ABSTRACT

LEONARD, CHRISTOPHER ALLEN. Data-Driven Methods for Partial Differential Equations and Discrete Convolution. (Under the direction of Dr. Alina Chertock and Dr. Semyon Tsynkov).

In the information age, data-driven methods are an increasingly popular tool used in science, engineering, business, and many other fields of study. Some of these methods include machine learning algorithms for image and speech recognition, email spam detection, and self-driving cars, but new applications are being discovered every day. Here, we apply data-driven methods to two of the most important types of equations in science, partial differential equations and convolutions. First, we show two ways neural networks are directly used to simulate partial differential equations (PDEs) using data driven methods where the underlying equation is unknown. Then, we show that neural networks can assist more classical numerical methods. Here, we use a neural network to help quantify the amount of artificial viscosity that should be used in a numerical PDE solver. Next, we will explore how neural networks can be used to discover properties of partial differential equations. We do this by using a neural network to predict the shape of the lacunae produced by the source term of the wave equation. Lastly, we will show how the quantized tensor train decomposition can be used to compute discrete convolutions, reducing the noise and storage space of our data.

Data-Driven Methods for Partial Differential Equations and Discrete Convolution

by
Christopher Allen Leonard

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Applied Mathematics

Raleigh, North Carolina
2022

APPROVED BY:

Dr. Alina Chertock Co-chair of Advisory Committee	Dr. Semyon Tsynkov Co-chair of Advisory Committee
Dr. Arvind Saibaba	Dr. Mohammad Farazmand

Dr. Marguerite Moore

BIOGRAPHY

Christopher Leonard was born in 1994, Augusta, Georgia. He went to The University of Alabama, where he obtained his bachelors degree in mathematics in 2016. After this, we went on to Georgia Southern University to do a master in computational mathematics, where he graduated in 2018. While at Georgia Southern University, he completed a masters thesis titled "Blow Up Solution of Bose-Einstein Condensates with Anisotropic Trapping Potential and Rotation" under the advisement of Dr. Shijun Zheng, and was awarded the Meritorious Graduate Student Award. He then went on to get his PhD at North Carolina State University, where he did his research under the advisement of Dr. Alina Chertock and Dr. Semyon Tsynkov.

ACKNOWLEDGEMENTS

First, I would like to give a very special thanks to my advisor, Dr. Alina Chertock, who always made time for me even when her schedule was completely full. She has been a great support to me throughout my graduate studies, and an amazing teacher for me to learn applied mathematics from. Secondly, I would like to give a special thanks to Dr. Semyon Tsynkov. As a second advisor to me, he has been a pleasure to work with, and without his insight, some of my projects would have been next to impossible to complete. Next, I would like to give thanks to my other collaborators, Dr. Alexander Kurganov and Dr. Sergey Utyuzhnikov, who have made invaluable contributions to my research. I would also like to thank my committee members, Dr. Arvind Saibaba and Dr. Mohammad Farazmand, who have both pointed me towards relevant articles that have helped me with my research.

Finally, I would like to thank my friends and family, specifically my parents (Brian and Mary Leonard), grandparents (Allen and Jean Strickland), and my girlfriend (Kristen Windoloski), who have all been a huge support to me throughout my graduate school journey. I would also like to thank my brother and sister (Timothy and Ginny), and my dog (Brooks) for their support as well.

TABLE OF CONTENTS

List of Tables . . . . . **vi**

List of Figures . . . . . **vii**

Chapter 1 Introduction **1**

Chapter 2 Partial Differential Equations and Neural Networks **4**
2.1 Neural Networks 6
2.1.1 Training the neural network 9
2.2 Finite Volume Method 11

Chapter 3 Simulating Partial Differential Equation Solutions with the help of Neural Networks **16**
3.0.1 Learning the Dynamics of PDEs with Neural Networks 16
3.0.2 Training the Neural Network 18
3.1 One Step Neural Network 20
3.1.1 Numerical Examples 20
3.2 Time Stepping Neural Network 27
3.2.1 Numerical Examples 30
3.3 Tuning Artificial Viscosity with Neural Networks 45
3.3.1 Finding the Best Coefficient 48
3.3.2 Collecting Training Data 49
3.3.3 Numerical Examples 51
3.4 Conclusion 55

Chapter 4 Finding the Shape of the Lacunae with Neural Networks **57**
4.1 Introducing the Lacunae of the Wave Equation 57
4.2 Construction of the Data Set 60
4.3 Numerical Results 63
4.4 Conclusion 69

Chapter 5 Convolutions and the Tensor Train Decomposition **70**
5.1 Convolution 72
5.2 Synthetic Aperture Radar (SAR) 74
5.3 Tensor Train Decomposition 76

Chapter 6 Computing Discrete Convolutions with the Quantized Tensor Train Decomposition **80**
6.1 Computing the Convolution with the QTT Decomposition 80
6.2 Denoising 85
6.3 Numerical Simulations 86

6.3.1 Example 1 89
6.3.2 Example 2 92
6.3.3 Example 3 94
6.4 Conclusion 97

References **98**

APPENDICES **104**
Appendix A Numerical Linear Algebra 105
A.1 Singular Value Decomposition 105
A.1.1 Randomized SVD 107
A.2 Gauss–Seidel Method 108
Appendix B Runge-Kutta Methods 110
B.1 Third Order Strong Stability-Preserving Runge-Kutta Method 110
B.2 Fourth Order Implicit-Explicit 111
Appendix C Central Upwind Method 113
Appendix D Variables 116

LIST OF TABLES

Table 1.1 Car price 2

Table 3.1 l_1-norm relative error table: 1D shallow water equations. 26
Table 3.2 l_1-norm relative error table: Burgers equation. 33
Table 3.3 l_2-norm relative error table: 1D wave equation. 39
Table 3.4 l_1-norm relative error table: 1D Isentropic Euler equations. 41
Table 3.5 l_1-norm relative error table: 2D Isentropic Euler equations. 45
Table 3.6 Comparing run time, error, and total variation for w from the all global solutions with $C = 0$, $C = 50$, and C determined by a neural network. . . 53

Table 6.1 l_2-norm relative error for $K = 20$ for examples 1 and 2, and $K = 10$ for example 3. 88
Table 6.2 Run time (seconds): Example 1 convolutions. 91
Table 6.3 Data storage for Example 1. 91
Table 6.4 Run times (seconds): Example 3 convolutions. 96
Table 6.5 Data storage for Example 3. 96

Table D.1 A list of variables and functions used in PDEs project 116
Table D.2 Size variables and the indices used to index them in PDEs project 118
Table D.3 A list of variables and functions used in convolution project 118
Table D.4 Size variables and the indices used to index them in convolution project . 119

LIST OF FIGURES

Figure 2.1 General architecture for a neural network. 8
Figure 2.2 Local wave speed of characteristics. 14

Figure 3.1 Architecture for the one-step neural network to solve the Camassa-Holm equation. 22
Figure 3.2 Comparison between the neural network approximation and the particle method solution for $x_j(t)$ (left) and $p_j(t)$ (right), $j = 1,2$. 22
Figure 3.3 Comparison between the neural network approximation and the particle method solution for $v(x,t)$ at time $t = 0$, $t = 2$, $t = 6$ and $t = 10$. 23
Figure 3.4 Architecture for the one-step neural network to solve the shallow water equation. 25
Figure 3.5 *Left:* Cell average for $\bar{h}$ at time $t = 10$. *Right:* Cell average for $\bar{q}$ at time $t = 10$. 26
Figure 3.6 *Left:* Sensors for one dimension. *Right:* Sensor locations for two dimensional domain. 30
Figure 3.7 Neural network architecture for Burgers equation. 33
Figure 3.8 Burgers equation at $t = 10$ with initial condition $u_0(x) = 0.9 - 0.2\cos(x) + 0.4\sin(x)$. 34
Figure 3.9 Burgers equation with initial condition $u_0(x) = 0.9e^{-0.4x^2}$ at time $t = 0$ (left) and time $t = 10$ (right). 35
Figure 3.10 Burgers equation with initial condition $u_0(x) = 0.6$ for $x < 0$ and $u_0(x) = -0.2$ for $x \geq 0$ at time $t = 0$ (left) and time $t = 10$ (right). 36
Figure 3.11 Burgers equation with initial condition $u_0(x) = -0.2$ for $x < 0$ and $u_0(x) = 0.6$ for $x \geq 0$ at time $t = 0$ (left) and time $t = 10$ (right). 36
Figure 3.12 Neural network architecture for the wave equation. 38
Figure 3.13 Reference (red) and neural network (blue) solution to (3.23) with initial condition parameters $a_1 = [0.6, 0.8, -0.7]$, $b_1 = [-0.4, 1.0, -0.4]$ and $a_2 = [0.1, 0.1, 0.7]$, $b_2 = [0.4, -0.7, -0.4]$. 39
Figure 3.14 Neural network architecture for the isentropic Navier-Stokes equations. . 41
Figure 3.15 Isentropic Euler equations at t=0.15 with initial condition parameters $\rho_L = 3.3$, $\rho_R = 0.7$. 42
Figure 3.16 Neural network architecture for the isentropic Navier-Stokes equations. . 43
Figure 3.17 Plot of ρ at $t = 0.15$ for the reference solution (Left) and neural network solution (Right) with $\rho_1 = 0.79$, $\rho_2 = 1.32$, $\rho_3 = 0.55$, and $\rho_4 = 0.99$. . . 44
Figure 3.18 Burgers equation with initial condition: $u(x,0) = 0.5 - \sin(x)$. 46
Figure 3.19 Burgers equation with initial condition: $u(x,0) = 0.5 - \sin(x)$. 47
Figure 3.20 Burgers equation with artificial viscosity where $C = 0$ (left), $C = 20$ (middle), and $C = 50$ (right). 48
Figure 3.21 Artificial viscosity coefficient neural network for the shallow water equation. 52

Figure 3.22 Comparison of C values calculated by neural network and C value calculated by the method in 3.3.2. 54
Figure 3.23 *Left:* All-global method with $C = 0$. *Right:* All-global method with C determined by neural network. 54
Figure 3.24 All-global method with C determined by neural network vs all-global method with $C = 50$. 55

Figure 4.1 The computational domain Ω with subdomain Q inside the dotted red line, and Q_f inside the blue circle. 61
Figure 4.2 Characteristic lines from the point (x_j, t_n). Green triangles indicate nodes within Δx of the characteristic lines at each time step. If there exist a green triangle and blue square at the same node, then $\Psi_{j,n} = 1$, else $\Psi_{j,n} = -1$. 63
Figure 4.3 Neural network architecture to find the lacunae of the wave equation. . . 64
Figure 4.4 Reconstruction of the shape of the lacuna (4.5) by a neural network for the case $I^{(m)} = 1$. *Top left:* Reference solutions Ψ^{ref}. *Top right:* Neural Network solutions Ψ^{NN}. *Bottom left:* The sets Q_f. *Bottom right:* $\Psi^{ref} - \Psi^{NN}$. 65
Figure 4.5 Reconstruction of the shape of the lacuna (4.5) by a neural network for the case $1 \leq I^{(m)} \leq 4$ with the same layout as in Figure 4.4 67
Figure 4.6 Reconstruction of the shape of the lacuna (4.5) by a neural network for the case $1 \leq I^{(m)} \leq 4$ with a hand crafted example such that the lacunae has a 'pocket'. 68

Figure 6.1 Kernel function (6.3) with $\Delta_x = 0.04\pi$. 88
Figure 6.2 *Top Left:* True convolution of data without noise, $\boldsymbol{I}$. *Top Right:* Function data with noise, $\boldsymbol{f}_\xi$. *Middle Left:* True convolution of data with noise, $\boldsymbol{I}_\xi$. *Middle Right:* Convolution using the **max rank TT-SVD** algorithm, $\boldsymbol{I}_{QTT_0}$. *Bottom Left:* Absolute error of $\boldsymbol{I}_\xi$. *Bottom Right:* Absolute error of $\boldsymbol{I}_{QTT_0}$. 90
Figure 6.3 *Top Left:* True convolution of data without noise, $\boldsymbol{I}$. *Top Right:* Zoomed in graph of $\boldsymbol{I}_\xi$, $\boldsymbol{I}_{QTT_0}$, and $\boldsymbol{I}_{ref}$. *Bottom Left:* Absolute error of $\boldsymbol{I}_\xi$. *Bottom Right:* Absolute error of $\boldsymbol{I}_{QTT_0}$. 93
Figure 6.4 *Top Left:* True convolution of data without noise, $\boldsymbol{I}$. *Top Right:* Function data with noise, $\boldsymbol{f}_\xi$. *Middle Left:* True convolution of data with noise, $\boldsymbol{I}_\xi$. *Middle Right:* Convolution using the **max rank TT-SVD** algorithm, $\boldsymbol{I}_{QTT_0}$. *Bottom Left:* Absolute error of $\boldsymbol{I}_\xi$. *Bottom Right:* Absolute error of $\boldsymbol{I}_{QTT_0}$. 95

CHAPTER

1

INTRODUCTION

Data-driven computational methods are playing a major role in modern science, engineering, and technological sciences. This is in large due to the fact that advances in hardware and software have enabled us to be able to store and process large amounts of data with relative ease. The area of study that focuses on handling and extracting information from these large sets of data is often referred to as 'Big Data', which seeks to find information from data sets that traditional computational and statistical methods are unable to handle.

Probably the most popular area of research in 'Big Data' is machine learning. Machine learning is a branch of artificial intelligence that entails training a computer to find patterns in data without being explicitly programmed in how to do so. This is usually accomplished by minimizing some cost function that depend on the machine learning model and the provided data set. Once the model is trained, it can then be used to make predictions given new data that is not in, but is closely related to, the original data set. These models usually fall into one of two categories, classification or regression. For classification models, the range of the machine learning model is an element of a discrete set. For example, there could be a function h whose input is a picture and the output is a string that indicates if there is a dog, a cat, neither a dog nor a cat, or both a dog and a cat, in the picture, i.e. range$(h) = \{$'dog','cat','neither','both'$\}$. Whereas for a regression model, the output is continuous. An example of this could be a function

g, whose input is information about a students past test and homework grades, and whose output is a prediction of their final exam grade, i.e. range$(g) = [0, 100]$.

Another important area of study in 'Big Data' is tensors and tensor decompositions. Tensors are used to store data that depends of multiple variables, just as matrix can be used to store data that depends on two variables. For a simplified example, consider a case when the price of a car depends only on if it is new or old, and if it black or white, represented in Table 1.1

Table 1.1: Car price

	Old	New
Black	a_{11}	a_{12}
White	a_{21}	a_{22}

,

where a_{ij} is the price of the car for all $i, j \in \{1, 2\}$. Then Table 1.1 could be represented by the matrix

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix},$$

where for a_{ij}, $i = 1$ corresponds to if the car is black, $i = 2$ correspond to if the car is white, $j = 1$ corresponds to if the car is old, and $j = 2$ corresponds to if the car is new. If however, the car price also depended on if the car is small or of the car is large, then a tensor $\mathscr{A}$ with elements a_{ijk} for all $i, j, k \in \{1, 2\}$, would need to be used to describe the price of the car. Here, i and j still describe the same thing as in the matrix but now there is another index k which describes if the car is large or small. Notice how adding one extra dimension requires us to store twice as many elements. In general, adding one dimension that has K variables requires us to multiply the number of elements by K. This is often referred to as the curse of dimensionality, and can be a huge issue for numerical computations. This is because the data may take up a lot of memory and can be too large to simulate in a reasonable amount of time. One way to work around this is to use what are known as tensor decompositions, where we can approximate the same information that is in a tensor with multiple smaller vectors, matrices, or tensors, whose total number of elements may be smaller than the original tensor.

In this thesis we will use these two popular data-driven methods, machine learning and tensor decompositions, and apply them to two of the most important types of equations in science and engineering, partial differential equations (PDEs) and convolutions. First, we will show

how neural networks can be applied to simulate and learn information about partial differential equations. The majority of this thesis will be on this topic. After that, we will discuss how the tensor train (TT) decomposition can be used to numerically solve convolution integrals.

We organize the thesis in the following way. In Chapter 2, we introduce partial differential equations and neural networks. We give some background on how neural networks have been used to help simulate or discover properties of PDEs, and discuss what our contributions to this field are. In this chapter, we discuss how neural networks are used in this thesis, as well as the finite volume method, which is a numerical method that is used often in our projects. In Chapter 3, we then present our methods of using neural networks to simulate PDEs. Then in Chapter 4, we show how a neural network can be used to find the shape of the lacunae produced by the source term of a wave equation. That will conclude the work that we do with neural networks and PDEs. After that, in Chapter 5, we introduce convolutions and the tensor train (TT) decomposition. Again, we will give some background on tensor decompositions, and what our contributions are. We then explain how to numerically approximate convolution integrals, as well as introduce the tensor train decomposition. Lastly, in Chapter 6, we present our methods of computing convolutions with the help of tensor decompositions, and how we use them to reduce noise in data.

CHAPTER

2

PARTIAL DIFFERENTIAL EQUATIONS AND NEURAL NETWORKS

Differential equations are perhaps the most important equations in science and engineering. They are the equations that describe how functions behave with respect to their derivatives, or rates of change with respect to some variable. These functions often depend on multiple independent variables, in which case the differential equations are called partial differential equations, due to the fact that the derivatives are partial derivatives with respect to each independent variable. In this thesis, we will be considering time-dependent partial differential equations that depend on one time variable and at least one spatial variable, which can be represented as

$$\boldsymbol{U}_t = \boldsymbol{F}(\boldsymbol{x}, t, \boldsymbol{U}, \boldsymbol{U}_{x_1}, \boldsymbol{U}_{x_2}, \boldsymbol{U}_{x_1 x_1}, \boldsymbol{U}_{x_1 x_2}, \boldsymbol{U}_{x_2 x_2}, \ldots), \tag{2.1}$$

where $\boldsymbol{U} = (u_1(\boldsymbol{x}, t), \ldots, u_{N_e}(\boldsymbol{x}, t))$ is a vector function of the spatial variable $\boldsymbol{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$ and the time variable $t \geq 0$, and $\boldsymbol{F}$ is a smooth function of its arguments. A general setting, presented in (2.1), includes among others the hyperbolic systems of conservation and balance laws, systems of convection-diffusion and convection-diffusion-reaction equations, dispersive equations and many others. Models spanned by (2.1) are widely used to describe a variety

of phenomena in physical, astrophysical, geophysical, meteorological, biological, chemical, financial, social and other scientific areas.

In recent years, machine learning techniques to solving and learning about differential equations has been a growing field of interest. This is in part due to the success machine learning has had in other fields such as computer vision and speech recognition. Along with these successes, another motivation to use neural networks is due to the universal approximation theorem [33]. This theorem showed that, under certain conditions, an artificial neural network can be used to approximate any continuous function $f : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$, where d_{in} and d_{out} are positive integers. Others, such as [54; 16], have shown how adding depth to the neural network can allow you to learn certain functions with less parameters than with a very wide but shallow neural network. This motivates us to use multi-layer, feedforward neural network for our work (see 2.1).

There are many different approaches to finding numerical solutions to PDEs with the help of neural networks. One of the most popular methods is known as a physics-informed neural network (PINN) [67], where it was shown that a neural network can be trained to satisfy the differential equation, along with the initial and boundary conditions. PINN have been used to quantify uncertainty in a PDEs [83; 81], solve stochastic PDEs (SPDE) [82; 38], solve high-dimensional PDEs [74], and even used to find solutions to PDEs that may be irregular [56]. While this approach very recently become popular, similar ideas to training neural networks have gone back as far as 1998 [46]. One of the major drawbacks of PINN is that they need to be retrained for any new initial or boundary conditions.

In [31], another approach to solving high-dimensional PDEs with neural networks is given. Other approaches use neural networks along side standard numerical methods and utilize the benefits of both. This has been done to speed up the method [78; 14; 42] and to improve accuracy of the method [77; 5]. In [3], it was shown that data from a physical system can help train a neural network to make accurate predictions with a low resolution grid. It was shown in [53; 6] that neural networks can be used as a solution map between infinite dimensional spaces where the solution map can take discretizations of arbitrary size. Neural networks can also be used to find properties of the PDE without trying to simulate the solution, as is done in [40] where a neural network is trained to obtain a physical quantity of interest when given the coefficients of the PDE. In [69; 70; 71] it was shown that data from a given system can actually be used to learn the underlying partial differential equations and not just used to help simulate new solutions.

We will investigate different uses of neural networks to solve and study solutions of partial differential equations. Similar to the work done in [8; 66], we will explore ways to simulate partial differential equations when we have solution information from (2.1) but the equation $\boldsymbol{F}$ is

unknown. In Section 3.1, we develop a method of simulating multiple time steps of solution data, with only a single neural network evaluation. Then in Section 3.2, we develop a time stepping method, where the neural network only depends on local data. This is a much more realistic setting for data collection, and allows us to simulate a wide range of initial value problems. In Section 3.3, we show a new way in which a neural network can assist in adding artificial viscosity for hyperbolic systems of equations, and help stabilize more traditional numerical methods. A similar idea has been proposed in [48], but with a different approach in how to help the artificial viscosity.

In Chapter 4, we use neural networks to find the shape of the lacunae produced by the wave equation with zero initial conditions and a source term. Using neural networks to find information about the wave equation has been studied by Eli Turkel et al, where neural networks have been trained to find obstacles in the path of propagating waves [36], simulate stable solutions even when the CFL condition is not met [63], and find the source term of the wave equation [37]. However, to the best of our knowledge, this is the first time neural networks have been used to find the shape of the lacunae produced by the wave equation.

2.1 Neural Networks

An artificial neural network is an operator $\boldsymbol{N}_\Theta(\cdot)$ with a parameter set $\Theta = (\Theta_1, \ldots, \Theta_{H+1})$, where Θ_η is itself a parameter set for each layer $\eta = 1, 2, ..., H+1$ (see equation (2.3)). The set Θ can have hundreds, thousands, or even millions of parameters, which allows $\boldsymbol{N}_\Theta$ to represent a wide range of functions depending on which parameters are chosen. Typically, the network is provided with a set of input-output pairs $\{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^M$, to "learn" which parameters to use by a training process. This training process tries to find the parameter set Θ such that

$$\boldsymbol{U}_m^{\text{out}} \approx \boldsymbol{N}_\Theta(\boldsymbol{U}_m^{\text{in}}), \quad m = 1, \ldots, M$$

by minimize a given loss function

$$L_\Theta = L(N_\Theta, \{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^M) \tag{2.2}$$

that depends on the neural network and the supplied set of data.

The process of training a neural network from a set of input-output data pairs is known as supervised learning. Other training processes include unsupervised learning and reinforcement learning. Unsupervised learning is a training algorithm that finds underlying patterns in the data

such as clusters, and reinforcement learning is an algorithm that uses a reward system to learn the best parameters for the machine learning model. While all three learning process have found success in different applications, supervised learning has been the most successful approach to training neural networks and is what is used for all of the models in this thesis.

Artificial neural networks are made from compositions of smaller functions (see equation (2.3)). Each of these smaller functions correspond to what is known as a layer of the neural network. This is visualized in Fig. 2.1 which depicts a general setup for feedforward neural network. Feedforward neural networks are a specific form of a neural network where the information is passed in one direction without looping back. This is opposed to say recurrent neural networks that contain cycles in their neural network architecture. All of the examples in this thesis will use feedforward neural networks, although many of them could be extended to other types of architectures such as recurrent neural networks.

A feedforward neural network with H hidden layers can be represented as the composition of parameterized functions

$$\boldsymbol{N}_{\Theta} = \boldsymbol{N}_{\Theta_{H+1}} \circ \boldsymbol{N}_{\Theta_H} \circ \ldots \circ \boldsymbol{N}_{\Theta_1}, \tag{2.3}$$

with

$$\boldsymbol{N}_{\Theta_\eta}(z) = \boldsymbol{\sigma}_\eta(A_\eta), \quad \eta = 1, 2, \ldots, H+1,$$

where A_η is the connection between layers $\eta - 1$ and η, and $\boldsymbol{\sigma}_\eta$ is the activation function for the layer η. Here, $\eta = 0$ corresponds to the input layer and $\eta = H+1$ corresponds to the output layer. A layer is said to be fully connected if

$$\boldsymbol{A}_\eta(z) = \boldsymbol{W}_\eta z + \boldsymbol{b}_\eta,$$

where $\boldsymbol{W}_\eta \in \mathbb{R}^{\omega_\eta \times \omega_{\eta-1}}$, $\boldsymbol{b}_\eta \in \mathbb{R}^{\omega_\eta}$. Here, ω_η is the number of nodes for layer η, $\boldsymbol{W}_\eta$ is known as the weight matrix, and $\boldsymbol{b}_\eta$ is the bias vector. Another type of layer is a convolutional layer, which in the simplest case can be defined as

$$\boldsymbol{A}_\eta(\boldsymbol{z}) = \boldsymbol{c}_\eta \star \boldsymbol{z} + \boldsymbol{b}_\eta,$$

where $\boldsymbol{c}_\eta \in \mathbb{R}^{\kappa_\eta}$, $\boldsymbol{b}_\eta \in \mathbb{R}$, and $\star$ is the cross-correlation operator. The vector $\boldsymbol{c}_\eta$ is known as a convolution kernel with a kernel size of κ_η. While the value of $\boldsymbol{b}_\eta$ is often a scalar value for convolution layers (and this is the default setting in PyTorch [64]), it can also be set such that $\boldsymbol{b} \in \mathbb{R}^{\omega_\eta}$. The parameters in the sets Θ_η are the element values for $\boldsymbol{W}_\eta$, $\boldsymbol{b}_\eta$, and $\boldsymbol{c}_\eta$. The activation functions $\boldsymbol{\sigma}_\eta$, $\eta = 1, \ldots, H+1$, are functions prescribed before the training process. Usually

these functions are nonlinear so that the neural network can learn nonlinear relationships from the data.

In Fig. 2.1 we depict a neural network architecture, where the green box represents the input layer, the blue boxes are the hidden layers, and the yellow box is the output layer. The type of connection between each layer is given by the letter above the arrows in the diagram, where usually a $\boldsymbol{W}_\eta$ represents a fully connected layer and $\boldsymbol{c}_\eta$ represents a convolutional layer, but in Fig. 2.1 we use the more general affine representation of $\boldsymbol{A}_\eta$. The activation functions σ_η, $\eta = 1, \ldots, H+1$, will be specified for each example as well.

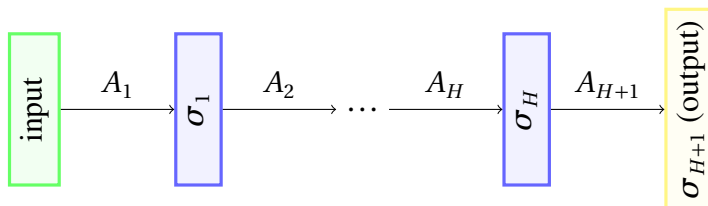

Figure 2.1: General architecture for a neural network.

Below, we give a brief list of activation functions that are used in this theses.

- The rectified linear unit (ReLU):

$$ReLU(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0. \end{cases} \tag{2.4}$$

- The leaky rectified linear unit (LeakyReLU):

$$LeakyReLU(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases}, \tag{2.5}$$

  where $0 < \alpha \ll 1$ is a small parameter value. We will use PyTorch's [64] default value of $\alpha = 0.01$.

- The hyperbolic tangent function:

$$\tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}. \tag{2.6}$$

- The Identity function:

$$I(z) = z. \tag{2.7}$$

- The semi-step activation function:

$$f_{ss}(z) = \begin{cases} 0 & z \leq 0 \\ 20z & 0 < z < \frac{1}{20} \\ 1 & z \geq \frac{1}{20}. \end{cases} \tag{2.8}$$

  This is an activation function that we created as a continuous approximation to a step function.

There are many other activation function such as the sigmoid function, binary step function, and softmax function just to name a few, but these will not be used by any of the neural networks in this thesis.

2.1.1 Training the neural network

Before we train a neural network, we need to define a few hyperparameters. These hyperparameters are

(a) Loss function: L_Θ (see (2.2)).

(b) Optimizer: The algorithm that is used to try and find the global minimum of the loss function.

(c) Learning rate: The initial step size that the optimization algorithm takes. Depending on the optimization algorithm, the step sizes may change between steps.

(d) Number of epochs: The number of times the optimization algorithm goes through the entire training data set. Denote the number of epochs as M_e.

(e) Batch size: The number of samples from the training data that propagates through the network for each update of the parameters. Denote the batch size as B_s.

(f) Number of hidden layers: H.

(g) Linear function for connection between layers $\boldsymbol{A}_\eta$ ($\boldsymbol{W}_\eta$ or $\boldsymbol{c}_\eta$), and their sizes, i.e. we need to define ω_η and $\omega_{\eta-1}$ for fully connected layers and κ_η for convolutional layers.

(h) Activation functions: σ_η, $\eta = 1, \ldots, H+1$.

We will identify the specific hyperparameters used for each example in their respective sections. We train the neural network $\boldsymbol{N}_\Theta$ using Algorithm 1.

Algorithm 1: Neural Network Training

---

input : Data set $\{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^M$, initial neural network model $\boldsymbol{N}_{\Theta^{(0)}}$, and hyperparameters (a)-(h) from above.

output : Trained neural network $\boldsymbol{N}_\Theta$

1. *Start*: Randomly split the data set $\{(\boldsymbol{U}_m^{\text{in}}, \boldsymbol{U}_m^{\text{out}})\}_{m=1}^M$ into two disjoint sets, the training set of size M_{tr}, $\{\boldsymbol{U}_{m_i}^{\text{in}}, \boldsymbol{U}_{m_i}^{\text{out}}\}_{i=1}^{M_{tr}}$, and the validation set of size M_{val}, $\{\boldsymbol{U}_{m_l}^{\text{in}}, \boldsymbol{U}_{m_l}^{\text{out}}\}_{l=1}^{M_{val}}$ where $M = M_{tr} + M_{val}$.

   (a) The training set is used in the optimization algorithm to find the parameter set Θ that minimizes the loss function L_Θ.

   (b) The validation set is used to check that the neural network can generalize to new data.

2. *Iterate*: For $e = 1, 2, \ldots, M_e$

   (a) Let $M_{tr} = (\beta - 1)B_s + r_b$, where $\beta, r_b \in \mathbb{N}$ and $r_b \leq B_s$ and randomly split the training set into β separate batches each of size B_s except the last which is of size r_b.

   (b) *Iterate*: For each batch=1,2,…,β

      - Update the neural network parameters using one step of the optimization algorithm.

   (c) With the current parameter set $\Theta^{(e)}$, calculate the loss value $L_{\Theta^{(e)}}$ using the new model $\boldsymbol{N}_{\Theta^{(e)}}$ and the validation set $\{\boldsymbol{U}_{m_l}^{\text{in}}, \boldsymbol{U}_{m_l}^{\text{out}}\}_{l=1}^{M_{val}}$ as the input into the model (see (2.2)).

   (d) If $L_{\Theta^{(e)}} < L_{\Theta^{(i)}}$ for all $i = 1, \ldots, e-1$, the set $\Theta = \Theta^{(e)}$, i.e. if this loss value is less than the loss value at the end of every other epoch before it, update Θ.

Remark 2.1.1. *Sometimes we will a third test set of size M_{test}, which is used to test the neural network on data that has no influence over the training of the neural network at all.*

Remark 2.1.2. *Finding good hyperparameters is often done by a running multiple trials of the*

training algorithm using different hyperparameters and identifying which one results in the best outcome.

Remark 2.1.3. *We will use the PyTorch [64] machine learning framework for all the neural network models in this thesis. The initial parameters in the set $\Theta^{(0)}$ are the random parameters set by PyTorch.*

2.2 Finite Volume Method

For much of this thesis we will be using neural networks as a substitute to or along side traditional finite volume methods, thus we will outline what a finite volume method is in this section.

For simplicity, consider the the conservation law given by

$$\boldsymbol{U}_t + \sum_{d=1}^{D} \boldsymbol{F}_d(\boldsymbol{U})_{x_d} = 0, \quad \boldsymbol{x} \in \mathbb{R}^D,\ t \geq 0 \tag{2.9}$$

without any source or diffusive term. Here, $\boldsymbol{U} = (u_1(\boldsymbol{x}, t), \ldots, u_{N_e}(\boldsymbol{x}, t))$ and $\boldsymbol{F} = (\boldsymbol{F}_1, \ldots, \boldsymbol{F}_D)$ describes the flux, where $\boldsymbol{F}_d : \mathbb{R}^{N_e} \to \mathbb{R}^{N_e}$, $d = 1, \ldots, D$. Integrating this equation over the connected domain $\Omega_j \subset \mathbb{R}^D$ with boundary $\partial\Omega_j$, we get

$$\frac{d}{dt} \int_{\Omega_j} \boldsymbol{U}\, d\boldsymbol{x} + \int_{\partial\Omega_j} \boldsymbol{F}(\boldsymbol{U})\, dS = 0, \tag{2.10}$$

where the second integral comes from the divergence theorem. Let $|\Omega_j|$ denote the D-dimensional volume (or Lebesgue measure) of the finite volume cell Ω_j. Then the average value of $\boldsymbol{U}$ over all Ω_j is given by

$$\bar{\boldsymbol{U}}_j = \frac{1}{|\Omega_j|} \int_{\Omega_j} \boldsymbol{U}\, d\boldsymbol{x}. \tag{2.11}$$

Dividing every term in equation (2.10) by $|\Omega_j|$, integrating in time from time t_n to time t_{n+1}, and moving terms to the right hand side, we can obtain

$$\bar{\boldsymbol{U}}_j(t_{n+1}) = \bar{\boldsymbol{U}}_j(t_n) - \frac{1}{|\Omega_j|} \int_{t_n}^{t_{n+1}} \int_{\partial\Omega_j} \boldsymbol{F}(\boldsymbol{U}(\boldsymbol{x}, t))\, d\boldsymbol{S}\, dt \tag{2.12}$$

This is a general form of the finite volume setup in $\mathbb{R}^D$, where if we can approximate the integrals on the right hand side of (2.12), then we can evolve $\bar{\boldsymbol{U}}_j$ in time.

Let $D=1$, and $\Omega_j=[x_{j-1/2},x_{j+1/2}]$, then (2.12) becomes

$$\bar{\boldsymbol{U}}_j(t_{n+1})=\bar{\boldsymbol{U}}_j(t_n)-\frac{1}{\Delta x_j}\Big(\int_{t_n}^{t_{n+1}}\boldsymbol{F}(\boldsymbol{U}(x_{j+1/2},t))\,d\,t-\int_{t_n}^{t_{n+1}}\boldsymbol{F}(\boldsymbol{U}(x_{j-1/2},t))\,d\,t\Big),$$

where $\Delta x_j=x_{j+1/2}-x_{j-1/2}$. Using the above as a guide, we get the fully discrete finite volume method in one dimension as

$$\bar{\boldsymbol{U}}_j^{n+1}=\bar{\boldsymbol{U}}_j^n-\Delta t\Big(\frac{\mathscr{F}_{j+1/2}^n-\mathscr{F}_{j-1/2}^n}{\Delta x_j}\Big), \tag{2.13}$$

where

$$\bar{\boldsymbol{U}}_j^{n+1}\approx\bar{\boldsymbol{U}}_j(t_n)$$

approximates the cell average at time t_n, and

$$\mathscr{F}_{j+1/2}^n\approx\frac{1}{\Delta t}\int_{t_n}^{t_{n+1}}\boldsymbol{F}(\boldsymbol{U}(x_{j+1/2},t)\,d\,t. \tag{2.14}$$

describes the numerical flux. This method is said to be conservative in the sense that if the domain $\mathbb{R}$ is partitioned into discrete cells $[x_{j-1/2},x_{j+1/2}]$, $j=-\infty,\ldots,\infty$ with $\Delta x_j=x_{j+1/2}-x_{j-1/2}$, then

$$\begin{aligned}\sum_{j=-\infty}^{\infty}\Delta x_j\bar{\boldsymbol{U}}_j^{n+1}&=\sum_{j=-\infty}^{\infty}\Delta x_j\Big[\bar{\boldsymbol{U}}_j^n-\Delta t\Big(\frac{\mathscr{F}_{j+1/2}^n-\mathscr{F}_{j-1/2}^n}{\Delta x_j}\Big)\Big]\\ \Longrightarrow\sum_{j=-\infty}^{\infty}\Delta x_j\bar{\boldsymbol{U}}_j^{n+1}&=\sum_{j=-\infty}^{\infty}\Delta x_j\bar{\boldsymbol{U}}_j^n.\end{aligned}$$

In practice, we are usually given a finite interval $[a_x,b_x]\subset\mathbb{R}$ as the numerical domain, partitioned by N_x cells $[x_{j-1/2},x_{j+1/2}]$, $j=1,\ldots,N_x$, thus

$$\begin{aligned}\sum_{j=1}^{N_x}\Delta x_j\bar{\boldsymbol{U}}_j^{n+1}&=\sum_{j=1}^{N_x}\Delta x_j\Big[\bar{\boldsymbol{U}}_j^n-\Delta t\Big(\frac{\mathscr{F}_{j+1/2}^n-\mathscr{F}_{j-1/2}^n}{\Delta x_j}\Big)\Big]\\ \Longrightarrow\sum_{j=1}^{N_x}\Delta x_j\bar{\boldsymbol{U}}_j^{n+1}&=\sum_{j=1}^{N_x}\Delta x_j\bar{\boldsymbol{U}}_j^n-\Delta t(\mathscr{F}_{N_x+1/2}^n+\mathscr{F}_{1/2}^n)\end{aligned}$$

i.e. the new total cell averages only differ by the amount of flux leaving and entering the domain $[a_x,b_x]$.

