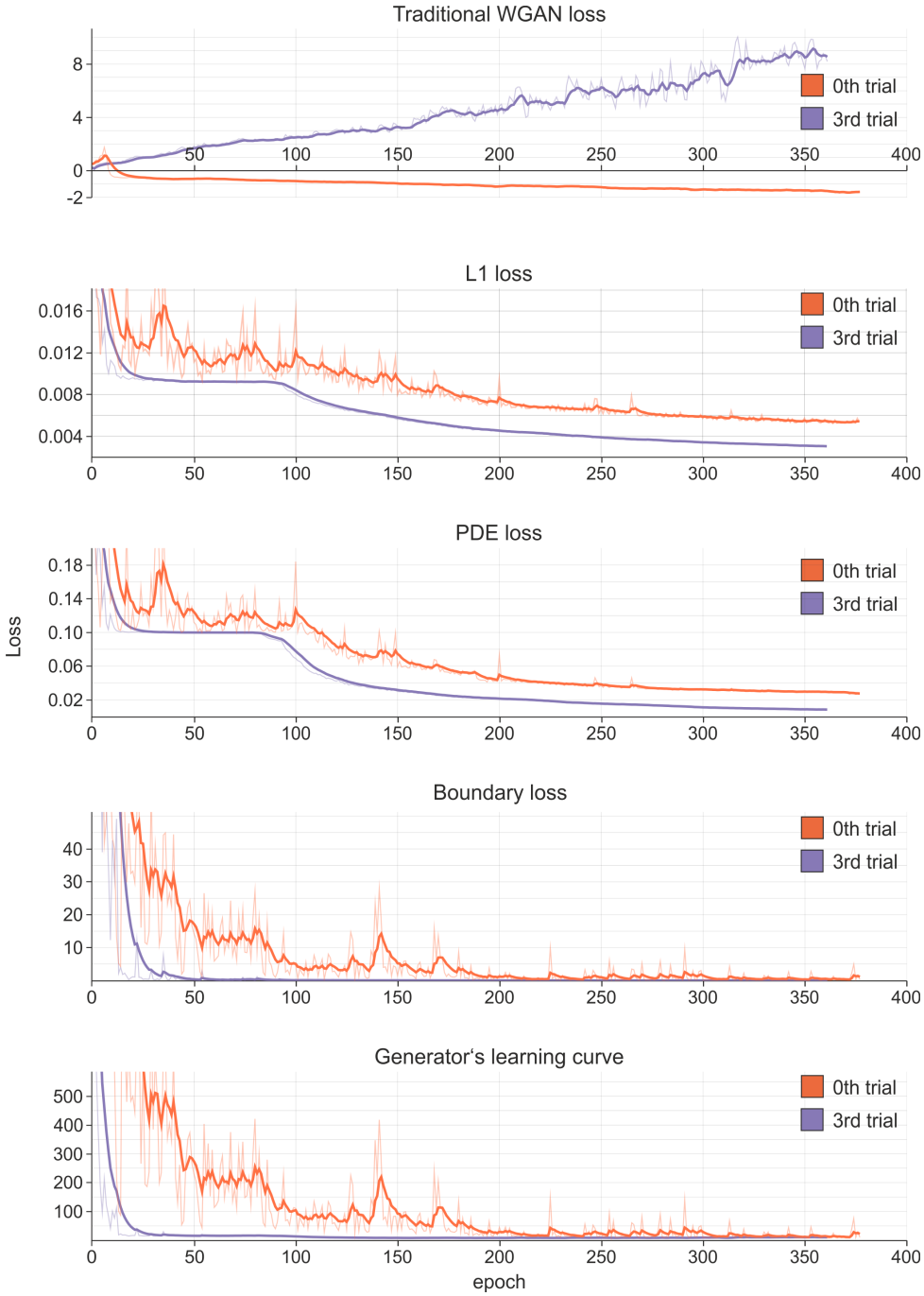

Figure 4.12: Generator's learning curve (lowest plot) and behavior of the unweighted summands of the generators's cost function.

The traditional WGAN-loss's behaviour differs strongly between trial 0 and trial 3. For trial 3, it increases almost linearly from 0 to ≈ 9 at the end of training. In contrast, for trial 0 it decreases after an initial increase from 0 to almost 2 at epoch 6. It starts becoming negative at epoch 9, after which it decreases almost linearly to ≈ -1.5. The other losses behave similar between trials, though trial 0's values fluctuate more and

are generally larger. For trial 0, after a fast reduction in loss and a phase with high fluctuations up to approximately epoch 100, the L1 and PDE show a decreasing tendency. The boundary loss decreases until epoch 200, after which it is constantly close to 0. The decrease is not linear, but rather interrupted by some plateaus and spikes. Trial 3's values for the L1 and PDE loss are almost constant until epoch 80 after an initial fast drop. Afterwards, both losses decrease without any fluctuations until the end of training, with the rate of decrease slowing over the epochs. The boundary loss in trial 3 falls rapidly towards 0 and is almost constant starting at epoch 80.

The learning curve of the generator of trial 0 is clearly dominated by the boundary loss, as the shapes of the two curves almost perfectly overlap. The learning curve of the generator of trial 4 has four phases: it decreases rapidly until epoch 20 when it plateaus until epoch 80, similar to the L1 and PDE loss. This almost constant value is overprinted with fluctuations stemming from the boundary loss. These fluctuations stop afterwards and the learning curve follows the downward trend of the L1 and PDE loss. Around epoch 200, the traditional WGAN loss takes over, where the generator's learning-curve shape and its tendency starts to follow it. It is interesting to see that the boundary loss of trial 0 is larger and has a higher amplitude than the one of trial 3, even though the weighting λ_b in trial 0 is seven times larger than the one in trial 3 (14 vs 2). This is possibly countered by the larger value of $\lambda_{\ell 1}$ of 246 in trial 3 compared to 180 in trial 0. The L1 loss does a per-pixel check between real and generated wavefields, so it will also penalize the values on the free-surface boundary.

The critic's cost is almost 0 throughout after an initial reduction for both trials (see Fig. 4.13). Again, this is a wanted behaviour as far as traditional WGAN go, as an optimal critic is required.

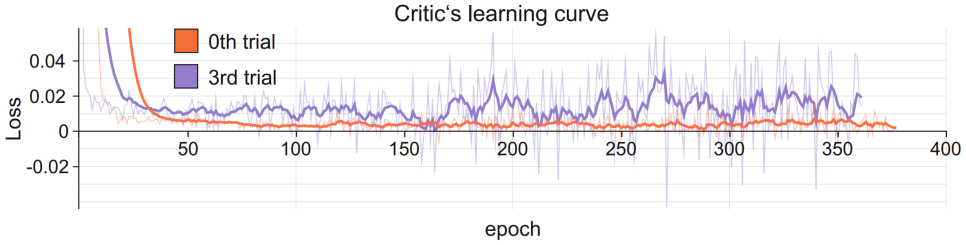

Figure 4.13: Critic's learning curve.

We again show the non-smoothed training and validation RMSEs, as the fluctuations between epochs have a high impact on which epoch is chosen for the best validation RMSE (see Fig. 4.14). While trial 3's validation RSME is smaller for most epochs, both end with a similar RSME of $\approx 5.6\,\%$. The training RMSEs of both trials closely follows the behaviour of the L1 and PDE loss from the generator (see Fig. 4.12) and reach their minimum at the end of training, with a value of $2.7\,\%$ for trial 0 and $1.6\,\%$ for trial 3. Similar to what we saw when we evaluated the generator's learning curve, trial 0 has more fluctuations for both RMSEs.

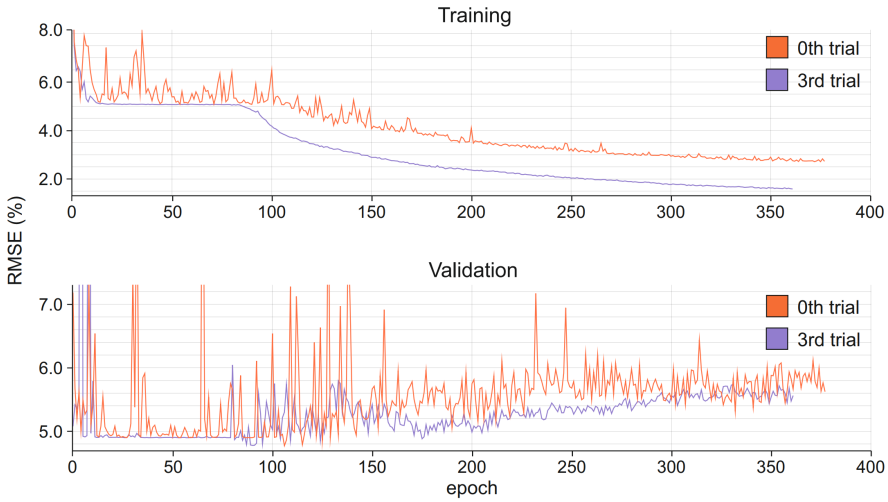

Figure 4.14: Non-smoothed training and validation RMSEs for the two best runs.

4.2.2 Visual inspection of generated pressure wavefields

We will visually compare the generator's output on the validation data set for both trials. For that, we make use of saved generator states. We saved the state of the generator - i.e., all its weights and biases - at the epochs of lowest validation RMSE and at the end of training. We compare the wavefields generated for the same three velocity distributions - uniform, two-layer, and three-layer. This is done for all 12 time steps on which the network was trained: [0.18, 0.25, 0.32, 0.39, 0.46, 0.53, 0.6, 0.67, 0.74, 0.81, 0.88, 0.95] seconds. We present wavefields for selected time steps in Fig. 4.15 for trial 0 and Fig. 4.16 for trial 3 (for all time steps see Fig. A.2 for trial 0 and Fig. A.3 for trial 3).

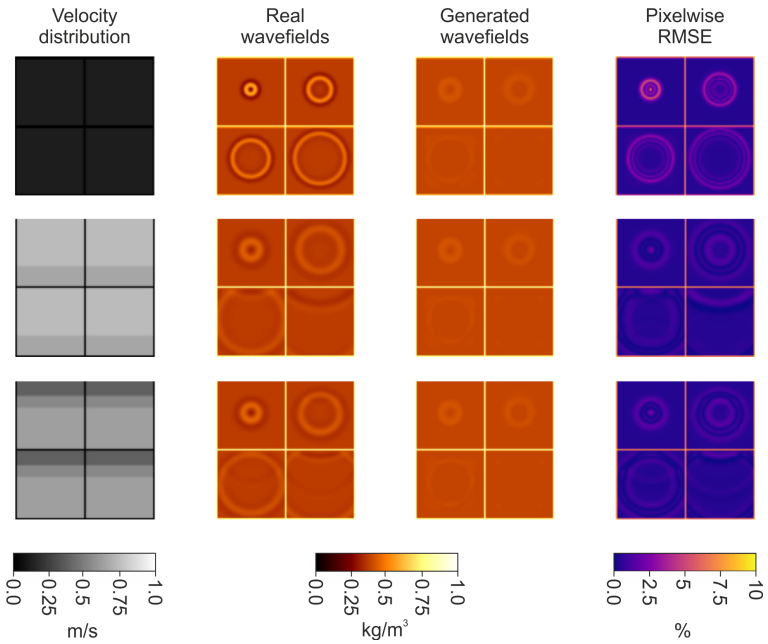

Figure 4.15: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from epoch 106 (with lowest validation RMSE) of trial 0. Wavefields are shown in a zig-zag pattern from top left to bottom right for time steps [0.18, 0.25, 0.32, 0.39] seconds.