For the numerical method (2.13) to converge, it must satisfy what is known as the CFL condition (named after Courant, Friedrichs, and Lewy). This necessary, but not sufficient, condition states that the numerical domain of dependence must contain the true domain of dependence of the partial differential equation. Thus, for the numerical method to converge, we must chose Δt small enough so that the information from the cell averages does not propagate too far. For example, consider the scalar equation for the inviscid Burgers equation

$$u_t + f(u)_x = 0.$$

The characteristics move with a wave speed of u, and thus if the flux is a function of the neighboring cell averages, i.e.

$$\mathscr{F}^n_{j+1/2} = \mathscr{F}(\bar{u}^n_j, \bar{u}^n_{j+1}), \tag{2.15}$$

then we need a time step such that

$$\nu = \frac{\Delta t \|\bar{u}\|_\infty}{\Delta x} < 1, \tag{2.16}$$

where $\|\bar{u}\|_\infty$ is the infinity vector norm for the finite volume discretization of u. This number ν is known at the Courant number, and the time step is typically chosen so that $\nu < 1$. For the general conservation law given in (2.9), the wave speeds are given by $\lambda_1 \leq \ldots \leq \lambda_{N_e}$ which are the eigen values of the Jacobian of $\boldsymbol{F}$, $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}$. We then have the CFL number given as

$$\nu = \frac{\Delta t}{\Delta x} \max_{1 \leq e \leq N_e} |\lambda_e|.$$

To illustrate the propagation speed see Fig. 2.2, and note that if the time step was too large then the characteristics would move too far away from the neighboring cells.

The main area of research for finite volume methods is determining how to numerically approximate the numerical fluxes, $\mathscr{F}^n_{j+1/2}$, $j = 0, \ldots, N_x$. To cite an example, consider a first order method and assume that the cells are of equal size, i.e. $\Delta x_j = \Delta x$ for all $j = 1, \ldots, N_x$. A naive first order approach to approximating the numerical flux may be

$$\mathscr{F}^n_{j+1/2} = \frac{1}{2}(\boldsymbol{F}(\bar{\boldsymbol{U}}^n_j) + \boldsymbol{F}(\bar{\boldsymbol{U}}^n_{j+1})), \quad j = 0, \ldots, N_x.$$

However, even for small time steps, the resulting numerical method will become unstable for hyperbolic equations.

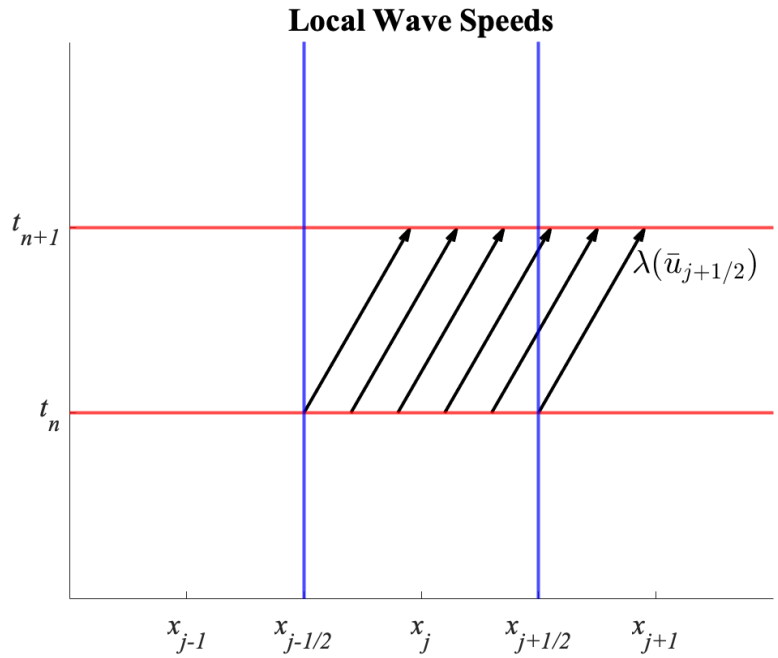

Figure 2.2: Local wave speed of characteristics.

An alternate approach, known as the Lax–Friedrichs method, is to approximate the flux values as

$$\mathscr{F}^n_{j+1/2} = \frac{1}{2}(\boldsymbol{F}(\bar{\boldsymbol{U}}^n_j) + \boldsymbol{F}(\bar{\boldsymbol{U}}^n_{j+1})) - \frac{\Delta x}{2\Delta t}(\bar{\boldsymbol{U}}_{j+1} - \bar{\boldsymbol{U}}_j), \quad j = 0, \ldots, N_x.$$

This method is stable for linear hyperbolic equations as long as the CFL condition is met but adds unnecessary numerical diffusion. Instead of using central differences to approximate the fluxes, a better approach might be to use the fact that information propagates along characteristic lines for hyperbolic equations. This leads to methods known as upwind methods. For this approach, consider the scalar Riemann problem around the cell boundary

$$\boldsymbol{U}(x,0) = \begin{cases} \boldsymbol{U}_L & x < x_{j+1/2} \\ \boldsymbol{U}_R & x > x_{j+1/2} \end{cases}. \tag{2.17}$$

For the upwind method, defining $\mathscr{F}^n_{j+1/2}$ is determined by the solution to the Riemann problem (2.17). The details in how to solve this Riemann problem is not the focus of this thesis, thus we will refer readers to [50].

There are many other approaches to approximating (2.14), such as the one we propose in Chapter 3 using artificial neural network, and the central upwind method (see Appendix C)

which is used often in this thesis.

For the case $D = 2$, $\Omega_{j,k} = [x_{j-1/2}, x_{j+1/2}] \times [y_{k-1/2}, y_{k+1/2}]$, and $\boldsymbol{F} = (\boldsymbol{F}_1, \boldsymbol{F}_2)^\top$ we get the finite volume method

$$\bar{\boldsymbol{U}}_{j,k}^{n+1} = \bar{\boldsymbol{U}}_{j,k}^{n} - \Delta t\left(\frac{\mathscr{F}_{j+1/2,k}^{n} - \mathscr{F}_{j-1/2,k}^{n}}{\Delta x} + \frac{\mathscr{G}_{j,k+1/2}^{n} - \mathscr{G}_{j,k-1/2}^{n}}{\Delta y}\right), \tag{2.18}$$

where

$$\begin{aligned}
\bar{\boldsymbol{U}}_{j,k}^{n+1} &\approx \bar{\boldsymbol{U}}_{j,k}(t_n) \\
\mathscr{F}_{j+1/2,k}^{n} &\approx \frac{1}{\Delta t \Delta y} \int_{t_n}^{t_{n+1}} \int_{y_{k-1/2}}^{y_{k+1/1}} \boldsymbol{F}_1(\boldsymbol{U}(x_{j+1/2}, y, t)\, d\, y\, d\, t \\
\mathscr{G}_{j,k+1/2}^{n} &\approx \frac{1}{\Delta t \Delta x} \int_{t_n}^{t_{n+1}} \int_{x_{j-1/2}}^{x_{j+1/1}} \boldsymbol{F}_2(\boldsymbol{U}(x, y_{k+1/2}, t)\, d\, x\, d\, t \\
\Delta x &= x_{j+1/2} - x_{j-1/2}, \quad \Delta y = y_{k+1/2} - y_{k-1/2}.
\end{aligned}$$

The two-dimensional finite volume method has similar properties as the one dimensional case discussed above (with appropriate modifications for the extra dimension). Often it is more convenient to write the above equation in the semi-discrete form

$$\frac{d}{d\,t}\bar{\boldsymbol{U}}_{j,k}(t) = -\left(\frac{\mathscr{F}_{j+1/2,k}(t) - \mathscr{F}_{j-1/2,k}(t)}{\Delta x} + \frac{\mathscr{G}_{j,k+1/2}(t) - \mathscr{G}_{j,k-1/2}(t)}{\Delta y}\right),$$

where

$$\begin{aligned}
\mathscr{F}_{j+1/2,k}(t) &\approx \boldsymbol{F}_1(\boldsymbol{U}(x_{j+1/2}, y_k, t) \\
\mathscr{G}_{j,k+1/2}(t) &\approx \boldsymbol{F}_2(\boldsymbol{U}(x_j, y_{k+1/2}, t).
\end{aligned}$$

Similarly there exist a semi-discrete form for the one dimensional case as well.

CHAPTER

3

SIMULATING PARTIAL DIFFERENTIAL EQUATION SOLUTIONS WITH THE HELP OF NEURAL NETWORKS

In this chapter we will show that neural networks can help us simulate the solutions of partial differential equations. The first two sections (Section 3.1 and Section 3.2) of this chapter will be presenting methods that use neural network to help simulate the solution of PDEs when no information about the equation is known, but instead we have solution data from previous solutions. Then, in Section 3.3 we show how a neural network is used alongside classical numerical methods to assist in simulating the solution to partial differential equations.

In Section 3.0.1 and Section 3.0.2 we will setting up the problems for The rest of this chapter.

3.0.1 Learning the Dynamics of PDEs with Neural Networks

Consider discretizing $\mathbb{R}^D$ into $N = N_1 \dots N_D$ points $\{\boldsymbol{x}_{\boldsymbol{j}_\ell}\}_{\ell=1}^N$ where

$$\boldsymbol{x}_{\boldsymbol{j}_\ell} = (x_{j_1}, \dots, x_{j_D})$$

$$\boldsymbol{j}_\ell = (j_1, \ldots, j_D)$$
$$\ell = j_1 + N_1 j_2 + \ldots j_D \Pi_{d=1}^{D-1} N_d,$$

and where $j_d = 0, \ldots, N_d - 1$, is the discretization in the dth dimension for all $d = 1, \ldots, D$. With a spatial discretization as the one above, a method of lines approach to solving time dependent PDEs, such as (2.1), involves solving a system of ordinary differential equations (ODEs) such as

$$\frac{d}{dt}\boldsymbol{U}(t) = \tilde{\boldsymbol{F}}(t, \boldsymbol{U}), \tag{3.1}$$

where $\boldsymbol{U} = [\boldsymbol{U}_{j_1}, \ldots, \boldsymbol{U}_{j_N}]^\top$ and $\tilde{\boldsymbol{F}} = [\tilde{\boldsymbol{F}}_{j_1}, \ldots, \tilde{\boldsymbol{F}}_{j_N}]^\top$. Typically, $\boldsymbol{U}_{j_\ell}(t)$ represents an approximation to the function value $\boldsymbol{U}(\boldsymbol{x}_{j_\ell}, t)$ for all $\ell = 1, \ldots, N$, and $\tilde{\boldsymbol{F}}_{j_\ell}$ depends on the value $\boldsymbol{U}_{j_\ell}$ and its numerical derivatives. We have $\boldsymbol{U}_{j_\ell} \in \mathbb{R}^{\tilde{N}_e}$ for each $l = 1, \ldots, N$, where typically $\tilde{N}_e = N_e$ from (2.1), but we make the distinction for when (3.1) is representing a particle method, in which case it represents the number of equations associated with each particle (see the Camassa-Holm equation in Section 3.1.1).

Solutions to (3.1) are often described using the flow map $\Phi : \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}^N$ such that

$$\boldsymbol{U}(t) = \Phi_t(t_0, \boldsymbol{U}^0), \tag{3.2}$$

where $\boldsymbol{U}^0 = \boldsymbol{U}(t_0)$. We will consider the case when (3.1) is an autonomous system (i.e. $\tilde{\boldsymbol{F}}$ does not explicitly depend on t), thus we can omit the input variable t_0 in the flow map (3.2). The flow map has the property that

$$\Phi_{t_2}(\boldsymbol{U}^0) = \Phi_{t_2}(\Phi_{t_1}(\boldsymbol{U}^0)),$$

where $t_2 > t_1$. Many numerical methods are aimed at trying to approximate this flow map, resulting in a numerical method

$$\boldsymbol{U}^{n+1} = \tilde{\Phi}_{t_{n+1}}(\boldsymbol{U}^n)$$

such that

$$\tilde{\Phi}_{t_{n+1}}(\boldsymbol{U}^n) = \Phi_{t_{n+1}}(\boldsymbol{U}^n) + \boldsymbol{\varepsilon}^{n+1},$$

where $\boldsymbol{U}^n = [\boldsymbol{U}_1(t_n), \ldots, \boldsymbol{U}_N(t_n)]^\top$, and $\boldsymbol{\varepsilon}^{n+1}$ depends on how $\tilde{\Phi}$ is defined and the step $\Delta t_n = t_{n+1} - t_n$.

Many numerical solutions to PDEs are aimed at approximating the flow map (3.2) described above, including finite difference, finite volume, finite element, and particle methods. With these methods, we can achieve small values for $\|\boldsymbol{\varepsilon}^n\| \ll 1$, $n = 1, \ldots, N_t$ using a numerical method, $\tilde{\Phi}_{t_{n+1}}(\boldsymbol{U}^n)$, which is continuous with respect to its inputs. This motives us to be able to train a

neural network N_Θ such that

$$N_\Theta(\boldsymbol{U}^n) \approx \Phi_{t_{n+1}}(\boldsymbol{U}^n)$$

for a constant time step $\Delta t = t_{n+1} - t_n$ for all $n = 0, \ldots, N_t - 1$, as is done in Section 3.2, or train the neural network N_Θ such that

$$N_\Theta(\boldsymbol{U}^0) \approx [\Phi_{t_1}(\boldsymbol{U}^0), \ldots, \Phi_{t_{N_t}}(\boldsymbol{U}^0)]$$

as is done in Section 3.1.

3.0.2 Training the Neural Network

To train an artificial neural network that can approximate the solution to (3.1) at discrete time steps

$$t_{n+1} = t_n + \Delta t_n; \quad n = 0, 1, 2, ..., N_t - 1,$$

where $t_0 = 0$ and Δt_n may or may not be constant, we need to have a set of data to learn from. Since we do not have data readily available to us, we create our own data. To train the networks, we assume that the initial condition to (3.1) is parameterized by $\hat{\boldsymbol{p}} = [\hat{p}_1, \ldots, \hat{p}_{N_p}]$, where each $\hat{p}_l \in [\alpha_l, \beta_l]$ for some $\alpha_l, \beta_l \in \mathbb{R}$, and N_p is the number of parameters. We then randomly select N_s parameters $\{\hat{\boldsymbol{p}}^i\}_{i=1}^{N_s}$ to get $\boldsymbol{U}(0; \hat{\boldsymbol{p}}^i)$ such that each element $\hat{p}_l^i$ comes from the uniform distribution $\mathscr{U}[\alpha_l, \beta_l]$. We run highly accurate simulations to get

$$\boldsymbol{U}_{j_\ell}^{n,i} \approx \boldsymbol{U}_{j_\ell}(t_n; \hat{\boldsymbol{p}}^i), \quad n = 0, \ldots, N_t, \quad i = 1, 2, ..., N_s$$

for some values of $\ell \in \{1, \ldots, N\}$ (possibly for all ℓ but not always). Using this data, we train our neural networks to simulate (3.1) when new parameters are given as the initial condition.

For each example, there is the initial cost of training the neural network that we need to consider. This cost is usually large, but only has to be considered once. We will describe the cost for each example in this chapter in their respective sections. After the neural network is trained, the only computational cost is in running an example with the network. The neural network solutions are run in Python using the PyTorch machine learning library and we compare them to those obtained by traditional numerical methods in Python using NumPy arrays. All the one dimensional neural network examples run on the CPU and the two dimensional example is tested on a CPU and GPU. We tested the one dimensional examples using the GPU as well but it did not make much of a difference in the run time thus we omit them. We believe comparing the run times of the solutions with these Python implementations is better than comparing our neural

network solution to a solution ran using a compiled language such as FORTRAN, as we are running the examples with more similar software tools. Note that there is still differences in software tools, thus the run times should only be viewed as rough comparison between the two method, and not as an absolute truth of which method is faster. Also, since we are generating large amounts of data, we use FORTRAN to create the data as it is much faster than Python, except for the Camassa-Holm equation in Section 3.1.1, where the training data is generated using python.

A more detailed explanation about how the data is collected will be given in the each method's respective section or for the specific examples, however we shall mention one approach that we use in many of the examples to collect accurate simulation data.

Consider the case when $D = 1$ and the numerical domain is $[a_x, b_x]$. To simulate an accurate solution to a PDE, we discretize the domain into $\tilde{N}_x = m_f N_x$ uniform cells $[\tilde{x}_{j-1/2}, \tilde{x}_{j+1/2}]$, where $\tilde{x}_{j+1/2} = \frac{\tilde{x}_{j+1} - \tilde{x}_j}{2}$,

$$\tilde{x}_{\tilde{j}} = a_x - \frac{\Delta \tilde{x}}{2} + \tilde{j}\Delta \tilde{x}, \quad j = 0, \ldots, \tilde{N}_x + 1$$

m_f is an integer larger than 1, and $\Delta \tilde{x} = \frac{b_x - a_x}{\tilde{N}_x}$. We then simulate the PDE solution on this fine grid discretization and from this simulation, collect the cell averages

$$\tilde{U}_{\tilde{j}}^n \approx \frac{1}{\Delta \tilde{x}} \int_{\tilde{x}_{\tilde{j}-1/2}}^{\tilde{x}_{\tilde{j}+1/2}} U(x, t_n)\, dx, \quad \tilde{j} = 1, \ldots, \tilde{N}_x,$$

and at every time step $t = n\Delta t$, $n = 0, \ldots, N_t$, for some constant time step Δt. We let

$$\bar{U}_j^n = \frac{1}{m_f} \sum_{l=1}^{m_f} \tilde{U}_{m_f(j-1)+l},$$

and use $\bar{U}_j^n$ to train the neural network. Note that if

$$\tilde{U}_{\tilde{j}}^n = \frac{1}{\Delta \tilde{x}} \int_{\tilde{x}_{\tilde{j}-1/2}}^{\tilde{x}_{\tilde{j}+1/2}} U(x, t_n)\, dx + \mathcal{O}(\Delta \tilde{x}^2), \quad \tilde{j} = 1, \ldots, \tilde{N}_x$$

then

$$\bar{U}_j^n = \frac{1}{\Delta x} \int_{x_{j-1/2}}^{x_{j+1/2}} U(x, t_n)\, dx + \mathcal{O}(\Delta \tilde{x}^2), \quad j = 1, \ldots, N_x$$

where $\Delta x = m_f \Delta \tilde{x}$ and $x_{j+1/2} = \tilde{x}_{m_f j+1/2}$.

Remark 3.0.1. *This fine grid discretization result can be generalized to higher spatial dimensions.*

3.1 One Step Neural Network

For our first approach, we train the neural network $\boldsymbol{N}_\Theta : \mathbb{R}^{N\times\tilde{N}_e} \to \mathbb{R}^{N\times\tilde{N}_e\times N_t}$ such that

$$\boldsymbol{N}_\Theta(\boldsymbol{U}^0) \approx [\boldsymbol{U}^1, \boldsymbol{U}^2, ..., \boldsymbol{U}^{N_t}], \tag{3.3}$$

where $\boldsymbol{U}^n = \boldsymbol{U}(t_n) \in \mathbb{R}^{N\times\tilde{N}_e}$, $t_n = n\Delta t$ for $n = 0, 1, 2, ..., N_t$ and some constant Δt, and Θ is the internal parameter set of the neural network (Note that here we use $[\boldsymbol{U}^1, \boldsymbol{U}^2, ..., \boldsymbol{U}^{N_t}]$ to represent a 3-mode tensor). Thus, this method takes in the array of initial values to the system (3.1) and returns an approximation to $\boldsymbol{U}(t)$ at multiple discrete time steps all at once. We assume that the initial condition in (3.1) is a known function that is parameterized by some set $\hat{\boldsymbol{p}}$ such that $\boldsymbol{U}(0) = \boldsymbol{U}(0;\hat{\boldsymbol{p}})$ and we randomly select N_s sets of parameters $\{\hat{\boldsymbol{p}}^i\}_{i=1}^{N_s}$ from a multivariate uniform distribution, such that $\boldsymbol{U}^{0,i}$ corresponds to $\boldsymbol{U}(0;\hat{\boldsymbol{p}}^i)$. We then collect data

$$\boldsymbol{U}^{n,i} \approx [\boldsymbol{U}_{j_1}(t_n;\hat{\boldsymbol{p}}^i), \ldots, \boldsymbol{U}_{j_N}(t_n;\hat{\boldsymbol{p}}^i)]$$

for all $n = 0, 1, ..., N_t$, and $i = 1, 2, ..., N_s$. To train the neural network, we choose the following loss function:

$$L_\Theta = \sum_{m=1}^{M} \|\boldsymbol{N}_\Theta(\boldsymbol{U}^{0,m}) - [\boldsymbol{U}^{1,m}, ..., \boldsymbol{U}^{N_t,m}]\|_F^2, \tag{3.4}$$

where $\|\cdot\|_F$ is the Frobenius norm, and try to find the parameter set Θ that minimizes (3.4).

3.1.1 Numerical Examples

The Camassa-Holm equation:

First, consider the Camassa-Holm equation, given by

$$\begin{aligned} &m_t + v\,m_x + 2m\,v_x = 0, \quad m = v - v_{xx} \\ &m(x,0) = m_0(x). \end{aligned} \tag{3.5}$$

where m is the momentum, and v is the fluid velocity. In [10; 11; 12] it was shown that we can approximate peakon solutions to this equation using the particle method. To do this, we look at solutions in the form

$$m^{N_x}(x,t) = \sum_{j=1}^{N_x} p_j(t)\delta(x - x_j(t)) \tag{3.6}$$

where N_x is the total number of particles, δ is the Dirac delta distribution, and $x_j(t)$ and $p_j(t)$ represent the location and weight of the jth particle respectively. This leads to solving a system of ODEs

$$\begin{cases} \dfrac{d\,x_j(t)}{d\,t} = v(x_j(t),t), & j=1,2,..,N_x \\ \dfrac{d\,p_j(t)}{d\,t} + v_x(x_j(t),t)p_j(t) = 0, & j=1,2,..,N_x. \end{cases} \tag{3.7}$$

Using (3.5) and (3.6) we have

$$v(x,t) = G * m = \frac{1}{2}\sum_{j=1}^{N} p_j(t)e^{-|x-x_j(t)|}, \tag{3.8}$$

where G is the Green's function

$$G(|x-y|) = \frac{1}{2}e^{-|x-y|}.$$

We train our neural network to approximate (3.7) with 2 particles ($N_x = 2$). To this end, we use the initial data $x_1(0), x_2(0) \sim \mathscr{U}[-2,7)$ and $p_1(0), p_2(0) \sim \mathscr{U}[0,6)$. Using the particle method, we then generate the data needed for the neural network by solving the system (3.7) up to time $t = 10$, collecting the data $x_j(t_n)$ and $p_j(t_n)$, where $t_n = 0.1n$, $n = 0,...,100$. For this neural network we use

$$\boldsymbol{U}^n = \begin{bmatrix} x_1(t_n) & p_1(t_n) \\ x_2(t_n) & p_2(t_n) \end{bmatrix}$$

and use a fully connected feedforward neural network with $H = 6$ hidden layers, each with 500 nodes, and the ReLU activation function (2.4) for each layer except the last, which has no activation function. The loss function we use is given in (3.4), and to train the neural network we use the Adam optimization algorithm [43] with an initial learning rate of 10^{-4}. We split the data set into batches of size 30 and then train the neural network for 20 epochs. The time to train the neural network was 627 seconds using a NVIDIA GeForce RTX 3060 laptop GPU. We represent the architecture for this neural network in Fig. 3.1, where each activation function is given as

$$\sigma_\eta = \begin{cases} ReLU & \eta = 1,\ldots,6 \\ I & \eta = 7 \end{cases}.$$

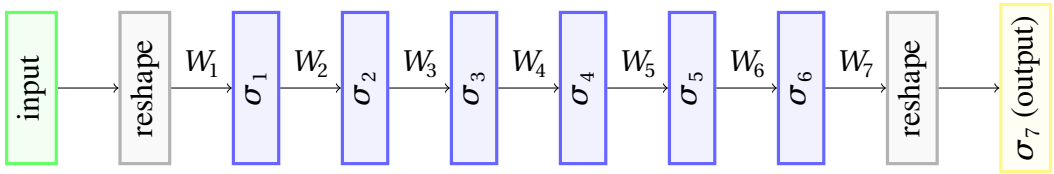

Figure 3.1: Architecture for the one-step neural network to solve the Camassa-Holm equation.

Remark 3.1.1. *To apply the neural network to any input matrix, we need to reshape the array into a vector with the same number of elements. Similarly, the output vector can be reshaped into a multidimensional array with the same number of outputs. With the notation of* $\boldsymbol{N}_\Theta : \mathbb{R}^{m_1 \times n_1} \to \mathbb{R}^{m_2 \times n_2}$*, the reshaping of these vectors and matrices are part of the neural network* $\boldsymbol{N}_\Theta$*.*

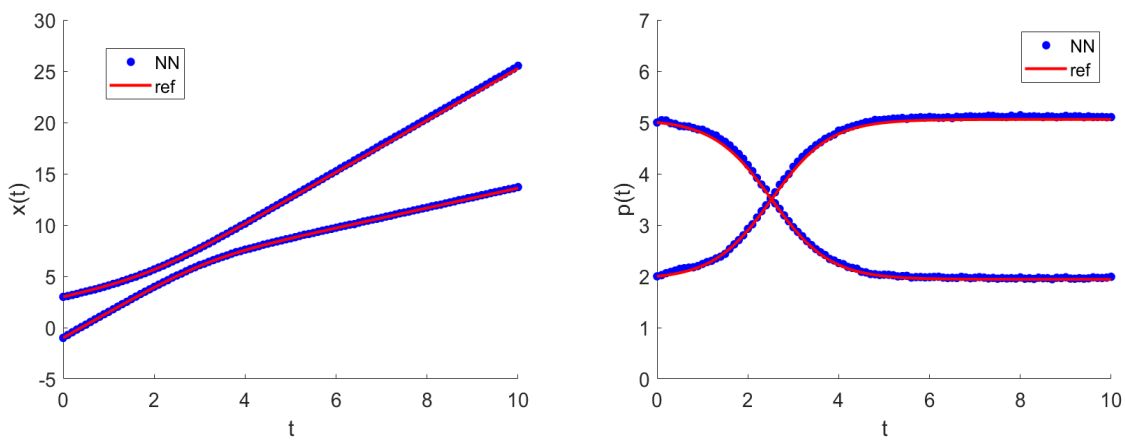

Figure 3.2: Comparison between the neural network approximation and the particle method solution for $x_j(t)$ (left) and $p_j(t)$ (right), $j = 1, 2$.

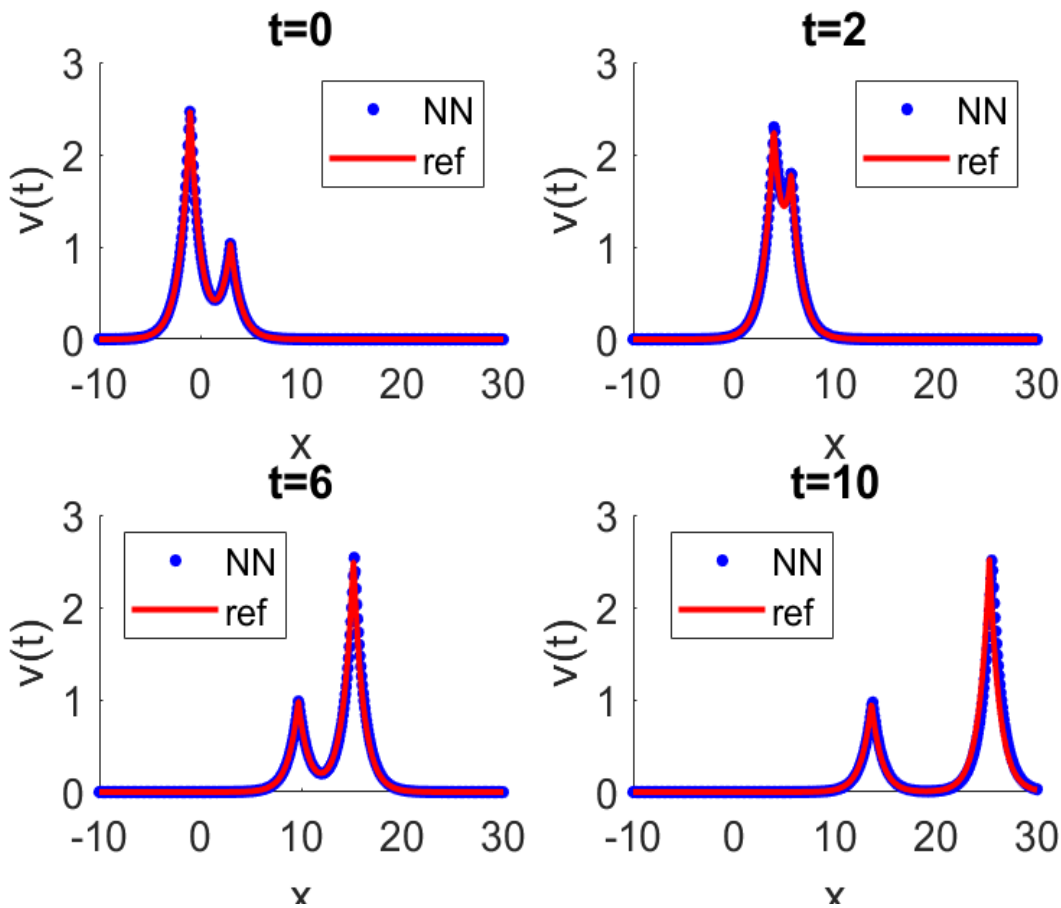

Figure 3.3: Comparison between the neural network approximation and the particle method solution for $v(x,t)$ at time $t=0$, $t=2$, $t=6$ and $t=10$.

In Fig. 3.2 and Fig. 3.3 we show the results from the trained network compared to the solution obtained by the particle method. The initial data is $[x_1(0), x_2(0), p_1(0), p_2(0)]^T = [-1, 3, 5, 2]^T$. As we can see, the result from the trained network look very similar to the results from the particle method. The particle method was implemented using Python and the NumPy package. The neural network solution took 0.0073 seconds to run while the particle method took 0.0237 seconds to run. We are able to obtain the solution fast using the neural network solution since the neural network is simple enough and computes all the steps at once.

The shallow water equations:

Next, we simulate the shallow water equations

$$h_t + (hu)_x = 0$$
$$(hu)_t + (hu^2 + \frac{1}{2}h^2)_x = 0$$

subject to the initial condition

$$h(x,0) = a e^{-bx^2} + 1 + c\cos(x), \quad u(x,0) = -k\frac{\partial h}{\partial x}(x),$$

where $a, b \in [0,2)$, and $c \in [0, 0.1)$, and $k \in [0,1)$. Here, h and u represent the depth and velocity of the water respectively. Let the conserved variable $q(x,t) = (hu)(x,t)$ be the momentum of the water. For the neural network, let $\bar{U}_j^n = [\bar{h}_j^n, \bar{q}_j^n]^\top$ be the cell averages of $[h, q]$ over the cells $[x_{j-1/2}, x_{j+1/2}]$, for $j = 1, ..., N_x$ and $n = 0, \ldots, N_t$. Here, $x_{j+1/2} = \frac{x_j + x_{j+1}}{2}$, where

$$x_j = -20 - \frac{\Delta x}{2} + j\Delta x, \quad j = 0, \ldots N_x + 1,$$

is the discritization of $[-20, 20]$ into $N_x = 100$ cells with $\Delta x = \frac{40}{N_x}$, and $t_n = n\Delta t$ is the uniform time discritization.

To collect data for the neural network, we take $a, b \sim \mathscr{U}[0,2)$, $c \sim \mathscr{U}[0,.1)$, and $k \sim \mathscr{U}[0,1)$ and simulate the solution on a fine grid with $\tilde{N}_x = 1000$ (see Section 3.0.2). We then simulate the solution for $M = 100{,}000$ examples, splitting the data set in $M_{tr} = 75{,}000$ and $M_{val} = 25{,}00$ training and validation sets respectively. At every $t = n\Delta t$, $n = 0, \ldots, N_t = 100$, where $\Delta t = 0.1$, we collect the data $\bar{U}^n$.

For this example, we use a neural network with $H = 6$ hidden layers. All of the connections between the layers are fully connected with 400 nodes except the connection between the 2nd and 3rd layers and the connection between the 5th and 6th layers, which are convolutional layers with a kernel size of 3. The activation function for the 1st and 4th layers is the ReLU activation function (2.4), the activation function for the 2nd and 5th layers is our own semi-step activation function (2.8), and the 3rd and 6th hidden layers do not have an activation function (or equivalently, the identity function). We train the neural network using the Adam optimizer with an initial learning rate of 10^{-4}. To train, we split the training data into batches of size 30 and train the network for 10 epochs. The time to train the neural network was 1622 seconds using a NVIDIA GeForce RTX 3060 laptop GPU. Note that we need to reshape the input and output of

the neural network so that there is a consistency with the array sizes (see Remark 3.1.1). The neural network architecture is represented in Fig. 3.4, where the activation function for each layer is given as

$$\sigma_\eta = \begin{cases} ReLU & \eta = 1,4 \\ f_{ss} & \eta = 2,5 \\ I & \eta = 3,6,7 \end{cases}.$$

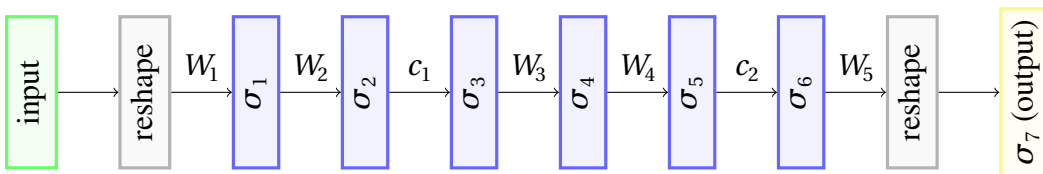

Figure 3.4: Architecture for the one-step neural network to solve the shallow water equation.

In Fig. 3.5 we look at how well the neural network approximated the function $\bar{h}(x,t)$ and $\bar{q}(x,t)$ at t=10 with the initial conditions

$$h(x,0) = 1.9e^{-1.1x^2} + 1 + .09\cos(x)$$
$$u(x,0) = -.7(-4.18xe^{-1.1x^2} - .09\sin(x)),$$

and compare it to solutions obtained by the second order central upwind scheme. The central upwind solutions in Fig. 3.5 and Table 3.1 below are computed on a grid with 100 cells, just like the neural network solution. The reference solution is also computed with the second order central upwind method but on a fine grid with 30,000 cells. As we can see from the plot and the table, the neural network solution is able to approximate the solution more accurately than the first and second order central upwind method, where the accuracy is determined by the relative error given in (3.9). The run time for this neural network solution is 0.074 seconds, whereas the run times for the first and second order central upwind methods are 0.033 seconds and 0.44 seconds respectively.

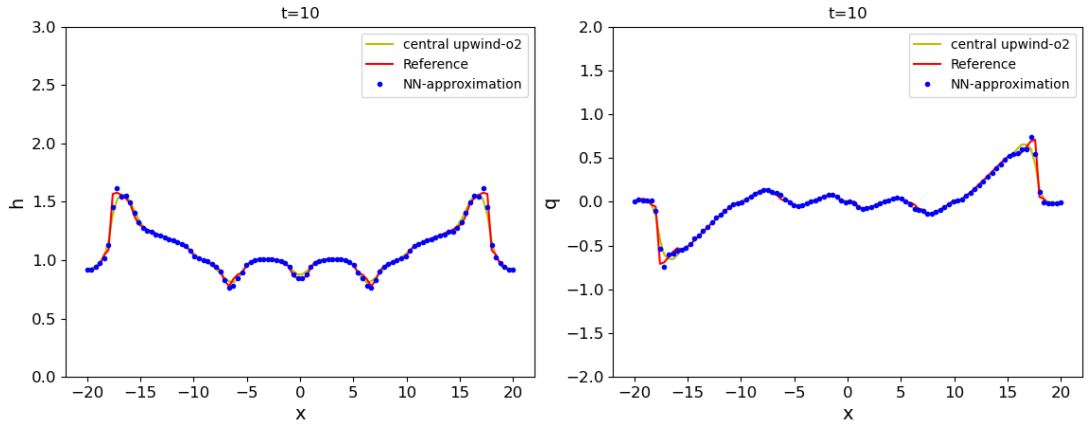

Figure 3.5: *Left:* Cell average for $\bar{h}$ at time $t = 10$. *Right:* Cell average for $\bar{q}$ at time $t = 10$.

Table 3.1: l_1-norm relative error table: 1D shallow water equations.

t	Neural Network		First Order Central Upwind		Second Order Central Upwind	
	$\bar{h}$	$\bar{q}$	$\bar{h}$	$\bar{q}$	$\bar{h}$	$\bar{q}$
2	0.0067	0.0444	0.0432	0.3817	0.0211	0.2076
4	0.0066	0.0387	0.0506	0.3475	0.0192	0.1542
6	0.0055	0.0356	0.0530	0.4150	0.0181	0.1553
8	0.0072	0.0493	0.0555	0.4184	0.0183	0.1307
10	0.0087	0.0636	0.0647	0.4325	0.0171	0.1337

The l^1 relative error at time t_n is defined as

$$E_1(u^n) = \frac{\|u^n - u^n_{ref}\|_{l^1}}{\|u^n_{ref}\|_{l^1}} \tag{3.9}$$

where the l^1 norm is

$$\|u\|_{l^1} = \Delta x \sum_{j=1}^{N_x} |u_j|$$

and u^n_{ref} is the reference solution. The relative error at different times t for this example are

given in Table 3.1. Note that while our method is able to produce a more accurate result on this grid, our method is limited to this grid size unless we train a new neural network. In such, we are not able to refine our grid and get a more accurate solution.