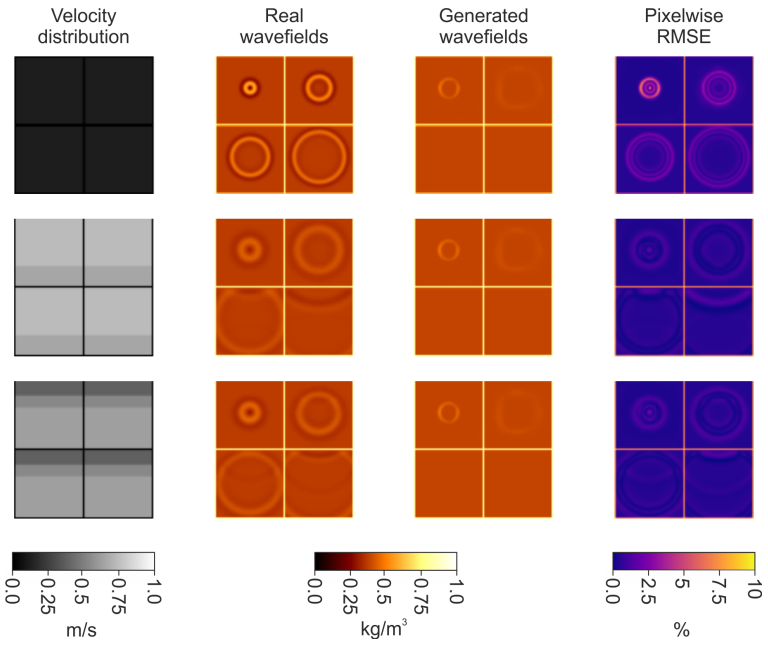

Figure 4.16: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from epoch 90 (with lowest validation RMSE) of trial 3. Wavefields are shown in a zig-zag pattern from top left to bottom right for time steps [0.18, 0.25, 0.32, 0.39] seconds.

It appears that the lowest validation RMSE were achieved by networks that are only able to generate wavefields for early time steps. Seemingly, it was enough to generate near optimal background pressure and only some overlapping structures between real and generated wavefields to achieve the best validation RMSE. This also explains why the lowest validation RMSE was found at in epochs in training (90 and 106), as these epochs coincide with the epochs where the training RMSE began reducing. This reduction is an indication that the generator starts to create wavefield images with structures other than the background in training. This change in output carries over to the validation data set. We can confirm that the generator produces more complex structures over time by checking the generated wavefields during training of trial 3 in Fig. 4.17. We use trial 3, as the training RMSE is smoother and has more clear phases in its development, as described above. We rely on gray-scale images as theses images cannot be post-processed or redone as we did with the other images. Additionally, the generated images are shuffled as they are from the training data set (see Chapter 3.2.3). The generated images of the epochs clearly follow the training RMSE. Starting at epoch 15, the generator produces the correct background value. At epoch 87, we first see early-time wavefields produced by the generator. The quality of the generated wavefields improves quickly through the next following epochs, with later-time wavefields at epoch 99 and visible differences between wavefields stemming from different velocity distributions at epoch 146. At epoch 296, the real and generated wavefields overlap in most cases and no visual improvements can be seen by going to one of the last epochs 355.

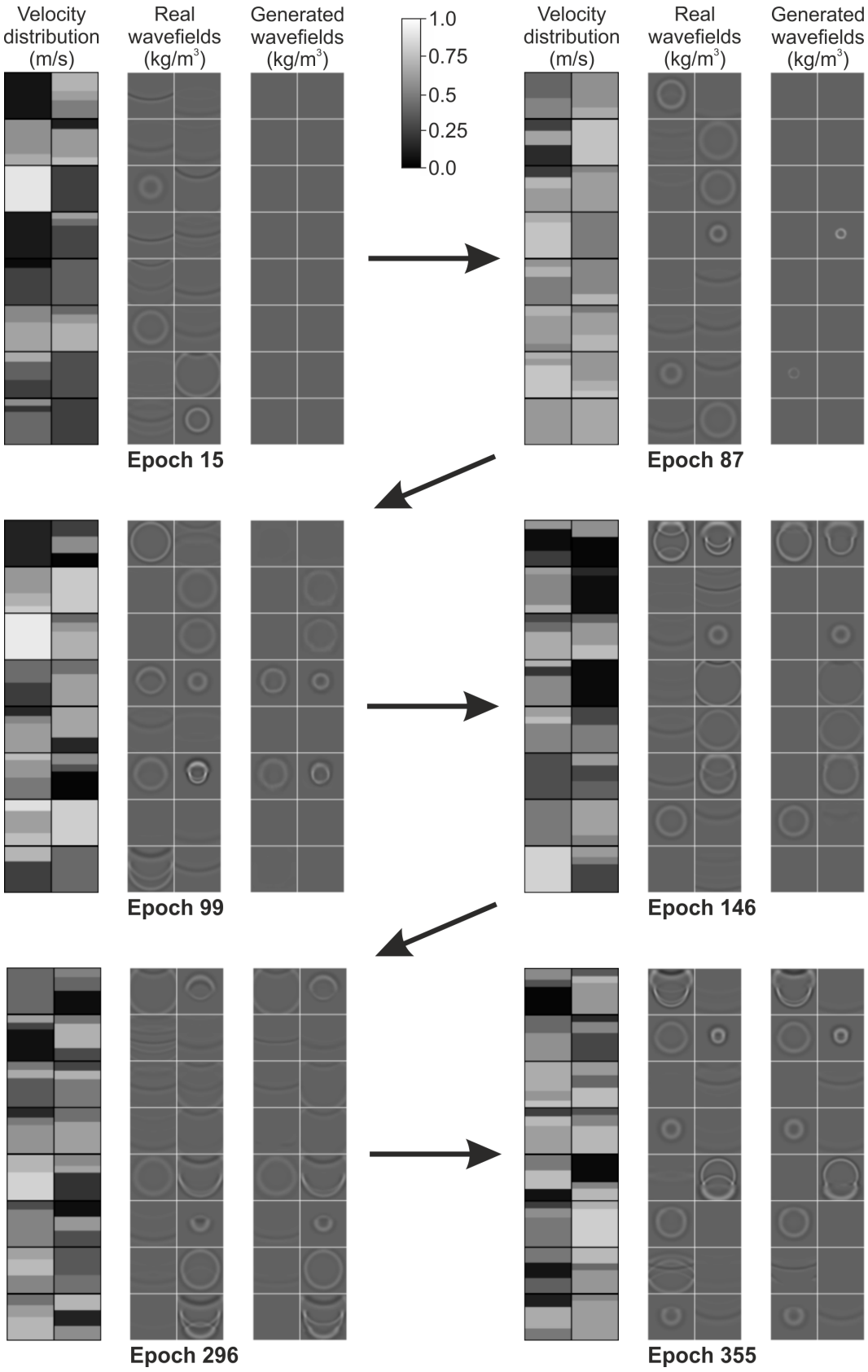

Figure 4.17: Generated pressure wavefields on the training data set over the epochs.

Moreover, in Fig. 4.18 for trial 0 and Fig. 4.19 for trial 3, we observe that the network generates more complex wavefields on the validation data set at the end of training (only selected wavefields are shown, for all time steps see Fig. A.4 for trial 0 and Fig. A.5 for trial 3). If we use the generator from the end of training on the validation dataset, we see that it starts to generate wavefields for later time steps. In both cases though, the generator produces the same wavefields independent of the underlying velocity distribution. This is in contrast to the wavefields generated on the training data set.

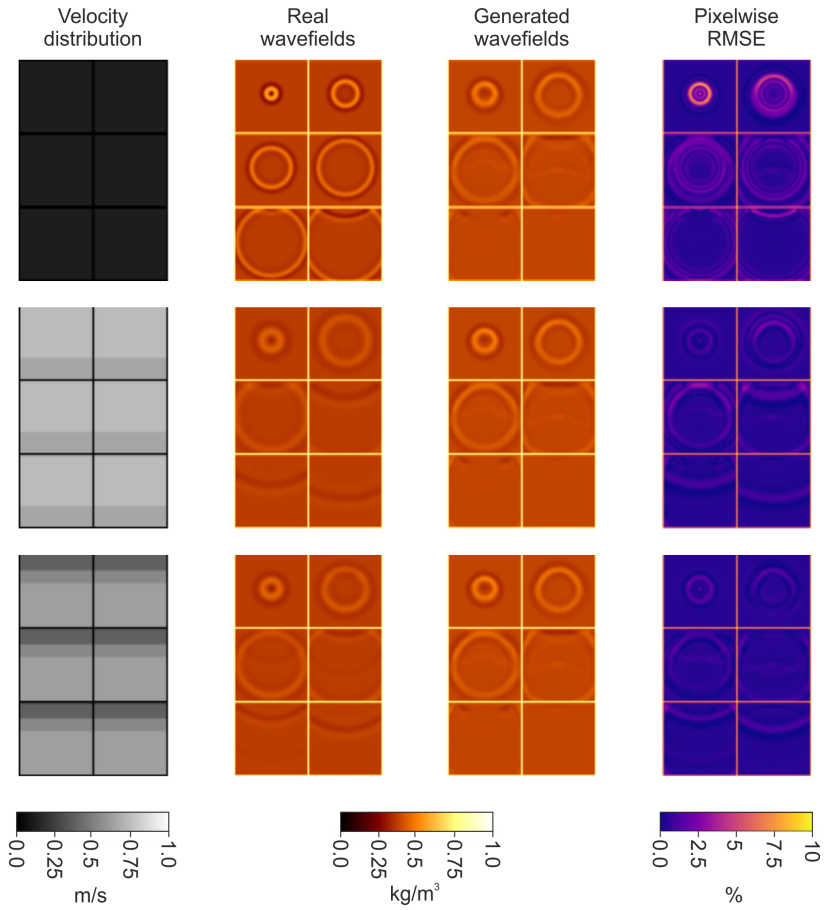

Figure 4.18: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from end of trial 0. Wavefields are shown in a zig-zag pattern from top left to bottom right for time steps [0.18, 0.25, 0.32, 0.39, 0.46, 0.53] seconds.

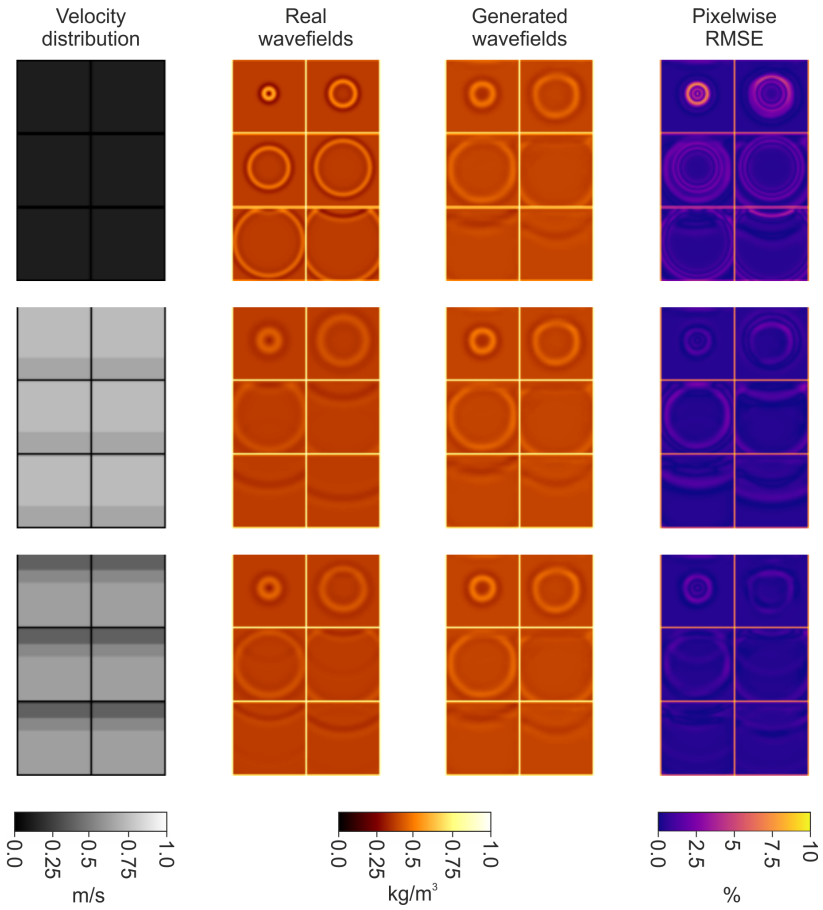

Figure 4.19: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from end of trial 3. Wavefields are shown in a zig-zag pattern from top left to bottom right for time steps [0.18, 0.25, 0.32, 0.39, 0.46, 0.53] seconds.