3.2 Time Stepping Neural Network

In this section, we introduce a time stepping neural network to numerically solve (3.1). For this method, note that if we integrate both sides of (3.1) in time from t_n to $t_n + \Delta t$ we get

$$\boldsymbol{U}_{j_\ell}(t_n + \Delta t) = \boldsymbol{U}_{j_\ell}(t_n) + \int_{t_n}^{t_n+\Delta t} \boldsymbol{F}_{j_\ell}(\boldsymbol{U}(t))\, d\, t. \tag{3.10}$$

Our goal is to find a neural network $\boldsymbol{N}_\Theta$ such that

$$\boldsymbol{N}_\Theta(\boldsymbol{U}_{j_\ell,sten}(t_n)) \approx \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \boldsymbol{F}_{j_\ell}(\boldsymbol{U}(t))\, d\, t, \qquad \ell = 1, 2, ..., N, \tag{3.11}$$

for some stencil $\boldsymbol{U}_{j_\ell,sten}$ around $\boldsymbol{U}_{j_\ell}$, which leads to the numerical method

$$\boldsymbol{U}_{j_\ell}^{n+1} = \boldsymbol{U}_{j_\ell}^{n} + \Delta t\, \boldsymbol{N}_\Theta(\boldsymbol{U}_{j_\ell,sten}^{n}), \qquad \ell = 1, 2, ..., N. \tag{3.12}$$

Remark 3.2.1. *If* (3.11) *is exact for all* $l = 1, 2, ..., N$*, then the method is an exact time stepping method. Also since this method is trying to approximate the exact integral value and not the limiting case, this method is not a first order numerical method in time i.e. decreasing the time step may not increase the accuracy.*

In particular, we will use use this time stepping method to numerically solve systems of equations given by

$$\boldsymbol{U}_t + \sum_{d=1}^{D} \boldsymbol{F}_d(\boldsymbol{U})_{x_d} = \sum_{d=1}^{D} \boldsymbol{\varepsilon}_d(\boldsymbol{U}_{x_d})_{x_d}, \quad x \in \mathbb{R}^D,\ t \geq 0, \tag{3.13}$$

where $\boldsymbol{U}(x,t) = (u_1(x,t), \ldots, u_{N_e}(x,t))^\top$, $\boldsymbol{F}_d : \mathbb{R}^{N_e} \to \mathbb{R}^{N_e}$ are the flux functions, and $\boldsymbol{\varepsilon}_d : \mathbb{R}^{N_e} \to \mathbb{R}^{N_e}$ are the diffusion terms for all $d = 1, \ldots, D$. Let

$$\boldsymbol{H}_d(\boldsymbol{U}) = \boldsymbol{F}_d(\boldsymbol{U}) - \boldsymbol{\varepsilon}_d(\boldsymbol{U}_{x_d}), \qquad d = 1, \ldots, D,$$

then we can rewrite (3.13) as

$$\boldsymbol{U}_t + \sum_{d=1}^{D} \boldsymbol{H}_d(\boldsymbol{U})_{x_d} = 0.$$

Consider the case when $D = 1$. We partition the spatial domain $[a_x, b_x]$ into uniform cells $[x_{j-1/2}, x_{j+1/2}]$ of size $\Delta x = \frac{b_x - a_x}{N_x}$ for all $j = 1, \ldots, N_x$, and where the center of each cell is given as

$$x_j = a_x - \frac{\Delta x}{2} + j\Delta x, \quad j = 1, \ldots, N_x.$$

Consider the finite volume framework in Section 2.2 with the numerical fluxes given as

$$\mathscr{F}^n_{j+1/2} \approx \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \boldsymbol{H}(\boldsymbol{U}(x_{j+1/2}, t))\, d\, t$$

and the cell averages

$$\bar{\boldsymbol{U}}^n_j \approx \int_{x_{j-1/2}}^{x_{j+1/2}} \boldsymbol{U}(x, t_n)\, d\, x.$$

We train a neural network $\boldsymbol{F}_\Theta : \mathbb{R}^{N_f \times N_e} \to \mathbb{R}^{N_e}$ on some stencil of size N_f such that

$$\boldsymbol{F}_\Theta(\bar{\boldsymbol{U}}^n_{j+N_f/2}, ..., \bar{\boldsymbol{U}}^n_{j-N_f/2+1}) \approx \mathscr{F}^n_{j+1/2}, \tag{3.14}$$

where Θ is the neural networks internal parameter set. Denote

$$\hat{\boldsymbol{F}}^n_{j+1/2} := \boldsymbol{F}_\Theta(\bar{\boldsymbol{U}}^n_{j+N_f/2}, ..., \bar{\boldsymbol{U}}^n_{j-N_f/2+1}), \quad j = 0, \ldots, N_x,$$

to train our network, we let

$$\hat{\boldsymbol{U}}^{n+1}_j(\Theta) = \bar{\boldsymbol{U}}^n_j - \frac{\Delta t}{\Delta x}[\hat{\boldsymbol{F}}^n_{j+1/2} - \hat{\boldsymbol{F}}^n_{j-1/2}] \tag{3.15}$$

and try to find the parameter set Θ such that $\hat{\boldsymbol{U}}^{n+1}_j \approx \bar{\boldsymbol{U}}^{n+1}_j$. Note that this is in the same form as (3.12) with

$$\boldsymbol{N}_\Theta(\boldsymbol{U}^n_{\boldsymbol{J}_l, sten}) = -\frac{1}{\Delta x}[\hat{\boldsymbol{F}}^n_{j+1/2} - \hat{\boldsymbol{F}}^n_{j-1/2}].$$

In two dimension, we discretize the domain $[a_x, b_x] \times [a_y, b_y]$ into uniform cells $[x_{j-1/2}, x_{j+1/2}] \times [y_{k-1/2}, y_{k+1/2}]$ for $j = 1, \ldots, N_x$ and $k = 1, \ldots, N_y$, and where the center of each cell is $[x_j, y_k]$,

$$x_j = a_x - \frac{\Delta x}{2} + j\Delta x, \quad j = 1, \ldots, N_x,$$

$$y_k = a_y - \frac{\Delta y}{2} + k\Delta y, \quad k = 1, \ldots, N_y,$$
$$\Delta x = \frac{b_x - a_x}{N_x}, \qquad \Delta y = \frac{b_y - a_y}{N_y}.$$

We use the finite volume framework to compute the solution with the numerical fluxes

$$\mathscr{F}^n_{j+1/2,k} \approx \frac{1}{\Delta t \Delta y} \int_{t_n}^{t_{n+1}} \int_{y_{k-1/2}}^{y_{k+1/2}} \boldsymbol{H}_1(\boldsymbol{U}(x_{j+1/2}, y, t))\, d\, y\, d\, t,$$
$$\mathscr{G}^n_{j,k+1/2} \approx \frac{1}{\Delta t \Delta x} \int_{t_n}^{t_{n+1}} \int_{x_{j-1/2}}^{x_{j+1/2}} \boldsymbol{H}_2(\boldsymbol{U}(x, y_{k+1/2}, t)\, d\, x\, d\, t.$$

Neural networks $\boldsymbol{F}_\Theta : \mathbb{R}^{N_f \times N_e} \to \mathbb{R}^{N_e}$ and $\boldsymbol{G}_\Theta : \mathbb{R}^{N_f \times N_e} \to \mathbb{R}^{N_e}$ are trained such that for some stencils of size N_f

$$\begin{aligned} \boldsymbol{F}_\Theta(\bar{\boldsymbol{U}}^n_{j+N_f/2,k}, \ldots, \bar{\boldsymbol{U}}^n_{j-N_f/2+1,k}) &\approx \mathscr{F}^n_{j+1/2,k}, \\ \boldsymbol{G}_\Theta(\bar{\boldsymbol{U}}^n_{j,k+N_f/2}, \ldots, \bar{\boldsymbol{U}}^n_{j,k-N_f/2+1}) &\approx \mathscr{G}^n_{j,k+1/2}, \end{aligned} \tag{3.16}$$

where Θ is the neural networks internal parameter set. Denote

$$\hat{\boldsymbol{F}}^n_{j+1/2,k} := \boldsymbol{F}_\Theta(\bar{\boldsymbol{U}}^n_{j+N_f/2,k}, \ldots, \bar{\boldsymbol{U}}^n_{j-N_f/2+1,k}),$$
$$\hat{\boldsymbol{G}}^n_{j,k+1/2} := \boldsymbol{G}_\Theta(\bar{\boldsymbol{U}}^n_{j,k+N_f/2}, \ldots, \bar{\boldsymbol{U}}^n_{j,k-N_f/2+1}),$$

to train our network, we let

$$\hat{\boldsymbol{U}}^{n+1}_{j,k}(\Theta) = \bar{\boldsymbol{U}}^n_{j,k} - \Delta t \Big[\frac{\hat{\boldsymbol{F}}^n_{j+1/2,k} - \hat{\boldsymbol{F}}^n_{j-1/2,k}}{\Delta x} + \frac{\hat{\boldsymbol{G}}^n_{j,k+1/2} - \hat{\boldsymbol{G}}^n_{j,k-1/2}}{\Delta y} \Big], \tag{3.17}$$

and try to find the parameter set Θ such that $\hat{\boldsymbol{U}}^{n+1}_{j,k} \approx \bar{\boldsymbol{U}}^{n+1}_{j,k}$.

To collect data for this method, we draw the parameters $\{\hat{\boldsymbol{p}}^i\}_{i=1}^{N_s}$ from a multivariate uniform distribution to define our initial conditions for N_s simulations, then run the simulation with a highly accurate solver to collect data at discrete locations. For the data collection locations, we assume there are 'sensors' in our spatial domain, which collect stencil data located around discrete points. In one dimension, let $\{x_{j_s}\}_{s=1}^S$ be the S sensor locations. Then we collect the data

$$[\boldsymbol{U}_{j_s - N_f/2}, \ldots, \boldsymbol{U}_{j_s + N_f/2}], \qquad s = 1, \ldots, S.$$

And for two dimensions, let $\{(x_{j_{s_1}}, y_{k_{s_2}}\}_{s_1,s_2=1}^{S_1,S_2}$ be the $S = S_1 S_2$ sensor locations. Then we collect

the data

$$\boldsymbol{U}_{j_{s_1}+\alpha,k_{s_2}+\beta},$$
$$\alpha,\beta=-N_f/2,\ldots,N_f/2, \qquad s_1=1,\ldots,S_1, \quad s_2=1,\ldots,S_2.$$

These sensors are visualized in Fig. 3.6 with the one dimensional case on the left and the two dimensional case on the right. Once we have the data, we train the neural networks to minimize a loss function of the form

$$L_\Theta=\sum_{m=1}^{M}\|\hat{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{\boldsymbol{j}_{\ell_m}}-\bar{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{\boldsymbol{j}_{\ell_m}}\|_p^p+\mu\sum_{m=1}^{M}\|\hat{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{\boldsymbol{j}_{\ell_m}}\|_p^p, \tag{3.18}$$

where $p\in\{1,2\}$, $\|\cdot\|_p$ is the l_p norm, and $0\le\mu\ll 1$. The term on the right is a regularization term used to help stabilize the numerical method. Here, $\hat{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m}$ and $\bar{\boldsymbol{U}}^{n_m+1,\hat{\boldsymbol{p}}^{i_m}}_{j_m}$ correspond to the neural network solution and reference solution respectively from the mth data in our data set, where $n_m\in\{0,\ldots,N_t-1\}$, $\ell_m\in\{1,\ldots,N\}$, and $i_m\in\{1,\ldots,N_s\}$.

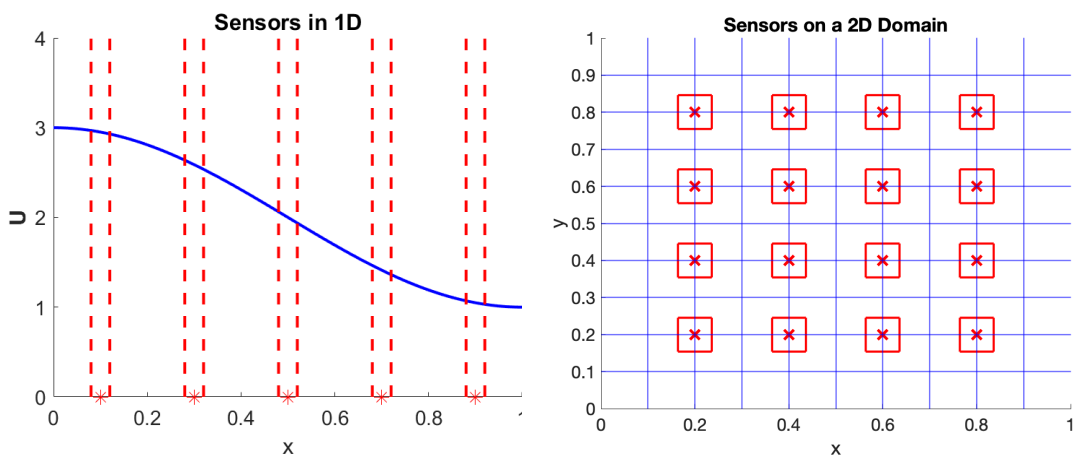

Figure 3.6: *Left:* Sensors for one dimension. *Right:* Sensor locations for two dimensional domain.

3.2.1 Numerical Examples

Here we will show numerical solutions to partial differential equations using the method described above. We apply the method to three one dimensional problems, showing that the method can work for scalar equations as well as both linear and nonlinear systems of equations.

In the last example, will show that the method can be applied to a two dimensional non-linear system of equations. For each of these examples, we will compare the solution from our method to a solution from a second order finite volume method on the same grid. For this we will compare the accuracy using the l_p error defined as

$$E_p(\boldsymbol{U}^n) = \frac{\|\boldsymbol{U}^n - \boldsymbol{U}^n_{ref}\|_p}{\|\boldsymbol{U}^n_{ref}\|_p}, \tag{3.19}$$

where in one dimension

$$\|\boldsymbol{U}^n\|_p = \left(\Delta x \sum_{j=1}^{N_x} |\boldsymbol{U}^n_j|^p\right)^{\frac{1}{p}},$$

and in two dimensions

$$\|\boldsymbol{U}^n\|_p = \left(\Delta x \Delta y \sum_{k=1}^{N_y} \sum_{j=1}^{N_x} |\boldsymbol{U}^n_{j,k}|^p\right)^{\frac{1}{p}}.$$

To train the neural networks in this section, we use a feedforward neural network that has both fully connected and convolutional layers. In each section, we show a representation of the architecture for the given problem. For each neural network, each convolutional layer has a kernel of length 5. We use $M = 200{,}000$ data points for the neural networks, splitting it into training, testing, and validation sets with sizes of $M_{train} = 120{,}000$, $M_{test} = 40{,}000$, and $M_{val} = 40{,}000$ for each of the respective sets. During the training procedure, we use a data batch size of 32, and train the neural networks for 1000 epochs using the Adam optimizer [64] with an initial learning rate of 10^{-4}. The only thing that changes between each example is the number of nodes for each fully connected layer, the number of inputs and outputs for the neural network, and the parameters p and μ for the loss function (3.18). We will also state the time it took to train each example in this section.

Once the neural network is trained, to find all the flux values for the time stepping method we treat the whole array of solution values and the appropriate stencils as an input batch to the neural network. For instance, if we have $\boldsymbol{U}^n = [\bar{\boldsymbol{U}}^n_1, \bar{\boldsymbol{U}}^n_2, \ldots, \bar{\boldsymbol{U}}^n_{N_x}]$, to find the flux values for the next numerical method we input

$$[\boldsymbol{U}^n_{1/2,sten}, \boldsymbol{U}^n_{3/2,sten}, \ldots, \boldsymbol{U}^n_{N_x+1/2,sten}]$$

as a single batch into the neural network, where

$$\boldsymbol{U}^n_{j+1/2,sten} = [\bar{\boldsymbol{U}}_{j-N_f/2+1}, \ldots, \bar{\boldsymbol{U}}_{j+N_f/2}]^\top, \quad j = 0, \ldots, N_x.$$

Note, we use the appropriate numerical boundary conditions (such as periodic or wall boundary conditions) depending on the problem for the stencil values outside our numerical domain.

Burgers Equation:

Consider the viscous Burgers equation

$$u_t + (\frac{u^2}{2})_x = \varepsilon u_{xx} \tag{3.20}$$

subject to the initial conditions

$$u(x,0) = a_0 + a_1 \cos(x) + b_1 \sin(x), \tag{3.21}$$

where $a_0, a_1, b_1 \in [-1,1)$, and $\varepsilon = 0.01$. To collect training data for our neural network, we discretize the computational domain $x \in [-1,1]$ into $N_x = 100$ cells of uniform size $\Delta x = \frac{2\pi}{N_x}$. We then simulate $N_s = 1000$ solutions where the parameters $a_0^i, a_1^i, b_1^i \sim \mathscr{U}([-1,1))$, $i = 1,\ldots,N_s$, are used to define the initial conditions. To simulate the data, we use a fine grid with $\tilde{N}_x = 1000$ (see Section 3.0.2), using the second order central upwind method for the flux term, and for the time integration we used the IMEX method from Appendix B.2. We ran the simulation for $N_t = 500$ time steps with each step size $\Delta t = 0.02$. To train the neural network, we collect the data

$$\begin{aligned} u_{j,in}^{n,\hat{p}^i} &= [\bar{u}_{j-2}^{n,\hat{p}^i}, \bar{u}_{j-1}^{n,\hat{p}^i}, \bar{u}_{j}^{n,\hat{p}^i}, \bar{u}_{j+1}^{n,\hat{p}^i}, \bar{u}_{j+2}^{n,\hat{p}^i}], \\ u_{j,out}^{n+1,\hat{p}^i} &= \bar{u}_{j}^{n+1,\hat{p}^i} \end{aligned}$$

for $j = 10, 20, \ldots, 90$ and $n = 0, \ldots N_t - 1$, where $\bar{u}_j^{n,\hat{p}_i}$ is the cell average approximation at (x_j, t_n) of the ith simulation. Note that $\{x_{10}, x_{20}, \ldots, x_{90}\}$ are our sensor locations.

For this neural network, we use 80 nodes for each fully connected layer. The number of inputs into the neural network is 4 data points around the cell interfaces and there is only one output, the neural network flux around that cell interface. The loss function is given as in (3.18) with $p = 1$ and $\mu = 0.1$. The time to train the neural network was 8 hours and 20 minutes (about 30 seconds for each epoch) using a NVIDIA GeForce RTX 3060 laptop GPU. The activation function for each layer is given as

$$\sigma_\eta = \begin{cases} ReLU & \eta = 1,3 \\ I & \eta = 2,4,5 \end{cases}.$$

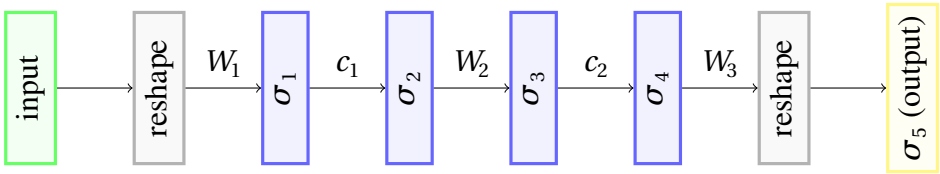

Figure 3.7: Neural network architecture for Burgers equation.

In Fig. 3.8, we can see a graph of an example with

$$(a_0, a_1, b_1) = (0.9, -0.2, 0.4)$$

at time $t = 10$. The neural network solution lines up very well with the reference solution and is even able to produce a more accurate solution around the sharp slope than the second order central upwind method. The run time for the neural network solution is 0.315 seconds and the run time for the central upwind solution is 0.877. Here, the second order central upwind method is computed with a variable time step determined by the CFL condition, and the neural network solution is computed with a constant time step $\Delta t = 0.02$ (note that this time step satisfied the CFL condition on the known solution for all steps). In Table 3.2, we show the error of the central upwind solution and the neural network solution for multiple time steps using the error defined in (3.19) with $p = 1$.

Table 3.2: l_1-norm relative error table: Burgers equation.

t	Neural Network	Second Order IMEX
2	.0030	.0048
4	.0021	.0050
6	.0025	.0046
8	.0022	.0042
10	.0022	.0037

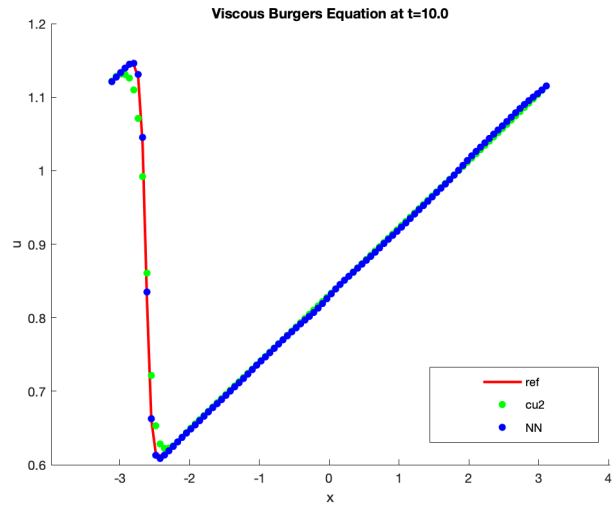

Figure 3.8: Burgers equation at $t = 10$ with initial condition $u_0(x) = 0.9 - 0.2\cos(x) + 0.4\sin(x)$.

As we can see, the neural network is able to produce an accurate result when the initial condition is of the same form as the initial condition in the training data. What we will show below is that our method can actually generalize to initial conditions that look different from the the initial conditions in the training data. We will show three different scenarios, all of which the neural network is trained with data that was produces using the initial condition (3.21). The first initial condition we consider is a periodic Gaussian function, thus it is smooth just like the initial conditions of the training data. The other two examples are cases when the initial condition is a piecewise constant function. For the piecewise constant functions, we show one example with a shock solution, and the another example is a rarefaction solution. The initial conditions are

$$u_0(x + 2K\pi) = 0.9e^{-0.4x^2}, \quad x \in [-\pi, \pi], \quad K \in \mathbb{Z},$$

$$u_0(x) = \begin{cases} 0.6 & x \le 0 \\ -0.2 & x > 0 \end{cases},$$

$$u_0(x) = \begin{cases} -0.2 & x \le 0 \\ 0.6 & x > 0 \end{cases},$$

for the Gaussian initial condition, shock solution, and rarefaction solution respectively.

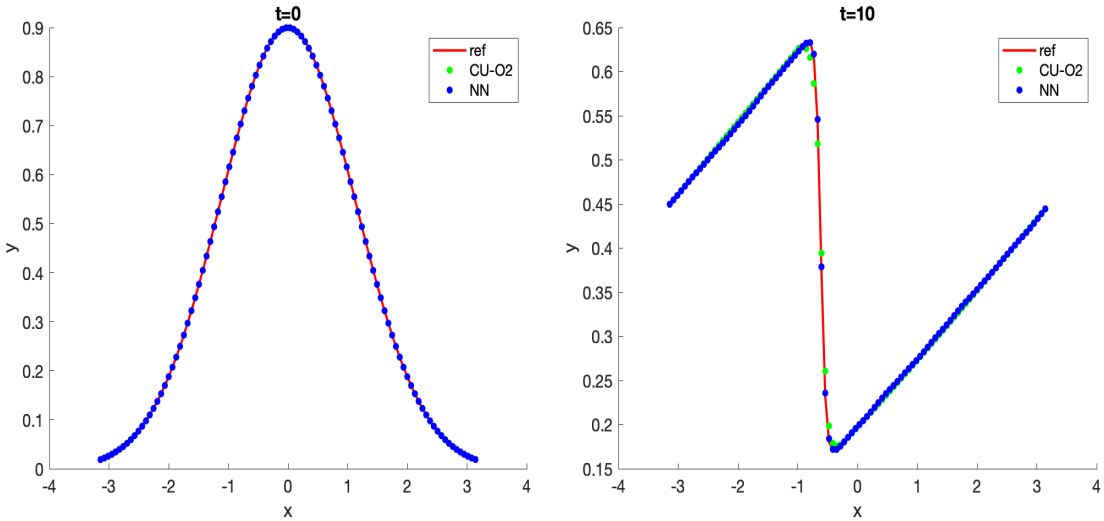

Figure 3.9: Burgers equation with initial condition $u_0(x) = 0.9e^{-0.4x^2}$ at time $t = 0$ (left) and time $t = 10$ (right).

From Fig. 3.9 and Fig. 3.10, we can see that our method is able to generalize to new initial conditions. This is because the neural network does not look at the equations values on the whole numerical domain, but instead uses only local values for its input. Thus, as long as those local values come from a similar distribution as the training data, our method should work. Because of this, the method in this section is able to compute a much wider range of functions than the method from Section 3.1.

Note that while the rarefaction solution in Fig. 3.11 does look good, our training data never saw any situations where a large jump became instantly smooth. Thus the central upwind method was able to produce a more accurate result, while with the examples in Fig. 3.9 and Fig. 3.10 our methods results were more accurate (these errors were similar to the errors in Table 3.2). The neural network's rarefaction solution had small overshoots for the first few time steps but smoothed back out as time went on. Thus, you should be careful to use this method only when you know the local data of your problem is similar to the local data in the training set.

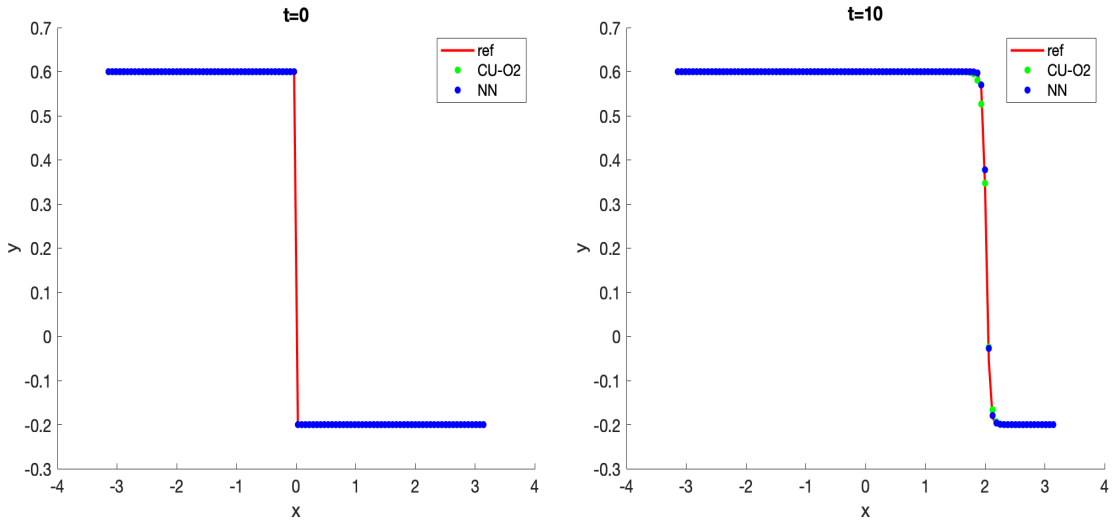

Figure 3.10: Burgers equation with initial condition $u_0(x) = 0.6$ for $x < 0$ and $u_0(x) = -0.2$ for $x \geq 0$ at time $t = 0$ (left) and time $t = 10$ (right).

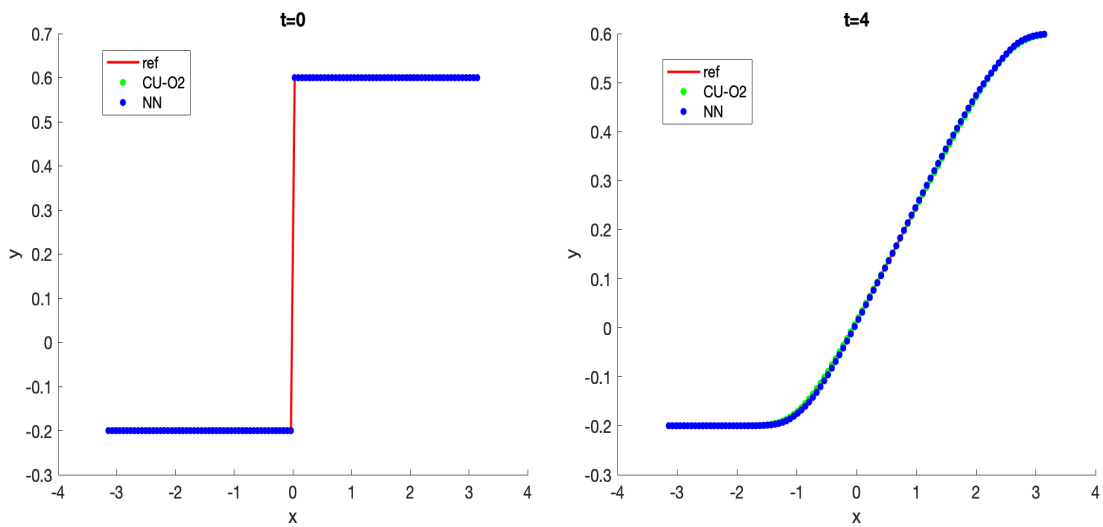

Figure 3.11: Burgers equation with initial condition $u_0(x) = -0.2$ for $x < 0$ and $u_0(x) = 0.6$ for $x \geq 0$ at time $t = 0$ (left) and time $t = 10$ (right).

The wave equation:

Next, we will look at the wave equation

$$\begin{aligned} &u_{tt} - u_{xx} = 0, \quad x \in \mathbb{R},\ t > 0, \\ &u(x,0) = \varphi_1(x), \quad x \in \mathbb{R}, \\ &u_t(x,0) = \varphi_2(x), \quad x \in \mathbb{R}, \end{aligned} \tag{3.22}$$

where

$$\begin{aligned} \varphi_1 &= \sum_{k=1}^{3} (a_{1k}\cos(kx) + b_{1k}\sin(kx))/k, \\ \varphi_2 &= \sum_{k=1}^{3} (a_{2k}\cos(kx) + b_{2k}\sin(kx))/k, \end{aligned}$$

for some $a_{lk}, b_{lk} \in [-1,1)$, $l = 1,2$ and $k = 1,2,3$. Thus, the initial conditions looks like the first few terms of a Fourier series. We can transform (3.22) and rewrite it as the system

$$\begin{aligned} &\boldsymbol{U}_t + A\boldsymbol{U}_x = 0, \\ &\boldsymbol{U}(x,0) = [\varphi_1'(x), \varphi_2(x)]^\top, \end{aligned} \tag{3.23}$$

where $U = [u_x, u_t]^T$ and

$$A = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}.$$

To simulate the solution, the computational domain $x \in [-\pi, \pi]$ is discretized into $N_x = 200$ cells, and $N_s = 1000$ simulations are run for $N_t = 1000$ time steps. For each time step $\Delta t = \frac{\Delta x}{2}$ where $\Delta x = \frac{2\pi}{N_x}$. The parameters for the initial conditions are drawn from the uniform distribution $a_{lk}^i, b_{lk}^i \sim \mathscr{U}([-1,1))$, for all $l = 1,2$, $k = 1,2,3$, and $i = 1\ldots, N_s$. The data collected is

$$\begin{aligned} \boldsymbol{U}_{j,in}^{n,\hat{\boldsymbol{p}}^i} &= [\bar{\boldsymbol{U}}_{j-1}^{n,\hat{\boldsymbol{p}}^i}, \bar{\boldsymbol{U}}_{j}^{n,\hat{\boldsymbol{p}}^i}, \bar{\boldsymbol{U}}_{j+1}^{n,\hat{\boldsymbol{p}}^i}], \\ \boldsymbol{U}_{j,out}^{n+1,\hat{\boldsymbol{p}}^i} &= \bar{\boldsymbol{U}}_{j}^{n+1,\hat{\boldsymbol{p}}^i} \end{aligned}$$

for $j = 40, 80, 120, 160$, $n = 0, 1, \ldots N_t - 1$, and $i = 1, \ldots, N_s$. Here, $\bar{\boldsymbol{U}}_j^{n,\hat{\boldsymbol{p}}^i} = \boldsymbol{U}(x_j, t_n; \hat{\boldsymbol{p}}^i)$ is the

exact value given by d'Alembert's formula

$$
\begin{aligned}
u(x,t) &= \frac{1}{2}(\varphi_1(x-t)+\varphi_1(x+t))+\frac{1}{2}\int_{x-t}^{x+t}\varphi_2(\xi)\,d\xi \\
\implies u_x(x,t) &= \frac{1}{2}(\varphi_1'(x-t)+\varphi_1'(x+t))+\frac{1}{2}(\varphi_2(x+t)-\varphi_2(x-t)) \\
\text{and } u_t(x,t) &= \frac{1}{2}(\varphi_1'(x+t)-\varphi_1'(x-t))+\frac{1}{2}(\varphi_2(x+t)+\varphi_2(x-t)).
\end{aligned}
$$

For this neural network, we use 80 nodes for each fully connected layer. The number of inputs into the neural network is 4 data points, 2 for each equation around the cell interfaces, and there are 2 outputs, the neural network fluxes for u_x and u_t around the cell interfaces. The loss function is given as (3.18) with $p=2$ and $\mu=0$. Note that due to the linearity of the problem, we do not have to use as large of a stencil as with Burgers equation. This linearity is also why we choose to use the l^2 norm for our minimization problem, i.e. why $p=2$. The time to train the neural network was 8 hours and 20 minutes (about 30 seconds for each epoch) using a NVIDIA GeForce RTX 3060 laptop GPU. Note that we need to reshape the input and output of the neural network so that there is a consistency with the array sizes (see Remark 3.1.1). In Fig. 3.12, we show the architecture of the neural network for this problem. The activation function for each layer is given by

$$
\sigma_\eta = \begin{cases} ReLU & \eta = 1,3 \\ I & \eta = 2,4,5 \end{cases}.
$$

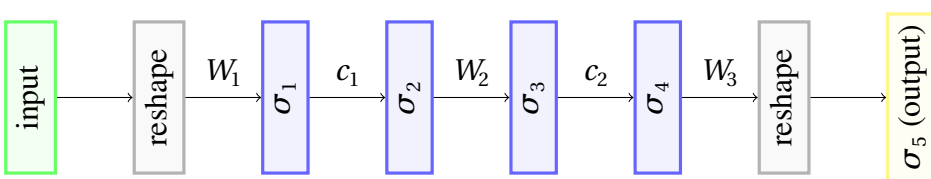

Figure 3.12: Neural network architecture for the wave equation.

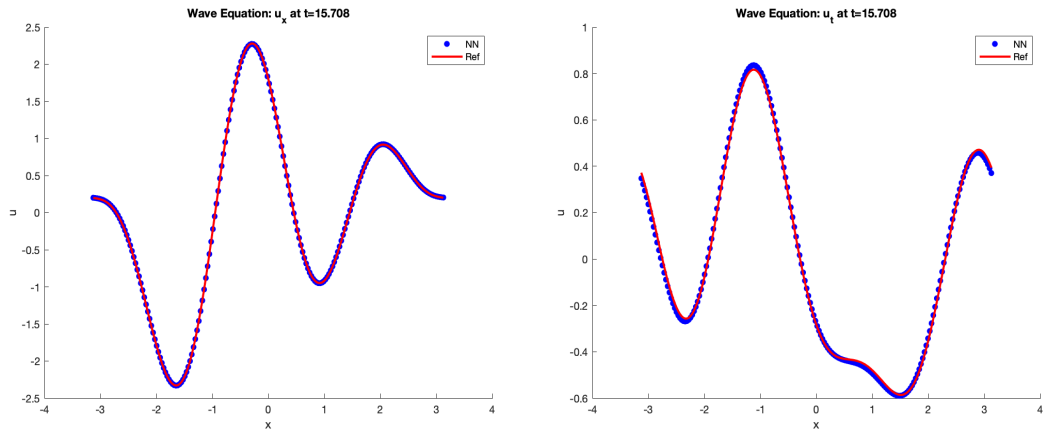

Figure 3.13: Reference (red) and neural network (blue) solution to (3.23) with initial condition parameters $a_1 = [0.6, 0.8, -0.7]$, $b_1 = [-0.4, 1.0, -0.4]$ and $a_2 = [0.1, 0.1, 0.7]$, $b_2 = [0.4, -0.7, -0.4]$.

t	Neural Network		First Order Upwind		Second Order Upwind	
	u_x	u_t	u_x	u_t	u_x	u_t
3.14	0.0009	0.0060	0.1201	0.1058	0.0015	0.0100
6.28	0.0019	0.0119	0.2212	0.1935	0.0031	0.0199
9.42	0.0028	0.0179	0.3066	0.2672	0.0047	0.0298
12.57	0.0038	0.0238	0.3793	0.3296	0.0038	0.0397
15.71	0.0047	0.0298	0.4417	0.3828	0.0078	0.0495

Table 3.3: l_2-norm relative error table: 1D wave equation.

In Fig. 3.13, we look at an example where $a_1 = [0.6, 0.8, -0.7]$, $b_1 = [-0.4, 1.0, -0.4]$ are the parameters for φ_1, and $a_2 = [0.1, 0.1, 0.7]$, $b_2 = [0.4, -0.7, -0.4]$ are the parameters for φ_2. Notice that the neural network solutions lines up very well with the reference solution which is found using d'Alembert's formula. In Table 3.3, we see that our neural network solution outperformed the first and second order upwind methods using the error (3.19) with $p = 2$. The run time for this neural network solution was 0.59 seconds, and the run times for the first and second order upwind methods are 0.03 seconds and 1.90 seconds respectively. Thus, our method was able to

compute the solution more accurate than the first and second order upwind method with a similar run time as the second order method. All three methods ran using a time step of $\Delta t = \frac{\Delta x}{2}$.

1D Navier Stokes Equation

Here we will consider a nonlinear system of equations, the isentropic Navier-Stokes equation

$$\begin{aligned} &\rho_t + (\rho u)_x = 0 \\ &(\rho u)_t + (\rho u^2 + p_r)_x = \varepsilon u_{xx}, \end{aligned} \tag{3.24}$$

subject to the initial conditions

$$\rho(x,0) = \begin{cases} \rho_L & x < .5, \\ \rho_R & x \geq .5 \end{cases}, \quad u(x,0) = 0,$$

where $p_r = \rho^\gamma$, $\gamma = 1.4$, $\varepsilon = 0.01$, and $\rho_L, \rho_R \in [.5, 5]$. This equation describe the flow of fluids where ρ, u, and p_r are the density, velocity, and pressure of the fluid respectively. Let $q = \rho u$ and $\boldsymbol{U} = [\rho, q]^\top$. To simulate the solution, the computational domain $x \in [0,1]$ is discretized into $N_x = 100$ cells, and $N_s = 1000$ simulations are run for $N_t = 150$ time steps. Each time step is $\Delta t = 0.001$, and the parameters for the initial conditions are drawn from the uniform distribution $\rho_{L,i}, \rho_{R,i} \sim \mathscr{U}([0.5, 5))$, for all $i = 1 \ldots, N_s$. The data collected is

$$\begin{aligned} \boldsymbol{U}_{j,in}^{n,\hat{\boldsymbol{p}}^i} &= [\bar{\boldsymbol{U}}_{j-2}^{n,\hat{\boldsymbol{p}}^i}, \bar{\boldsymbol{U}}_{j-1}^{n,\hat{\boldsymbol{p}}^i}, \bar{\boldsymbol{U}}_{j}^{n,\hat{\boldsymbol{p}}^i}, \bar{\boldsymbol{U}}_{j+1}^{n,\hat{\boldsymbol{p}}^i}, \bar{\boldsymbol{U}}_{j+2}^{n,\hat{\boldsymbol{p}}^i}] \\ \boldsymbol{U}_{j,out}^{n+1,\hat{\boldsymbol{p}}^i} &= \bar{\boldsymbol{U}}_{j}^{n+1,\hat{\boldsymbol{p}}^i} \end{aligned}$$

for $j = 10, 20, \ldots, 90$, $n = 0, 1, \ldots N_t - 1$, and $i = 1, \ldots, N_s$. Here $\bar{\boldsymbol{U}}_j^{n,\hat{\boldsymbol{p}}^i} \approx \boldsymbol{U}(x_j, t_n; \hat{\boldsymbol{p}}^i)$ is computed on a fine grid with $\tilde{N}_x = 1000$ cells, using the central upwind method for the numerical fluxes and the IMEX method for the time integration, see Appendix B.2.