Finally, we use the test data set to check the network's performance on time steps not present in the training set. We choose time steps that are shifted by 0.03 s compared to the standard ones, so [0.21, 0.28, 0.35, 0.42, 0.49, 0.56, 0.63, 0.70, 0.77, 0.84, 0.91, 0.98] seconds. We rely on the saved model at the end of training as we saw that the longer the training takes, the more wavefield features are generated. We concentrate our examination on the network from trail 3, as the average validation RMSE at the end of training is better than trial 0's (see Fig. 4.14). In Fig. 4.20, we observe that the wavefields generated are again the same, independent of the velocity distribution that was given to the generator (for all timesteps, see Fig. A.6). Additionally, they are the same as the wavefields that were produced using the validation data set on time steps that were used during training. The generator simply generates the wavefields of the time steps from training instead of interpolating between them. We verify this in Fig. 4.21, where we see that the generator outputs the wavefields from the time steps present in training that are lower than the one given, except of the closest one to the next later one. An example would be [0.19, 0.20, 0.21, 0.22, 0.23] seconds all being interpreted as 0.18 s from training and 0.24 s as 0.25 s from training.

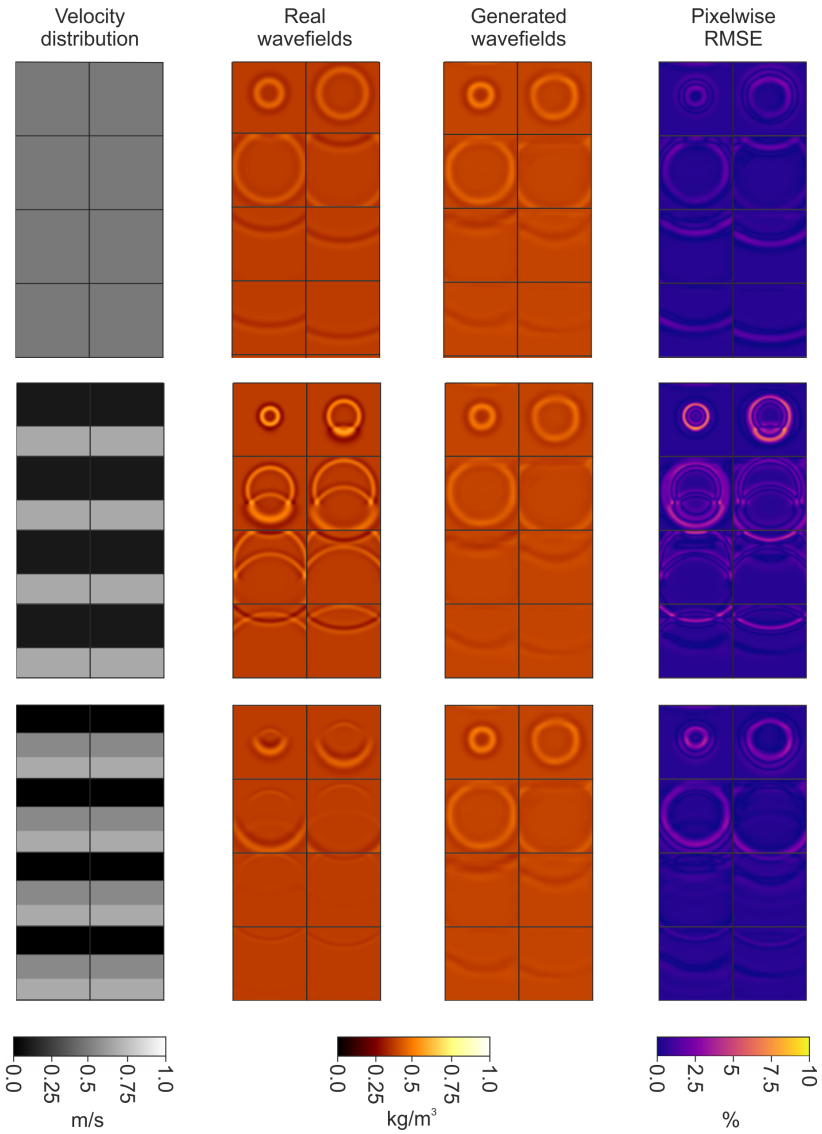

Figure 4.20: Generated pressure wavefields and pixel-wise RMSE on the test data set using generator from end of trial 3. Wavefields are shown in a zig-zag pattern from top left to bottom right on time steps not present in training ([0.21, 0.28, 0.35, 0.42, 0.49, 0.56, 0.63, 0.70] seconds).

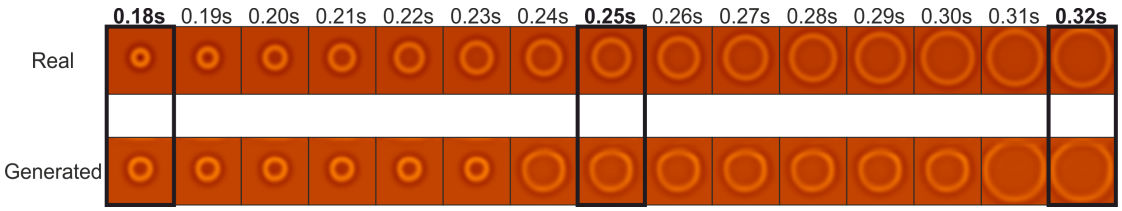

Figure 4.21: Generated pressure wavefields on the test data set using generator from end of trial 3. Boxes indicate time steps that are used during training.

In conclusion, we find that the generators from the epochs with the lowest validation RMSE of both trials produce wavefields at early time steps that only overlap partially with the ground truth. Additionally, on the validation and test data set, the same wavefields are always produced, irrespective of the underlying velocity distribution.

4.2.3 Computational time comparison

We briefly compare the time it takes to generate a wavefield using the generator vs. using SpecFem2D. The parameters for the simulation using SpecFem2D are exactly the same as described in Chapter 3.1. We do five simulations per velocity distribution-type (uniform, two-layer, and three-layer). For one second of simulated time, 1000 wavefields are simulated. We average the computational time and conclude that calculating a single pressure wavefield takes $\approx 0.015\,\mathrm{s}$ on average, using a serial calculation on an Intel Core i7-3820 @ 3.60 GHz. For the generator, we average the time over all samples in the test dataset, which are 1080 generated pressure wavefields. On average, it took 0.028 s on the Geforce RTX 4080 to generate a single pressure wavefield. This is about twice the amount of time that the numerical simulation needs. The main advantage of using the generator though is that it can produce wavefields without the need to calculate wavefields at earlier time steps. Imagine we are interested in the wavefield at $t = 0.5\,\mathrm{s}$. For the numerical simulation to calculate that, it needs to calculate 499 previous wavefields when using a time step of 0.01 s. The overall time to get the result is then $500 \cdot 0.015\,\mathrm{s} = 7.5\,\mathrm{s}$, now making it a much slower option than using the generator.

5. Discussion

The primary objective of our research is to explore the possibility of efficiently generating pressure wavefields at arbitrary time steps for varying velocity distributions. In the following sections, we will delve into the performance of our model in this regard, compare our results to those obtained in Kadeethum, et al. [1] upon which we based our approach, and discuss the limitations and potential solutions to our method.

5.1 CcGAN performance

Our physics-informed CcGAN presented predominantly poor outcomes. The biggest issue is the generator's incapability to produce pressure wavefields that vary depending on the inputted velocity distribution. It resembles a common problem in training traditional GANs: mode collapse, where the generator ends up producing only a limited variety of samples, often almost identical [115]. This is usually seen on all data sets - including the training data set - whereas in our case the issue was only seen on the validation and test data set. The issue of model collapse in traditional GANs is usually explained as the generator simply memorizing a few training examples that fool the discriminator [116]. One of the supposed advantages of using WGANs instead of traditional GANs is that they prevent the issue of mode collapse [99], though Lala, et al. [117] show that mode collapse can still happen in certain cases. Additionally, conditioning GANs is also known to help mitigate mode collapse, as conditioned with labels should lead to generating samples with more variety in modes [72]. However, Shahbazi, et al. [118] show that in the case of a limited data set, conditioning can lead to mode collapse, even in cases where unconditioned GANs performed well. We made sure that the issue was not due to improper saving and loading the generator's state when redoing the plots shown in Chapter 4.2, as the outputs during the learning phase showed similar results.

Moreover, contrary to our initial assumption, the non-physics-informed model consistently outperformed the physics-informed model for the uniform velocity distribution. Additionally, we saw that high λ_{ℓ_1} were preferred in the physics-informed trials. This pattern underlines the critical role that the L1 loss plays, even when physical consistency-based losses are present. This is even more obvious in the non-physics-informed approach, where the L1 loss is solely responsible for reducing the PDE loss to a lower level than in the physics-informed approach. This result suggests that adding the physics consistency-based losses might interfere with the generator's cost function, making finding an optimal solution more challenging, as optimization need to be taking into account the different parts of the cost function. It would be interesting to see how the network behaves if λ_{ℓ_1} was to be set to zero, so that only the physics-consistency based losses are added to the traditional WGAN loss.

The performance of a deep learning approach can be highly dependent on the choice of hyperparameters[119]. Using Optuna for our hyperparameter search, we were able to

try many different configurations and showed that this is also the case for our CcGAN. Optuna requires an initial range of hyperparamters in which it is supposed to find an optimal set. These ranges are mostly based around values from Kadeethum, et al. [93]. Particular attention was given to the weights λ of the cost functions. The gradient penalty's λ_{gp} value was chosen in the range of 5 to 15, based on the recommendation of Arjovsky, et al. [99], who found a value of 10 to be effective in their experiments. Kadeethum, et al. [93] used an L1-loss weight of $\lambda_{\ell_1} = 250$. As we additionally incorporate the physical consistency-losses, we let Optuna search in a range from 5 to 250 to see if the reliance on the L1 loss reduces. For λ_{PDE}, we chose values ranging from 50 to 400 to allow the network to give substantial importance to the physics-consistency of the model. The λ_{b} values chosen were much smaller, ranging from 0 to 15. This decision was driven by the observation that the magnitude of the boundary loss values was in the tens, significantly higher than the L1 and PDE loss values, which are in the range of 0.010s and 0.10s/0.010s, respectively. Interestingly, a coding error unexpectedly provided insightful results regarding λ_{b}. Originally, the lower limit of it was intended to be 0.2, but due to asking Optuna to only pick integer values for λ_{b}, it was interpreted as 0. We saw that the boundary loss had a very large impact on the generator's learning curve in one of the trials chosen for the layered velocity distribution. This level of influence is possibly undesirable, as evidenced by the best trial for the uniform velocity distribution preferring a λ_{b} of 0.

The best validation RMSE was often found quite early in the trials: specifically trials 51 and 26 out of 100 for the uniform velocity distributions and 0 and 3 out of 14 for the layered velocity distribution. It is important to note that Optuna performs an initial random search of ten trials, which aids in familiarizing it with the cost function landscape. Therefore, we would expect Optuna to find better hyperparameters after this initial phase and even more so the longer the hyperparameter search continues. However, Optuna struggled with conducting an effective hyperparameter search for a two possible reasons. First, not enough trials were done for the layered velocity distribution data set. Second, that the lowest validation RMSE was early in the learning process where almost no wavefields were produced. Consequently, the metric rarely corresponded to a genuinely good performance, so Optuna tried to optimize for an unwanted behavior.