For this neural network, we use 300 nodes for each fully connected layer. The number of inputs into the neural network is 8 data points, 4 for each equation around the cell interfaces, and there are 2 outputs, the neural network fluxes around the cell interfaces. The loss function (3.18) with $p = 1$ and $\mu = 0.1$. The time to train the neural network was 8 hours and 20 minutes (about 30 seconds for each epoch) using a NVIDIA GeForce RTX 3060 laptop GPU. Note that we need to reshape the input and output of the neural network so that there is a consistency with the array sizes (see Remark 3.1.1). The neural network architecture is depicted in Fig. 3.14, where the

activation function for each layer is given by

$$\sigma_\eta = \begin{cases} ReLU & \eta = 1,3 \\ I & \eta = 2,4,5 \end{cases}.$$

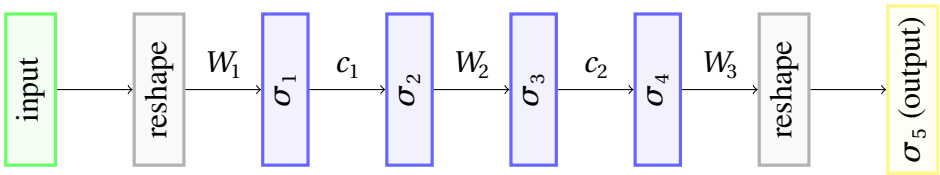

Figure 3.14: Neural network architecture for the isentropic Navier-Stokes equations.

In Fig. 3.15 we look at an example where $\rho_L = 3.3$, $\rho_R = 0.7$. The reference solution is computed with the second order central upwind method on a fine grid with 1000 cells. From the plot, our method seems to compare very well with the second order central upwind method computed on the same grid, possibly even a little better around the sharp slopes. In Table 3.5 this close approximation is confirmed as the neural network solution accuracy and the central upwind method have very similar errors. This error is taken as in (3.19) with $p = 1$. The neural network solution ran in 0.141 seconds and the IMEX method ran in 0.481 seconds. The neural network simulation ran with a time step $\Delta t = 0.001$ which, based on observed simulations, is a much smaller time step than the CFL condition requires. The second order central upwind method ran using an adaptive time step with the CFL number $\nu = 0.5$ (see (2.16)).

Table 3.4: l_1-norm relative error table: 1D Isentropic Euler equations.

t	Neural Network		Second Order IMEX	
	ρ	q	ρ	q
0.05	0.00257	0.0303	0.00337	0.0355
0.10	0.0024	0.0151	0.0035	0.0191
0.15	0.00257	0.0108	0.00352	0.0129

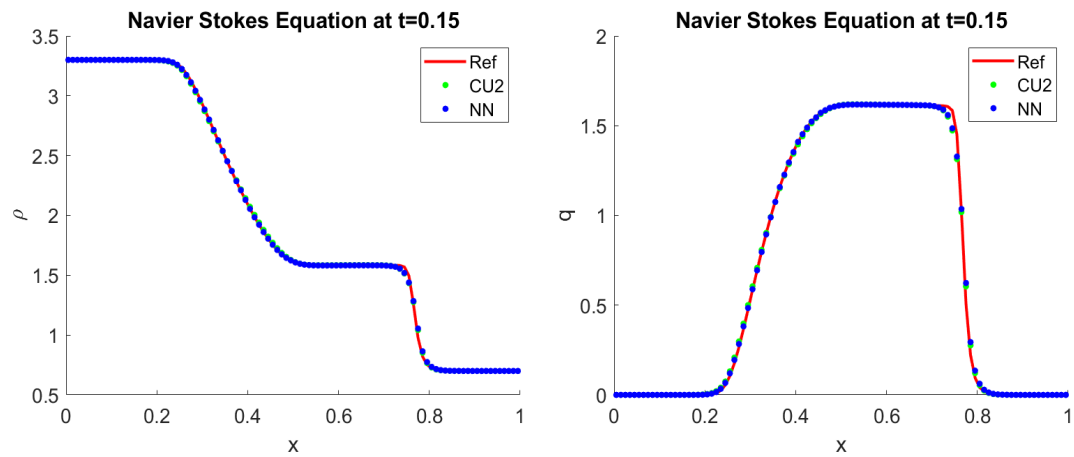

Figure 3.15: Isentropic Euler equations at t=0.15 with initial condition parameters $\rho_L = 3.3$, $\rho_R = 0.7$.

2D Navier Stokes Equation

Here we will consider the two dimension isentropic Navier-Stokes equation

$$\begin{aligned} &\rho_t + (\rho u)_x + (\rho v)_y = 0; \quad x \in \mathbb{R},\ t > 0, \\ &(\rho u)_t + (\rho u^2 + \rho^\gamma)_x + (\rho u v)_y = \varepsilon u_{xx}, \\ &(\rho v)_t + (\rho u v)_x + (\rho v^2 + \rho^\gamma)_y = \varepsilon v_{yy}, \end{aligned} \tag{3.25}$$

subject to the initial condition

$$\rho(x,y,0) = \begin{cases} \rho_1 & x < 0.5 \text{ and } y < 0.5, \\ \rho_2 & x < 0.5 \text{ and } y \geq 0.5, \\ \rho_3 & x \geq 0.5 \text{ and } y < 0.5, \\ \rho_4 & x \geq 0.5 \text{ and } y \geq 0.5 \end{cases}, \quad u(x,y,0) = v(x,y,0) = 0,$$

where $\gamma = 1.4$, $\varepsilon = 0.005$, and $\rho_1, \rho_2, \rho_3, \rho_4 \in [.5, 1.5]$. Here, ρ, u, and v are the fluids density, velocity in the x direction, and velocity in the y direction respectively. Let $q_1 = \rho u$, $q_2 = \rho v$, and $\boldsymbol{U} = [\rho, q_1, q_2]$. To simulate the solution, the numerical domain $[0,1] \times [0,1]$ is discretized into $N_x = N_y = 80$ cells, and $N_s = 1000$ simulations are run for $N_t = 150$ time steps. For each time step $\Delta t = 0.001$, and the parameters for the initial conditions are drawn from the uniform

distribution $\rho_{1,i}, \rho_{2,i}, \rho_{3,i}, \rho_{4,i} \sim \mathscr{U}([0.5, 1.5))$, for all $i = 1 \ldots, N_s$. The data collected is

$$
\begin{aligned}
\boldsymbol{U}^{n,\hat{\boldsymbol{p}}^i}_{j,k,in_x} &= [\bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j-2,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j-1,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j+1,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j+2,k}] \\
\boldsymbol{U}^{n,\hat{\boldsymbol{p}}^i}_{j,k,in_y} &= [\bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k-2}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k-1}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{j,k}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{k,j+1}, \bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_{k,j+2}] \\
\boldsymbol{U}^{n+1,\hat{\boldsymbol{p}}^i}_{j,k,out} &= \bar{\boldsymbol{U}}^{n+1,\hat{\boldsymbol{p}}^i}_{j,k}
\end{aligned}
$$

for $j, k = 10, 20, \ldots, 70$, $n = 0, 1, \ldots N_t - 1$, and $i = 1, \ldots, N_s$. Here $\bar{\boldsymbol{U}}^{n,\hat{\boldsymbol{p}}^i}_j \approx \boldsymbol{U}(x_j, t_n; \hat{\boldsymbol{p}}^i)$ which is computed using a fine grid simulation with $\tilde{N}_x = \tilde{N}_y = 240$, using the central upwind method to find the numerical fluxes, and the third order string stability-preserving Runge-Kutta method for the time integration, see Appendix B.1 (accounting for the diffusive term in the time step to guarantee stability).

For this neural network, we use 300 nodes for each fully connected layer. The number of inputs into the neural network is 12 data points, 4 for each equation around the cell interfaces, and there were 3 outputs, the neural network fluxes around the cell interfaces. The loss function is given as in (3.18) with $p = 1$ and $\mu = 0.1$. The time to train the neural network was 13 hours and 35 minutes (about 50 seconds for each epoch) using a NVIDIA GeForce RTX 3060 laptop GPU. Note that we need to reshape the input and output of the neural network so that there is a consistency with the array sizes (see Remark 3.1.1). The neural network architecture is depicted in Fig. 3.16 where the activation function for each layer is given as

$$
\sigma_\eta = \begin{cases} ReLU & \eta = 1, 3 \\ I & \eta = 2, 4, 5 \end{cases}.
$$

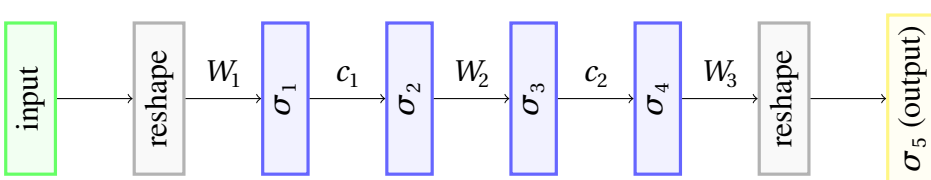

Figure 3.16: Neural network architecture for the isentropic Navier-Stokes equations.

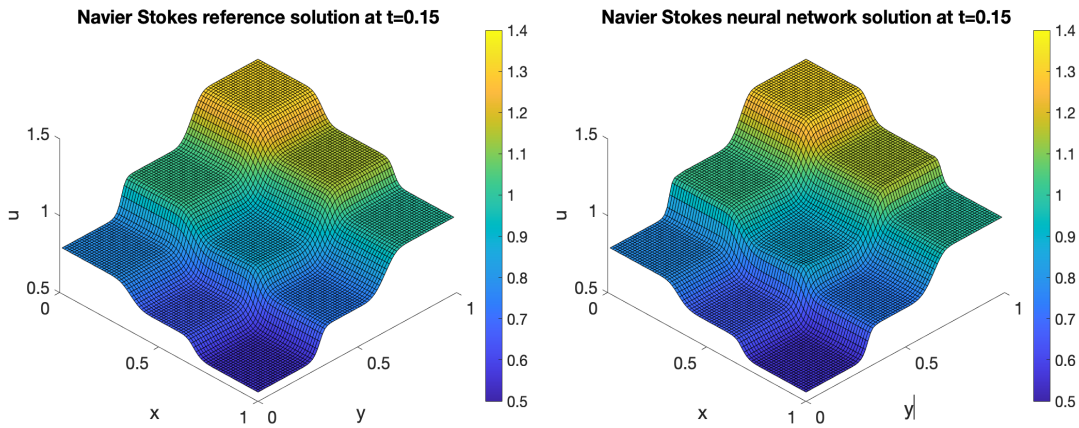

Figure 3.17: Plot of ρ at $t = 0.15$ for the reference solution (Left) and neural network solution (Right) with $\rho_1 = 0.79$, $\rho_2 = 1.32$, $\rho_3 = 0.55$, and $\rho_4 = 0.99$.

In Fig. 3.17, we look at an example where $\rho_1 = 0.79$, $\rho_2 = 1.32$, $\rho_3 = 0.55$, and $\rho_4 = 0.99$. The reference solution was found using a fine grid simulation with 800×800 cells in both the x and y dimensions. Looking at the plots, there is almost no discernible difference between the neural network solution and the reference solution. We can see the difference in errors between the neural network solution and the second order central upwind solution in Table 3.5 using the error (3.19) with $p = 1$. From here we see that the neural network solution is comparable to the solution from the second order central upwind method. The run time for the neural network solution is 3.45 seconds using the GPU and 7.05 seconds using the CPU. For the second order central upwind method, the run time is 36.88 second. The neural network solution uses a time step $\Delta t = 0.001$ which, based on observed simulations, is a much smaller time step than the CFL condition requires. The second order central upwind method is ran using an adaptive time step based on the CFL condition. Note that since $\varepsilon < \Delta x$, the time step restriction due to the diffusive term was larger than the time step restriction of the hyperbolic flux terms, thus the speed for the second order central upwind method would not have been faster if we used an IMEX method like we did in the one dimensional case.

Table 3.5: l_1-norm relative error table: 2D Isentropic Euler equations.

t	Neural Network			Central Upwind-O2		
	ρ	q_1	q_2	ρ	q_1	q_2
0.05	0.0016	0.0207	0.0497	0.0031	0.0687	0.0700
0.10	0.0019	0.0173	0.0317	0.0035	0.0395	0.0415
0.15	0.0021	0.0147	0.0243	0.0037	0.0289	0.0308

3.3 Tuning Artificial Viscosity with Neural Networks

Consider the system of one-dimensional hyperbolic balance laws given by

$$\boldsymbol{U}_t + \boldsymbol{F}(\boldsymbol{U})_x = \boldsymbol{S}(x,t,\boldsymbol{U}), \quad x \in \mathbb{R},\ t \geq 0, \tag{3.26}$$

where $\boldsymbol{U}(x,t) = (u_1(x,t),\ldots,u_s(x,t))$, $\boldsymbol{F}$ is a nonlinear flux, and $\boldsymbol{S}$ is the source. Equations of this type are known to have weak solutions with discontinuities in the form of shocks. When not handled appropriately, these discontinuities can cause issues with traditional numerical methods, causing them to produce non-physical oscillations in the solution and causing instabilities. For example, the solution in Fig. 3.18 is from simulating the inviscid Burgers equation with the second order central upwind method using the forward difference for the cell reconstructions (see Appendix C). In this example, the initial condition is the smooth function

$$u(x,0) = 0.5 - \sin(x)$$

on the numerical domain $x \in [-\pi,\pi]$ with periodic boundary conditions. Around the time $t = 1$, the solution start producing oscillations.

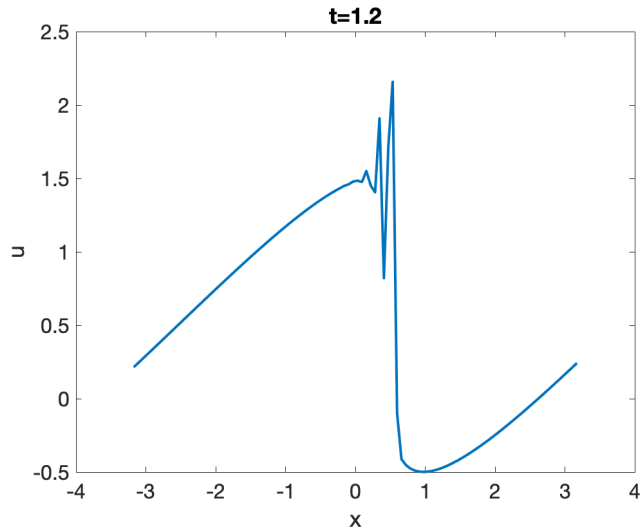

Figure 3.18: Burgers equation with initial condition: $u(x,0) = 0.5 - \sin(x)$.

There are different methods to suppress these oscillations. One method is to use what are known as limiters. An example of a limiter is the minmod limiter (see Appendix C) which we used to produce the graph in Fig. 3.19. As we can see, using this limiter we are able to remove all of the oscillations. However, these limiters can be expensive to compute and cause the order of accuracy to decrease near these oscillations. Thus, we will consider stabilizing numerical methods using another approach, which is by adding artificial viscosity. For this, we add an extra term to (3.26) to get the new equation

$$\boldsymbol{U}_t + \boldsymbol{F}(\boldsymbol{U})_x = \boldsymbol{S}(x,t,\boldsymbol{U}) + C(\varepsilon(\boldsymbol{U})\boldsymbol{U}_x)_x. \tag{3.27}$$

The last term in (3.27) is known as adaptive artificial viscosity (AAV) and is used smooth out discontinuities and stabilize numerical methods. It should have the property that

$$\varepsilon(\boldsymbol{U}) \sim \begin{cases} \Delta & \text{non-smooth regions,} \\ \Delta^{\alpha} & \text{smooth regions} \end{cases},$$

where $\Delta = \max(\Delta x, \Delta t)$ is the maximum between the spatial and temporal discretization step size for the numerical method, and $\alpha > 1$ so that $\Delta^{\alpha} \approx 0$.

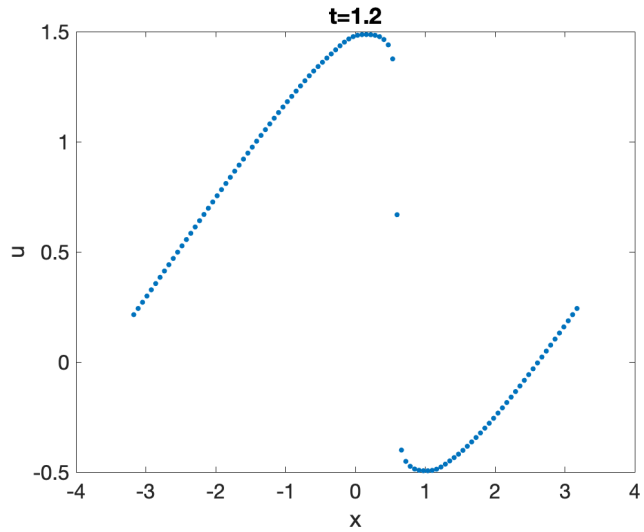

Figure 3.19: Burgers equation with initial condition: $u(x,0) = 0.5 - \sin(x)$.

There are numerous different numerical methods that can be used to solve (3.27), as well as many different ways to define $\varepsilon(\boldsymbol{U})$ and its discrete approximation. In this thesis, we are not as interested in the numerical method as a whole but instead, for a given method, how to find the optimal value for the coefficient C on the right hand side of (3.27).

Traditionally, the value of C is determined by running multiple simulations and choosing which one looks the best. In Fig. 3.20, we can see three different solutions to Burgers equation using a numerical method which requires AAV, where the value of C is different for each simulation. In the first simulation, when $C = 0$, we can see that there are a lot of oscillations in the solution. In the last simulation, when $C = 50$, the oscillations are gone, however there is excessive smoothing in the solution. Thus, we want a solution like the one on the middle, when $C = 20$, where the oscillations are dissipated, yet the shock is not overly smoothed out. The value of C that gives the best solution depends on the state of the solution at a given time. Thus, the optimal C value can change depending on the initial condition to the problem, as well as when the solution evolves in time. It would be ideal to come up with a method of determining what the best C value is without having to run multiple simulations. For this, we propose using an artificial neural network to find the best value for the constant C.

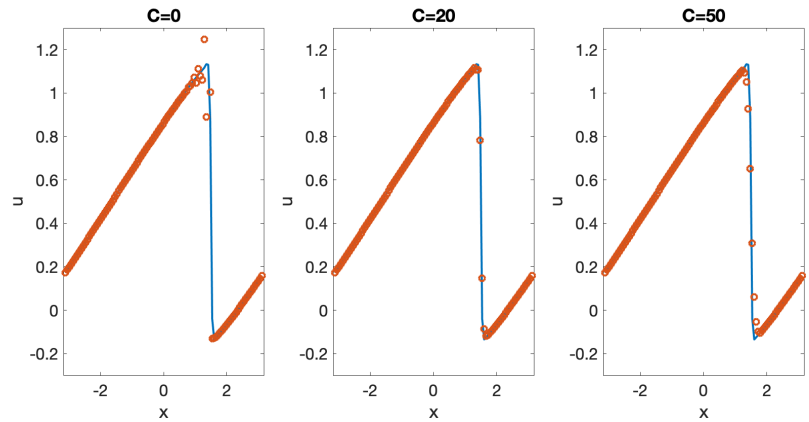

Figure 3.20: Burgers equation with artificial viscosity where $C = 0$ (left), $C = 20$ (middle), and $C = 50$ (right).

3.3.1 Finding the Best Coefficient

For this section we are going to assume that the our numerical domain is $x \in [a_x, b_x]$, discretized by

$$x_j = a_x - \frac{\Delta x}{2} + j\Delta x, \quad j = 0, \ldots, N_x + 1,$$

where $\Delta x = \frac{b_x - a_x}{N_x}$. Let

$$\bar{U}_j^n \approx \frac{1}{\Delta x} \int_{x_{j-1/2}}^{x_{j+1/2}} U(x, t_n)\, dx, \quad j = 1, \ldots, N_x$$

be an approximation to (3.26) over the cells $[x_{j-1/2}, x_{j+1/2}]$, where $x_{j-1/2} = \frac{x_{j-1} + x_j}{2}$, at time t_n, that came from an accurate and stable numerical method. Similarly, let

$$\bar{U}_j^{n,C} \approx \frac{1}{\Delta x} \int_{x_{j-1/2}}^{x_{j+1/2}} U^C(x, t_n)\, dx, \quad j = 1, \ldots, N_x$$

where U^C is the solution to (3.27) with the viscosity coefficient C, be a numerical solution that does not use any limiters. We will denote $\bar{U}^n = [\bar{U}_1^n, \ldots, \bar{U}_{N_x}^n]$ and $\bar{U}^{n,C} = [\bar{U}_1^{n,C}, \ldots, \bar{U}_{N_x}^{n,C}]$ as the vector of the solution values at all points in our numerical domain at time t_n.

In order to train a neural network to determine the best value for C, we first have to determine a way to quantify what the best value is. For this, we assume that the C value is most influential around the location where $\varepsilon(U)$ is at a maximum. Let $\varepsilon_{j+1/2}^n$ be the numerical approximation to

our viscous term $\varepsilon(\boldsymbol{U}(x_{j+1/2}, t_n))$. Then, let

$$j^n_{max} = \underset{j=0,\ldots,N_x}{\arg\max}\, \varepsilon^n_{j+1/2}$$

and

$$\boldsymbol{U}^n_{j^n_{max},sten} := [\bar{\boldsymbol{U}}^n_{j^n_{max}-M_s/2+1}, \ldots, \bar{\boldsymbol{U}}^n_{j^n_{max}}, \bar{\boldsymbol{U}}^n_{j^n_{max}+1}, \ldots, \bar{\boldsymbol{U}}^n_{j^n_{max}+M_s/2}], \tag{3.28}$$

be the stencil of size M_s around $x_{j^n_{max}+1/2}$. The input into our neural network will be the stencils $\boldsymbol{U}^n_{j^n_{max},sten}$ for every time step that is used in calculating $\varepsilon^n_{j+1/2}$. For example, in [44] an artificial viscosity was used where $\varepsilon^n_{j+1/2}$ depends on the solution at times t_n and t_{n-1}, thus the input for the neural network would be $\boldsymbol{U}^n_{j^n_{max},sten}$ and $\boldsymbol{U}^{n-1}_{j^n_{max},sten}$. We will denote the set of all these required stencils as $\Phi_{j^n_{max}}$ which consist of β time steps. At time t_n, we want to find the smallest value of C such that at time $t_{\tilde{n}} = t_n + \Delta T$,

$$TV(\boldsymbol{U}^{\tilde{n},C}_{j^{\tilde{n}}_{max},sten}) \leq TV(\boldsymbol{U}^{\tilde{n},ref}_{j^{\tilde{n}}_{max},sten})(1+\delta_{tol})$$

where

$$TV(\boldsymbol{U}^n_{j^n_{max},sten}) = \sum_{k=-M_s/2+1,\ldots,M_s/2-1} |\bar{\boldsymbol{U}}^n_{j^n_{max}+k+1} - \bar{\boldsymbol{U}}^n_{j^n_{max}+k}|,$$

is the total variation around the point $x_{j^n_{max}}$, δ_{tol} is some prescribed tolerance and ΔT is some prescribed time evolution (note we use ΔT instead of Δt so that it is not confused with the time step of any particular numerical method. Typically $\Delta T \gg \Delta t$.). In other words, we are trying to find the smallest C value such that the total variation around $x_{j^{\tilde{n}}_{max}}$ from the AAV solution is within some tolerance of the total variation of the reference solution.

Remark 3.3.1. *This time step ΔT determines how often we update the value of C in the simulation, but another approach is to update after so many time steps. For instance, we could let $t_{\tilde{n}} = t_{n+b}$ for some constant b.*

3.3.2 Collecting Training Data

To train the neural network, we will generate our own input and output data to learn from. For this, assume the initial condition to (3.26) is a parameterized function $\boldsymbol{U}_0(x;\hat{\boldsymbol{p}})$ where $\hat{\boldsymbol{p}}$ is a multivariate parameter. To collect data, we will select $\{\hat{\boldsymbol{p}}^i\}_{i=1}^{N_s}$ from a multivariate uniform distribution, and use each of these to define the initial conditions $\{\boldsymbol{U}_0(x,\hat{\boldsymbol{p}}^i)\}_{i=1}^{N_s}$ for each

simulation. We will also assume that the optimal C values for each simulation come from a discrete set $\{C_l\}_{l=0}^{N_c}$ such that $C_0 < C_1 < \ldots < C_{N_c}$.

For each simulation, we find

$$j_{max}^0 = \underset{j=0,\ldots,N_x}{\arg\max}\, \varepsilon_{j+1/2}^0$$

at time $t = t_0$, and the first input data $\Phi_{j_{max}^0}$. Note that $t_0 = 0$ if $\varepsilon_{j+1/2}^0$ does not depend on previous time steps, otherwise we need to simulate the solution for enough time steps to calculate $\varepsilon_{j+1/2}^0$. Let $\bar{\boldsymbol{U}}^{n_1,ref}$ be the reference solution computed with an accurate and stable numerical method up to time $t_{n_1} = t_0 + \Delta T$. We then calculate $\bar{\boldsymbol{U}}^{n_1,C_l}$, starting with $l = 0$ and increasing it by one until we find the smallest C_l such that

$$TV(\boldsymbol{U}_{j_{max}^{n_1},sten}^{n_1,C_l}) \leq TV(\boldsymbol{U}_{j_{max}^{n_1},sten}^{n_1,ref})(1+\delta_{tol}).$$

This smallest C_l value, denoted C_{opt}^0, will be the output data corresponding to the input data $\Phi_{j_{max}^0}$. We then repeat this process going forward in time for

$$t_{n_k} = t_0 + k\Delta T, \quad k = 1,\ldots,N_T+1$$

collecting the input data $\Phi_{j_{max}^{n_k}}$ and output data C_{opt}^k. Here, we assume the solution at t_{n_k} for the next time step $t_{n_{k+1}}$ is taken as $\bar{\boldsymbol{U}}^{n_k,C_{opt}^{k-1}}$ from the previous time step, i.e. the solutions $\bar{\boldsymbol{U}}^{n_{k+1},ref}$ and $\bar{\boldsymbol{U}}^{n_{k+1},C_l}$, are computed from the solution $\bar{\boldsymbol{U}}^{n_k,C_{opt}^{k-1}}$.

With this process, we generate data from $i = 1,\ldots,N_s$ simulation, taking data at the times $t_0, t_{n_1},\ldots,t_{N_T}$, thus we end up with $N_s(N_T+1)$ data pairs. Let $M = N_s(N_T+1)-1$ and $m = (i-1)(N_T+1)+k$, $i = 1,\ldots,N_s$ and $k = 0,\ldots,N_T$. Let

$$\Phi^m := \Phi_{j_{max}^{n_k}} \tag{3.29}$$

from the ith simulation and let

$$\Psi^m := C_{opt}^k \tag{3.30}$$

be the corresponding optimal C value. Then the data set for the neural network can be written as $\{\Phi^m, \Psi^m\}_{m=0}^M$ where Φ^m is the input data and Ψ^m is the output data. Given this data set we train

a fully connected feedforward neural network, $\boldsymbol{N}_\Theta : \mathbb{R}^{M_s \times \beta} \to \mathbb{R}$, such that

$$\boldsymbol{N}_\Theta(\Phi^m) \approx \Psi^m. \tag{3.31}$$

While the training output data, Ψ^m, takes discrete values (i.e. the values in the set $\{C_l\}_{l=1}^{N_c}$) we train our neural network as a regression problem that is continuous with respect to its the input. In particular, the neural network has the output layer

$$\boldsymbol{N}_{\Theta_{H+1}}(z) = \frac{C_{N_C}}{2}(\tanh(z)+1)$$

thus $\text{Range}(\boldsymbol{N}_\Theta) = (0, C_{N_C})$. We use this approach instead of a classification neural network for two reasons. First, classification neural networks usually do not have any correlation between output classes. However for our network, inputs that give close C values from our data set should give close outputs from our neural network, even if the solution is not perfect. Second, the optimal C value should come from a continuous set, but for the sake of gathering data we can not compute the solution for every possible C value in an infinite set. Thus we are left with approximating the optimal C values from a discrete set.

3.3.3 Numerical Examples

Shallow Water Equation

Here we consider the shallow water equations

$$\begin{aligned} w_t + (hu)_x &= 0 \\ (hu)_t + \left(hu^2 + \frac{h^2}{2}\right)_x &= -hB_x, \end{aligned}$$

with the following initial conditions

$$w(x,0) = \begin{cases} w_L, & x < 0 \\ w_R, & x > 0 \end{cases}, \quad u(x,0) = 0,$$

and source term

$$B(x) = 0.4e^{-2x^2},$$

where h, u, and B are the depth, velocity, and bottom topography of the water respectively. Here, $w = h + B$ is the water surface, and $w_L, w_R \in [0.5, 2.5]$. Let $\boldsymbol{U} = [w, q]^\top$ where $q = hu$.

We train the neural network to find the optimal C value for the all-global scheme with artificial viscosity (A. Chertock and A. Kurganov, personal communication, May 25, 2021).

In this example, we trained a fully connected feedforward neural network using the Adam optimization algorithm [43] to find the parameter set Θ that minimizes the mean square error loss function

$$L_\Theta = \frac{1}{M}\sum_{m=0}^{M} |\boldsymbol{N}_\Theta(\Phi^m) - \Psi^m|^2. \tag{3.32}$$

This neural network has $H = 3$ hidden layers, each with 256 nodes, and the activation functions used are

$$\sigma_h(z) = ReLU(z), \quad h = 1, \ldots, H \tag{3.33}$$

$$\sigma_{H+1}(z) = \frac{C_{N_C}}{2}(\tanh(z) + 1). \tag{3.34}$$

Note that we need to reshape the input and output of the neural network so that there is a consistency with the array sizes (see Remark 3.1.1). In Fig. 3.21 we depict the architecture diagram for this neural network.

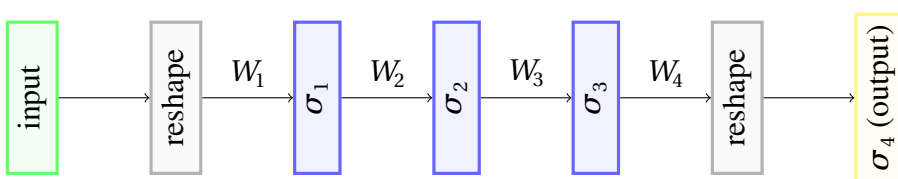

Figure 3.21: Artificial viscosity coefficient neural network for the shallow water equation.

To collect data to train the network, we use the method from Section 3.3.2 with $\Delta T = 0.05$, the initial conditions $w_L, w_R \sim \mathcal{U}(0.5, 2.5)$, $\delta_{tol} = .03$, $M_s = 10$, and $C_l = l$ for $l = 0, \ldots, 50 = N_C$. We simulate the data on the numerical domain $[-1, 1]$ with $N_x = 200$, up to the time $t = 0.5$ (thus, $N_t = \frac{0.5}{0.05} = 10$), taking $N_s = 10{,}000$ different initial conditions, thus we get $M = (10{,}000)(10) = 100{,}000$ total data pairs. We split this data set into $M_{tr} = 80{,}000$ and $M_{val} = 20{,}000$ training and validation data pairs respectively. The reference solution is computed using the second order central upwind method on the same grid. We train the neural network for 100 epochs, and the time to train was 822 seconds using a NVIDIA GeForce RTX 3060 laptop GPU.

In Fig. 3.22, we compare the neural network solution for C verses the reference values C_l for 25 random samples taken from the test data set. In Fig. 3.23 and Fig. 3.24 we see the numerical

solution for an example with

$$w(x,0) = \begin{cases} w_L = 0.6 & x \leq 0 \\ w_R = 1.45 & x > 0 \end{cases}.$$

On the left of Fig. 3.23 is the solution computed with the all global scheme without artificial viscosity, and on the right is the all-global scheme solution with the viscosity coefficient obtained by the neural network. In Fig. 3.24, we compare the all-global scheme computed with the neural network to the all-global scheme computed with $C = 50$. Notice how when $C = 0$ there are oscillations and when $C = 50$ there is too much smoothing. The neural network for this example computed $C \approx 30$ for all time steps which got rid of most of the oscillations without excessively smoothing the solution.

In Table 3.6, we compare the run time, relative error, and total variation of the solution w from the neural network with the solutions when $C = 0$ and when $C = 50$. Here the relative error is defined as in (3.19) and the total variation is defined as

$$TV(w) = \sum_j |w_{j+1} - w_j|.$$

Usually, the solution with a small C value has a small error and a large total variation while solutions with a large C value have a high error and small total variation. We can see from Table 3.6 that this is what happened with our example. While our method has a lower error than both the other methods, in general this is not always true. Our method is good at keeping a balance between the error and the total variation, which usually means the error and total variation is between the two extreme cases.

Table 3.6: Comparing run time, error, and total variation for w from the all global solutions with $C = 0$, $C = 50$, and C determined by a neural network.

-	CU2	Neural Network	AG:C=0	AG:C=50
run time	$.018s$	$.011s$	$0.006s$	$0.006s$
E	-	0.0015	0.0017	0.0031
TV	0.906	0.934	1.0733	0.910

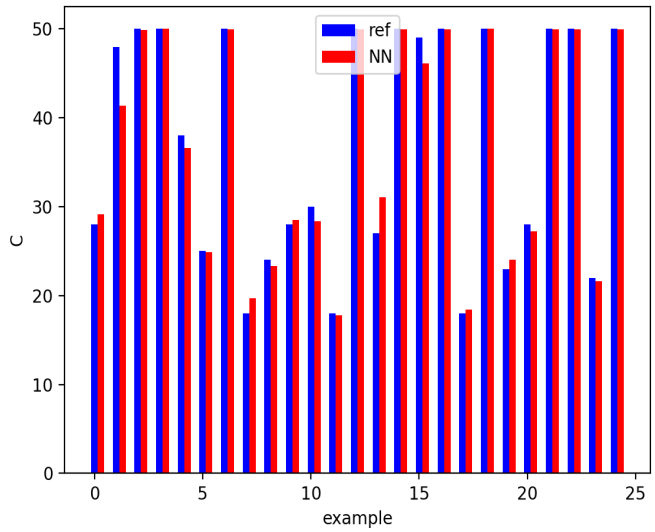

Figure 3.22: Comparison of C values calculated by neural network and C value calculated by the method in 3.3.2.

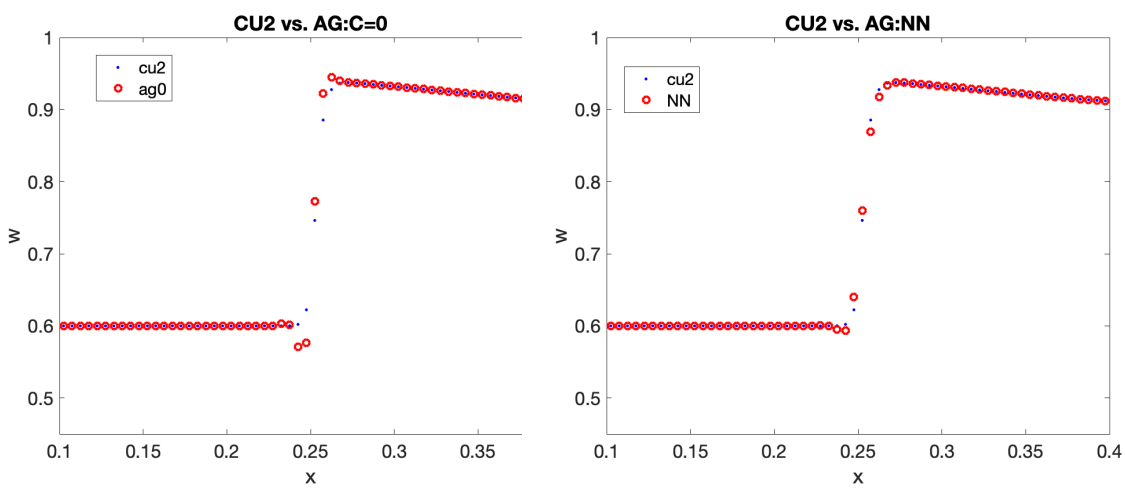

Figure 3.23: *Left:* All-global method with $C = 0$. *Right:* All-global method with C determined by neural network.

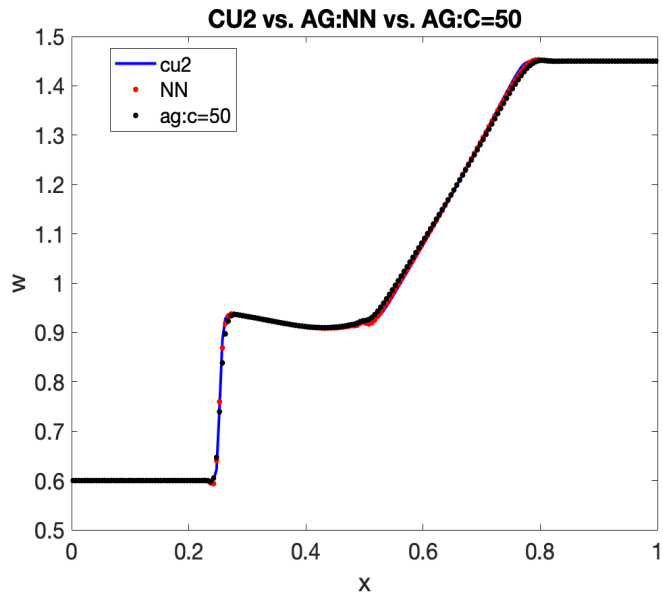

Figure 3.24: All-global method with C determined by neural network vs all-global method with $C = 50$.