One beneficial aspect of our work is that when specific time steps are provided, the generator does show the progression of the actual wavefield. This means we can request a wavefield at a certain time step without needing to calculate the preceding time steps. Unfortunately, this only worked for time steps that were also used in training. This means that we did not achieve continuous time stepping. Nevertheless, this feature theoretically reduces the computational effort compared to classical numerical simulations as it eliminates the need to generate all preceding wavefields. However, this advantage is only theoretical in nature due to the significant discrepancies between the generated output and the ground truth, rendering our current model an inadequate option for practical applications.

5.2 Comparison to other work

Our approach draws heavily on the work of Kadeethum, et al. [1], yet their success did not transfer to our research, despite the similarity in having an underlying 2D time-dependent PDE. First, we compare data-set sizes. Our layered data set contains $10,800$ wavefield samples, which is larger than the data sets used in Kadeethum, et al. [1]. They experimented with multiple sizes, ranging from $1,250$ to $10,000$ samples, observing promising results across the board, with the results improving with larger data sets. This suggest that our model's performance issues are unlikely to be related to the size of the used data set.

Even though both experiments are interested in generating pressure distributions - pressure wavefields in our case - the feature and frequency content of the inputs and outputs are noticeably different. The inputs from Kadeethum, et al. [1] feature more rich and varied samples, rather than simply uniform or layered. This raises the possibility that our velocity distributions might not encode enough features for the network to distinguish between uniform, two-layer, and three-layer velocity distributions. However, this assumption is partially invalidated by the variety of wavefields generated for different inputs during the training phase.

In terms of outputs, our generated wavefields essentially form high amplitude, sharp structures against a uniform background. This is in contrast to the smooth, low frequency output of Kadeethum, et al. [1]. Our generator can easily identify and draw the correct uniform background value, but takes many epochs before starting to generate structures that resemble wavefields. It is possible that generating low-frequency, smooth outputs is an easier task, especially when using the L1 loss. As Isola, et al. [74] have pointed out, this loss is responsible of enforcing low-frequency content, while the generator-discriminator interplay tends to be responsible for the high-frequency content. Contrary to this assumption, our network showed the capability to generate varied, high-frequency wavefields, especially when overfitting on the training dataset. in conclusion, the exact cause of the poor performance of our experiment relative to that of Kadeethum, et al. [1] remains unclear.

5.3 Limitations and potential improvements

Our current research is characterized by several limitations, a number of which have been intentionally imposed in order to simplify the problem at hand. In this section, we will discuss the main limitations and potential improvements.

One of the key disadvantages of our approach compared to other physics-informed neural networks is its reliance on a grid. This is because we worked with images, which inherently represent data in a grid-like format. Additionally we need the grid's step size for the computation of the physics-consistency score. As GANs always work on images, this issue cannot be overcome. Moreover, the model is currently limited to a single domain size and a 128×128 grid. While convolutional neural networks can handle images of varying sizes, our use of fully-connected layers at certain points currently prohibits us

from training on different domain/grid sizes. Future research could work on overcoming this limitation to achieve more generalized applicability.

We have based our analysis on the simplified case of a 2D acoustic wave equation. This means we do not take into account other wave types like S-waves. It might be feasible to ask a modified generator to output two wavefields, one each for S-waves and P-waves. It would be interesting to see how or if the transformation between wave types at boundaries can be captured. Furthermore, incorporating these waves would necessitate the use of a more complex wave equation for the physics-consistency loss.

Another aspect of our simplification is the 2D nature of our model. Real-world wave propagation is in a 3D subsurface, so it would make sense to try to capture that behavior in the model as well. If we conceptualize 3D images as stacked 2D gray-scale images, we could theoretically feed in a 3D velocity distribution and output 3D wavefield images using a GAN. CNNs lend themselves to such a problem, as they are inherently able to work on multi-channel images. An exemplary work using GANs to generate 3D images is by Wu, et al. [120]. In our context, using 3D images would require some changes to the network, especially when it comes to the time-inputting scheme. Due to its complex nature, we assume that this will be quite difficult. Additionally, the increase in data size from 3D images, the accompanying need for more GPU memory, and subsequent increase in training time imposes a practicality limit when using our current hardware.

We use only a single source location and type for all the ground truth wavefields. Therefore, the network was only able to produce wavefields that are generated by that specific source. As Kadeethum, et al. [93] point out, the framework of CcGANs allows for multiple (continuous) inputs to the network, not just the time steps. We could therefore explore the potential of including variables such as source location and type as additional inputs. While source location can be encoded using coordinate values, more possibilities exist for the type of source. We could focus solely on different source frequencies as single values, or add different source types as a one-hot encoded vector. This addition of the source information might necessitate a modification of the physics-consistency loss to incorporate the source into the PDE again. A further note to the input mechanism for the condition made by Kadeethum, et al. [93] is that the CcGAN framework is able to handle data stemming from simulations with non-constant time-stepping.

We have only used a single network structure without verifying if others might yield better results. One change that would be interesting to explore is using batch normalization layers in the critic, as done by Kadeethum, et al. [1], who did so even though Gulrajani, et al. [98] specifically said that these are inappropriate in the context of WGAN-gp's. Additionally, we only used a physics-informed generator. Daw, et al. [121] used a physics-informed discriminator instead, pointing out that this will influence the generator physics-consistency as well due to the feedback the discriminator gives to the generator. This method is more difficult to implement, which is why we chose a physics-informed generator only, but a physics-informed discriminator is a natural extension. A possible change related to the network's cost functions is Wang, et al. [122]'s dynamic weight updating process to find the optimal weighting parameters λ and

dynamically adapting them during training. In addition, this could potentially help with the hyperparameter tuning since not taking these parameters into account reduces the search space.

Another limitation is that we were bound by the accuracy of the SpecFem2D simulation used to generate the ground truth data. While physics-informed neural networks have demonstrated potential to improve the accuracy of coarse simulation outputs or noisy measurements [32], [123], it would be worth exploring how a successful model would handle this aspect. Additionally, our physics-loss based on finite differences has inherent errors, which might affect the physics-loss and consequently the accuracy of the generated wavefields. This is especially true, because we took a step size of $dt = 0.1\,\mathrm{s}$ for the finite-difference approximation of the second partial derivative. This is 10 times larger than the step size that SpecFem2D uses for its simulation. There was a two-fold reasoning in taking such a large step size. First, saving every wavefield from Optuna in an $dt = 0.01\,\mathrm{s}$ interval would have taken too much time storage when generating wavefields for the 900 velocity distributions. Second, we already had to employ some memory saving techniques when processing the data set of simulated wavefields. This would have become even more challenging to do on a more extensive data set. Eventually, we decided to write only every tenth simulated wavefield to file after visual inspection of PDE losses on real wavefields with different dt's.

Finally, in this thesis we only considered simple uniform or horizontally layered velocity distributions. This decision was made to check if the network can perform on these simple velocity distributions to warrant further research towards more realistic subsurface velocity distributions. In reality, subsurface velocity distributions can contain heterogeneous distributions with complex structures. As such, it may be beneficial for future work to consider irregular velocity distributions to enhance the applicability of the model. To introduce more variability into the data sets, synthetic models with varying geological features, such as faults, inclusions, and salt domes, could be created.

6. Conclusion

The primary motivation that we outlined for solving the wave equation using a physics-informed deep learning approach was to accelerate seismic imaging by substituting the traditional numerical solver in FWI with a NN. Due to the issues observed in our approach, we think that it might be advantageous to focus on direct inversion using deep learning. Although in many cases it is extremely dependent on large data sets, the fact that only a single output - a subsurface image - needs to be produced per input, i.e., measurement data, reduces the complexity of the issues compared to the time-dependent nature of the wave equation. In this context, Wang, et al. [34]'s approach of the two networks in a cyclic interplay from Chapter 1 seems to be very promising, especially as it does not rely on training data. Nonetheless, further research on using physics-informed deep learning as a solver for the wave equation should still continue due to the importance of simulating wave propagation in many different disciplines, e.g., for simulating seismic waves induced by earthquakes.

In this regard, we again reviewed some promising alternatives to our CcGAN due to its performance issues. An important trait of our problem is its time-dependent nature, so network structures that are designed to handle these kinds of issues, such as LSTM-RNNs, seem to be well-suited. However, as indicated by Rodriguez-Torrado, et al. [124], this approach is currently limited to one-dimensional problems. This is due to the increase in memory load for higher-dimensional problems, as many or all preceding results need to be accessed to calculate the results at the next time step. It is again important to note that our CcGAN does not suffer from this issue, as it treats wavefields independently.

Another option to address the complexity introduced by the time dependency, as well as the intricacy of the reflections and the sharp contrast between a uniform background and the actual wave, is solving the wave equation in the frequency domain. This is commonly done by employing the Helmholtz equation [125]. Instead of relying on physics-informed NNs to solve this equation, Fourier Neural Operators (FNOs) have very recently emerged as a promising alternative [126]. They are a type of NN designed to approximate the solution operator of a PDE, which under certain assumptions can be understood as a convolution kernel. Leveraging the convolution theorem that states that convolution is equivalent to multiplication in the Fourier space, convolutional layers are replaced with Fourier layers [127]. The advantage of this approach is that it can learn a family of PDEs, is a mesh-free method, and has demonstrated the ability to solve the wave equation for complex velocity distributions and to generalize to similar ones [126].

In conclusion, although our current implementation of a physics-informed CcGAN for wavefield generation presents multiple limitations and unsatisfactory performance, it lays the groundwork for future research. We have identified several potential enhancements and research directions and are confident that it is worthwhile to pursue them to achieve a fast, universal wave equation solver using physics-informed deep learning to advance wave-based seismic imaging.