3.4 Conclusion

In these projects, we showed how neural networks can be used to simulate partial differential equations. In the Section 3.1 and Section 3.2, we were able to show two different approaches to simulating the solution to partial differential equations without explicit knowledge of the underlying dynamics. With the first approach, we retrieved cell averages for multiple time steps with only one neural network function evaluation, where the input to the neural network was the initial condition to the PDE. For the specific initial conditions that these networks were trained for, these networks were able to produce accurate simulated data very quickly. As far as I know, know one else has used this kind of method for solving time dependent PDEs. For the second approach, we use a time stepping method to simulate the dynamics of the PDEs. As this kind of approach has been done before in, such as in [8], our method uses only local solution data to train the neural network. This allows our method to work for a wide range of conditions. Also, I believe this to be a more practical approach to using a neural network to simulate unknown PDEs, as collecting data on a large set of nodal points could be very hard to accomplish in a real setting. In Section 3.3, we showed that neural networks can be used to tune the amount of artificial viscosity that is used for a numerical method. This allows us to run our simulations without having to check multiple scales of magnitude to see which viscosity constant will give

us the best results.

Some possible future work for these projects include

- Adding noise to the collected data to make it more realistic.
- Combining the results from the time stepping method in Section 3.2, to the artificial viscosity method in Section 3.3, in order to get a stabilized neural network method that works for hyperbolic equations.
- Continue the work in Section 3.3 in higher spatial dimensions.

CHAPTER

4

FINDING THE SHAPE OF THE LACUNAE WITH NEURAL NETWORKS

4.1 Introducing the Lacunae of the Wave Equation

Consider the inhomogeneous scalar wave (d'Alembert) equation in the three-dimensional space (3D):

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \Delta u = f(\boldsymbol{x}, t), \quad \boldsymbol{x} \in \mathbb{R}^3, \quad t \geqslant 0, \tag{4.1}$$

subject to zero initial conditions, and with the source term f that is compactly supported on a bounded (3+1)D domain $Q_f \subset \mathbb{R}^3 \times [0, +\infty)$. The solution u to equation (4.1) is given by the Kirchhoff integral:

$$u(\boldsymbol{x}, t) = \frac{1}{4\pi} \iiint\limits_{|\boldsymbol{x}-\boldsymbol{\xi}| \leqslant c\,t} \frac{f(\boldsymbol{\xi}, t - |\boldsymbol{x}-\boldsymbol{\xi}|/c)}{|\boldsymbol{x}-\boldsymbol{\xi}|} d\boldsymbol{\xi}. \tag{4.2}$$

The integration in (4.2) is performed in space over the ball of radius $c\,t$ centered at $\boldsymbol{x}$, but as f is taken at retarded moments of time, this can be interpreted as integration in the (3+1)D space-time over the surface of a backward characteristic cone of equation (4.1) (light cone of the

past) with the vertex $(\boldsymbol{x}, t)$. This surface may or may not intersect with the support Q_f of the right-hand side f. If there is no intersection, then $u(\boldsymbol{x}, t) = 0$, which implies, in particular, that the solution $u = u(\boldsymbol{x}, t)$ of equation (4.1) will have a lacuna (secondary lacuna in the sense of Petrowsky [65]):

$$u(\boldsymbol{x}, t) \equiv 0 \quad \forall (\boldsymbol{x}, t) \in \bigcap_{(\boldsymbol{\xi}, \tau) \in Q_f} \left\{ (\tilde{\boldsymbol{x}}, \tilde{t}) \middle| |\tilde{\boldsymbol{x}} - \boldsymbol{\xi}| < c(\tilde{t} - \tau),\ \tilde{t} > \tau \right\} \stackrel{\text{def}}{=} \Lambda. \tag{4.3}$$

Mathematically, the lacuna Λ is the intersection of all forward characteristic cones (i.e., light cones of the future) of the wave equation (4.1) once the vertex of the cone sweeps the support Q_f of the right-hand side $f(\boldsymbol{x}, t)$. From the standpoint of physics, Λ is the part of space-time where the waves generated by a compactly supported source have already passed and the solution has become zero again. The primary lacuna (as opposed to secondary lacuna (4.3)) is the part of space-time ahead of the propagating fronts where the waves have not reached yet.

The phenomenon of lacunae is inherently three-dimensional (more precisely, it pertains to spaces of odd dimension). The surface of the lacuna includes the trajectory of aft (trailing) fronts of the propagating waves. The existence of sharp aft fronts in odd-dimension spaces is known as the (strong) Huygens' principle, as opposed to the so-called wave diffusion, which takes place in spaces of even dimension [80; 13].

The question of identifying the hyperbolic equations and systems that admit the diffusionless propagation of waves has been first formulated by Hadamard [27; 28; 29]. He, however, did not know any other examples besides the d'Alembert equation (4.1). The notion of lacunae was introduced and studied by Petrowsky in [65], where conditions for the coefficients of hyperbolic equations that guaranteed the existence of lacunae have been obtained (see also [13, Chapter VI]). Subsequent developments can be found in [1; 2]. However, since work [65] no other constructive examples of either scalar equations or systems that satisfy the Huygens' principle have been found except for the wave equation (4.1) and its equivalents. Specifically, it was shown in [55] that in the standard $(3+1)$D space-time with Minkowski metric, the only scalar hyperbolic equation that has lacunae is the wave equation (4.1). The first examples of nontrivial diffusionless equations (i.e., irreducible to the wave equation) were constructed in [75; 47; 76], but the space must be $\mathbb{R}^d$ for odd $d \geqslant 5$. Examples of nontrivial diffusionless systems (as opposed to scalar equations) in the standard Minkowski $(3+1)$D space-time were presented in [72; 4; 25], as well as examples of nontrivial scalar Huygens' equations in a $(3+1)$D space-time equipped with a different metric (the plane wave metric that contains off-diagonal terms), see [4; 25; 24]. It was shown in [49] that the wave equation on the d-dimensional sphere, where $d \geqslant 3$ is odd, satisfies the Huygens' principle; this spherical wave equation can be transformed to the Euclidean wave

equation locally, but not globally.

While the examples of nontrivial diffusionless equations/systems built in [75; 47; 76; 72; 4; 25; 24] are primarily of a theoretical interest, the original wave equation (4.1) accounts for a variety of physically relevant (albeit sometimes simplified) models in acoustics, electromagnetism, elastodynamics, etc. Accordingly, understanding the shape of the lacunae (4.3) is of interest for the aforementioned application areas as lacunae represent the regions of "quietness" where the corresponding wave field is zero.

The objective of our work is to determine the shape of the lacunae in the solutions of the wave equation using fully connected artificial neural networks. In the current section, we adopt a simplified scenario to construct, test, and verify the proposed machine learning approach. Specifically, while the true phenomenon of lacunae is 3D and applies to solutions given by the Kirchhoff integral (4.2), hereafter we conduct the analysis and simulations in a one-dimensional (1D) setting. The domain of dependence for $u(\boldsymbol{x}, t)$ determined by the Kirchhoff integral is the surface of the backward light cone. To mimic that in 1D, we consider the function $u = u(x, t)$ and define its domain of dependence as the sum of two backward propagating rays:

$$\{(\xi, \tau) : \xi - x = \pm c(\tau - t), \;\; \xi \in \mathbb{R}, \;\; \tau \leqslant t\} \stackrel{\text{def}}{=} \mathscr{L}(x, t). \tag{4.4}$$

We do not need a full specification of u for our subsequent considerations. We only need a sufficient condition for $u(x, t)$ to be equal to zero, which we take as

$$u(x, t) = 0 \quad \text{if} \quad \mathscr{L}(x, t) \cap Q_f = \emptyset,$$

where Q_f is a given bounded domain in (1+1)D space-time: $Q_f \subset \mathbb{R} \times [0, +\infty)$. Accordingly, the function u is going to have a lacuna:

$$u(x, t) \equiv 0 \quad \forall (x, t) \in \left\{(\tilde{x}, \tilde{t}) : \mathscr{L}(\tilde{x}, \tilde{t}) \cap Q_f = \emptyset\right\} \stackrel{\text{def}}{=} \Lambda_1(Q_f). \tag{4.5}$$

The lacuna $\Lambda_1(Q_f)$ combines both the secondary and primary lacuna as per the discussion in Section 4.1. A purely secondary lacuna would be given by [cf. (4.3)]

$$u(x, t) \equiv 0 \quad \forall (x, t) \in \bigcap_{(\xi, \tau) \in Q_f} \left\{(\tilde{x}, \tilde{t}) : |\tilde{x} - \xi| < c(\tilde{t} - \tau), \; \tilde{t} > \tau\right\} \subset \Lambda_1(Q_f). \tag{4.6}$$

We emphasize that the proposed 1D construct is not accurate on the substance. Its only purpose is to provide an inexpensive testing framework for the neural networks described in this section. This construct is designed as a direct counterpart of the physical 3D setting and does

not represent a true solution of the 1D wave equation:

$$\frac{1}{c^2}u_{tt} - u_{xx} = f(x,t), \quad x \in \mathbb{R},\ t > 0. \tag{4.7}$$

The solution of (4.7) subject to zero initial conditions is given by the d'Alembert integral [cf. the Kirchhoff integral (4.2)]:

$$u(x,t) = \frac{c}{2}\int_0^t d\tau \int_{x+c(\tau-t)}^{x-c(\tau-t)} f(\xi,\tau)d\xi. \tag{4.8}$$

Unlike (4.4), the domain of dependence for the 1D solution (4.8) contains not only the two rays, but the entire in-between region as well:

$$\{(\xi,\tau):\ x + c(\tau - t) \leqslant \xi \leqslant x - c(\tau - t),\ \ \tau \leqslant t\} \stackrel{\text{def}}{=} \mathscr{D}(x,y).$$

Therefore, the solution $u = u(x,t)$ defined by (4.8) will not, generally speaking, have secondary lacunae as presented by (4.6).[1]

In Section 4.2, we provide a description of the numerical algorithm to build the data set, and in Section 4.3, we present the numerical results.

4.2 Construction of the Data Set

In this section, we describe the construction of the data set needed to train the neural network. To this end, we introduce a computational domain $\Omega := [a,b] \times [0,T]$ and discretize it in space and time using a uniform spatial $x_1, \ldots, x_{N_x}$ and temporal $t_1, \ldots, t_{N_t}$ mesh so that

$$\begin{aligned} x_j &= a + (j-1)\Delta x, \quad j = 1, \ldots, N_x \\ t_n &= (n-1)\Delta t, \quad n = 1, \ldots, N_t, \end{aligned}$$

where $\Delta x = \frac{b-a}{N_x - 1}$ and $\Delta t = \frac{T}{N_t - 1}$.

[1]The 1D case is special in the sense that while the dimension of the space is odd, the wave equation (4.7) is not Huygens' when driven by a source term. It may, however, demonstrate a Huygens' behavior with respect to the initial data [80].

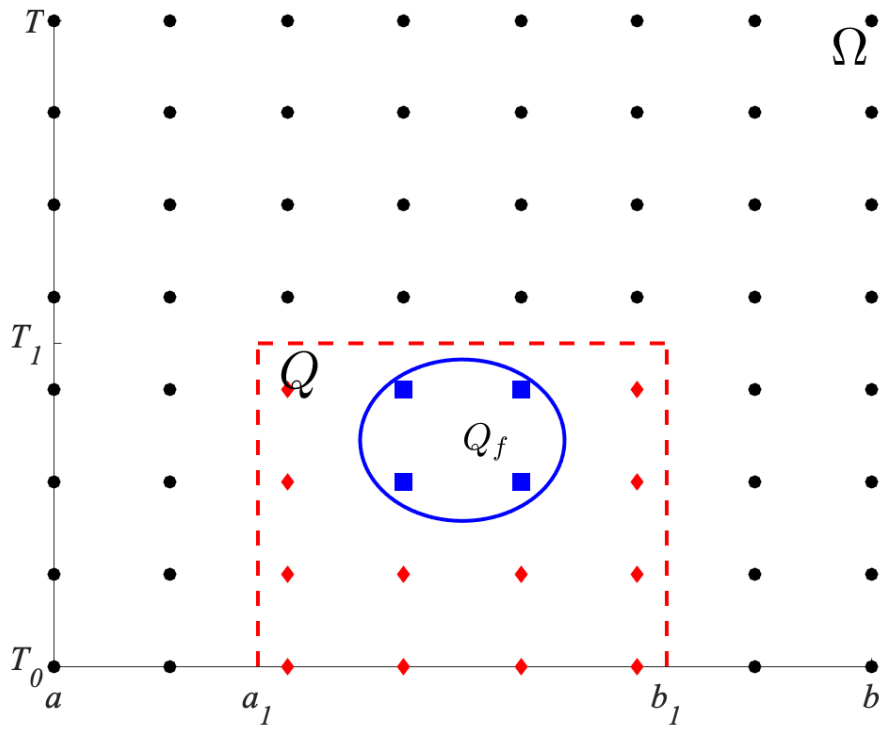

Figure 4.1: The computational domain Ω with subdomain Q inside the dotted red line, and Q_f inside the blue circle.

We assume that the domain Q_f that defines the lacuna Λ_1 in (4.5) lies inside a subdomain $Q \subset \Omega$, which, for the simplicity of implementation, is taken as a box $Q = [a_1, b_1] \times [T_0, T_1] \subseteq \Omega$, where $a \le a_1 < b_1 \le b$ and $0 \le T_0 < T_1 \le T$. We can then denote the nodes inside Q by

$$x_{j_\ell} = x_{j_1+(\ell-1)}, \quad \ell = 1, \ldots, N_x^{'} \le N_x,$$
$$t_{n_p} = t_{n_1+(p-1)}, \quad p = 1, \ldots, N_t^{'} \le N_t,$$

where x_{j_1} and t_{n_1} are the smallest x_j and t_n values that are inside the set Q and $x_{j_1+N_x^{'}-1}$ and $t_{n_1+N_t^{'}-1}$ are the largest x_j and t_n values that are inside the set Q, see Fig. 4.1.

The shape of the lacuna can then be determined by identifying the set of nodes, for which the characteristic lines $\mathscr{L}(x_j, t_n)$ emerging from the node $(x_j, t_n) \in \Omega,\ j = 1, \ldots, N_x,\ n = 1, \ldots, N_t$ (see formula (4.4)), pass through the domain Q_f. We therefore construct M training data sets by implementing Algorithm 2.

Algorithm 2: Generate lacunae data set

start : Introduce the computational domain Ω and its discretization by a uniform mesh $(x_j, t_n) \in \Omega,\ j = 1, \dots, N_x,\ n = 1, \dots, N_t$, and identify the set of nodes $(x_{j_\ell}, t_{n_p}) \in Q,\ \ell = 1, \dots, N_x^{'},\ p = 1, \dots, N_t^{'}$.

for *m=1,2,...,M* **do**

1. Generate a random positive integer $I^{(m)}$ and a set of domains $Q_{f_i^{(m)}} \subset Q$ for each $i = 1, \dots, I^{(m)}$, and define
$$Q_{f^{(m)}} = \Big(\bigcup_{i=1}^{I^{(m)}} Q_{f_i^{(m)}} \Big). \tag{4.9}$$

2. Construct the following matrices $\Phi^{(m)}$ and $\Psi^{(m)}$:

   - Matrix $\Phi^{(m)}$ with entries $\phi_{\ell,p}^{(m)}$ that indicate for each node $(x_{j_\ell}, t_{n_p}) \in Q$ whether or not it belongs to the domain $Q_{f^{(m)}}$, namely,
$$\phi_{\ell,p}^{(m)} = \begin{cases} 1, & (x_{j_\ell}, t_{n_p}) \in Q_{f^{(m)}}, \\ -1, & \text{otherwise.} \end{cases}$$

   - Matrix $\Psi^{(m)}$ with entries $\psi_{j,n}^{(m)}$ indicates whether or not the characteristic lines $\mathscr{L}(x_j, t_n)$ intersect with the domain $Q_{f^{(m)}}$, namely,
$$\psi_{j,n}^{(m)} = \begin{cases} -1, & (x_j, t_n) \in \Lambda_1(Q_{f^{(m)}}), \\ 1, & \text{otherwise.} \end{cases} \tag{4.10}$$
   In practice, we check whether there is a node $(x_{j_\ell}, t_{n_p}) \in Q_{f^{(m)}}$ and point $(\xi, t_{n_p}) \in \mathscr{L}(x_j, t_n)$, such that $|\xi - x_{j_\ell}| < \Delta x$, in which case $\psi_{j,n}^{(m)} = 1$; otherwise $\psi_{j,n}^{(m)} = -1$. Note that, the point (ξ, t_{n_p}) is not, generally speaking, a grid node. The constriction of $\Psi^{(m)}$ is visualized in Fig. 4.2.

output : the set $\{\Phi^{(m)}, \Psi^{(m)}\}_{m=1}^{M}$

Remark 1. In one spatial dimension, this approach to constructing the matrices $\Psi^{(m)},\ m = 1, \dots, M$, could be done with a more exact approach by finding the lowest and highest characteristic lines coming from the the connected regions $Q_{f_i^{(m)}}$, and finding the nodes between those lines. However, we believe a similar approach to the one described in Algorithm 2 will work better in 3D so we use it here as well.

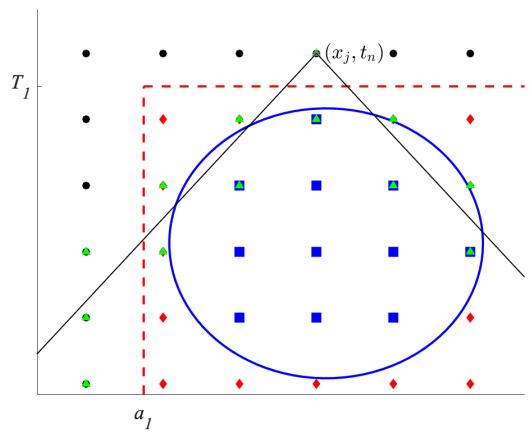

Figure 4.2: Characteristic lines from the point (x_j, t_n). Green triangles indicate nodes within Δx of the characteristic lines at each time step. If there exist a green triangle and blue square at the same node, then $\Psi_{j,n} = 1$, else $\Psi_{j,n} = -1$.

4.3 Numerical Results

In this section, we conduct a number of numerical experiments to demonstrate the performance of using a neural network for detecting the lacunae. All of our models are built and trained using the open source library PyTorch.

To define the set $Q_{f_i^{(m)}}$ in (4.9), we draw uniformly distributed values $x_i^{(m)} \sim \mathscr{U}(a_1, b_1)$, $t_i^{(m)} \sim \mathscr{U}(0, T_1)$, and $r_i^{(m)} \sim \mathscr{U}(0, R)$ for $i = 1, 2, ..I^{(m)}$, and then let

$$Q_{f_i^{(m)}} = \{(x, t) \in [a_1, b_1] \times [0, T_1] : (x - x_i^{(m)})^2 + (t - t_i^{(m)})^2 \leq (r_i^{(m)})^2\}. \tag{4.11}$$

Recall that $I^{(m)}$ is the number of sets on the right hand side of (4.9). As mentioned in Algorithm 2, this integer is randomly generated, and in our numerical examples it is an integer between 1 and 4. If $I^{(m)} > 1$, then we may have a disconnected set $Q_{f^{(m)}}$.

We train a neural network $\boldsymbol{N}_\Theta : \mathbb{R}^{N_x' \times N_t'} \longrightarrow \mathbb{R}^{N_x \times N_t}$ such that for a given set Q_f represented by the matrix Φ as described in Algorithm 2, the neural network will produce

$$\boldsymbol{N}_\Theta(\Phi) \approx \Psi,$$

where Ψ is the matrix representing the location of the lacunae $\Lambda_1(Q_f)$ and its compliment $(\Lambda_1(Q_f))^c$. For all of the examples, the computational domains are $\Omega = [-20, 20] \times [0, 20]$ and $Q = [-10, 10] \times [0, 10]$. We discretize Ω and Q such that $N_x = 64$, $N_t = 64$, $N_x' = 32$, and $N_t' = 32$.

The max radius in (4.11) we use is $R = 5$. We generated $M = 10{,}000$ data pairs for our training process, randomly splitting the data such that $M_{tr} = 8{,}000$ and $M_{val} = 2{,}000$. For these examples, we use a fully connected feedforward neural network $\boldsymbol{N}_\Theta : \mathbb{R}^{N_x' \times N_t'} \longrightarrow \mathbb{R}^{N_x \times N_t}$ to classify the data.

To train the neural network, we minimized the loss function

$$L_\Theta = \frac{1}{M} \sum_{m=1}^{M} \|\Psi^{(m)} - \boldsymbol{N}_\Theta(\Phi^{(m)})\|_F,$$

where $\|\cdot\|_F$ is the Frobenius norm using the Adam optimization algorithm with an initial learning rate of 10^{-4}. We ran the algorithm with batch sizes if 32 data samples for 200 epochs. The neural network consisted of $H = 3$ hidden layers with each hidden layer having 256 nodes. All of the activation functions for the hidden layers are the $LeakyReLU$ activation function (2.5), and the activation function for the output layer is the tanh activation function. Since the output activation function is the tanh activation function, then each element of the output matrix is in $(-1, 1)$. For both examples below, we train the neural networks for 100 epochs, which took about 150 seconds using a NVIDIA GeForce RTX 3060 laptop GPU. Note that since the input is a matrix, we need to reshape it into a vector in our neural network and then reshape the vector output back into a matrix, see Remark 3.1.1. The neural network architecture is given in Fig. 4.3.

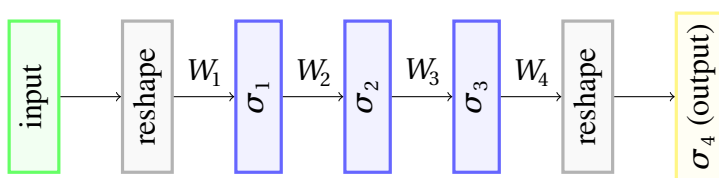

Figure 4.3: Neural network architecture to find the lacunae of the wave equation.

Lacunae neural network case 1: $I^{(m)} = 1$

In the first example, we consider a particular case where $I^{(m)} = 1$ for each $m = 1, \ldots, M$. After training the neural network, $\boldsymbol{N}_\Theta$, we apply it to a test set that is independent of the training and validation sets. For a given input matrix Φ, let Ψ^{ref} denote the reference solution matrix with elements $\psi^{ref}_{j,n} \in \{-1, 1\}$, $j = 1, \ldots, N_x$, $n = 1, \ldots, N_t$, determining whether or not the node (x_j, t_n) is in the lacuna, see (4.10).

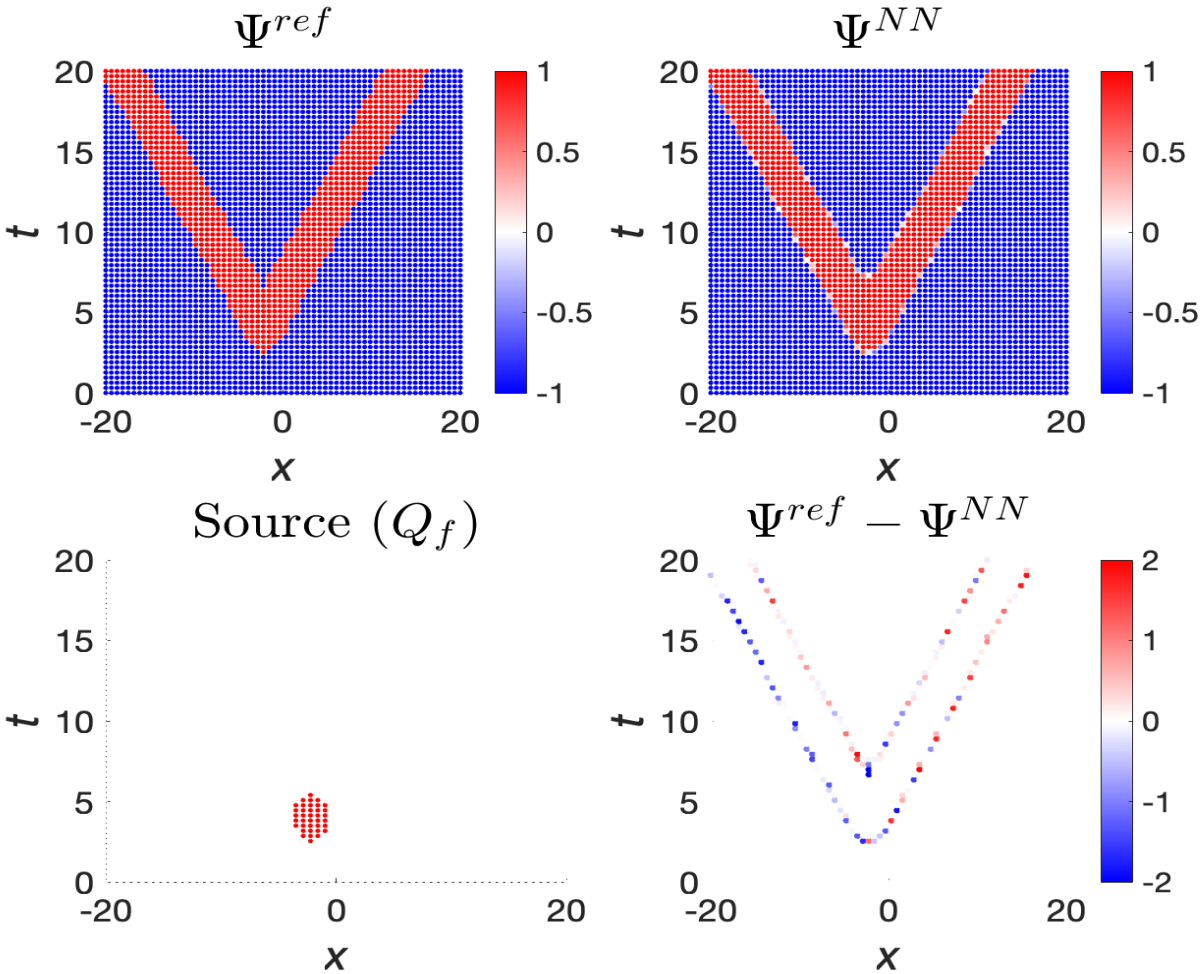

Figure 4.4: Reconstruction of the shape of the lacuna (4.5) by a neural network for the case $I^{(m)} = 1$. *Top left:* Reference solutions Ψ^{ref}. *Top right:* Neural Network solutions Ψ^{NN}. *Bottom left:* The sets Q_f. *Bottom right:* $\Psi^{ref} - \Psi^{NN}$.

For the neural network solution, $\Psi^{NN} := \boldsymbol{N}_\Theta(\Phi) \in (-1, 1)$, we state that it correctly identifies that the node (x_j, t_n) is in the right set, $\Lambda_1(Q_f)$ or $\Lambda_1(Q_f)^c$, if

$$(\psi_{j,n}^{NN} \leq 0 \text{ and } \psi_{j,n}^{ref} = -1) \quad \text{or} \quad (\psi_{j,n}^{nn} > 0 \text{ and } \psi_{j,n}^{ref} = 1),$$

and incorrectly identifies which set the node is in if

$$(\psi_{j,n}^{NN} > 0 \text{ and } \psi_{j,n}^{ref} = -1) \quad \text{or} \quad (\psi_{j,n}^{nn} \leq 0 \text{ and } \psi_{j,n}^{ref} = 1).$$

We can then determine how accurate the neural network is by calculating

$$accuracy = \frac{\text{\# of nodes correctly identified}}{\text{total \# of nodes}}. \tag{4.12}$$

From 1,000 test cases, the neural network identified which set each node belonged to with approximately 99.12% accuracy. In Fig. 4.4 we show an example taken from the test set. The top left represents the values for the reference solutions Ψ^{ref}, the top right represents the values of the neural network solution Ψ^{NN}, the bottom left shows nodes in the set Q_f, and the bottom right represents the difference $\Psi^{ref} - \Psi^{NN}$. For the top graphs, the values range from $[-1, 1]$ with -1 indicating nodes in $\Lambda_1(Q_f)$ with blue dots and 1 indicating nodes in $\Lambda_1(Q_f)^c$ with red dots. For the reference solution, the element values are either -1 or 1 so all the nodes are either dark blue or dark red respectively. The neural network has values between -1 and 1, thus the node colors in its graph might be different shades of red, blue, and white. On the plot for $\Psi^{ref} - \Psi^{NN}$, the white nodes indicates that the two solution are close to each other, the red nodes indicates that the Ψ^{ref} is greater than Ψ^{NN} and the blue nodes indicate Ψ^{ref} is less than Ψ^{NN}. Note that, the interior of the sets $\Lambda_1(Q_f)$ and $\Lambda_1(Q_f)^c$ for the neural network solution are clearly defined as very close to -1 or 1, but there is some uncertainty from the neural networks along the boundary. It is expected that the neural network would in general have more difficulty learning the edges of these sets.

Lacunae neural network case 2: $1 \leq I^{(m)} \leq 4$

In this case, we generate our data choosing $I^{(m)}$ to be a random integer such that $1 \leq I^{(m)} \leq 4$ for each $m = 1, \ldots, M$. Once trained, the neural network was able to predict which set, $\Lambda_1(Q_f)$ or $(\Lambda_1(Q_f))^c$, each node belongs to with approximately 98.55% accuracy over 1,000 test cases where the accuracy is calculated as in (4.12). In this section we show 2 examples (using the same layout as in Fig. 4.4). The first example, shown in Fig. 4.5, is taken from the test set. In the second example, shown in Fig. 4.6, we chose Q_f such that the lacuna (4.5) has a "pocket," i.e., a fully enclosed area. It is a part of the secondary lacuna (4.6).

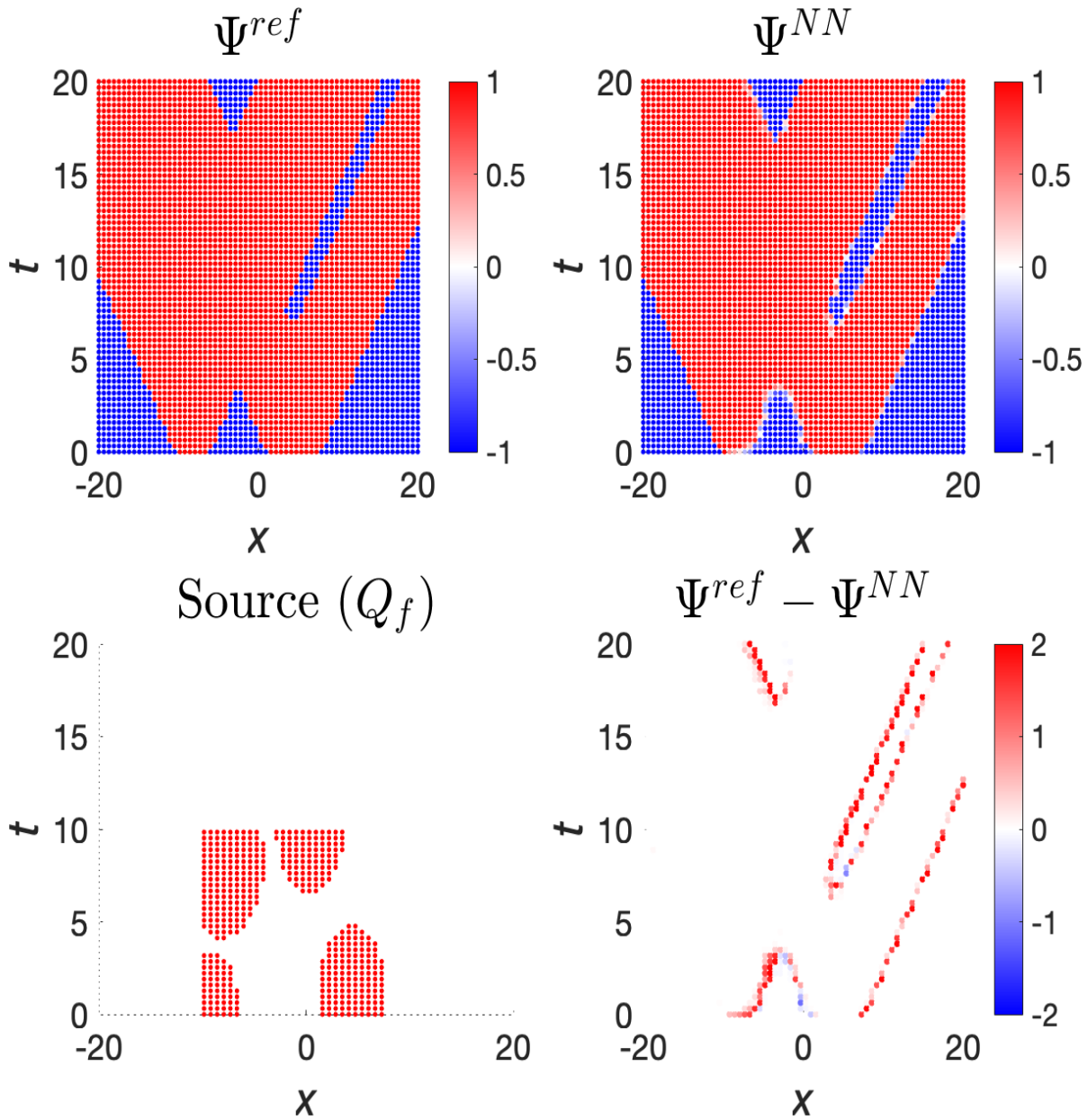

Figure 4.5: Reconstruction of the shape of the lacuna (4.5) by a neural network for the case $1 \leq I^{(m)} \leq 4$ with the same layout as in Figure 4.4

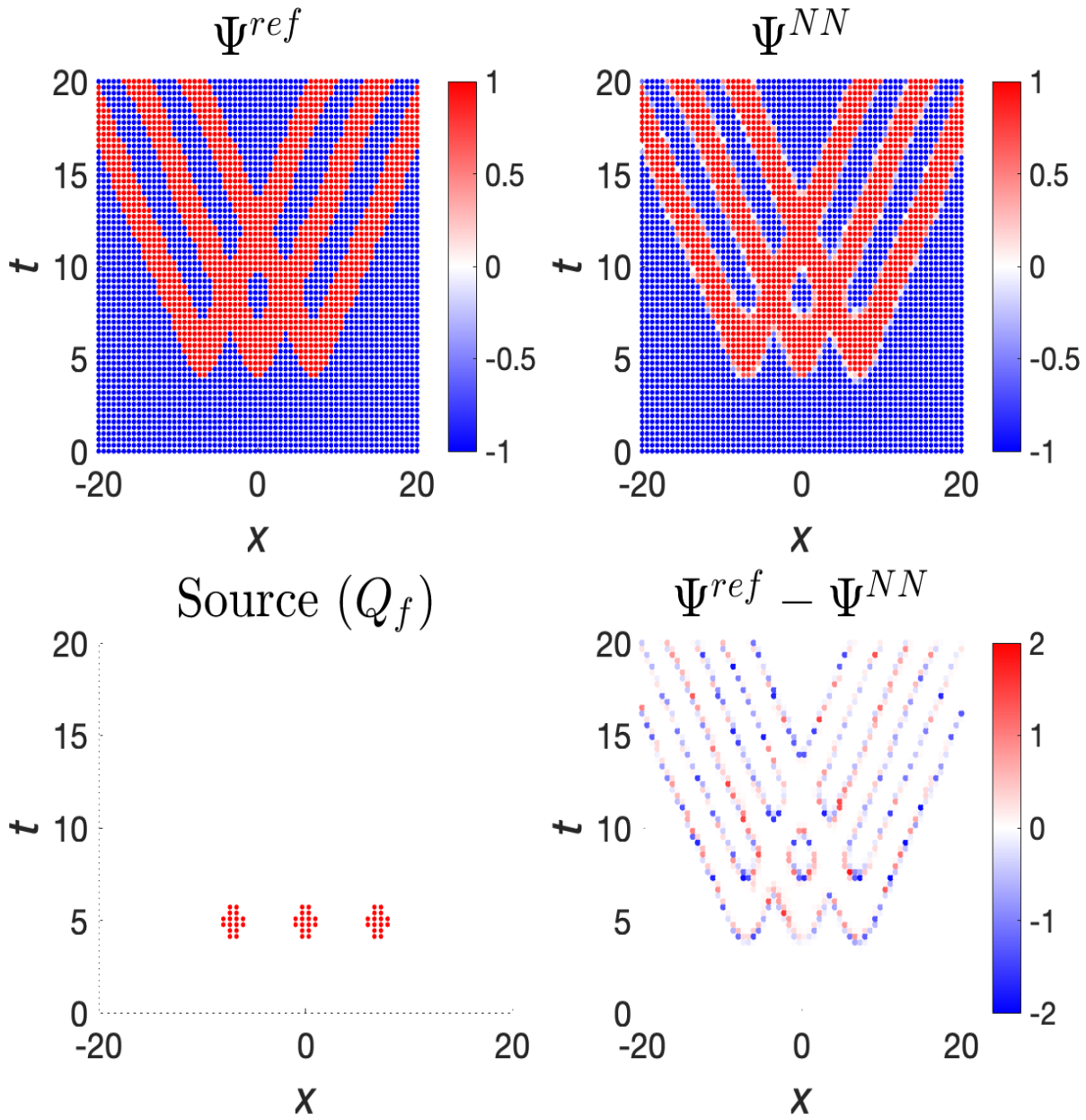

Figure 4.6: Reconstruction of the shape of the lacuna (4.5) by a neural network for the case $1 \leq I^{(m)} \leq 4$ with a hand crafted example such that the lacunae has a 'pocket'.

4.4 Conclusion

We have demonstrated that a fully connected neural network can accurately reconstruct the shape of the lacunae in an artificial one-dimensional setting introduced in Section 4.1. While we have trained our network to find the shape of a combined lacuna (4.5), we anticipate that having it identify only the secondary lacunae (4.6) would not present any additional issues. A challenging next step is to extend the proposed machine learning approach to a realistic three-dimensional setting where the secondary lacunae are defined according to (4.3) and account for the actual physics of the solutions to the wave equation (4.1).

CHAPTER

5

CONVOLUTIONS AND THE TENSOR TRAIN DECOMPOSITION

Convolution operations are used in different practical applications. They often involve large arrays of data and require optimization with respect to memory and computational cost. While input data are usually available only in a discrete form, the standard realization based on a vector-matrix representation is not often efficient since it leads to the use of sparse matrices. To resolve this problem, a tensor decomposition looks very attractive because it might reduce the volume of data very drastically minimizing the number of zero elements. In addition, arithmetic operations between tensors can be realized in an efficient way.