Appendix

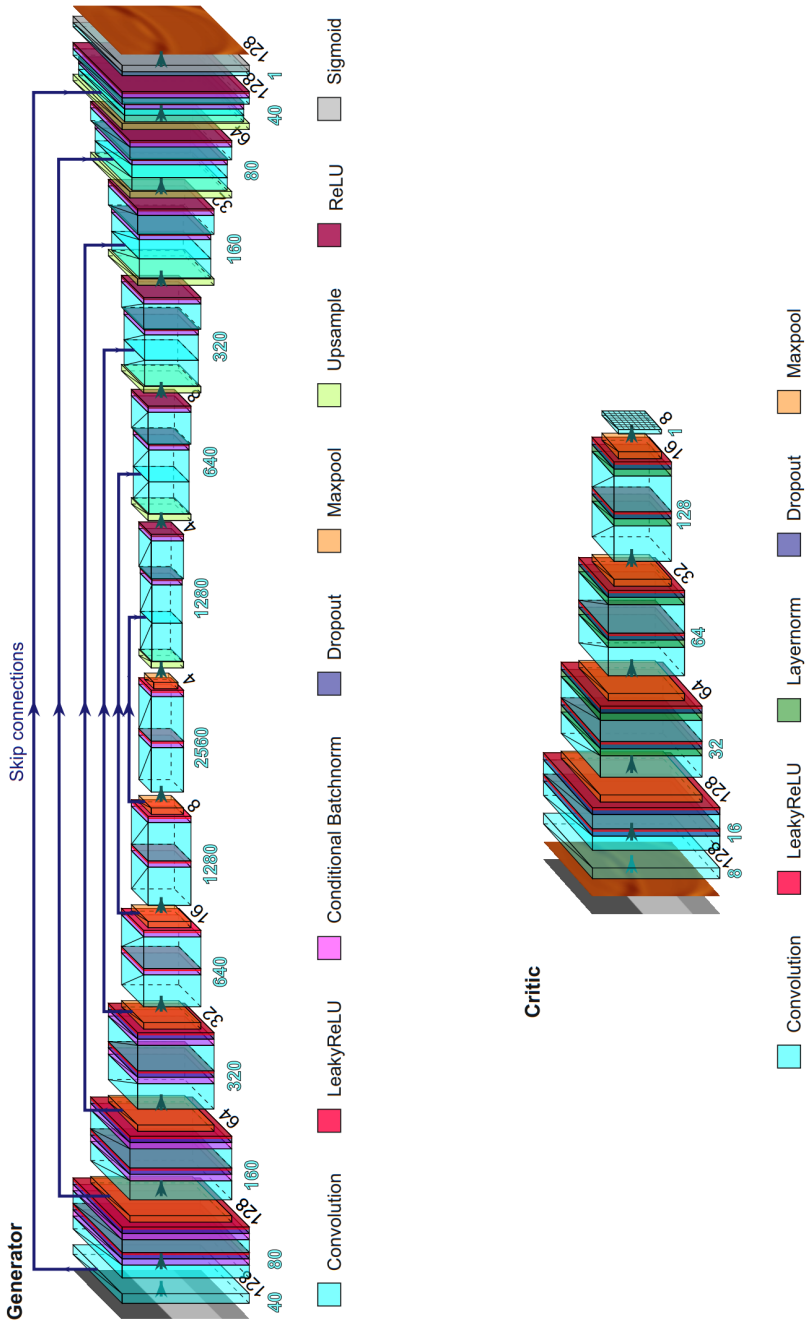

Figure A.1: In-depth look at generator and critic architecture, visualized using PlotNeuralNet. Blue numbers indicate the channels after a convolution, the black numbers the height and width of the image before being passed to the maxpooling or upsampling layer. The amount of channels in the generator is exemplary.

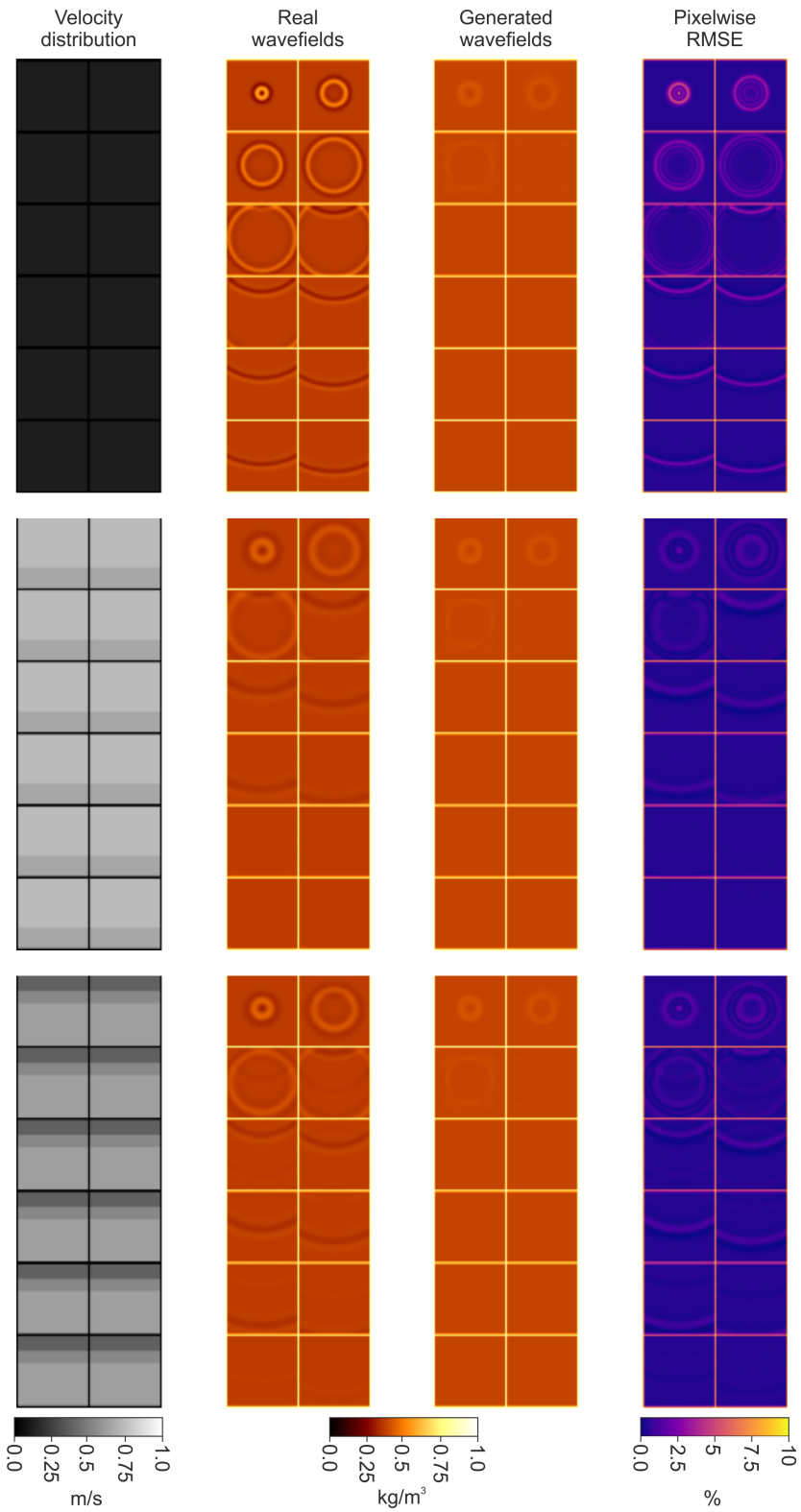

Figure A.2: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from epoch 106 (with lowest validation RMSE) of trial 0. Wavefields are shown in a zig-zag pattern from top left to bottom right for timesteps in training.

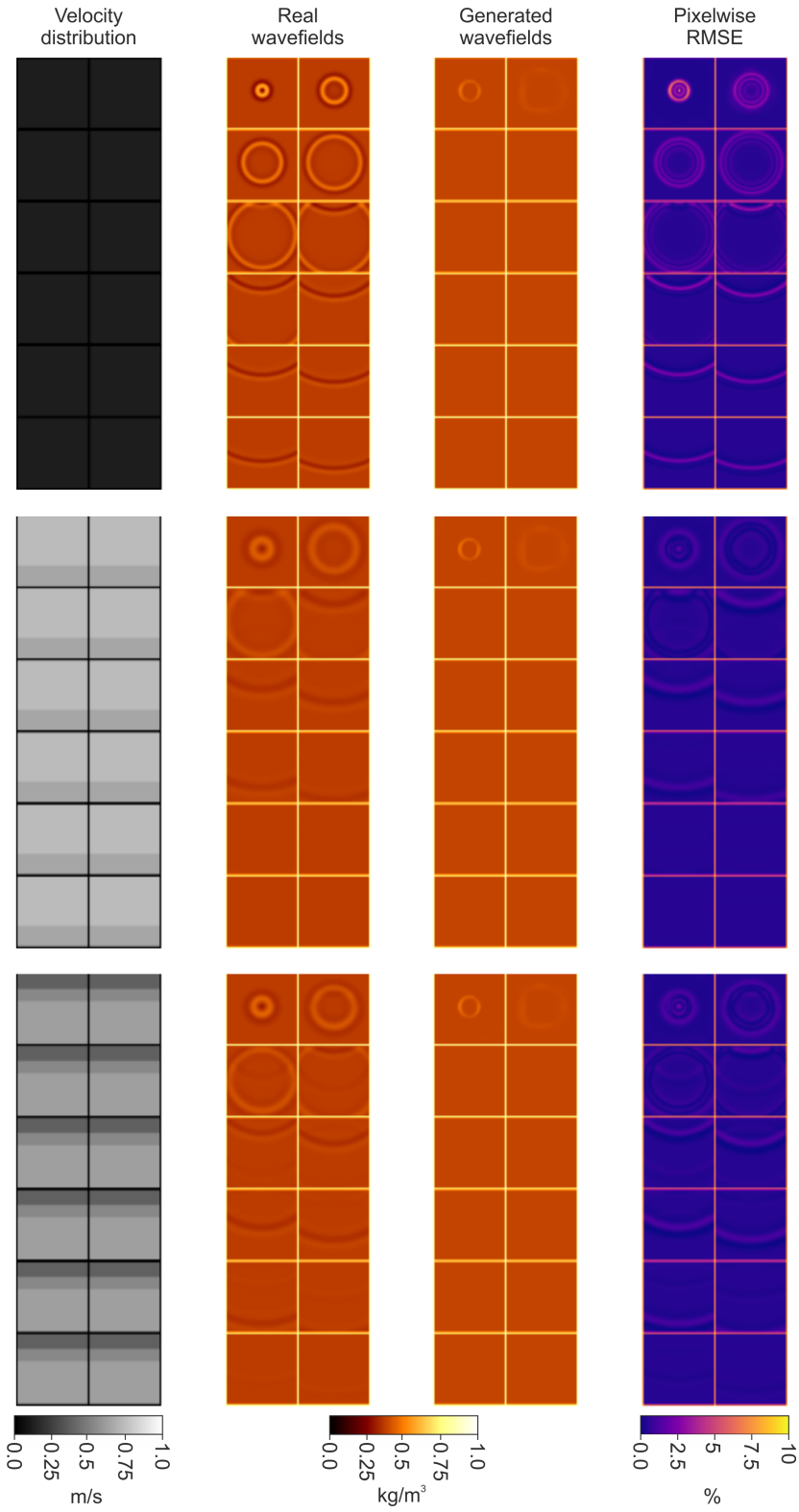

Figure A.3: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from epoch 90 (with lowest validation RMSE) of trial 3. Wavefields are shown in a zig-zag pattern from top left to bottom right for timesteps in training.