There are different forms of tensor decomposition. The most popular approach is based on the canonical decomposition [32] where a multidimensional array is represented (might be approximately) via a sum of outer products of vectors. For matrices, this kind of decomposition is reduced to the skeleton decomposition. However, it is well known that in the cases of multiple tensor dimensions, also known as tensor modes, it is unstable. The Tucker decomposition [79] represents a natural stable generalization of the canonical decomposition. It is able to provide a high compression rate. The main drawback of it is related to the so called curse of dimensionality. This means the complexity of the algorithm growths exponentially with the number of tensor

modes. A breakthrough in this direction was made by Oseledets and Tyrtyshnikov [61], [62] who proposed the Tensor Train (TT) Decomposition. Effectively, TT decomposition represents a generalization of the classical SVD decomposition to the case of multiple modes. It can also be interpreted as a hierarchical Tucker Decomposition [26].

Computing the TT decomposition fully can be very expensive if we use the standard TT-SVD algorithm given by Algorithm 3. For this reason, there are many modifications to this algorithms to help speed it up. One such modification is done in [52], where for sparse tensor data, an algorithm is given that is able to give the exact same result as the TT-SVD algorithm in a fraction of the time. Another approach uses the column space of the unfolding tensors to develop an algorithm to compute the TT cores in parallel, see [73]. Probably the most popular approach to efficiently compute the TT decomposition is to use a randomized algorithm, for this we refer to [34; 7; 18].

Maximal compression with TT decomposition can be reached in case only $2^d \times 2^d$ matrices are involved as proposed in the so called Quantized TT (QTT) algorithm [58]. As shown in [39], the convolution realized for multilevel Toeplitz matrices via QTT has a complexity that is logarithmic with respect to the number of elements in each mode, N, and proportional to the number of modes. It is proven that the result cannot be asymptotically improved. However, this algorithm is improved for finite and practically important $N \sim 10^4$ in [68] thanks to the cross convolution in the Fourier (image) space. The improvement is demonstrated for convolutions with three modes with the Newton potential. It is to be noted that QTT can be also applied to the Fast Fourier Transform (FTT) to decrease its complexity. In this way, Dolgov et al. [15] develop the super-fast FFT (QTT-FFT). It beats the standard FFT for extremely large N such as $N \sim 2^{60}$ for one mode tensors and $N \sim 2^{20}$ for tensors with three modes.

For practical applications, a very important issue is denoising. Real-life data, such as radar signals, are typically contaminated with noise. Denoising is not addressed in the papers we have cited previously. However, TT decomposition itself potentially has the property of denoising, owing to the SVD incorporated in the algorithm [17], [21]. In the current work, we propose and implement the low-rank modifications for the previously developed TT-SVD algorithm of [59]. These modifications speed up the computations. We also demonstrate the denoising capacity of numerical convolutions computed using the QTT decomposition. Specifically, we employ a common model for synthetic aperture radar (SAR) signal processing based on the convolution with a sinc imaging kernel (called the generalized ambiguity function) [19, Chapter 2] and show that, when a convolution with this kernel is evaluated in the QTT format, the noise level in the resulting image is substantially reduced compared to that in the original data.

Most papers on tensor convolution only consider the run time cost of the convolution after

the tensor decomposition has been applied to the objective function and the kernel function and either ignore the cost of the actual tensor decompositions or put as a side note. In this thesis, we consider every step of computing the convolution using the QTT-FFT algorithm including how to decompose the arrays into the QTT format, compute the QTT-FFT algorithm once in that format, and then extract the data back after the computation is done (Section 6.1). As the QTT decomposition is computationally expensive, we consider several approaches to speed up the decomposition run time. Without these modifications to the TT decomposition algorithm (see Algorithm 3) the convolution can take an extremely long time to compute and is not a practical approach. We go into more detail on this in section 6.3.1.

The methods we use to speed up our TT decompositions are based on truncating SVD ranks in the decomposition algorithm (Algorithm 3), and when we do this we area able to greatly reduce the noise in the data (Section 6.2). Thus, in Chapter 6, we present methods to compute convolutions in a reasonable time, while greatly reducing the noise in the data at the same time. Our contribution include developing and analysing new approached to speeding up the tensor train decomposition, see 6.1. In 6.2, we consider the effects these convolutions have on removing noise in data. Finally in 6.3, we show numerical examples and compare our results with other approaches to computing convolutions.

5.1 Convolution

The convolution operation is widely used in different applications in signal processing, data imaging, physics, and probability just to name a few. This operation is a way to combine two signals, usually represented as functions, in a way that produces a third signal with meaningful information. The convolution between two functions f and g is defined as

$$I(\boldsymbol{x}) = [f * g](\boldsymbol{x}) = \int_{\mathbb{R}^D} f(\boldsymbol{y}) g(\boldsymbol{x}-\boldsymbol{y})\, d\boldsymbol{y}, \quad \forall \boldsymbol{x} \in \mathbb{R}^D. \tag{5.1}$$

Often to compute the convolution numerically, we assume the support of f and g, denoted $\text{supp}(f)$ and $\text{supp}(g)$ respectively, are compact. For simplicity, in this thesis, we assume $\text{supp}(f) = \text{supp}(g) = [-L, L]^D$ for some $L \in \mathbb{R}$. Next, we discretize the domain $[-L, L]^D$ uniformly into N^D points such that

$$\begin{aligned} &\boldsymbol{x}_{j_1,\ldots,j_D} = (x_{j_1}, \ldots, x_{j_D}), \\ &x_{j_d} = -L + \frac{\Delta x}{2} + j_d \Delta x, \quad j_d = 0, \ldots, N-1, \quad d = 1, \ldots, D, \end{aligned}$$

where $\Delta x = \frac{2L}{N}$. We then let $\boldsymbol{f}$ and $\boldsymbol{g}$ be D-dimensional arrays such that

$$\begin{aligned} \boldsymbol{f}_{j_1,\ldots,j_D} &= f(x_{j_1,\ldots,j_D}) \\ \boldsymbol{g}_{j_1,\ldots,j_D} &= g(x_{j_1,\ldots,j_D}) \end{aligned} \tag{5.2}$$

for all $j_d = 0,\ldots,N-1$ and $d = 1,\ldots,D$. This leads to the discrete convolution $\boldsymbol{I}$ such that

$$\boldsymbol{I}_{j_1,\ldots,j_D} := \Delta x^D \sum_{i_D} \cdots \sum_{i_1} \boldsymbol{f}_{i_1,\ldots,i_D} \boldsymbol{g}_{j_1,\ldots,j_D+\frac{1}{2}(N_1,\ldots,N_D-1)-i_1,\ldots,i_D} \approx I(x_{j_1,\ldots,j_D}) \tag{5.3}$$

for all $j_d = 0,\ldots,N_d-1$, $d = 1,\ldots,D$ and where the sums are over all values that lead to legal subscripts. This Riemann sum approximation (5.3) to the integral (5.1) uses the midpoint rule, thus has $\mathcal{O}(\Delta x^2)$ accuracy.

Remark 5.1.1. *The convolution defined in* (5.3) *is equivalent to Matlab's* ***convn*** *function with the optional shape input set to 'same' and then multiplied by* Δx^D.

To compute this convolution directly takes $\mathcal{O}(N^{2D})$ operations, but it can be reduced to $\mathcal{O}(N^D \log(N^D))$ by using the fast Fourier transform (FFT) and the discrete convolution theorem. The FFT algorithm is an efficient algorithm used to compute the discrete Fourier transforms (DFT), where if $\hat{\mathscr{V}} \in \mathbb{R}^{N\times\ldots\times N}$ is the D-dimensional DFT of $\mathscr{V} \in \mathbb{R}^{N\times\ldots\times N}$ then

$$\hat{\mathscr{V}}_{\alpha_1,\ldots,\alpha_D} := DFT(\mathscr{V}) = \sum_{j_1,\ldots j_D=0}^{N-1} \mathscr{V}_{j_1,\ldots,j_D} \omega_N^{j_1\alpha_1} \cdots \omega_N^{j_D\alpha_D}$$

for all $\alpha_d = 0,\ldots,N-1$, $d = 1,\ldots,D$, where $\omega_N = e^{-\frac{2\pi\hat{\imath}}{N}}$ and $\hat{\imath} = \sqrt{-1}$ is the imaginary unit. Similarly, the D-dimensional inverse discrete Fourier transform (IDFT), such that

$$\mathscr{V} = IDFT(DFT(\mathscr{V})),$$

of the array $\hat{\mathscr{V}} \in \mathbb{R}^{N\times\ldots\times N}$ is given by

$$\mathscr{V}_{j_1,\ldots,j_D} = \frac{1}{N^D} \sum_{\alpha_1,\ldots,\alpha_D=0}^{N-1} \hat{\mathscr{V}}_{\alpha_1,\ldots,\alpha_D} \omega_{N_1}^{-j_1\alpha_1} \cdots \omega_{N_D}^{-j_D\alpha_D}$$

$j_d = 0,\ldots,N-1$, $d = 1,\ldots,D$.

Using the discrete Fourier transform, we can compute the circular convolution $\boldsymbol{I}^c = (\mathscr{V} \circledast \mathscr{W})$

defined as

$$\boldsymbol{I}^c_{j_1,\ldots,j_D} = \sum_{i_1,\ldots,i_D=0}^{N-1} \mathscr{V}_{i_1,\ldots,i_D} \hat{\mathscr{W}}_{j_1,\ldots,j_D-i_1,\ldots,i_D}$$
$$\hat{\mathscr{W}}_i = \mathscr{W}_j, \quad i \equiv j \bmod(N),$$

by taking the DFT of $\mathscr{W}$ and $\mathscr{V}$, multiplying the results together, and then taking the IDFT of the given result. Thus we have

$$\boldsymbol{I}^c = IDFT(DFT(\mathscr{W}) \odot DFT(\mathscr{V}))$$

where $\odot$ is Hadamard product (element wise product) of D-dimensional arrays. The circular convolution is the same as the convolution of two periodic functions (up to a constant scaling), thus to obtain the convolution given in (5.3) (also known as a linear convolution), we need to pad the vectors $\boldsymbol{f}$ and $\boldsymbol{g}$ with at least $N-1$ zeros. For example, given the vectors $\boldsymbol{f}^0, \boldsymbol{g}^0 \in \mathbb{R}^{2N-1}$ with

$$\boldsymbol{f}^0_j = \begin{cases} \boldsymbol{f}_j & 0 \le j \le N-1 \\ 0 & j > N-1 \end{cases}, \quad \text{and} \quad \boldsymbol{g}^0_j = \begin{cases} \boldsymbol{g}_j & 0 \le j \le N-1 \\ 0 & j > N-1 \end{cases},$$

and $\boldsymbol{I}^c = (\boldsymbol{f}^0 \circledast \boldsymbol{g}^0)$ as the circular convolution between them, the linear convolution $\boldsymbol{I}$ in (5.3) is given by

$$\boldsymbol{I}_j = \Delta x^D \boldsymbol{I}^c_{j+\frac{N-1}{2}}, \quad j = 0, \ldots, N-1.$$

In this thesis, we let g be a predefined kernel, such as the SAR generalized ambiguity function (GAF) (see Section 5.2 and [19, Chapter 2] for detail) and f be a smooth gradually varying function contaminated with white noise. To compute the convolution, we use the QTT decomposition [41] and the QTT-FFT algorithm [15]. The QTT decomposition is a special case of the more general TT decomposition (see Section 5.3 and [59] for detail).

5.2 Synthetic Aperture Radar (SAR)

SAR is a coherent remote sensing technology capable of producing two-dimensional images of the Earth's surface from overhead platforms (airborne or spaceborne). SAR illuminates the chosen area on the surface of the Earth with microwaves (specially modulated pulses) and generates the image by digitally processing the returns (i.e., reflected signals). SAR processing involves the application of the matched filter and summation along the synthetic array, which

is a collection of successive locations of the SAR antenna along the flight path. Matched filtering yields the image in the direction normal to the platform flight trajectory or orbit (called cross-track or range), while summation along the array yields the image in the direction parallel to the trajectory or orbit (along-the-track or azimuth).

Mathematically, each of the two signal processing stages can be interpreted as convolution of the signal received by the SAR antenna with a known function. Equivalently, it can be represented as a convolution of the ground reflectivity function, which is the unknown quantity that SAR aims to reconstruct, with the imaging kernel or generalized ambiguity function. The advantage of this equivalent representation is that, it leads to a very convenient partition: the GAF depends on the characteristics of the imaging system whereas the ground reflectivity function is determined by the properties of the target. Moreover, image representation via GAF allows one to see clearly how signal compression (a property that pertains to SAR interrogating waveforms) enables SAR resolution, i.e., the capacity of the sensor to distinguish between closely located targets.

In the simplest possible imaging scenario, when the propagation of radar signals between the antenna and the target is assumed unobstructed and a number of additional assumptions also hold, the GAF in either range or azimuthal direction is given by the sinc (or spherical Bessel) function:

$$g(x) = A\,\mathrm{sinc}\left(\pi \frac{x}{\Delta_x}\right) \equiv A \frac{\sin\left(\pi \frac{x}{\Delta_x}\right)}{\pi \frac{x}{\Delta_x}}, \tag{5.4}$$

where the constant A is determined by normalization, x denotes a given direction, and the quantity Δ_x is the resolution in this direction. From formula (5.4) we see that, the resolution is defined as half-width of the sinc main lobe, i.e., the distance from is central maximum to the first zero. When x is the range direction (cross-track), the resolution Δ_x is inversely proportional to the SAR signal bandwidth, see [19, Section 2.4.4]. When x is the azimuthal direction (along-the-track), the resolution is inversely proportional to the length of the synthetic array, i.e., synthetic aperture, see [19, Section 2.4.3]. Note that, lower values of Δ_x correspond to better resolution in the sense that SAR can tell between the targets located closer to one another. It can also be shown that, as $\Delta_x \to 0$ the GAF given by (5.4) converges to the δ-function in the sense of distributions [20, Section 3.3]. In this case, the image, which is a convolution of the ground reflectivity with the GAF, coincides with ground reflectivity. This would be an ideal situation, because the image would reconstruct the unknown ground reflectivity exactly. This situation, however, is never realized in practice, because having $\Delta_x \to 0$ requires either the SAR bandwidth (range direction) or synthetic aperture (azimuthal direction) become infinitely large, which is not possible.

The literature on SAR imaging is vast. Among the more mathematical sources, we mention

the monographs [9] and [19].

5.3 Tensor Train Decomposition

Consider the K-mode, tensor $\mathscr{A} \in \mathbb{C}^{M_1 \times \ldots \times M_K}$ such that

$$\mathscr{A} = a(i_1, \ldots, i_K), \quad i_k = 0, \ldots, M_k - 1, \quad k = 1, \ldots, K, \tag{5.5}$$

where M_k is the size of each mode, and $a(i_1, \ldots, i_K) \in \mathbb{C}$ are the elements of the tensor $\mathscr{A}$ for all $i_k = 0, \ldots, M_k - 1$ and $k = 1, \ldots, K$. The tensor train format of $\mathscr{A}$ decomposes the tensor into K cores $\mathscr{A}^{(k)} \in \mathbb{C}^{r_{k-1} \times M_k \times r_k}$ such that

$$a(i_1, \ldots, i_K) = \boldsymbol{A}^{(1)}_{i_1} \boldsymbol{A}^{(2)}_{i_2} \cdots \boldsymbol{A}^{(K)}_{i_K}, \tag{5.6}$$

where the matrices $\mathscr{A}^{(k)}(:, i_k, :) = \boldsymbol{A}^{(k)}_{i_k} \in \mathbb{C}^{r_{k-1} \times r_k}$, for all $i_k = 0, \ldots, M_k - 1$, $k = 1, \ldots, K$ (In Matlab notation, $\boldsymbol{A}^{(k)}_{i_k} = \text{squeeze}(\mathscr{A}^{(k)}(:, i_k, :))$, where squeez() is used to conver the $\mathbb{C}^{r_{k-1} \times 1 \times r_k}$ tensor into a $\mathbb{C}^{r_{k-1} \times r_k}$ matrix).

We can also represent the TT decomposition as the product of tensor contraction operators. Define the tensor contraction between the tensors $\mathscr{A} \in \mathbb{C}^{M_1 \times \ldots \times M_K}$ and $\mathscr{B} \in \mathbb{C}^{M_K \times \ldots \times M_{\bar{K}}}$ (note that the first dimension size of $\mathscr{B}$ equals the last dimension size of $\mathscr{A}$) as $\mathscr{C} = \mathscr{A} \circ \mathscr{B} \in \mathbb{C}^{M_1 \times \ldots \times M_{K-1} \times M_{K+1} \times \ldots \times M_{\bar{K}}}$ where

$$\mathscr{C}(i_1, \ldots, i_{K-1}, i_{K+1}, \ldots, i_{\bar{K}}) = \sum_{p=0}^{M_K - 1} \mathscr{A}(i_1, \ldots, p) \mathscr{B}(p, \ldots, i_{\bar{K}}).$$

Then the TT format of $\mathscr{A}$ can be represented as

$$\mathscr{A} = \mathscr{A}^{(1)} \circ \ldots \circ \mathscr{A}^{(K)}.$$

Since we are interested in the case when $a(i_1, \ldots, i_K) \in \mathbb{C}$, we impose the condition $r_0 = r_K = 1$. The matrix dimensions $\{r_k\}_{k=0}^{K}$ are referred to as the TT-ranks of the tensor decomposition and the 3−mode tensors $\mathscr{A}^{(k)}$ are the TT-cores. Let $M = \max_{1 \le k \le K} M_k$ and $r = \max_{1 \le k \le K-1} r_k$, then the the tensor $\mathscr{A}$, which has $\mathscr{O}(M^K)$ elements, can be represented with $\mathscr{O}(M K r^2)$ elements in the TT format.

Before we show how to find the TT-cores, we first need to define a few properties of tensors.

First, let the matrix $\boldsymbol{A}^{\{k\}}$ be the k-th unfolding of the tensor $\mathscr{A}$ such that

$$\boldsymbol{A}^{\{k\}}(\alpha,\beta) = a(i_1,\ldots,i_K), \tag{5.7}$$

$$\alpha = i_1 + i_2 M_1 + \ldots + i_k \Pi_{l=1}^{k-1} M_l, \quad \beta = i_{k+1} + i_{k+2} M_{k+1} + \ldots + i_K \Pi_{l=k+1}^{K-1} M_l. \tag{5.8}$$

Thus, we have that $\boldsymbol{A}^{\{k\}} \in \mathbb{C}^{M_1 M_2 \ldots M_k \times M_{k+1} M_{k+2} \ldots M_K}$ which we write as

$$\boldsymbol{A}^{\{k\}} = a(i_1 \ldots i_k, i_{k+1} \ldots i_K). \tag{5.9}$$

We denote the process of unfolding a tensor $\mathscr{A}$ into a matrix $\boldsymbol{A}^{\{k\}} \in \mathbb{C}^{M_1 M_2 \ldots M_k \times M_{k+1} M_{k+2} \ldots M_K}$ as

$$\boldsymbol{A}^{\{k\}} = \text{reshape}(\mathscr{A}, [M_1 M_2 \ldots M_k, M_{k+1} M_{k+2} \ldots M_K])$$

and folding a matrix into a tensor $\mathscr{A} \in \mathbb{C}^{M_1 \times \ldots \times M_K}$ as

$$\mathscr{A} = \text{reshape}(\boldsymbol{A}^{\{k\}}, [M_1, M_2, \ldots, M_K]).$$

(Note this is to be consistent with the Matlab function reshape()).

From [59] it can be shown that there exist a TT-decomposition of $\mathscr{A}$ such that

$$r_k = \text{rank}(A^{\{k\}}), \quad k = 1, \ldots, K. \tag{5.10}$$

Denote the Frobenius norm of a tensor $\mathscr{A} \in \mathbb{C}^{M_1 \times \ldots \times M_K}$ as

$$\|\mathscr{A}\|_F = \sqrt{\sum_{i_1=0}^{M_1-1} \ldots \sum_{i_K=0}^{M_K-1} |a(i_1,\ldots,i_K)|^2}, \tag{5.11}$$

and the ε_k-rank of the matrix $\boldsymbol{A}^{\{k\}}$ as

$$\text{rank}_{\varepsilon_k}(\boldsymbol{A}^{\{k\}}) = \min\{\text{rank}(\boldsymbol{B}) : \|\boldsymbol{A}^{\{k\}} - \boldsymbol{B}\|_F \leq \varepsilon_k\}. \tag{5.12}$$

Given a set $\{\varepsilon_k\}_{k=1}^K$, we can approximate the tensor $\mathscr{A}$ with a tensor $\tilde{\mathscr{A}}$ in the TT format such that it has TT-ranks $\tilde{r}_k \leq \text{rank}_{\varepsilon_k}(\boldsymbol{A}^{\{k\}})$ and

$$\|\mathscr{A} - \tilde{\mathscr{A}}\|_F \leq \varepsilon, \quad \varepsilon^2 = \varepsilon_1^2 + \ldots + \varepsilon_{K-1}^2. \tag{5.13}$$

In Algorithm 3, we present the TT-SVD algorithm [59] which computes a TT-decomposition

of a tensor $\mathscr{A}$ with a prescribed accuracy ε. In Section 6.1, we present some modifications to this algorithm that relax the prescribed tolerance and allow us to compute an approximate decomposition faster. For a tensor $\mathscr{A} \in \mathbb{C}^{M_1 \times \ldots \times M_K}$, define

$$|\mathscr{A}| = \text{number of elements in } \mathscr{A} = M_1 M_2 \ldots M_K.$$

Algorithm 3: TT-SVD

input : $\mathscr{A}$, ε
output : TT-Cores: $\mathscr{A}^{(1)}, \mathscr{A}^{(2)}, \ldots, \mathscr{A}^{(K)}$
$\tau := \frac{\varepsilon}{\sqrt{M-1}} \|\mathscr{A}\|_F$
$r_0 := 1$;
for *k=1,...,K-1* **do**
  $\boldsymbol{A}^{\{k\}} := \text{reshape}(\mathscr{A}, [M_k r_{k-1}, \frac{|\mathscr{A}|}{M_k r_{k-1}}])$
  Compute truncated SVD: $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^* + \boldsymbol{E} = \boldsymbol{A}^{\{k\}}$ such that $\|\boldsymbol{E}\|_F \leq \tau$
  $r_k := \text{rank}(\boldsymbol{\Sigma}) = \text{rank}_\tau(\boldsymbol{A}^{\{k\}})$
  $\mathscr{A}^{(k)} := \text{reshape}(\boldsymbol{U}, [r_{k-1}, M_k, r_k])$
  $\mathscr{A} := \boldsymbol{\Sigma}\boldsymbol{V}^*$
end
$\mathscr{A}^{(K)} := \mathscr{A}$

The TT-decomposition can also be applied to tensors with a small number of modes by using the quantized tensor train decomposition (QTT). For instance, let $\boldsymbol{v} \in \mathbb{C}^{2^K}$ be a vector (1-mode tensor). To apply the QTT-decomposition of $\boldsymbol{v}$, we reshape it into the K-mode tensor $\mathscr{V} \in \mathbb{C}^{2 \times \ldots \times 2}$ such that

$$\mathscr{V}(i_1, i_2, \ldots, i_K) = \boldsymbol{v}(i)$$

where

$$i = \sum_{k=1}^{K} i_k 2^{k-1}, \quad i_k = 0, 1,$$

then compute the TT-decomposition of the tensor $\mathscr{V}$ (you can think of $i_K \ldots i_1$ as the binary representation of i). Extending the QTT-decomposition to matrices (2-mode tensors) $\boldsymbol{V} \in \mathbb{C}^{2^K \times 2^K}$ can be done similarly by reshaping them into $2K$-mode tensors $\mathscr{V} \in \mathbb{C}^{2 \times \ldots \times 2}$, then computing the TT-decomposition of $\mathscr{V}$.

We can approximate the discrete Fourier transform of a vector $\boldsymbol{v} \in \mathbb{R}^{2^K}$ (or 2D discrete Fourier transform of a matrix $\boldsymbol{V} \in \mathbb{R}^{2^K \times 2^K}$) in the QTT format using what is known as the QTT-FFT approximation algorithm [15]. Let $\hat{\boldsymbol{v}} = DFT(\boldsymbol{v})$ be the discrete Fourier transform of $\boldsymbol{v}$ and let $\mathscr{V}$ and $\hat{\mathscr{V}}$ be the tensors in the QTT-format that represent the vectors $\boldsymbol{v}$ and $\hat{\boldsymbol{v}}$ respectively.

Given $\mathcal{V}$, the QTT-FFT approximation algorithm can approximate $\hat{\mathcal{V}}$ with a tensor $\tilde{\mathcal{V}}$ such that

$$\|\tilde{\mathcal{V}} - \hat{\mathcal{V}}\|_F \leq \varepsilon \tag{5.14}$$

for some given tolerance ε. Similarly, we could prescribe some max TT-rank, $\hat{R}_{\max}$, for the QTT-FFT algorithm such that $\tilde{r}_k \leq \hat{R}_{\max}$ for all TT-ranks of $\tilde{\mathcal{V}}$, $\{\tilde{r}_k\}_{k=0}^{K}$. The QTT-FFT algorithm can easily be modified to the inverse Fourier transform of a vector (or matrix) in the QTT format, which we denote as the QTT-iFFT algorithm.

CHAPTER

6

COMPUTING DISCRETE CONVOLUTIONS WITH THE QUANTIZED TENSOR TRAIN DECOMPOSITION

6.1 Computing the Convolution with the QTT Decomposition

Recall the integral (5.1) given as

$$I(\boldsymbol{x}) = [f * g](\boldsymbol{x}) = \int_{\mathbb{R}^D} f(\boldsymbol{y}) g(\boldsymbol{x} - \boldsymbol{y})\, d\boldsymbol{y},$$

where f is the function of interest, and g is a given kernel. In practice, we often are not given the function f explicitly, but instead are given noisy data

$$f_\xi(\boldsymbol{x_j}) = f(\boldsymbol{x_j}) + \xi_j \tag{6.1}$$

at discrete points $\boldsymbol{x}_{\boldsymbol{j}}$, $\boldsymbol{j} = (j_1, \ldots, j_D)$. In particular, representing the ground reflectivity function for SAR reconstruction in the form (6.1) helps one model the noise in the received data. We assume that $\xi_{\boldsymbol{j}}$ is white noise from a normal distribution with the standard deviation σ, i.e. $\xi_{\boldsymbol{j}} \sim \mathcal{N}(0, \sigma^2)$.

We propose numerically approximating the integral (5.1) using the quantized tensor train (QTT) decomposition. For this, we assume $\text{supp}(f_\xi), \text{supp}(g) \in [-L, L]^D$ and discretize them into D-dimensional arrays $\boldsymbol{f}_\xi$ and $\boldsymbol{g}$ as in (5.2) with $N = 2^{K-1} - 1$. To represent the arrays in the QTT format, we pad them with zeros such that the new arrays are D-mode tensors in $\mathbb{R}^{2^K \times \ldots \times 2^K}$. We can relax the condition on the size N, but to compute the convolution with an FFT algorithm, we need to zero-pad each dimension with at least $N-1$ extra zeros (see Section 5.1). Also, for the QTT-decomposition we need each dimension to be of size 2^K for some $K \in \mathbb{N}$. Let $\boldsymbol{\mathcal{F}}_\xi, \boldsymbol{\mathcal{G}}$ be the zero padded tensors representing $\boldsymbol{f}_\xi$ and $\boldsymbol{g}$ respectively in the QTT format. Here, we assume that the discretization of f, $\boldsymbol{f}$, has a low, but not exactly known, TT-rank in the QTT-format. This is motivated by the fact that many standard piecewise smooth functions naturally have a low TT-rank, see [60; 23; 41].

In order to find approximations to these tensors in the TT-format, we modify the original TT-SVD algorithm. This is because with the full TT-SVD algorithm, if the tolerance $\boldsymbol{\varepsilon}$ is small, see equation (5.14), the TT-decomposition has close to full rank. Not only does it take a very long time to compute these decompositions, but most of the noise is still present. However, if $\boldsymbol{\varepsilon}$ is too large, then the TT-SVD algorithm looses too much information about the true function f. For these reasons, we present slight modifications to the TT-SVD algorithm. They are needed to significantly reduce the computing time, as illustrated by the example in Section 6.3.1.

We will consider three different modifications to the TT-SVD algorithm. These modifications are as follows:

(1) Set some max rank R_{max} and truncate the SVD in Algorithm 3 with ranks less than or equal to this threshold. Denote this method as the **max rank TT-SVD** algorithm.

(2) Set some max rank R_{max} and replace the SVD in Algorithm 3 with a randomized SVD (RSVD) given in [30] with max ranks set to R_{max} (see Appendix A.1.1). Denote this method as the **max rank TT-RSVD** algorithm. Note that for this algorithm we also need to prescribe an oversampling parameter p. There are several randomized SVD algorithms we could chose from, but due to simplicity and effectiveness, we use the approach described in Appendix A.1.1. This algorithm realizes the direct SVD.

(3) Truncate the SVD in Algorithm 3 based on when there is a relative drop in singular values,

i.e. if $\frac{\sigma_{k+1}}{\sigma_k} < \delta$ for a given threshold delta, then truncate the singular values less than σ_k. Denote this method as the **SV drop off TT-SVD** algorithm.

For the **max rank TT-RSVD**, if the unfolding matrices $\boldsymbol{A}^{\{k\}} \in \mathbb{R}^{m_k \times n_k}$, where $\min(m_k, n_k) \leq R_{\max} + p$, then we revert to the **max rank TT-SVD** algorithm (without the randomized SVD).

We can modify the QTT-FFT and QTT-iFFT algorithms in a similar manner to the our modifications of the TT-SVD algorithms to get low rank approximation to the discrete Fourier transform representations of $\mathscr{F}_\xi$ and $\mathscr{G}$. For this, we replace the SVD in the QTT-FFT algorithm (QTT-iFFT) with the truncated SVD algorithms (1)-(3) given above, but with possibly a different max rank which we will denote $\hat{R}_{max}$ for (1) and (2), or different threshold $\hat{\delta}$ for (3). For the examples in Section 6.3, we will distinguish between R_{max} and $\hat{R}_{max}$, however we will use the same threshold for δ in the TT-SVD algorithm, as well as the QTT-FFT algorithm thus we will not distinguish between the two. Note that using the threshold (1) in the QTT-FFT algorithm is not new, and is mentioned in [15].

With these above modifications to the TT-SVD algorithm and QTT-FFT (QTT-iFFT) algorithms, we propose the following algorithm (Algorithm 4) to approximate the convolution between the D-dimensional arrays $\boldsymbol{f}$ and $\boldsymbol{g}$. For this algorithm, denote

- $QFFT_{\hat{R}_{max}(\hat{\delta})}$: QTT-FFT algorithm with a max rank of $\hat{R}_{max}$ (or threshold $\hat{\delta}$),
- $QiFFT_{\hat{R}_{max}(\hat{\delta})}$: QTT-iFFT algorithm with a max rank of $\hat{R}_{max}$ (or threshold $\hat{\delta}$).

Algorithm 4: QTT convolution

input : $\boldsymbol{f}_\xi, \boldsymbol{g}$
output : $\boldsymbol{I}$
Step 1: $\mathscr{F}_\xi$ = reshape($\boldsymbol{f}_\xi$, [2, ..., 2]), $\mathscr{G}$ = reshape($\boldsymbol{g}$, [2, ..., 2])
Step 2: Decompose $\mathscr{F}_\xi$ and $\mathscr{G}$ into the QTT format using one of the modified TT-SVD algorithms.
Step 3: $\mathscr{I} = QiFFT_{\hat{R}_{max}(\hat{\delta})}(QFFT_{\hat{R}_{max}(\hat{\delta})}(\mathscr{F}_\xi) \odot QFFT_{\hat{R}_{max}(\hat{\delta})}(\mathscr{G}))$.
Step 4: Retrieve $\boldsymbol{I}$ from $\mathscr{I}$. (see Algorithm 5)

In Theorem 6.1.2, we show the asymptotic run time behavior of computing a convolution in one spacial dimension ($D = 1$) with the **max rank TT-SVD** algorithm. First, we prove an auxiliary result about the size of the unfolding matrices for this algorithm, see Lemma 6.1.1. For Theorem 6.1.2, we consider the whole process of converting the vector into the QTT-format, computing the convolution, then converting the convolution in the QTT format back into a vector, as is deminstrated in Algorithm 4. For the last step, to convert a tensor in the TT-format back into the standard format, we use the 'full' algorithm from the Matlab toolbox **oseledets/TT-Toolbox**. This is given in Algorithm 5. We then reshape this tensor into a vector, which has a negligible run time.

Algorithm 5: Full

input : $\mathscr{A}^{(1)}, \mathscr{A}^{(2)}, ..., \mathscr{A}^{(K)}$, and size of output tensor $[M_1, \ldots, M_k]$
output : $\mathscr{A} \in \mathbb{C}^{M_1 \times \ldots \times M_k}$
Let $\boldsymbol{A} = \mathscr{A}^{(1)}$
for k=2,...,K **do**
  $\boldsymbol{A} = \text{reshape}(\boldsymbol{A}, [\frac{|\boldsymbol{A}|}{r_{k-1}}, r_{k-1}])$
  $\boldsymbol{B} = \text{reshape}(\mathscr{A}^{(k)}, [r_{k-1}, 2r_k])$
  $\boldsymbol{A} = \boldsymbol{A}\boldsymbol{B}$
end
$\mathscr{A} = \text{reshape}(\boldsymbol{A}, [M_1, \ldots, M_k])$

Lemma 6.1.1. *Let $\mathscr{A} \in \mathbb{R}^{2\times\ldots\times 2}$ be a K-mode tensor. Let $\{\boldsymbol{A}^{\{k\}}\}_{k=1}^{K-1}$ be the unfolding matrices of $\mathscr{A}$ in the max rank TT-SVD algorithm with a max rank of $R_{\max}$ and with each $\boldsymbol{A}^{\{k\}} \in \mathbb{C}^{m_k \times n_k}$. Then*

$$m_k = 2r_{k-1} \leq 2R_{\max} \quad \textit{and} \quad n_k = 2^{K-k}.$$

Proof. Since $M_k = 2$ for all k, the proof for $m_k = 2r_{k-1} \leq 2R_{\max}$ is trivial by the first line inside the for loop in Algorithm 3. For n_k, we do a proof by induction. First, note that $|\boldsymbol{A}^{\{1\}}| = 2^K$ and $r_0 = 1$, thus

$$n_1 = \frac{|\boldsymbol{A}^{\{1\}}|}{2r_0} = \frac{2^K}{2} = 2^{K-1}.$$

Assume $n_\ell = 2^{K-\ell}$ for all $1 \leq \ell \leq k-1$. Then,

$$\begin{aligned} n_k &= \frac{|\boldsymbol{A}^{\{k\}}|}{2r_{k-1}} \\ &= \frac{|\boldsymbol{\Sigma}_{k-1}\boldsymbol{V}_{k-1}^*|}{2r_{k-1}} \\ &= \frac{r_{k-1}n_{k-1}}{2r_{k-1}} \\ &= \frac{n_{k-1}}{2} \\ &= \frac{2^{K-(k-1)}}{2} = 2^{K-k}. \end{aligned}$$

Thus, we get

$$m_k = 2r_{k-1} \leq 2R_{\max} \quad \text{and} \quad n_k = 2^{K-k}.$$

□

Theorem 6.1.2. *Let $\boldsymbol{f}_\xi, \boldsymbol{g} \in \mathbb{R}^{2^{K-1}-1}$ for some positive integer K. Then the computational complexity, $C_{QTT\text{-}conv}$, of approximating the convolution $\boldsymbol{f}_\xi * \boldsymbol{g}$ with the max rank TT-SVD and*

max rank QTT-SVD algorithms described above is

$$C_{QTT\text{-}conv} \leq \mathcal{O}(R_{\max}^2 2^K)$$

where $R_{\max}$ is the prescribed max rank for both the TT-SVD algorithms and the QTT-FFT algorithm.

Proof. We show that the computational complexity is dominated asymptotically by the max rank TT-SVD algorithms and the full tensor algorithm. First, let C_{svd} be the computational cost of the SVD in big $\mathcal{O}$ notation. Then, for a matrix $\boldsymbol{A} \in \mathbb{C}^{m\times n}$, $C_{svd}(\boldsymbol{A}) = \mathcal{O}(mn\min(m,n))$. Note that, in Algorithm 3 (as well as in our max rank modifications), the computational complexity is dominated by the SVD algorithm. Denote the unfolding matrices at the kth iterations as $\boldsymbol{A}^{\{k\}} \in \mathbb{C}^{m_k\times n_k}$. Hence, the computational cost of the max rank TT-SVD algorithm is

$$\begin{aligned}
\sum_{k=1}^{K-1} C_{svd}(\boldsymbol{A}^{\{k\}}) &= \sum_{k=1}^{K-1} \mathcal{O}(m_k n_k \min(m_k, n_k)) \\
&\leq \sum_{k=1}^{K-1} \mathcal{O}((2R_{\max})^2 2^{K-k}) \\
&= 4R_{\max}^2 \sum_{k=1}^{K-1} \mathcal{O}(2^k) \\
&= 4R_{\max}^2 \mathcal{O}(2^K - 2) \\
&= \mathcal{O}(R_{\max}^2 2^K).
\end{aligned}$$

From [15], we have that for the QTT-FFT and QTT-iFFT algorithms, the computational complexity is $\mathcal{O}(K^2 R_{\max}^3)$. In Algorithm 5, the computational complexity comes from the multiplication $\boldsymbol{AB}$ in every loop. For the kth loop, $\boldsymbol{A} \in \mathbb{C}^{2^{k-1}\times r_{k-1}}$ and $\boldsymbol{B} \in \mathbb{R}^{r_{k-1}\times 2r_k}$ for $k = 2,\ldots,K$, thus the computational complexity is

$$\begin{aligned}
C_{full} = \sum_{k=2}^{K} cost(\boldsymbol{AB}) &= \sum_{k=2}^{K} \mathcal{O}(2^{k-1} r_{k-1} 2 r_k) \\
&\leq R_{\max}^2 \sum_{k=2}^{K} \mathcal{O}(2^k) \\
&= R_{\max}^2 \mathcal{O}(2^{K+1} - 4) \\
&= \mathcal{O}(R_{\max}^2 2^K).
\end{aligned}$$

Hence, the total computational complexity is

$$\mathcal{O}(R_{\max}^2 2^K) + \mathcal{O}(K^2 R_{\max}^3) + \mathcal{O}(R_{\max}^2 2^K) = \mathcal{O}(R_{\max}^2 2^K).$$

□

For the randomized SVD, we have the computational complexity $C_{rsvd}(A^{\{k\}}) = \mathcal{O}(m_k n_k (R_{\max} + p)) = \mathcal{O}(2^{K-k} R_{\max}(R_{\max} + p))$. Thus, the run time for the convolution with max rank TT-RSVD is similar when p is small. In D spatial dimensions, we can obtain a similar result but with replacing K with DK in the max rank TT-SVD algorithm and the full tensor algorithm, and the QTT-FFT algorithm is $\mathcal{O}(DK^2 R_{\max}^3)$. Hence, the total run time complexity in D spatial dimensions is $\mathcal{O}(R_{\max}^2 2^{DK})$.