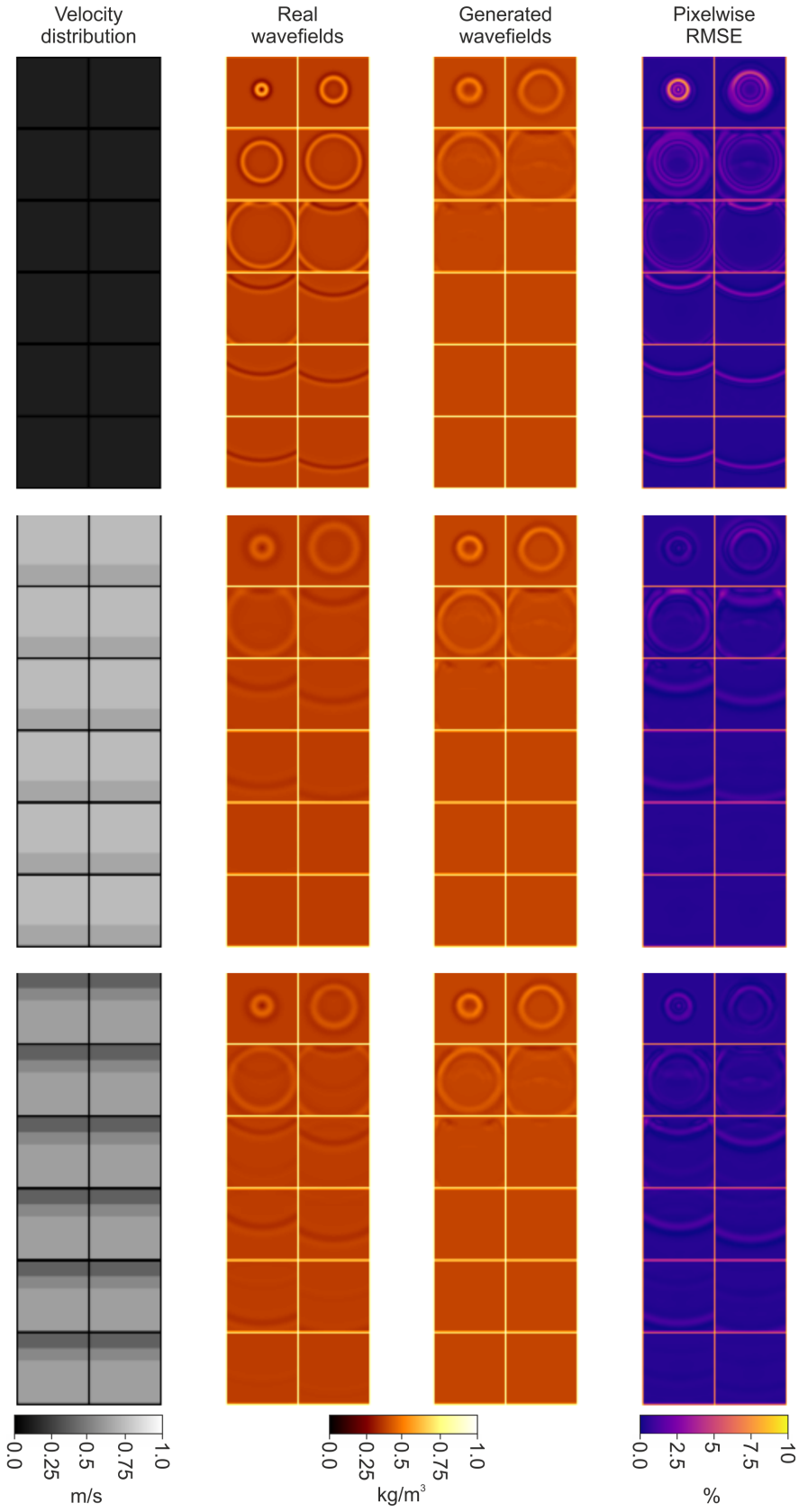

Figure A.4: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from end of trial 0. Wavefields are shown in a zig-zag pattern from top left to bottom right for timesteps in training.

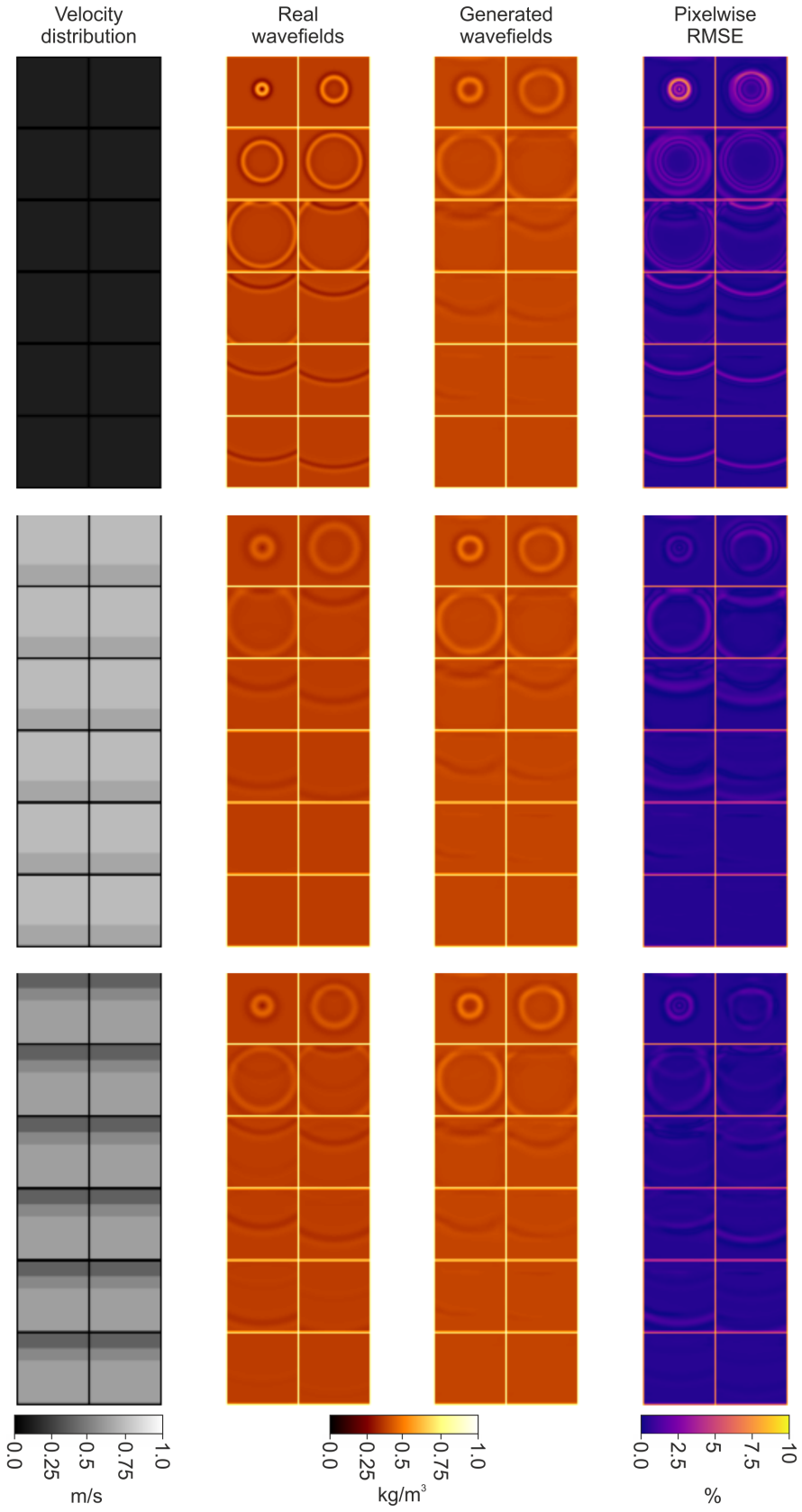

Figure A.5: Generated pressure wavefields and pixel-wise RMSE on the validation data set using generator from end of trial 3. Wavefields are shown in a zig-zag pattern from top left to bottom right for timesteps in training.

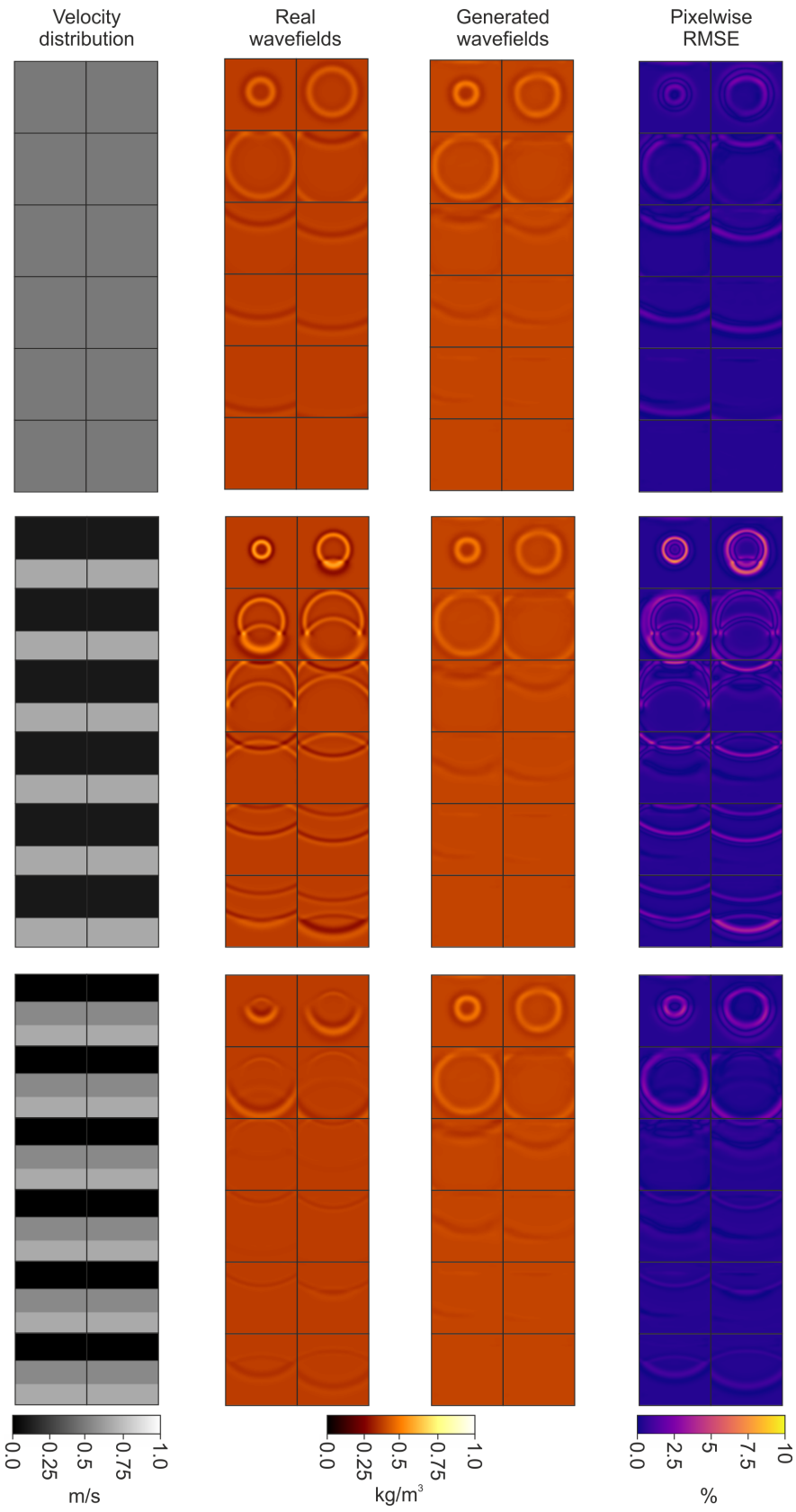

Figure A.6: Generated pressure wavefields and pixel-wise RMSE on the test data set using generator from end of trial 3. Wavefields are shown in a zig-zag pattern from top left to bottom right on time steps not present in training ([0.21,0.28, 0.35, 0.42, 0.49, 0.56, 0.63, 0.70, 0.77, 0.84, 0.91, 0.98] seconds)

Bibliography

[1] T. Kadeethum, D. O'Malley, Y. Choi, H. S. Viswanathan, N. Bouklas, and H. Yoon, "Continuous Conditional Generative Adversarial Networks for Data-Driven Solutions of Poroelasticity with Heterogeneous Material Properties," *Computers & Geosciences*, vol. 167, p. 105 212, Oct. 2022.

[2] D. Fleisch and L. Kinnaman, *A Student's Guide to Waves* (Student's Guides). Cambridge: Cambridge University Press, 2015.

[3] Science Mission Directorate. National Aeronautics and Space Administration, *Anatomy of an Electromagnetic Wave*, 2010. [Online]. Available: `https://science.nasa.gov/ems/02%5C_anatomy` (visited on 06/25/2023).

[4] S. Kaur, P. Singh, V. Tripathi, and R. Kaur, "Recent Trends in Wireless and Optical Fiber Communication," *International Conference on Intelligent Engineering Approach(ICIEA-2022)*, vol. 3, no. 1, pp. 343–348, Jun. 2022.

[5] M. S. Zahrani, "Telecommunications Network using Electromagnetic Waves," *Information Technology Journal*, vol. 9, no. 3, pp. 430–437, 2010.

[6] A. P. Sarvazyan, M. W. Urban, and J. F. Greenleaf, "Acoustic Waves in Medical Imaging and Diagnostics," *Ultrasound in Medicine & Biology*, vol. 39, no. 7, pp. 1133–1146, Jul. 2013.

[7] "Wave–Particle Duality: De Broglie, Einstein, and Schrödinger," in *Critical Appraisal of Physical Science as a Human Enterprise: Dynamics of Scientific Progress*, M. Niaz, Ed., Dordrecht: Springer Netherlands, 2009, pp. 159–165.

[8] LIGO Scientific Collaboration and Virgo Collaboration, B. P. Abbott, R. Abbott, *et al.*, "Observation of Gravitational Waves from a Binary Black Hole Merger," *Physical Review Letters*, vol. 116, no. 6, p. 061 102, Feb. 2016.

[9] G. T. Schuster, "Seismic Imaging, Overview," in *Encyclopedia of Solid Earth Geophysics*, ser. Encyclopedia of Earth Sciences Series, H. K. Gupta, Ed., Dordrecht: Springer Netherlands, 2011, pp. 1121–1134.

[10] Ö. Yilmaz, *Seismic Data Analysis: Processing, Inversion, and Interpretation of Seismic Data*. Society of Exploration Geophysicists, Jan. 2001.

[11] P. M. Shearer, *Introduction to Seismology*, Second. Cambridge: Cambridge University Press, 2009.

[12] Y. Cui, K. B. Olsen, T. H. Jordan, *et al.*, "Scalable Earthquake Simulation on Petascale Supercomputers," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 13, pp. 1–20.

[13] A. Tarantola, "Inversion of Seismic Reflection Data in the Acoustic Approximation," *Geophysics*, vol. 49, no. 8, pp. 1259–1266, Aug. 1984.

[14] H. Igel, *Computational Seismology: A Practical Introduction*. Oxford University Press, Nov. 2016. (visited on 06/25/2023).

[15] K. Leng, T. Nissen-Meyer, M. van Driel, K. Hosseini, and D. Al-Attar, "AxiSEM3D: Broad-Band Seismic Wavefields in 3-D Global Earth Models with Undulating Discontinuities," *Geophysical Journal International*, vol. 217, no. 3, pp. 2125–2146, Jun. 2019.

[16] L. Zhu, W. Zhang, J. Kou, and Y. Liu, "Machine Learning Methods for Turbulence Modeling in Subsonic Flows around Airfoils," *Physics of Fluids*, vol. 31, no. 1, p. 015 105, Jan. 2019. (visited on 06/25/2023).

[17] C. Li, P. Yuan, Y. Liu, *et al.*, "Fast Flow Field Prediction of Hydrofoils Based on Deep Learning," *Ocean Engineering*, vol. 281, p. 114 743, Aug. 2023.

[18] J. Schmidt, M. R. G. Marques, S. Botti, and M. A. L. Marques, "Recent Advances and Applications of Machine Learning in Solid-State Materials Science," *npj Computational Materials*, vol. 5, no. 1, p. 83, Aug. 2019.

[19] A. W. Senior, R. Evans, J. Jumper, *et al.*, "Improved Protein Structure Prediction Using Potentials from Deep Learning," *Nature*, vol. 577, no. 7792, pp. 706–710, Jan. 2020.

[20] T. Perol, M. Gharbi, and M. Denolle, "Convolutional Neural Network for Earthquake Detection and Location," *Science Advances*, vol. 4, no. 2, e1700578,

[21] L. Lehmann, M. Ohrnberger, M. Metz, and S. Heimann, "Accelerating Low-Frequency Ground Motion Simulation for Finite Fault Sources Using Neural Networks," *Geophysical Journal International*, ggad239, Jun. 2023.

[22] M. Malfante, M. Dalla Mura, J. I. Mars, J.-P. Métaxian, O. Macedo, and A. Inza, "Automatic Classification of Volcano Seismic Signatures," *Journal of Geophysical Research: Solid Earth*, vol. 123, no. 12, pp. 10, 645–10, 658, Dec. 2018.

[23] M. Azarafza, M. Azarafza, H. Akgün, P. M. Atkinson, and R. Derakhshani, "Deep Learning-Based Landslide Susceptibility Mapping," *Scientific Reports*, vol. 11, no. 1, p. 24 112, Dec. 2021.

[24] Z. Zhang and Y. Lin, *Data-driven Seismic Waveform Inversion: A Study on the Robustness and Generalization*, Jun. 2019. eprint: `1809.10262` (eess). (visited on 06/25/2023).

[25] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations*, Nov. 2017. arXiv: `1711.10561 [cs, math, stat]`.

[26] J. Hermann, Z. Schätzle, and F. Noé, "Deep-Neural-Network Solution of the Electronic Schrödinger Equation," *Nature Chemistry*, vol. 12, no. 10, pp. 891–897, Oct. 2020.

[27] K. Kashinath, M. Mustafa, A. Albert, *et al.*, "Physics-Informed Machine Learning: Case Studies for Weather and Climate Modelling," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, no. 2194, p. 20 200 093, Feb. 2021.

[28] P. Borate, J. Rivière, C. Marone, A. Mali, D. Kifer, and P. Shokouhi, "Using a Physics-Informed Neural Network and Fault Zone Acoustic Monitoring to Predict Lab Earthquakes," *Nature Communications*, vol. 14, no. 1, p. 3693, Jun. 2023.

[29] T. Okazaki, T. Ito, K. Hirahara, and N. Ueda, "Physics-Informed Deep Learning Approach for Modeling Crustal Deformation," *Nature Communications*, vol. 13, no. 1, p. 7092, Nov. 2022.

[30] S. Karimpouli and P. Tahmasebi, "Physics Informed Machine Learning: Seismic Wave Equation," *Geoscience Frontiers*, vol. 11, no. 6, pp. 1993–2001, Nov. 2020.

[31] B. Moseley, A. Markham, and T. Nissen-Meyer, *Solving the Wave Equation with Physics-Informed Deep Learning*, Jun. 2020. arXiv: `2006.11894 [physics]`.

[32] M. Rasht-Behesht, C. Huber, K. Shukla, and G. E. Karniadakis, "Physics-Informed Neural Networks (PINNs) for Wave Propagation and Full Waveform Inversions," *Journal of Geophysical Research: Solid Earth*, vol. 127, no. 5, e2021JB023120, 2022.

[33] C. Song and T. A. Alkhalifah, "Wavefield Reconstruction Inversion via Physics-Informed Neural Networks," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–12, 2022.

[34] Z. Wang, S. Wang, C. Zhou, and W. Cheng, "Dual Wasserstein Generative Adversarial Network Condition: A Generative Adversarial Network-Based Acoustic Impedance Inversion Method," *GEOPHYSICS*, R401–R411, Nov. 2022.

[35] Y. Ren, X. Xu, S. Yang, L. Nie, and Y. Chen, "A Physics-Based Neural-Network Way to Perform Seismic Full Waveform Inversion," *IEEE Access*, vol. 8, pp. 112 266–112 277, 2020.

[36] A. Adler, M. Araya-Polo, and T. Poggio, "Deep Learning for Seismic Inverse Problems: Toward the Acceleration of Geophysical Analysis Workflows," *IEEE Signal Processing Magazine*, vol. 38, no. 2, pp. 89–119, 2021.

[37] M. Båth, "Seismic Waves," in *Introduction to Seismology*, ser. Wissenschaft Und Kultur, Second, Basel: Birkhäuser Basel, 1979, pp. 61–103.

[38] K. Roy Chowdhury, "Seismic Data Acquisition and Processing," in *Encyclopedia of Solid Earth Geophysics*, ser. Encyclopedia of Earth Sciences Series, H. K. Gupta, Ed., Dordrecht: Springer Netherlands, 2011, pp. 1081–1097.

[39] A. Cannata, F. Cannavò, S. Moschella, S. Gresta, and L. Spina, "Exploring the Link between Microseism and Sea Ice in Antarctica by Using Machine Learning," *Scientific Reports*, vol. 9, no. 13050, Sep. 2019.

[40] P. Richards, *Mathematical Methods in the Earth Sciences, W4950*, Lecture Notes; Columbia University, NY, 2006. [Online]. Available: `https://www.ldeo.columbia.edu/~richards/webpage_rev_Jan06/Ch2_ElasticWaves_in_Solids.pdf` (visited on 05/10/2023).

[41] J. Billingham and A. C. King, *Wave Motion* (Cambridge Texts in Applied Mathematics). Cambridge: Cambridge University Press, 2001.

[42] Y. Lin, J. Theiler, and B. Wohlberg, "Physics-Guided Data-Driven Seismic Inversion: Recent Progress and Future Opportunities in Full-Waveform Inversion," *IEEE Signal Processing Magazine*, vol. 40, no. 1, pp. 115–133, Jan. 2023.

[43] H. Chauris, "Chapter 5: Full Waveform Inversion," in *Seismic Imaging: A Practical Approach*, EDP Sciences, Feb. 2021, pp. 123–146.

[44] R. Huang, Z. Zhang, Z. Wu, Z. Wei, J. Mei, and P. Wang, "Full-Waveform Inversion for Full-Wavefield Imaging: Decades in the Making," *The Leading Edge*, vol. 40, no. 5, pp. 324–334, May 2021.

[45] Y. Lin and L. Huang, "Acoustic- and Elastic-Waveform Inversion Using a Modified Total-Variation Regularization Scheme," *Geophysical Journal International*, vol. 200, no. 1, pp. 489–502, Jan. 2015.

[46] J. Virieux and S. Operto, "An Overview of Full-Waveform Inversion in Exploration Geophysics," *GEOPHYSICS*, vol. 74, no. 6, WCC1–WCC26, Nov. 2009.

[47] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[48] P. Wang, "On Defining Artificial Intelligence," *Journal of Artificial General Intelligence*, vol. 10, no. 2, pp. 1–37, 2019.

[49] J. McCarthy, *WHAT IS ARTIFICIAL INTELLIGENCE?* 2007. [Online]. Available: `http://www-formal.stanford.edu/jmc/whatisai/node1.html` (visited on 05/11/2023).

[50] C. M. Bishop, *Pattern Recognition and Machine Learning* (Information Science and Statistics). Springer New York, NY, Aug. 2006.

[51] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[52] R. Sathya and A. Abraham, "Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification," *International Journal of Advanced Research in Artificial Intelligence (IJARAI)*, vol. 2, no. 2, 2013.

[53] J. Delua, *Supervised vs. Unsupervised Learning: What's the Difference?* 2021. [Online]. Available: `https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning` (visited on 05/15/2023).

[54] E. Kavlakoglu, *AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?* May 2020. [Online]. Available: `https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks` (visited on 05/15/2023).

[55] K. Hornik, M. Stinchcombe, and H. White, "Multilayer Feedforward Networks Are Universal Approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989.

[56] M. van der Baan and C. Jutten, "Neural Networks in Geophysical Applications," *GEOPHYSICS*, vol. 65, no. 4, pp. 1032–1047, Jul. 2000.

[57] I. H. Sarker, "Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions," *SN Computer Science*, vol. 2, no. 6, p. 420, Aug. 2021.

[58] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark," *Neurocomputing*, vol. 503, pp. 92–108, Sep. 2022.

[59] B. Ramsundar and R. Zadeh, *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. O'Reilly Media, 2018.

[60] S. Ruder, *An Overview of Gradient Descent Optimization Algorithms*, Jun. 2017. arXiv: `1609.04747 [cs]`.

[61] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," English (US), *Journal of Machine Learning Research*, vol. 38, pp. 192–204, 2015, 18th International Conference on Artificial Intelligence and Statistics, AISTATS 2015 ; Conference date: 09-05-2015 Through 12-05-2015.

[62] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[63] J. Brownlee, *Difference Between a Batch and an Epoch in a Neural Network*, Aug. 2022. [Online]. Available: `https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/` (visited on 06/16/2023).