6.2 Denoising

It is well known that the SVD can remove noise from matrix data as is seen in [17; 35], but little research has been done in denoising with tensor decompositions. In [51] and [57], the Tucker decomposition was used to help remove noise from point cloud data and from electron holograms respectively. In [21] it was shown that the TT-decomposition may have some advantages to denoising as opposed the Tucker decomposition. This is because a low rank Tucker matrix guarantees a low TT-rank for the data, however the converse statement is not always true.

Let $\mathscr{F}$ be the low TT-rank tensor that represents $\boldsymbol{f}$ in the QTT format. Then for some core tensors $\mathscr{F}^{(k)} \in \mathbb{R}^{r_{k-1} \times 2 \times r_k}$ with tensor slices $\mathscr{F}^{(k)}(:, i_k, :) = \boldsymbol{f}_{i_k}^{(k)} \in \mathbb{R}^{r_{k-1} \times r_k}$, $i_k = 0, 1$. Each element of $\mathscr{F}$ can be represented in the TT format as

$$\mathscr{F}(i_1, \ldots, i_k) = \boldsymbol{f}_{i_1}^{(1)} \ldots \boldsymbol{f}_{i_K}^{(K)}, \quad i_k = 0, 1, \quad k = 1, \ldots, K,$$

where each $\boldsymbol{f}_{i_k}^{(k)}$ is a low rank matrix. In practice, it is unlikely the data collected has a low rank TT decomposition since almost all real radar data has noise due to hardware limitations or other signals interfering with the data. Instead, we have the noisy data $\boldsymbol{f}_\xi$ whose tensor representation is

$$\mathscr{F}_\xi = \mathscr{F} + \boldsymbol{\xi},$$

where $\boldsymbol{\xi}$ is the realizations of the random noise in the TT format. The tensor $\mathscr{F}_\xi$ almost surely has full TT-rank when represented exactly in the QTT format. Ideally, we would like to be able to find an approximate TT decomposition $\tilde{\mathscr{F}}$ with TT-cores $\tilde{\mathscr{F}}^{(k)}$, $k = 1, \ldots, K$, using the noisy

data such that $\tilde{\mathscr{F}}^{(k)} \approx \mathscr{F}^{(k)}$. However, it is hard to guarantee any sort of bound on this. We argue though, that by using our proposed methods, when given the noisy data $\mathscr{F}_\xi$, we can find a TT decomposition $\tilde{\mathscr{F}}$ with low rank such that $\tilde{\mathscr{F}} \approx \mathscr{F}$.

Consider the first iteration of the for loop of algorithm 3, with $\mathscr{A} = \mathscr{A}_0 + \mathscr{A}_\xi$ as the sum of a smooth tensor ($\mathscr{A}_0$) and a noisy tensor ($\mathscr{A}_\xi$). Then after it is reshaped we get the matrix

$$\boldsymbol{A}^{\{1\}} = \boldsymbol{A}_0^{\{1\}} + \boldsymbol{A}_\xi^{\{1\}},$$

where $\boldsymbol{A}_0^{\{1\}}$ is a low rank matrix and $\boldsymbol{A}_\xi^{\{1\}}$ is added noise. Let $\boldsymbol{A}^{\{1\}} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^* + \boldsymbol{E}$ be the truncated SVD of $\boldsymbol{A}^{\{1\}}$ and $\boldsymbol{A}_0^{\{1\}} = \boldsymbol{U}_0\boldsymbol{\Sigma}_0\boldsymbol{V}_0^*$ be the SVD of $\boldsymbol{A}_0^{\{1\}}$. Note that $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^* \approx \boldsymbol{A}_0^{\{1\}}$ does not imply that $\boldsymbol{U} \approx \boldsymbol{U}_0$, and thus the TT-core $\mathscr{A}^{(1)}$ is not guaranteed to be approximately equal to $\mathscr{A}_0^{(1)}$, where $\mathscr{A}_0^{(1)}$ is the first TT-core of $\mathscr{A}_0$. However, if we let $\mathscr{A}^2 = \mathscr{A}$ on the second iteration of the loop in Algorithm 3 (and similarly for $\mathscr{A}_0$), we do get that the elements of the tensor contraction $\mathscr{A}^{(1)} \circ \mathscr{A}^2 \approx \mathscr{A}_0^{(1)} \circ \mathscr{A}_0^2$. Similarly, if we can approximate the noise free component on every iteration of the for loop, then we get an approximation for the tensor $\mathscr{A}_0$. While we do not have a theoretical bound on this error, our experiments in Section 6.3 show that this method works well at removing the noise. In fact, since our method computes multiple SVD, it is able to reduce a lot more noise than if we just did a single SVD, and is able to do so without excessive smearing.

6.3 Numerical Simulations

In this section, we present some examples in both one and two spatial dimensions. The original code for the TT-decompositions, as well as the QTT-FFT algorithms, comes from the Matlab toolbox **oseledets/TT-Toolbox**. We have modified it accordingly for the **max rank TT-SVD**, **max rank TT-RSVD**, and **SV drop off TT-SVD** algorithm, as discussed in Section 6.1. For all our examples, we compare the run time and errors of computing the convolution (5.1) using several methods. The error for every example is the l_2 relative error

$$E_2(\boldsymbol{I}) = \frac{\|\boldsymbol{I} - \boldsymbol{I}_{\text{ref}}\|_2}{\|\boldsymbol{I}_{\text{ref}}\|_2}, \tag{6.2}$$

where in one spatial dimension

$$\|\boldsymbol{I}\|_2 = \sqrt{\frac{1}{N}\sum_{j=1}^{N} |\boldsymbol{I}_j|^2},$$

and in two spatial dimensions

$$\|\boldsymbol{I}\|_2 = \sqrt{\frac{1}{N^2}\sum_{j,k=1}^{N} |\boldsymbol{I}_{j,k}|^2}.$$

The reference solution, $\boldsymbol{I}_{\text{ref}}$, is the discrete convolution (5.3) computed without any noise. In all of the examples, we compare our methods against computing the convolution with the randomized TT-SVD algorithm from [34], as well as computing the true noisy convolution with FFT. In two space dimensions, we also approximate the convolution using a low matrix rank approximation to the noisy data $\boldsymbol{f}_\xi$, where the truncated rank is determined by the true matrix rank of $\boldsymbol{f}$.

For all of these examples, we use the normalized sinc imaging kernel that corresponds to the GAF (5.4) truncated to a sufficiently large interval $[-L, L]$:

$$g(x) = \frac{\operatorname{sinc}(\pi \frac{x}{\Delta_x})}{\int_{-L}^{L} \operatorname{sinc}(\pi \frac{x}{\Delta_x})\, dx}, \quad x \in [-L, L] \tag{6.3}$$

for $D = 1$, and

$$g(x, y) = \frac{\operatorname{sinc}(\pi \frac{x}{\Delta_x})\operatorname{sinc}(\pi \frac{y}{\Delta_y})}{\iint_{-L}^{L} \operatorname{sinc}(\pi \frac{x}{\Delta_x})\operatorname{sinc}(\pi \frac{y}{\Delta_y})\, dx\, dy}, \quad (x, y) \in [-L, L] \times [-L, L] \tag{6.4}$$

for $D = 2$, where the resolution Δ_x in (6.3) and $\Delta_x = \Delta_y$ in (6.4) is a given parameter. The one-dimensional kernel (6.3) for $\Delta_x = 0.04\pi$ is shown in Figure 6.1.

In Table 6.1, we present the relative error for each example for $K = 20$ when $D = 1$, and for $K = 10$ when $D = 2$. In this table, the convolution $\boldsymbol{f}_\xi * \boldsymbol{g}$ is denoted $\boldsymbol{I}_\xi$ and is computed using the FFT algorithm, the QTT-convolution computed by the **max rank TT-SVD** algorithm is denoted as $\boldsymbol{I}_{QTT_0}$, the QTT-convolution computed by the **max rank TT-RSVD** is denoted $\boldsymbol{I}_{QTT_r}$, and the convolution computed using the **SV drop off TT-SVD** algorithm is denoted I_δ. In turn, the convolutions computed using the randomized TT-decomposition is denoted $\boldsymbol{I}_{RTT}$, and in two dimensions, the convolution computed using low rank approximations of $\boldsymbol{f}$ is denoted $\boldsymbol{I}_{lr}$. For I_δ and $\boldsymbol{I}_{lr}$, we also denote what parameter δ and what truncation matrix rank R are used respectively for each example as a subscript of the error.

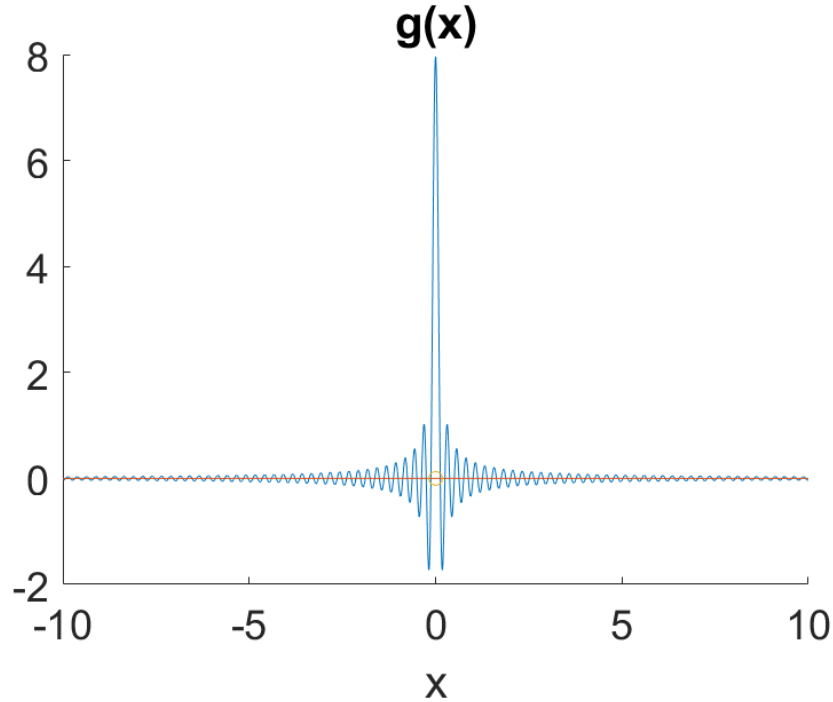

Figure 6.1: Kernel function (6.3) with $\Delta_x = 0.04\pi$.

For each example, we show the TT-ranks of the original function without noise, $\boldsymbol{f}$, in the QTT format given by the tensor $\mathscr{F}$. This QTT approximation is computed with Algorithm 3 with the tolerance $\varepsilon = 10^{-10}$. We compute these TT-ranks for $K = 20$ when $D = 1$, and for $K = 10$ when $D = 2$. However, it is worth noting that these TT-ranks do not change much for any size data. Notice that the max TT-ranks that we choose for our algorithms are less than the TT-ranks of $\boldsymbol{f}$ from Algorithm 3, yet still provide a good estimate.

Table 6.1: l_2-norm relative error for $K = 20$ for examples 1 and 2, and $K = 10$ for example 3.

example	$\boldsymbol{I}_{\xi}$	$\boldsymbol{I}_{QTT_0}$	$\boldsymbol{I}_{QTT_r}$	$\boldsymbol{I}_{\delta}$	$\boldsymbol{I}_{RTT}$	$\boldsymbol{I}_{lr}$
1	0.0383	**0.0028**	0.0102	$0.0280_{\delta=0.02}$	0.0430	-
2	0.0131	**0.0011**	0.0075	$0.0068_{\delta=0.01}$	0.0201	-
3	0.1142	**0.0151**	0.0447	$0.1650_{\delta=0.09}$	0.1534	$0.0470_{rank=23}$

The following test cases were considered.

6.3.1 Example 1

For this example let

$$f(x) = e^{-(\frac{3x}{10})^2}(0.4\sin(8\pi x) - 0.7\cos(6\pi x)), \quad x \in [-10, 10],$$

and

$$f_\xi(x_j) = f(x_j) + \xi_j,$$
$$x_j = -10 + \frac{\Delta x}{2} + j\Delta x, \quad j = 0, \ldots, N-1, \quad \Delta x = \frac{20}{N}, \quad N = 2^{K-1} - 1,$$

with $\xi_j \sim \mathcal{N}(0, 0.02)$. We also set the resolution $\Delta_x = 4\Delta x$ in (6.3), where Δx is the size of the spatial discretization, thus the width of the main lobe of the sinc is $8\Delta x$ on the x-axis.

As we can see in figure 6.2, the FFT-QTT algorithm removed a lot of the noise in the data compared to the true convolution. For $K = 20$, we also tried computing the convolution using the original TT-SVD algorithm given in Algorithm 3 with multiple values of $\boldsymbol{\varepsilon}$. The smallest error, as defined in (6.2), occurred when $\boldsymbol{\varepsilon} = 0.01$ and gave an relative error of $E_2(\boldsymbol{I}) = 0.03202$. This is close to the error of the true convolution of the noisy data, and took over 100 seconds to compute. However, as we can see in table 6.2, the run times for all of our methods on the same grid took less than a second. This gives us a good indication the original TT-SVD algorithm is not suitable for removing noise from data.

The max TT-rank of the discretization of $f(x)$ in the QTT format, $\boldsymbol{\mathcal{F}}$, is 17, yet we were able to achieve our approximation using a max rank of $R_{max} = 10$ for the **max rank TT-SVD** and **max rank TT-RSVD** algorithms and $\hat{R}_{max} = 15$ for the QTT-FFT algorithm. Thus even if we do not know the exact TT-rank, we can still compute a good approximation.

In Table 6.2 we show run times for different grid sizes for each method. As we can see, computing the convolution with FFT is faster than our methods for these values of K, however the convolution with our QTT methods get closer to the FFT run time as K increases. This is shown in the last column of table 6.2 where we see the ratio of the **max rank TT-SVD** convolution method to the FFT convolution method is getting smaller as K grows. This help verify our theoretical results that for some constant max rank R_{max} (and $\hat{R}_{max}$), the **max rank TT-SVD** convolution method is asymptotically faster than computing the convolution with FFT. The amount of data needed for our method to outperform the FFT method may be impractical for most real world application in 1-2 spatial dimensions.

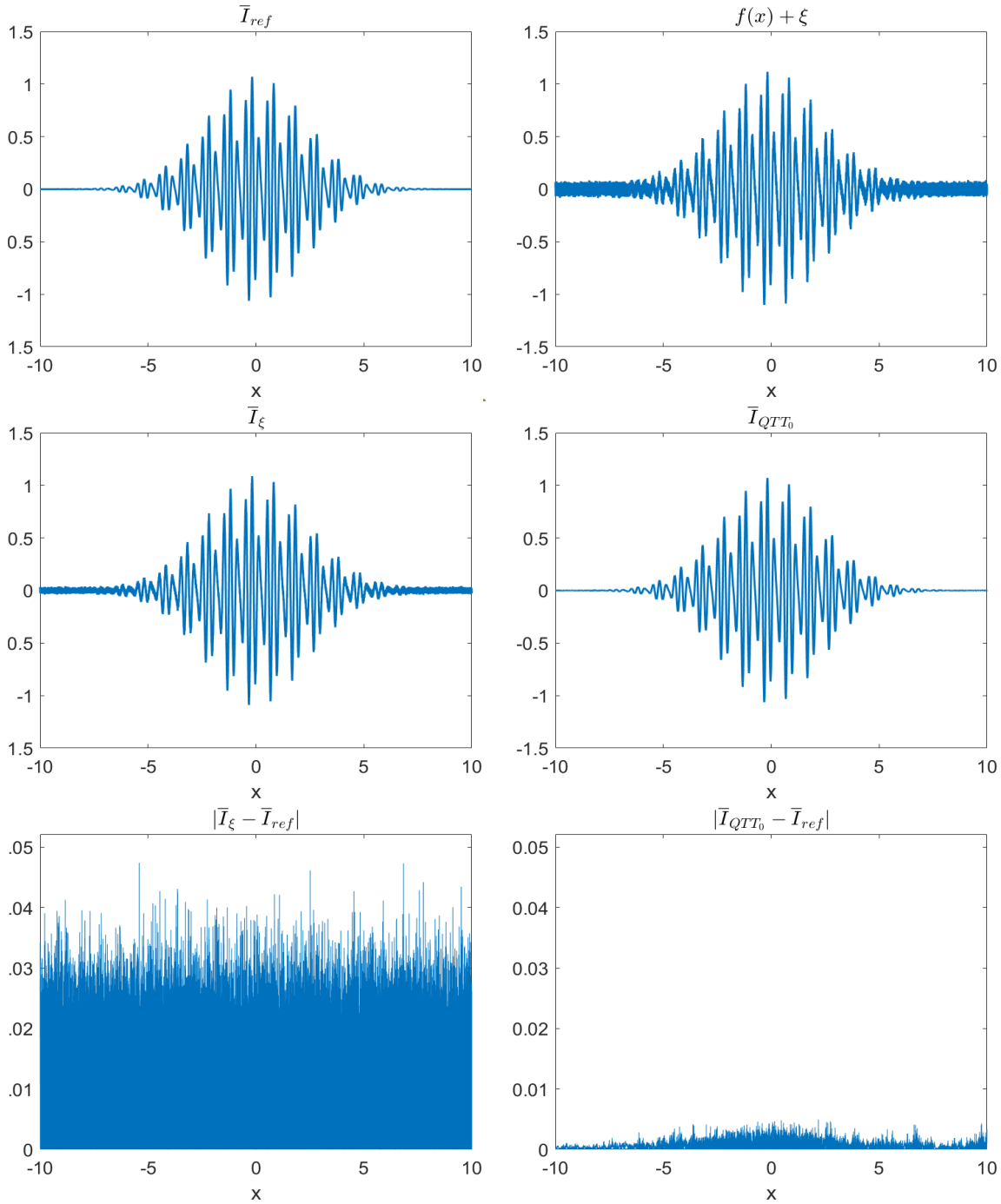

Figure 6.2: *Top Left:* True convolution of data without noise, $\boldsymbol{I}$. *Top Right:* Function data with noise, $\boldsymbol{f}_{\xi}$. *Middle Left:* True convolution of data with noise, $\boldsymbol{I}_{\xi}$. *Middle Right:* Convolution using the **max rank TT-SVD** algorithm, $\boldsymbol{I}_{QTT_0}$. *Bottom Left:* Absolute error of $\boldsymbol{I}_{\xi}$. *Bottom Right:* Absolute error of $\boldsymbol{I}_{QTT_0}$.

Table 6.2: Run time (seconds): Example 1 convolutions.

K	$\boldsymbol{I}_{\xi}$	$\boldsymbol{I}_{QTT_0}$	$\boldsymbol{I}_{QTT_r}$	$\boldsymbol{I}_{\delta}$	$\boldsymbol{I}_{RTT}$	$\boldsymbol{I}_{QTT_0}/\boldsymbol{I}_{\xi}$
16	**0.005**	0.325	0.369	$1.485_{\delta=0.02}$	0.414	65
20	**0.067**	0.653	0.694	$0.479_{\delta=0.02}$	0.609	9.7463
24	**1.21**	4.41	4.86	$3.10_{\delta=0.02}$	2.80	3.6446
26	**5.77**	17.75	19.68	$13.93_{\delta=0.02}$	10.97	3.0763
28	66.6	95.5	108.7	$79.0_{\delta=0.02}$	**62.49**	1.4339

Table 6.3: Data storage for Example 1.

K	$\boldsymbol{f}_{\xi}$	$\boldsymbol{\mathscr{F}}_{\xi}$
16	65,536	2088
20	1,048,576	2888
24	16,777,216	3688
26	67,108,864	4088
28	268,435,456	4488

Table 6.3 shows the number of elements to represent the data $\boldsymbol{f}_{\xi}$ fully, versus how many elements are required to store the data in the QTT-format with a prescribed max rank of $R_{max} = 10$, $\boldsymbol{\mathscr{F}}_{\xi}$, in example 1. As we can see, storing all the elements takes a lot of data and grows exponentially in K, while storing the elements in the QTT format takes a lot less data and only grows linearly in K. These values for the QTT-data storage can be found by looking at the sized of the core tensors. For the tensor $\boldsymbol{\mathscr{F}}_{\xi}$ in the QTT format and with a max TT-rank of $R_{max} = 10$, we have the TT-cores

$$
\begin{aligned}
&\boldsymbol{\mathscr{F}}_{\xi}^{(1)}, \boldsymbol{\mathscr{F}}_{\xi}^{(K)} \in \mathbb{R}^{1\times 2\times 2},\\
&\boldsymbol{\mathscr{F}}_{\xi}^{(2)}, \boldsymbol{\mathscr{F}}_{\xi}^{(K-1)} \in \mathbb{R}^{2\times 2\times 4},\\
&\boldsymbol{\mathscr{F}}_{\xi}^{(3)}, \boldsymbol{\mathscr{F}}_{\xi}^{(K-2)} \in \mathbb{R}^{4\times 2\times 8},\\
&\boldsymbol{\mathscr{F}}_{\xi}^{(4)}, \boldsymbol{\mathscr{F}}_{\xi}^{(K-3)} \in \mathbb{R}^{8\times 2\times 10},\\
&\boldsymbol{\mathscr{F}}_{\xi}^{(k)} \in \mathbb{R}^{10\times 2\times 10}, \quad k = 5, \ldots, K-4.
\end{aligned}
$$

Thus, the number of elements that make up this QTT tensors is:

$$\text{number of QTT elements} = 2(1\times2\times2)+2(2\times2\times4)+2(4\times2\times8)+2(8\times2\times10)+(K-8)(10\times2\times10)$$

The **max rank TT-RSVD** algorithm was not able to produce results as good as the **max rank TT-SVD** (see Table 6.1 for relative error comparison and Table 6.2 for a run time comparison) but was still able to produce a reasonably low error. While the run time for the **max rank TT-SVD** was faster than the **max rank TT-RSVD** for all of our methods, the **max rank TT-RSVD** can be faster for tensors with larger mode sizes. This is due to the SVD in **max rank TT-SVD** algorithm with mode sizes, M_k, may be computed on a matrix with $m_k = M_k R_{\max}$ rows, while for the **max rank TT-RSVD** algorithm, the SVD is computed on a matrix with $m_k = R_{\max} + p$ rows. When $M_k = 2$ (such as for the QTT decomposition) the difference in the sizes of m_k does not make up for the extra amount of work the RSVD algorithm does. Although in this thesis we focus on the QTT-decomposition and thus $M_k = 2$, we believe this is important to note as the **max rank TT-RSVD** algorithm can speed up the TT-decomposition for higher mode tensor data and still produce accurate approximations. We verify this by computing the **max rank TT-SVD** algorithm and the **max rank TT-RSVD** algorithm on a tensor with $K = 8$ modes with each mode of size $M_k = 10$, $k = 1, \ldots, K$. Each element of this tensor was taken from the uniform distribution $\mathscr{U}[0,1)$. The max rank TT-SVD algorithm took 9.57 seconds and the **max rank TT-RSVD** algorithm only took 5.12 seconds, almost half the time of the **max rank TT-SVD** algorithm.

6.3.2 Example 2

If we were to choose a coarser resolution for the example of Section 6.3.1 (i.e., a wider sinc function), we could reduce the noise using the standard convolution, at the cost of smoothing out the solution's peaks. Doing this gives similar results for the true convolution and with our methods (Section 6.1). In this section, we show an example where the ground reflectivity is very oscillatory. Here, the resolution Δ_x determined by the GAF must be small (i.e., the sinc function must be "skinny"). Otherwise, if the sinc window is close to or larger than the characteristic scale of variation of the ground reflectivity, then the convolution can smooth out the true oscillations instead of just the noise, loosing most of the information in f.

We choose the ground reflectivity as

$$f(x) = e^{-(3x)^2}(0.9\sin(\frac{2x\pi}{5\Delta x}) + 1.4\cos(\frac{x\pi}{3\Delta x})), \quad x \in [-1,1],$$

and

$$f_\xi(x_j) = f(x_j) + \xi_j,$$
$$x_j = -1 + \frac{\Delta x}{2} + j\Delta x, \quad j = 0, \ldots, N-1, \quad \Delta x = \frac{2}{N}, \quad N = 2^{K-1} - 1,$$

with $\xi_j \sim \mathcal{N}(0, .01)$. We use $\Delta_x = 2\Delta x$ and the max TT-rank of the discretization of the smooth function $f(x)$ in the QTT-format is 26.

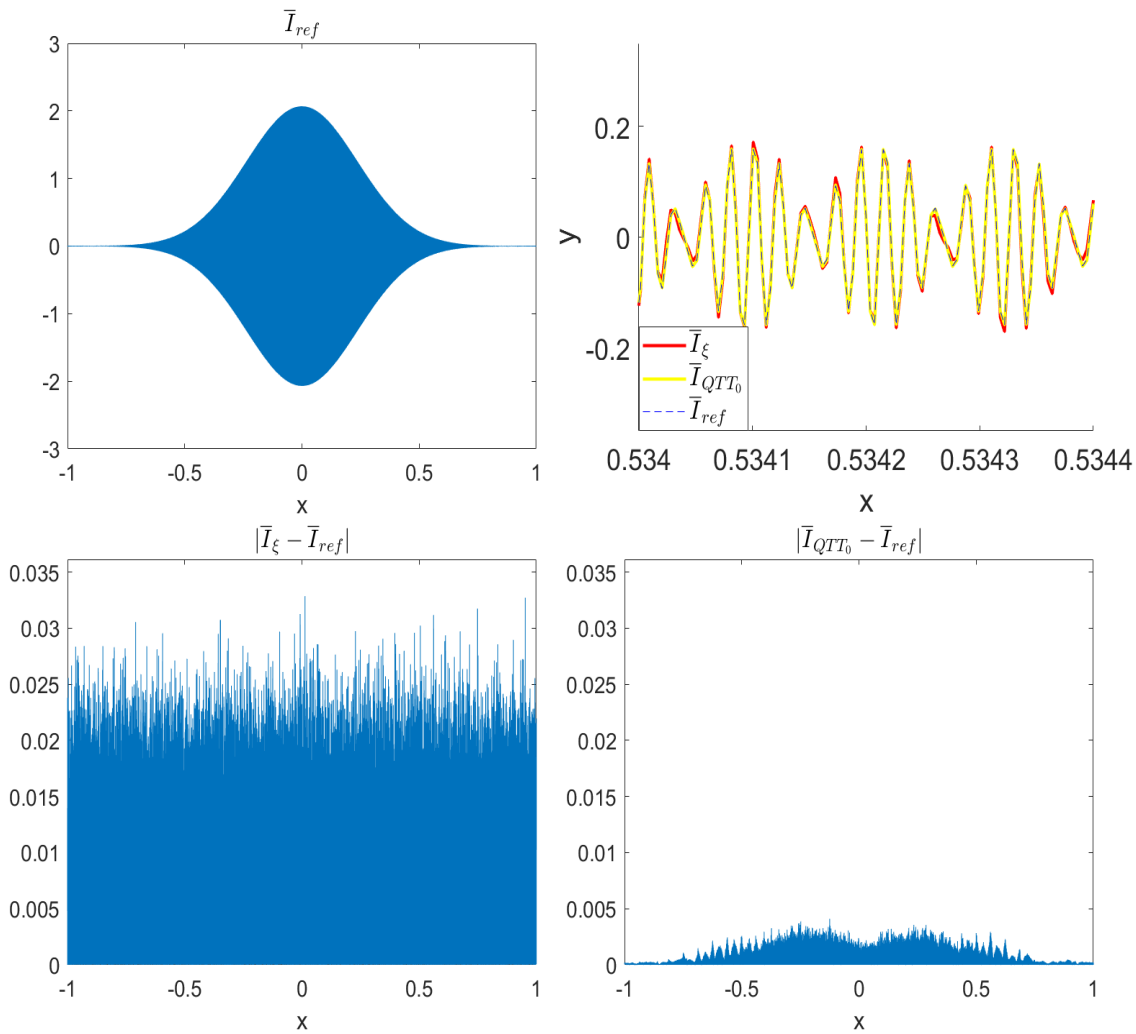

Figure 6.3: *Top Left:* True convolution of data without noise, $\boldsymbol{I}$. *Top Right:* Zoomed in graph of $\boldsymbol{I}_\xi$, $\boldsymbol{I}_{QTT_0}$, and $\boldsymbol{I}_{ref}$. *Bottom Left:* Absolute error of $\boldsymbol{I}_\xi$. *Bottom Right:* Absolute error of $\boldsymbol{I}_{QTT_0}$.

Again, we use the max ranks of $R_{max} = 10$ and $\hat{R}_{max} = 15$ for the **max rank TT-SVD** and QTT-SVD algorithms respectively. As the function is too oscillatory to see a lot of useful information in the full graph (see top left of figure 6.3), we show a zoomed in plot of the graph of $\boldsymbol{I}_{\xi}$, $\boldsymbol{I}_{QTT_0}$, and $\boldsymbol{I}_{\text{ref}}$ (see top right of figure 6.3). While there is some error, the QTT-FFT convolution agrees with the true convolution $\boldsymbol{I}_{ref}$ very well, whereas $\boldsymbol{I}_{\xi}$ has a larger noticeable differences. This is verified by the graphs of the absolute error given in Fig. 6.3, where the bottom left shows the error for $\boldsymbol{I}_{\xi}$ and the bottom right shows the error for $\boldsymbol{I}_{QTT_0}$.

6.3.3 Example 3

Let

$$f(x,y) = e^{-((2x)^2+(2y)^2)}(\sin(2\pi x) - \cos(7\pi y) + \cos(4\pi x y) - \sin(3\pi x y),$$
$$(x,y) \in [-1,1] \times [-1,1]$$

and

$$f_{\xi}(x_j, y_k) = f(x_j, y_k) + \xi_{j,k},$$
$$x_j = -1 + \frac{\Delta x}{2} + j\Delta x, \quad y_k = -1 + \frac{\Delta x}{2} + k\Delta x,$$
$$j,k = 0,\ldots,N-1, \quad \Delta x = \frac{2}{N}, \quad N = 2^{K-1} - 1,$$

with $\xi_{j,k} \sim \mathcal{N}(0, 0.1)$. We have $\Delta_x = \Delta_y = 2\Delta x$, thus the diameter of the main lobe of the sinc is $4\Delta x$ on the xy-plane. Here, we show a 2D example whose discretization of a the smooth function f has a matrix rank of 23, and a TT-rank of 26 when represented in the QTT format. We still use the ranks $R_{max} = 10$ and $\hat{R}_{max} = 15$ for our **max rank TT-SVD** (**max rank TT-RSVD**) and max rank QTT-FFT, thus our TT-ranks are much smaller than the true TT-ranks. In Fig. 6.4 and Table 6.1, notice that our method is still able to capture the shape of the original function with an error that is an order of magnitude smaller than the error from the true convolution using FFT. The plots on the bottom of Fig. 6.4 are a side view of the error graphs, as it is easier to see compare the errors in this view. The 2D examples are quite similar to the previous test case, thus it is reasonable to assume our method works about the same regardless of the spatial dimension.

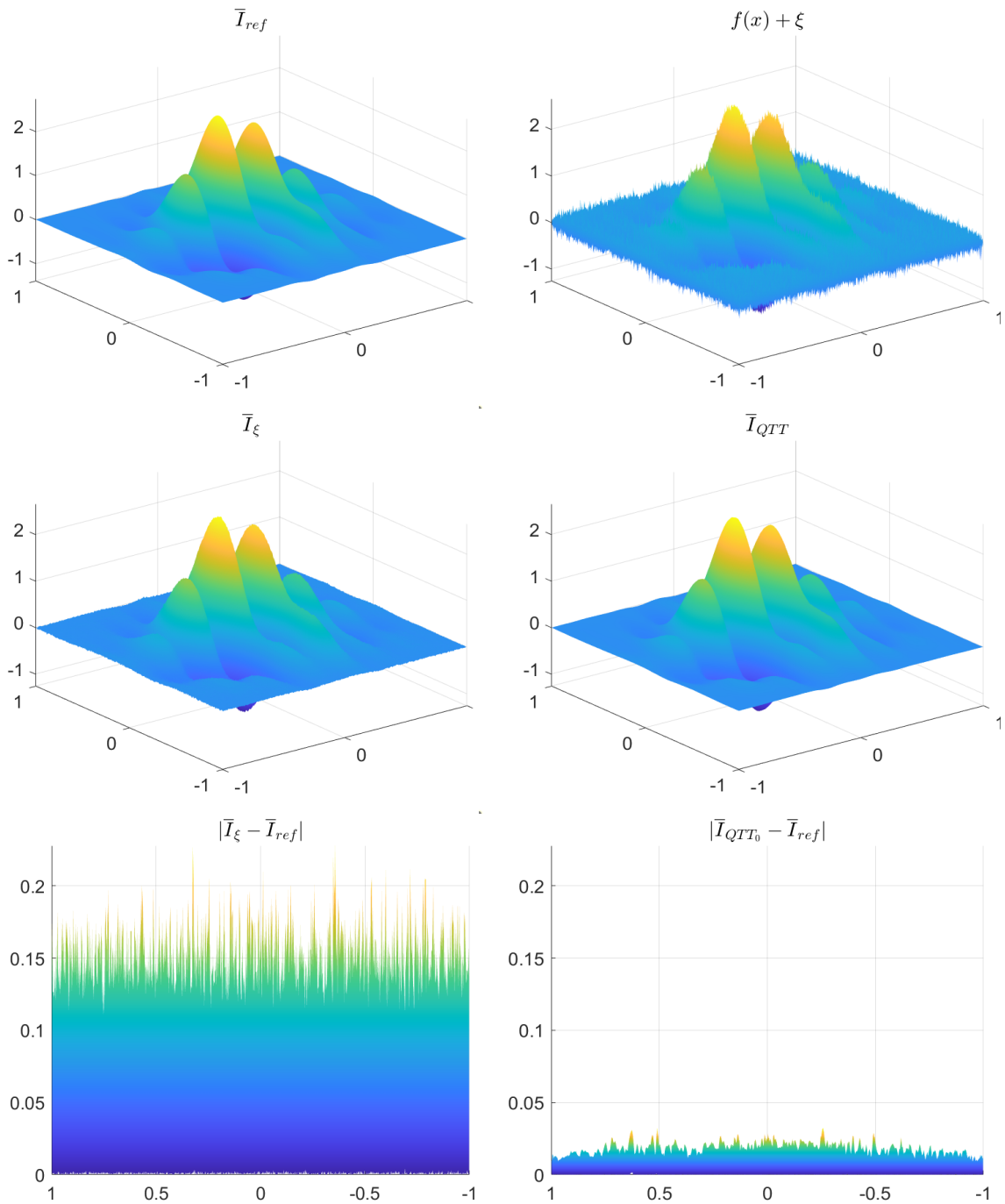

Figure 6.4: *Top Left:* True convolution of data without noise, $\boldsymbol{I}$. *Top Right:* Function data with noise, $\boldsymbol{f}_{\xi}$. *Middle Left:* True convolution of data with noise, $\boldsymbol{I}_{\xi}$. *Middle Right:* Convolution using the **max rank TT-SVD** algorithm, $\boldsymbol{I}_{QTT_0}$. *Bottom Left:* Absolute error of $\boldsymbol{I}_{\xi}$. *Bottom Right:* Absolute error of $\boldsymbol{I}_{QTT_0}$.

In table 6.4 we compare the run times for the different methods of computing the convolution in two spatial dimensions. We get a similar results as the one dimensional case, where the fastest run time is from the convolution with FFT, but with the **max rank TT-SVD** and **max rank TT-RSVD** methods approaching its run time asymptotically. In 2D, when there is the same amount of data as in the 1D case (for example $2^{14\times 14}$ in 2D compared to 2^{28} in 1D) the 2D examples does not run as fast as the 1D example. This is due to the extra work in the 2D QTT-FFT algorithm from [15].

Table 6.4: Run times (seconds): Example 3 convolutions.

K	$\boldsymbol{I}_{\xi}$	$\boldsymbol{I}_{QTT_0}$	$\boldsymbol{I}_{QTT_r}$	$\boldsymbol{I}_{\delta}$	$\boldsymbol{I}_{RTT}$	$\boldsymbol{I}_{lr}$	$\boldsymbol{I}_{QTT_0}/\boldsymbol{I}_{\xi}$
8	**0.0034**	0.2213	0.310	$7.102_{\delta=0.04}$	0.297	$0.0090_{rank=2}$	65.088
10	**0.0629**	0.9209	1.428	$0.670_{\delta=0.04}$	1.321	$0.154_{rank=2}$	14.651
12	**0.948**	8.6462	11.258	$22.79_{\delta=0.04}$	10.27	$5.15_{rank=2}$	9.121
14	**58.67**	147.8	151.05	$1087_{\delta=0.04}$	164.5	$286.2_{rank=2}$	2.519

Again we compare the amount of data stored in the full format versus in the QTT-format. Note that the spatial dimension of the original function does not matter in how much storage it takes, just the dimensionality of the data. For example, it takes just as much data to store a vector in $\mathbb{R}^{2^{20}}$ as it does to store a matrix in $\mathbb{R}^{2^{10}\times 2^{10}}$ in the QTT format with a max rank of R_{max}.

Table 6.5: Data storage for Example 3.