[64] C. Olah, *Calculus on Computational Graphs: Backpropagation*, Blog, Aug. 2015. [Online]. Available: `http://colah.github.io/posts/2015-08-Backprop/` (visited on 05/25/2023).

[65] R. C. Staudemeyer and E. R. Morris, *Understanding LSTM – a Tutorial into Long Short-Term Memory Recurrent Neural Networks*, Sep. 2019. arXiv: `1909.09586 [cs]`.

[66] J. Zhou, G. Cui, S. Hu, *et al.*, "Graph Neural Networks: A Review of Methods and Applications," *AI Open*, vol. 1, pp. 57–81, Jan. 2020.

[67] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A Gentle Introduction to Graph Neural Networks," *Distill*, vol. 6, no. 9, e33, Sep. 2021.

[68] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27, Curran Associates, Inc., 2014.

[69] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[70] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, "A Review on Generative Adversarial Networks: Algorithms, Theory, and Applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 3313–3332, 2023.

[71] I. Goodfellow, *NIPS 2016 Tutorial: Generative Adversarial Networks*, Apr. 2017. arXiv: 1701.00160 [cs].

[72] M. Mirza and S. Osindero, *Conditional Generative Adversarial Nets*, Nov. 2014. arXiv: 1411.1784 [cs, stat].

[73] G. Zhu, H. Zhao, H. Liu, and H. Sun, "A Novel LSTM-GAN Algorithm for Time Series Anomaly Detection," in *2019 Prognostics and System Health Management Conference (PHM-Qingdao)*, Oct. 2019, pp. 1–6.

[74] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, Nov. 2018. arXiv: 1611.07004 [cs].

[75] M. Nießner, *Introduction to Deep Learning (I2DL) (IN2346)*, Lecture Notes; Technical University of Munich, GER, 2023. [Online]. Available: https://niessner.github.io/I2DL/ (visited on 06/14/2023).

[76] F.-F. Li, Y. Li, and R. Gao, *CS231n: Deep Learning for Computer Vision*, Lecture Notes; Stanford University, CAL, 2023. [Online]. Available: https://cs231n.github.io/convolutional-networks/#conv (visited on 05/30/2023).

[77] A. Radford, L. Metz, and S. Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, Jan. 2016. arXiv: 1511.06434 [cs].

[78] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, Jul. 2015, pp. 448–456.

[79] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[80] S. Cai, Y. Shu, G. Chen, B. C. Ooi, W. Wang, and M. Zhang, *Effective and Efficient Dropout for Deep Convolutional Neural Networks*, Jul. 2020. arXiv: 1904.03392 [cs].

[81] PyTorch, *Dropout2d — PyTorch 2.0 documentation*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Dropout2d.html (visited on 06/11/2023).

[82] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-Informed Machine Learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, Jun. 2021.

[83] M. Reichstein, G. Camps-Valls, B. Stevens, *et al.*, "Deep Learning and Process Understanding for Data-Driven Earth System Science," *Nature*, vol. 566, no. 7743, pp. 195–204, Feb. 2019.

[84] S. Markidis, "The Old and the New: Can Physics-Informed Deep-Learning Replace Traditional Linear Solvers?" *Frontiers in Big Data*, vol. 4, 2021.

[85] X. Jia, J. Willard, A. Karpatne, *et al.*, "Physics-Guided Machine Learning for Scientific Discovery: An Application in Simulating Lake Temperature Profiles," *ACM/IMS Trans. Data Sci.*, vol. 2, no. 3, May 2021.

[86] D. Komatitsch and J.-P. Vilotte, "The Spectral Element Method: An Efficient Tool to Simulate the Seismic Response of 2D and 3D geological structures," *Bulletin of the Seismological Society of America*, vol. 88, no. 2, pp. 368–392, Apr. 1998.

[87] M. B. Hafeez and M. Krawczuk, "A Review: Applications of the Spectral Finite Element Method," *Archives of Computational Methods in Engineering*, vol. 30, no. 5, pp. 3453–3465, Jun. 2023.

[88] D. Komatitsch and J. Tromp, "A Perfectly Matched Layer Absorbing Boundary Condition for the Second-Order Seismic Wave Equation," *Geophysical Journal International*, vol. 154, no. 1, pp. 146–153, Jul. 2003.

[89] Z. Xie, D. Komatitsch, R. Martin, and R. Matzen, "Improved Forward Wave Propagation and Adjoint-Based Sensitivity Kernel Calculations Using a Numerically Stable Finite-Element PML," *Geophysical Journal International*, vol. 198, no. 3, pp. 1714–1747, Jul. 2014.

[90] B. Engquist and A. Majda, "Absorbing Boundary Conditions for the Numerical Simulation of Waves," *Mathematics of Computation*, vol. 31, no. 139, pp. 629–651, 1977.

[91] T. Lay and T. C. Wallace, "Chapter 4 - surface waves and free oscillations," in *Modern Global Seismology*, ser. International Geophysics, R. Dmowska and J. R. Holton, Eds., vol. 58, Academic Press, 1995, pp. 116–172.

[92] X. Ding, Y. Wang, Z. Xu, W. J. Welch, and Z. J. Wang, "Continuous Conditional Generative Adversarial Networks: Novel Empirical Losses and Label Input Mechanisms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 7, pp. 8143–8158, 2023.

[93] T. Kadeethum, D. O'Malley, J. N. Fuhg, *et al.*, *A framework for data-driven solution and parameter estimation of PDEs using conditional generative adversarial networks*, 2021. arXiv: `2105.13136 [cs.LG]`.

[94] NVIDIA, *What is PyTorch?* 2023. [Online]. Available: `https://www.nvidia.com/en-us/glossary/data-science/pytorch/` (visited on 06/20/2023).

[95] O. Ronneberger, P. Fischer, and T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015. arXiv: `1505.04597 [cs.CV]`.

[96] H. Wang, P. Cao, J. Wang, and O. R. Zaiane, "UCTransNet: Rethinking the Skip Connections in U-Net from a Channel-Wise Perspective with Transformer," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 3, pp. 2441–2449, 2022.

[97] H. de Vries, F. Strub, J. Mary, H. Larochelle, O. Pietquin, and A. Courville, *Modulating early visual processing by language*, 2017. arXiv: `1707.00683 [cs.CV]`.

[98] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved Training of Wasserstein GANs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 5769–5779.

[99] M. Arjovsky, S. Chintala, and L. Bottou, *Wasserstein GAN*, 2017. arXiv: 1701.07875 [stat.ML].

[100] C. Villani, "The Wasserstein Distances," in *Optimal Transport: Old and New*, A. Chenciner and S. Varadhan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 93–111.

[101] J. L. Ba, J. R. Kiros, and G. E. Hinton, *Layer Normalization*, Jul. 2016. arXiv: 1607.06450 [cs, stat].

[102] Y. Yang and P. Perdikaris, "Adversarial uncertainty quantification in physics-informed neural networks," *Journal of Computational Physics*, vol. 394, pp. 136–152, 2019.

[103] PyTorch, *Automatic Differentiation with torch.autograd*, 2023. [Online]. Available: https://pytorch.org/tutorials/beginner/basics/autogradqs%5C_tutorial.html (visited on 06/13/2023).

[104] B. Fornberg, "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids," *Mathematics of Computation*, vol. 51, no. 184, pp. 699–706, 1988.

[105] PyTorch, *Reproducibility — PyTorch 2.0 documentation*, 2023. [Online]. Available: https://pytorch.org/docs/stable/notes/randomness.html (visited on 06/17/2023).

[106] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, Jan. 2017. arXiv: 1412.6980 [cs].

[107] T. Schaul, I. Antonoglou, and D. Silver, *Unit Tests for Stochastic Optimization*, Feb. 2014. arXiv: 1312.6055 [cs].

[108] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, *Optuna: A Next-generation Hyperparameter Optimization Framework*, Jul. 2019. arXiv: 1907.10902 [cs, stat].

[109] Optuna Contributors, *Efficient Optimization Algorithms*, 2018. [Online]. Available: https://optuna.readthedocs.io/en/stable/tutorial/10_key_features/003_efficient_optimization_algorithms.html#pruning (visited on 06/20/2023).

[110] N. Selvaraj, *Hyperparameter Tuning Using Grid Search and Random Search in Python*, 2022. [Online]. Available: https://www.kdnuggets.com/2022/10/hyperparameter-tuning-grid-search-random-search-python.html (visited on 06/18/2023).

[111] J. Brownlee, *How to Identify Overfitting Machine Learning Models in Scikit-Learn*, Nov. 2020. [Online]. Available: https://machinelearningmastery.com/overfitting-machine-learning-models/ (visited on 06/16/2023).

[112] PyTorch, *Torch.cuda.memory_ allocated — PyTorch 2.0 documentation*, 2023. [Online]. Available: `https://pytorch.org/docs/stable/generated/torch.cuda.memory_allocated.html` (visited on 06/19/2023).

[113] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, Apr. 2004.

[114] J. Brownlee, *How to use Learning Curves to Diagnose Machine Learning Model Performance*, Feb. 2019. [Online]. Available: `https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/` (visited on 06/18/2023).

[115] A. Srivastava, L. Valkov, C. Russell, M. U. Gutmann, and C. Sutton, "VEEGAN: Reducing Mode Collapse in GANs Using Implicit Variational Learning," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 3310–3320.

[116] Y. Saatci and A. G. Wilson, "Bayesian GAN," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017.

[117] S. Lala, M. Shady, A. Belyaeva, and M. Liu, "Evaluation of mode collapse in generative adversarial networks," *High Performance Extreme Computing*, 2018.

[118] M. Shahbazi, M. Danelljan, D. P. Paudel, and L. Van Gool, *Collapse by Conditioning: Training Class-conditional GANs with Limited Data*, Mar. 2022. eprint: `2201.06578` (cs). (visited on 06/23/2023).

[119] M. Feurer and F. Hutter, "Hyperparameter Optimization," in *Automated Machine Learning: Methods, Systems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., Cham: Springer International Publishing, 2019, pp. 3–33.

[120] J. Wu, C. Zhang, T. Xue, B. Freeman, and J. Tenenbaum, "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29, Curran Associates, Inc., 2016.

[121] A. Daw, M. Maruf, and A. Karpatne, "PID-GAN: A GAN Framework Based on a Physics-informed Discriminator for Uncertainty Quantification with Physics," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 237–247.

[122] S. Wang, Y. Teng, and P. Perdikaris, "Understanding and Mitigating Gradient Flow Pathologies in Physics-Informed Neural Networks," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, A3055–A3081, 2021.

[123] M. Bode, M. Gauding, Z. Lian, *et al.*, "Using Physics-Informed Enhanced Super-Resolution Generative Adversarial Networks for Subfilter Modeling in Turbulent Reactive Flows," *Proceedings of the Combustion Institute*, vol. 38, no. 2, pp. 2617–2625, Jan. 2021.

[124] R. Rodriguez-Torrado, P. Ruiz, L. Cueto-Felgueroso, *et al.*, "Physics-Informed Attention-Based Neural Network for Hyperbolic Partial Differential Equations: Application to the Buckley–Leverett Problem," *Scientific Reports*, vol. 12, no. 1, p. 7557, May 2022.

[125] S. Fu and K. Gao, "A Fast Solver for the Helmholtz Equation Based on the Generalized Multiscale Finite-Element Method," *Geophysical Journal International*, vol. 211, no. 2, pp. 797–813, Nov. 2017.

[126] B. Li, H. Wang, S. Feng, X. Yang, and Y. Lin, *Solving Seismic Wave Equations on Variable Velocity Models with Fourier Neural Operator*, 2023. arXiv: 2209.12340 [cs.LG].

[127] Z. Li, N. Kovachki, K. Azizzadenesheli, *et al.*, *Fourier Neural Operator for Parametric Partial Differential Equations*, 2021. arXiv: 2010.08895 [cs.LG].