K	$\boldsymbol{f}_{\xi}$	$\mathscr{F}_{\xi}$
8	65,536	2088
10	1,048,576	2888
12	16,777,216	3688
14	268,435,456	4488

6.4 Conclusion

In this chapter, we have shown that the QTT decomposition along with the QTT-FFT algorithm can effectively be used to remove noise from signals with full TT-ranks when the true signal is of low rank. As we have seen in the numerical examples, we were able to drastically remove the amount of noise from the signal compared to if we took the convolution the traditional way of using the FFT algorithm. This came at a cost of run time but our methods still ran at a reasonable speed which got closer to the FFT run time as the dimensionality of the data increased. This was shown by three different examples, two in one spatial dimension and one in two spatial dimensions. We were even able to show that our method works on very oscillatory data where it is required to have a sinc kernel with a narrow main lobe. Using approximate TT-ranks smaller than the TT-ranks of the true signal data, we were able to recover most of the signal. This indicates that as long as the signal is reasonably smooth, the QTT decomposition can effectively be used for noise reduction for high dimensional data, even if the true TT-rank is unknown.

From our three new methods, the **max rank TT-SVD** convolution algorithm did much better than the **max rank TT-RSVD** and the **SV drop off TT-SVD** convolution algorithms. As was stated before, the **max rank TT-RSVD** can outperform the **max rank TT-SVD** algorithm when there are larger mode sizes than are used in this thesis. For this reason we present this algorithm, as we have not seen it in the literature anywhere else. The **SV drop off TT-SVD** convolution algorithms did not produce as accurate of a method as the **max rank TT-SVD** or the **max rank TT-RSVD** algorithm, however in some cases it did run faster and this method may give a higher degree of confidence that the truncated singular values are of little importance. Unfortunately, this method can also lead to long run times as is seen in table 6.4 when $K = 14$.

REFERENCES

[1] M. F. Atiyah, R. Bott, and L. Gårding. Lacunas for hyperbolic differential operators with constant coefficients. I. *Acta Math.*, 124:109–189, 1970.

[2] M. F. Atiyah, R. Bott, and L. Gårding. Lacunas for hyperbolic differential operators with constant coefficients. II. *Acta Math.*, 131:145–206, 1973.

[3] Yohai Bar-Sinai, Stephan Hoyer, Jason Hickey, and Michael P. Brenner. Learning data-driven discretizations for partial differential equations. *Proc. Natl. Acad. Sci. USA*, 116(31):15344–15349, 2019.

[4] M. Belger, R. Schimming, and V. Wünsch. A survey on Huygens' principle. *Z. Anal. Anwendungen*, 16(1):9–36, 1997. Dedicated to the memory of Paul Günther.

[5] Deniz A. Bezgin, Steffen J. Schmidt, and Nikolaus A. Adams. A data-driven physics-informed finite-volume scheme for nonclassical undercompressive shocks. *Journal of Computational Physics*, 437:110324, 2021.

[6] Kaushik Bhattacharya, Bamdad Hosseini, Nikola B. Kovachki, and Andrew M. Stuart. Model reduction and neural networks for parametric pdes. *arXiv e-prints*, page arXiv:2005.03180, 2020.

[7] Maolin Che and Yimin Wei. Randomized algorithms for the approximations of Tucker and the tensor train decompositions. *Adv Comput Math*, 45:395–428, 2019.

[8] Zhen Chen, Victor Churchhill, Kailiang Wu, and Dongbin Xiu. Deep neural network modeling of unknown partial differential equations in nodal space. *Arxiv*, 2021.

[9] Margaret Cheney and Brett Borden. *Fundamentals of Radar Imaging*, volume 79 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 2009.

[10] Alina Chertock, Jian-Guo Liu, and Terrance Pendleton. Convergence analysis of the particle method for the Camassa-Holm equation. In *Hyperbolic problems—theory, numerics and applications. Volume 2*, volume 18 of *Ser. Contemp. Appl. Math. CAM*, pages 365–373. World Sci. Publishing, Singapore, 2012.

[11] Alina Chertock, Jian-Guo Liu, and Terrance Pendleton. Convergence of a particle method and global weak solutions of a family of evolutionary PDEs. *SIAM J. Numer. Anal.*, 50(1):1–21, 2012.

[12] Alina Chertock, Jian-Guo Liu, and Terrance Pendleton. Elastic collisions among peakon solutions for the Camassa-Holm equation. *Appl. Numer. Math.*, 93:30–46, 2015.

[13] R. Courant and D. Hilbert. *Methods of Mathematical Physics. Volume II*. Wiley, New York, 1962.

[14] Filipe De Avila Belbute-Peres, Thomas Economon, and Zico Kolter. Combining differentiable PDE solvers and graph neural networks for fluid flow prediction. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 2402–2411, Virtual, 13–18 Jul 2020. PMLR.

[15] Sergey Dolgov, Boris N. Khoromskij, and Dmitry Savostyanov. Superfast Fourier transform using qtt approximation. *Journal of Fourier Analysis and Applications*, 18:915–953, 2012.

[16] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. *Arxiv*, 2016.

[17] Brenden P. Epps and Eric M. Krivitzky. Singular value decomposition of noisy data: noise fltering. *Experiments in Fluids*, 60, 2019.

[18] Krzysztof Fonał and Rafał Zdunek. Distributed and randomized tensor train decomposition for feature extraction. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[19] Mikhail Gilman, Erick Smith, and Semyon Tsynkov. *Transionospheric synthetic aperture imaging*. Applied and Numerical Harmonic Analysis. Birkhäuser/Springer, Cham, Switzerland, 2017.

[20] Mikhail Gilman and Semyon Tsynkov. A mathematical perspective on radar interferometry. *Inverse Problems & Imaging*, 16(1):119–152, 2022. doi: 10.3934/ipi.2021043.

[21] Xiao Gong, Wei Chen, Jie Chen, and Bo Ai. Tensor denoising using low-rank tensor train decomposition. *IEEE Signal Processing Letters*, 27:1685–1689, 2020.

[22] Sigal Gottlieb, Chi-Wang Shu, and Eitan Tadmor. Strong stability-preserving high-order time discretization methods.

[23] Lars Grasedyck. *Polynomial Approximation in Hierarchical Tucker Format by Vector–Tensorization*. Institut f¨ur Geometrie und Praktische Mathematik RWTH Aachen, 2010.

[24] Paul Günther. Ein Beispiel einer nichttrivialen Huygensschen Differentialgleichung mit vier unabhängigen Variablen. *Arch. Rational Mech. Anal.*, 18:103–106, 1965. [German].

[25] Paul Günther. *Huygens' principle and hyperbolic equations*, volume 5 of *Perspectives in Mathematics*. Academic Press Inc., Boston, MA, 1988. With appendices by V. Wünsch.

[26] W. Hackbusch and S. Kühn. A new scheme for the tensor representation. *J. Fourier Anal. Appl.*, 15(5):706–722, 2009.

[27] J. Hadamard. *Lectures on Cauchy's Problem in Linear Partial Differential Equations*. Yale University Press, New Haven, 1923.

[28] J. Hadamard. *Problème de Cauchy*. Hermann et cie, Paris, 1932. [French].

[29] J. Hadamard. The problem of diffusion of waves. *Ann. of Math. (2)*, 43:510–522, 1942.

[30] N. HALKO, P. G. MARTINSSON, and J. A. TROPP. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[31] Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[32] R.A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an explanatory multimodal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, December 1970.

[33] Kurt Hornik, Maxwell Stinchombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[34] Benjamin Huber, Reinhold Schneider, and Sebastian Wolf. A randomized tensor train singular value decomposition. In *Compressed sensing and its applications*, Appl. Numer. Harmon. Anal., pages 261–290. Birkhäuser/Springer, Cham, 2017.

[35] Sunil K. Jha and R. D. S. Yadava. Denoising by singular value decomposition and its application to electronic nose data processing. *IEEE Sensors Journal*, 11(1):35–44, 2011.

[36] Adar Kahana, Eli Turkel, Shai Dekel, and Dan Givoli. Obstacle segmentation based on the wave equation and deep learning. *Journal of Computational Physics*, 413:109458, 2020.

[37] Adar Kahana, Eli Turkel, Shai Dekel, and Dan Givoli. A physically-informed deep-learning model using time-reversal for locating a source from sparse and highly noisy sensors data. *Journal of Computational Physics*, 470:111592, 2022.

[38] Sharmila Karumuri, Rohit Tripathy, Ilias Bilionis, and Jitesh Panchal. Simulator-free solution of high-dimensional stochastic elliptic partial differential equations using deep neural networks. *Journal of Computational Physics*, 404:109120, 2020.

[39] VLADIMIR A. KAZEEV, BORIS N. KHOROMSKIJ, and EUGENE E. TYRTYSHNIKOV. Multilevel Toeplitz matrices generated by tensor-structured vectors and convolution with logarithmic complexity. *SIAM Journal on Scientific Computing*, 35(3):A1511–A1536, 2013.

[40] YUEHAW KHOO, JIANFENG LU, and LEXING YING. Solving parametric pde problems with artificial neural networks. *European Journal of Applied Mathematics*, pages 1–15, 2020.

[41] Boris N. Khoromskij. O(d logn)-quantics approximation of n-d tensors in high-dimensional numerical modeling. *Constr Approx*, 34:257–280, 2011.

[42] Byungsoo Kim, Vinicius C. Azevedo, Nils Thuerey, Theodore Kim, Markus Gross, and Barbara Solenthaler. Deep fluids: A generative network for parameterized fluid simulations. *Computer Graphics Forum*, 38(2):59–70, 2019.

[43] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, December 2014.

[44] Alexander Kurganov and Yu Liu. New adaptive artificial viscosity method for hyperbolic systems of conservation laws. 2012.

[45] ALEXANDER KURGANOV, SEBASTIAN NOELLE, and GUERGANA PETROVA. Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations.

[46] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[47] J. E. Lagnese and K. L. Stellmacher. A method of generating classes of Huygens' operators. *J. Math. Mech.*, 17:461–472, 1967.

[48] Aaron Larsen and Britton Olson. Approximating an artificial viscosity operator with neural networks in a shock-capturing scheme. In *APS Division of Fluid Dynamics Meeting Abstracts*, APS Meeting Abstracts, page T24.006, January 2021.

[49] Peter D. Lax and Ralph S. Phillips. An example of Huygens' principle. *Comm. Pure Appl. Math.*, 31(4):415–421, 1978.

[50] Randall J. Leveque. *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.

[51] Jianze Li, Xiao-Ping Zhang, and Tuan Tran. Point cloud denoising based on tensor Tucker decomposition. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 4375–4379, 2019.

[52] Lingjie Li, Wenjian Yu, and Kim Batselier. Faster tensor train decomposition for sparse data. *Journal of Computational and Applied Mathematics*, 405:113972, 2022.

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

[54] Shiyu Liang and R. Srikant. Why deep neural networks for function approximation? *Arxiv*, 2017.

[55] M. Matthisson. Le probléme de Hadamard relatif á la diffusion des ondes. *Acta Math.*, 71:249–282, 1939. [French].

[56] Craig Michoski, Miloš Milosavljević, Todd Oliver, and David R. Hatch. Solving differential equations using deep neural networks. *Neurocomputing*, 399:193–212, 2020.

[57] Yuki Nomura, Kazuo Yamamoto, Satoshi Anada, Tsukasa Hirayama, Emiko Igaki, and Koh Saitoh. Denoising of series electron holograms using tensor decomposition. *Microscopy*, 70(3):255–264, 09 2020.

[58] I. V. OSELEDETS. Approximation of 2d × 2d matrices using tensor decomposition. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2130–2145, 2010.

[59] I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[60] I. V. Oseledets. Constructive representation of functions in low-rank tensor formats. *Constr Approx*, 37:1–18, 2013.

[61] I. V. Oseledets and E. E. Tyrtyshnikov. Breaking the curse of dimensionality, or how to use SVD in many dimensions. *SIAM J. Sci. Comput.*, 31(5):3744–3759, 2009.

[62] Ivan Oseledets and Eugene Tyrtyshnikov. TT-cross approximation for multidimensional arrays. *Linear Algebra Appl.*, 432(1):70–88, 2010.

[63] Oded Ovadia, Adar Kahana, Eli Turkel, and Shai Dekel. Beyond the courant-friedrichs-lewy condition: Numerical methods for the wave problem using deep learning. *Journal of Computational Physics*, 442:110493, 2021.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[65] I. Petrowsky. On the diffusion of waves and the lacunas for hyperbolic equations. *Matematicheskii Sbornik (Recueil Mathématique)*, 17 (59)(3):289–370, 1945.

[66] Tong Qin, Kailiang Wu, and Dongbin Xiu. Data driven governing equations approximation using deep neural networks. *J. Comput. Phys.*, 395:620–635, 2019.

[67] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.*, 378:686–707, 2019.

[68] M. V. RAKHUBA and I. V. OSELEDETS. Fast multidimensional convolution in low-rank tensor formats via cross approximation. *SIAM Journal on Scientific Computing*, 37(2):A565–A582, 2015.

[69] Samuel Rudy, Alessandro Alla, Steven L. Brunton, and J. Nathan Kutz. Data-driven identification of parametric partial differential equations. *SIAM J. Appl. Dyn. Syst.*, 18(2):643–660, 2019.

[70] Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *SCIENCE ADVANCES*, 3, 2017.

[71] Hayden Schaeffer. Learning partial differential equations via data discovery and sparse optimization. *Proc. A.*, 473(2197):20160446, 20, 2017.

[72] R. Schimming. A review of Huygens' principle for linear hyperbolic differential equations. In N. H. Ibragimov and L. V. Ovsyannikov, editors, *Proceedings of the Joint IUTAM/IMU Symposium "Group-Theoretical Methods in Mechanics"*, pages 214–225, USSR, Novosibirsk, August 1978. USSR Acad. Sci., Siberian Branch, Institute of Hydrodynamics — Computing Center.

[73] Tianyi Shi, Maximilian Ruth, and Alex Townsend. Parallel algorithms for computing the tensor-train decomposition, 2021.

[74] Justin Sirignano and Konstantinos Spiliopoulos. A deep learning algorithm for solving partial differential equations. *J. Comput. Phys.*, 375(3874585):1339–1364, 2018.

[75] Karl L. Stellmacher. Ein Beispiel einer Huyghensschen Differentialgleichung. *Nachr. Akad. Wiss. Göttingen. Math. Phys. Kl. Math.-Phys. Chem. Abt.*, 1953:133–138, 1953. [German].

[76] Karl L. Stellmacher. Eine Klasse huyghenscher Differentialgleichungen und ihre Integration. *Math. Ann.*, 130:219–233, 1955. [German].

[77] Ben Stevens and Tim Colonius. Enhancement of shock-capturing methods via machine learning. *Theor. Comput. Fluid Dyn.*, 34(4):483–496, 2020.

[78] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin. Accelerating Eulerian fluid simulation with convolutional networks. *ArXiv e-prints*, 2016.

[79] L.R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31:279–311, 1966.

[80] V. S. Vladimirov. *Equations of Mathematical Physics*. Dekker, New-York, 1971.

[81] Yibo Yang and Paris Perdikaris. Adversarial uncertainty quantification in physics-informed neural networks. *J. Comput. Phys.*, 394:136–152, 2019.

[82] Dongkun Zhang, Ling Guo, and George Em Karniadakis. Learning in modal space: Solving time-dependent stochastic pdes using physics-informed neural networks. *SIAM Journal on Scientific Computing*, 42(2):A639–A665, 2020.

[83] Dongkun Zhang, Lu Lu, Ling Guo, and George Em Karniadakis. Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems. *Journal of Computational Physics*, 397:108850, 2019.

APPENDICES

APPENDIX

A

NUMERICAL LINEAR ALGEBRA

Numerical linear algebra is a very large (perhaps the largest) field of study in numerical analysis, thus, this appendix will only focus on a small number of methods that are used in this thesis.

A.1 Singular Value Decomposition

To understand the singular value decomposition, we first must know what a unitary matrix is.

Definition A.1.1. *A matrix $\boldsymbol{U} \in \mathbb{C}^{n\times n}$ is unitary if $\boldsymbol{U}^* = \boldsymbol{U}^{-1}$ where $\boldsymbol{U}^*$ is the conjugate transpose of the matrix $\boldsymbol{U}$ and $\boldsymbol{U}^{-1}$ is the matrix inverse.*

The singular value decomposition (SVD) of a matrix $\boldsymbol{A} \in \mathbb{C}^{m\times n}$ is a matrix factorization

$$\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^* = \boldsymbol{A} \tag{A.1}$$

where $\boldsymbol{U} \in \mathbb{C}^{m\times m}$ and $\boldsymbol{V} \in \mathbb{C}^{n\times n}$ are unitary matrices, and $\boldsymbol{\Sigma} \in \mathbb{R}^{m\times n}$ is a diagonal matrix. In the case when $m \geq n$, the matrices are of the form

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_n \\ \boldsymbol{0} \end{bmatrix}, \quad \boldsymbol{U} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m], \quad \boldsymbol{V} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n] \tag{A.2}$$

where

$$\boldsymbol{\Sigma}_n = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \sigma_n \end{bmatrix} \in \mathbb{R}^{n\times n},$$

$\boldsymbol{u}_i \in \mathbb{C}^m \ \forall i$, $\boldsymbol{v}_j \in \mathbb{C}^n \ \forall j$, $\mathbf{0} \in \mathbb{R}^{m-n,n}$ is a zero matrix, and $\sigma_k \in \mathbb{R} \ \forall k$ such that $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$. Similarly if $n > m$, then

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_m & \mathbf{0} \end{bmatrix}.$$

The values σ_k, $k = 1, \ldots, n$, are known as the singular values of the matrix $\boldsymbol{A}$, $\boldsymbol{u}_i \in \mathbb{C}^m$, $i = 1, \ldots, m$, are the left singular vectors of $\boldsymbol{A}$, and $\boldsymbol{v}_j \in \mathbb{C}^m$, $j = 1, \ldots, n$, are the right singular vectors of $\boldsymbol{A}$. The singular values are given by

$$\sigma_k = \sqrt{\lambda_k(\boldsymbol{A}^*\boldsymbol{A})} = \sqrt{\lambda_k(\boldsymbol{A}\boldsymbol{A}^*)}, \quad k = 1, \ldots n,$$

where $\lambda_k(\boldsymbol{A}^*\boldsymbol{A})$ and $\lambda_k(\boldsymbol{A}\boldsymbol{A}^*)$ are the eigenvalues of $\boldsymbol{A}^*\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^*$ respectively, and the singular vector are the eigenvectors of $\boldsymbol{A}^*\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{A}^*$ such that

$$\boldsymbol{A}\boldsymbol{A}^*\boldsymbol{u}_i = \sigma_i^2 \boldsymbol{u}_i$$
$$\boldsymbol{A}^*\boldsymbol{A}\boldsymbol{v}_j = \sigma_j^2 \boldsymbol{v}_j.$$

The rank of a matrix $\boldsymbol{A}$ is equal to the number of nonzero singular values.

We will now go over some properties of the SVD.

Definition A.1.2. *The Frobenius norm, $\|\cdot\|_F$, of a matrix $\boldsymbol{A} \in \mathbb{C}^{m\times n}$ is defined as*

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |\boldsymbol{A}_{i,j}|^2}$$

Theorem A.1.3. *The Frobenius norm of $\boldsymbol{A}$ is equal to the square root of the sum of singular values squared,*

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{k=1}^{n} \sigma_k^2}.$$

Given a matrix $\boldsymbol{A}$, Theorem (A.1.4) gives some insight into how to find the 'closest' matrix to $\boldsymbol{A}$ that has a rank of K. We will denote this matrix as $\boldsymbol{A}_K$.

Theorem A.1.4. *(Eckart-Young-Mirsky). Let $\boldsymbol{A} \in \mathbb{C}^{m\times n}$ with a singular value decomposition given as in* (A.1)*. Let $K \leq \min(m,n)$ and*

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}^{(1)} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}^{(2)} \end{bmatrix}$$

where $\boldsymbol{\Sigma}^{(1)} \in \mathbb{R}^{K\times K}$ and $\boldsymbol{\Sigma}^{(2)} \in \mathbb{R}^{m-K,n-k}$. Then

$$\boldsymbol{A}_K := \boldsymbol{U}^{(1)}\boldsymbol{\Sigma}^{(1)}(\boldsymbol{V}^{(1)})^* = \underset{\boldsymbol{B} \text{ of rank} \leq K}{\operatorname{argmin}} \|\boldsymbol{A} - \boldsymbol{B}\|_F.$$

where $\boldsymbol{U}^{(1)} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_K]$ and $\boldsymbol{V}^{(1)} = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_K]$. Furthermore

$$\|\boldsymbol{A}_K\|_F = \sqrt{\sum_{k=1}^{K} \sigma_k^2} \quad \text{and} \quad \|\boldsymbol{A} - \boldsymbol{A}_K\|_F = \sqrt{\sum_{k=K+1}^{n} \sigma_k^2}.$$

This theorem shows one of the reasons why the SVD is so useful. With it we know how to get the most useful information from a matrix using the least amount of data. The SVD along with this theorem has many practical applications. These include principle component analysis (PCA), which uses the SVD to reduce the dimension of a data set in order to interpret the most influential components, i.e. the largest eigenvalues and their eigenvectors of the covariance matrix for the data. Another application is in signal processing, and in particular removing noise from data.

A.1.1 Randomized SVD

Here we give a brief overview of the randomized SVD (RSVD) decomposition from [30]. To compute the RSVD of the matrix $A \in \mathbb{R}^{m\times n}$, the first step is to find $Q \in \mathbb{R}^{m\times(k+p)}$ such that

$$\boldsymbol{A} \approx \boldsymbol{Q}\boldsymbol{Q}^*\boldsymbol{A}$$

where $\boldsymbol{Q}$ has orthonormal columns and whose columns are approximations for the range of $\boldsymbol{A}$. Here, k is the number of singular values that we want in our approximation to be close to the singular values of $\boldsymbol{A}$, and p is what is known as an oversampling parameter. To find $\boldsymbol{Q}$, we use the following Algorithm 6.

Algorithm 6: Solving the Fixed-Rank Problem

input : $\boldsymbol{A}$, k, p

output : $\boldsymbol{Q}$

Draw random matrix $\boldsymbol{\Omega} \in \mathbb{R}^{n\times(k+p)}$ such that $\boldsymbol{\Omega}_{i,j} \sim \mathcal{N}(0,1)$.

Let $\boldsymbol{Y} = \boldsymbol{A}\boldsymbol{\Omega}$.

Compute QR factorization $\boldsymbol{Q}\boldsymbol{R} = \boldsymbol{Y}$.

Once we have obtained $\boldsymbol{Q}$, we can compute the low rank RSVD using Algorithm 7 (Algorithm 5.1 in [30]).

Algorithm 7: RSVD

input : $\boldsymbol{A}$, $\boldsymbol{Q}$, k

output : $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^*$

1. Let $\boldsymbol{B} = \boldsymbol{Q}^*\boldsymbol{A}$.
2. Compute SVD: $\tilde{\boldsymbol{U}}\boldsymbol{\Sigma}\boldsymbol{V}^* = \boldsymbol{B}$.
3. Let $\boldsymbol{U} = \boldsymbol{Q}\tilde{\boldsymbol{U}}$.

With these algorithms, we obtain an approximation $\tilde{\boldsymbol{A}}$ to $\boldsymbol{A}$ such that

$$\|\boldsymbol{A} - \tilde{\boldsymbol{A}}\| \leq (1 + 11\sqrt{k+p}\sqrt{\min(m,n)})\sigma_{k+1}, \tag{A.3}$$

with probability $1 - 6p^{-p}$. If we truncate the SVD to only the leading k singular values in Algorithm 7, then the error on the left hand side of (A.3) only increases by at most σ_{k+1}. The computational complexity for each step of this algorithm is given as

- $\mathcal{O}(mn(k+p))$
- $\mathcal{O}((k+p)^2 n)$
- $\mathcal{O}((k+p)^2 m)$,

Thus, for $k + p < \min(m,n)$, the overall algorithm requires $\mathcal{O}(mn(k+p))$ operations.

A.2 Gauss–Seidel Method

The Gauss–Seidel (GS) method is an iterative algorithm used to find $\boldsymbol{x}$ in the inverse problem

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

where $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is an invertible matrix, and $\boldsymbol{b} \in \mathbb{R}^{n}$. The algorithm is as follows:

Algorithm 8: Gauss–Seidel Method

input : $\boldsymbol{A}$, $\boldsymbol{b}$, ϵ, and initial guess $\boldsymbol{x}^{(0)}$
output : $\boldsymbol{x}$
$k = 0$
while $\|\boldsymbol{A}\boldsymbol{x}^{(k)} - b\| \geq \epsilon$ **do**
  for $j = 1, \ldots, n$ **do**
    $\boldsymbol{x}_i^{(k+1)} = \frac{1}{\boldsymbol{A}_{ii}}(b_i - \sum_{j=1}^{i-1} \boldsymbol{A}_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} \boldsymbol{A}_{ij} x_j^{(k)})$
  end
  k = k+1
end
$\boldsymbol{x} = \boldsymbol{x}^{(k)}$;

This algorithm is known to converge when $\boldsymbol{A}$ is strictly diagonally dominant or if $\boldsymbol{A}$ is symmetric and positive definite.

APPENDIX

B

RUNGE-KUTTA METHODS

B.1 Third Order Strong Stability-Preserving Runge-Kutta Method

Here we will show the steps for the third order strong stability-preserving Runge-Kutta method (RK3-SSP) developed in [22]. First, consider the ordinary differential equation

$$\frac{d}{dt}u = L(u). \tag{B.1}$$

The (RK3-SSP) is the following convex combination of explicit Euler steps

$$\begin{aligned} u^{(1)} &= u^n + \Delta t\, L(u^n) \\ u^{(2)} &= .75\, u^n + .25(u^{(1)} + \Delta t\, L(u^{(1)})) \\ u^{n+1} &= \frac{1}{3}u^n + \frac{2}{3}(u^{(2)} + \Delta t\, L(u^{(2)})). \end{aligned}$$

This Runge-Kutta method is often used to solve the method of lines approximation for hyperbolic conservation law

$$u_t(x,t) = -f(u)_x,$$

where $L(u)$ is the discrete approximation for $-f(x)_x$. Let $u^n = [u_1^n, \ldots, u_{N_x}^n]$, and u_j^n, $j = 1, \ldots, N_x$ come from the discretization of a finite difference, volume, or element method. Then the (RK3-SSP) has the property that the total variation is strictly non-increasing in time, i.e.

$$TV(u^{n+1}) \leq TV(u^{n+1}),$$

where

$$TV(u^n) = \sum_{j=1}^{N_x} |u_j - u_{j+1}|.$$

B.2 Fourth Order Implicit-Explicit

Here we will show a fourth order implicit-explicit (IMEX) Runge-Kutta method used to approximate systems of differential equations that have a stiff and non-stiff term. Let

$$u_t = F(u) + R(u),$$

where F is the non-stiff term of the ODE and R is the stiff term. Then a 4th order IMEX method is given as

$$\begin{aligned}
u^{(1)} &= u^n + \frac{1}{4}\Delta t R(u^{(1)}) \\
u^{(2)} &= u^n + \frac{1}{2}\Delta t F(u^{(1)}) + \frac{1}{4}\Delta t R(u^{(2)}) \\
u^{(3)} &= u^n + \frac{1}{2}\Delta t (F(u^{(1)}) + F(u^{(2)})) + \frac{1}{3}\Delta t (R(u^{(1)}) + R(u^{(2)}) + R(u^{(3)})) \\
u^{n+1} &= u^n + \frac{1}{3}\Delta t (F(u^{(1)}) + F(u^{(2)}) + F(u^{(3)}) + R(u^{(1)}) + R(u^{(2)}) + R(u^{(3)})).
\end{aligned}$$

In the case when R is linear, this can be solved by some linear solver. For example, if we numerically solve the viscous Burgers equation

$$u_t + (\frac{u^2}{2})_x = \varepsilon u_{xx},$$

using the finite volume method, then we would get a system like

$$\bar{u}_j^{n+1} = \bar{u}_j^n - \frac{\Delta t}{\Delta x}(\mathscr{F}_{j+1/2}^n - \mathscr{F}_{j-1/2}^n) + \Delta t(\epsilon \frac{\bar{u}_{j-1}^{n+1} - 2\bar{u}_j^{n+1} + \bar{u}_{j+1}^{n+1}}{\Delta x^2}),$$

where $\mathscr{F}_{j+1/2}^n$, $j = 0, \ldots, N_x$ are the numerical fluxes for $\frac{u^2}{2}$, and $\bar{u}_j^n$ are the cell averages of u (see Section 2.2). If we let $u^n = [\bar{u}_1^n, \ldots, \bar{u}_{N_x}^n]^\top$, $R \in \mathbb{R}^{N_x \times N_x}$ such that

$$\begin{aligned} R_{j,j} &= \frac{-2\varepsilon}{\Delta x^2}, \quad j = 1, \ldots, N_x, \\ R_{j,j+1} &= \frac{\varepsilon}{\Delta x^2}, \quad j = 1, \ldots, N_x - 1, \\ R_{j,j-1} &= \frac{\varepsilon}{\Delta x^2}, \quad j = 2, \ldots, N_x, \end{aligned} \tag{B.2}$$

(with boundary conditions handled appropriately), and $F(u) = \frac{-1}{\Delta x}[\mathscr{F}_{3/2} - \mathscr{F}_{1/2}, \ldots, \mathscr{F}_{N_x+1/2} - \mathscr{F}_{N_x-1/2}]^\top$ then the IMEX method comes down to solving the systems of equations

$$\begin{aligned} (I - \frac{\Delta t}{4}R)u^{(1)} &= u^n \\ (I - \frac{\Delta t}{4}R)u^{(2)} &= u^n + \frac{\Delta t}{2}F(u^{(1)}) \\ (I - \frac{\Delta t}{3}R)u^{(3)} &= u^n + \frac{\Delta t}{3}(F(u^{(1)}) + Ru^{(1)} + Ru^{(2)}) \\ u^{n+1} &= u^n + \frac{\Delta t}{3}(F(u^{(1)}) + F(u^{(2)}) + F(u^{(3)}) + Ru^{(1)} + Ru^{(2)} + Ru^{(3)}). \end{aligned}$$

APPENDIX

C

CENTRAL UPWIND METHOD

The central upwind method [45] was developed to simulate systems of hyperbolic conservation laws. The method for the one dimensional and two dimensional cases is given below.

One-dimensional case $(D = 1)$**:**

Here we will consider the conservation law (2.9), in one spatial dimension. We will also assume that the system is strictly hyperbolic, thus the Jacobian of $\boldsymbol{F}$ has s unique eigenvalues.

Consider the piecewise polynomial reconstruction of $\boldsymbol{U}$ at time t_n given by

$$\tilde{\boldsymbol{U}}(x, t^n) = p_j^n(x), \quad x \in (x_{j-1/2}, x_{j+1/2}), \text{ for all } j = 1, \ldots, N_x.$$

For second order methods, it is enough to define $p_j^n(x)$ as the linear function

$$p_j^n(x) = \bar{\boldsymbol{U}}_j + (\bar{\boldsymbol{U}}_x)_j (x - x_j),$$

where the derivative $(\bar{\boldsymbol{U}}_x)_j$ is approximated using a limiter. In particular, it can be defined using

the minmod limiter as

$$(\bar{\boldsymbol{U}}_x)_j = \text{minmod}(\theta \frac{\boldsymbol{U}_{j+1} - \boldsymbol{U}_j}{\Delta x}, \frac{\boldsymbol{U}_{j+1} - \boldsymbol{U}_{j-1}}{2\Delta x}, \theta \frac{\boldsymbol{U}_j - \boldsymbol{U}_{j-1}}{\Delta x}),$$

where $\theta \in [1,2]$ and

$$\text{minmod}(x_1, x_2, x_3) = \begin{cases} \min(x_1, x_2, x_3), & \text{if } x_i > 0,\ \forall i \\ \max(x_1, x_2, x_3), & \text{if } x_i < 0,\ \forall i \\ 0, & \text{else} \end{cases}.$$

For the rest of this section, we will drop the superscript n for simplicity but we will still assume we are at time t_n.

Define

$$\boldsymbol{U}^+_{j+1/2} = p_{j+1}(x_{j+1/2}),$$
$$\boldsymbol{U}^-_{j+1/2} = p_j(x_{j+1/2}),$$

to be the values of $\tilde{\boldsymbol{U}}(x_{j+1/2}, t_n)$ on the right and left hand side of the cell boundary at $x_{j+1/2}$ respectively. Let

$$a^+_{j+1/2} = \max\{\lambda_s\big(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}(\boldsymbol{U}^-_{j+1/2})\big), \lambda_s\big(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}(\boldsymbol{U}^+_{j+1/2})\big), 0\},$$
$$a^-_{j+1/2} = \min\{\lambda_1\big(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}(\boldsymbol{U}^-_{j+1/2})\big), \lambda_1\big(\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}(\boldsymbol{U}^+_{j+1/2})\big), 0\}$$

where $\lambda_1 < \ldots < \lambda_s$ are the eigenvalues of the Jacobian matrix $\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}}$. Then the numerical flux $\mathscr{F}^n_{j+1/2}$ for the central upwind method is defined as follows

$$\mathscr{F}_{j+1/2} = \frac{a^+_{j+1/2}\boldsymbol{F}(\boldsymbol{U}^-_{j+1/2}) - a^-_{j+1/2}\boldsymbol{F}(\boldsymbol{U}^+_{j+1/2})}{a^+_{j+1/2} - a^-_{j+1/2}} + \frac{a^+_{j+1/2}a^-_{j+1/2}}{a^+_{j+1/2} - a^-_{j+1/2}}(\boldsymbol{U}^+_{j+1/2} - \boldsymbol{U}^-_{j+1/2}).$$

Two-dimensional case ($D = 2$):

Here we consider the two-dimensional conservation law

$$\boldsymbol{U}_t(x, y, t) + \boldsymbol{F}_1(\boldsymbol{U})_x + \boldsymbol{F}_2(\boldsymbol{U})_y = 0, \quad (x, y) \in \mathbb{R}^2,\ t > 0. \tag{C.1}$$

The piecewise linear reconstruction for the two dimensional case is given as

$$\begin{aligned}
&\tilde{\boldsymbol{U}}_{j,k} = \boldsymbol{U}_{j,k} + (\bar{\boldsymbol{U}}_x)_{j,k}(x - x_j) + (\bar{\boldsymbol{U}}_y)_{j,k}(y - y_k),\\
&(x, y) \in [x_{j-1/2}, x_{j+1/2}] \times [y_{k-1/2}, y_{k+1/2}],\\
&j = 1, \ldots, N_x, \quad k = 1, \ldots, N_y,
\end{aligned}$$

where reconstructed values at the cell interfaces are

$$\begin{aligned}
\boldsymbol{U}^E_{j,k} &= \boldsymbol{U}_{j,k} + (\bar{\boldsymbol{U}}_x)_{j,k}\frac{\Delta x}{2},\\
\boldsymbol{U}^W_{j,k} &= \boldsymbol{U}_{j,k} - (\bar{\boldsymbol{U}}_x)_{j,k}\frac{\Delta x}{2},\\
\boldsymbol{U}^N_{j,k} &= \boldsymbol{U}_{j,k} + (\bar{\boldsymbol{U}}_y)_{j,k}\frac{\Delta y}{2},\\
\boldsymbol{U}^S_{j,k} &= \boldsymbol{U}_{j,k} - (\bar{\boldsymbol{U}}_y)_{j,k}\frac{\Delta y}{2}.
\end{aligned}$$

Using the above values, we can calculate the local speeds around the cell interfaces as

$$\begin{aligned}
a^+_{j+1/2,k} &= \max\{\lambda_s\big(\frac{\partial \boldsymbol{F}_1}{\partial \boldsymbol{U}}(\boldsymbol{U}^E_{j,k})\big), \lambda_s\big(\frac{\partial \boldsymbol{F}_1}{\partial \boldsymbol{U}}(\boldsymbol{U}^W_{j+1,k})\big), 0\},\\
a^-_{j+1/2,k} &= \min\{\lambda_1\big(\frac{\partial \boldsymbol{F}_1}{\partial \boldsymbol{U}}(\boldsymbol{U}^E_{j,k})\big), \lambda_1\big(\frac{\partial \boldsymbol{F}_1}{\partial \boldsymbol{U}}(\boldsymbol{U}^W_{j+1,k})\big), 0\},\\
b^+_{j,k+1/2} &= \max\{\mu_s\big(\frac{\partial \boldsymbol{F}_2}{\partial \boldsymbol{U}}(\boldsymbol{U}^N_{j,k})\big), \mu_s\big(\frac{\partial \boldsymbol{F}_2}{\partial \boldsymbol{U}}(\boldsymbol{U}^S_{j,k+1})\big), 0\},\\
b^-_{j,k+1/2} &= \min\{\mu_1\big(\frac{\partial \boldsymbol{F}_2}{\partial \boldsymbol{U}}(\boldsymbol{U}^N_{j,k})\big), \mu_1\big(\frac{\partial \boldsymbol{F}_2}{\partial \boldsymbol{U}}(\boldsymbol{U}^S_{k,k+1})\big), 0\},
\end{aligned}$$

where $\lambda_1 < \ldots < \lambda_s$ are the eigenvalues of the Jacobian matrix $\frac{\partial \boldsymbol{F}_1}{\partial \boldsymbol{U}}$, and $\mu_1 < \ldots < \mu_s$ are the eigenvalues of the Jacobian matrix $\frac{\partial \boldsymbol{F}_2}{\partial \boldsymbol{U}}$. We then define the numerical fluxes as

$$\mathscr{F}_{j+1/2,k} = \frac{a^+_{j+1/2,k}\boldsymbol{F}_1(\boldsymbol{U}^E_{j,k}) - a^-_{j+1/2}\boldsymbol{F}_1(\boldsymbol{U}^W_{j+1,k})}{a^+_{j+1/2,k} - a^-_{j+1/2,k}} + \frac{a^+_{j+1/2,k}a^-_{j+1/2,k}}{a^+_{j+1/2,k} - a^-_{j+1/2,k}}(\boldsymbol{U}^W_{j+1,k} - \boldsymbol{U}^E_{j,k}),$$

and

$$\mathscr{G}_{j,k+1/2} = \frac{b^+_{j,k+1/2}\boldsymbol{F}_2(\boldsymbol{U}^N_{j,k}) - b^-_{j,k+1/2}\boldsymbol{F}_2(\boldsymbol{U}^S_{j,k+1})}{b^+_{j,k+1/2} - b^-_{j,k+1/2}} + \frac{b^+_{j,k+1/2}b^-_{j,k+1/2}}{b^+_{j,k+1/2} - b^-_{j,k+1/2}}(\boldsymbol{U}^S_{j,k+1} - \boldsymbol{U}^N_{j,k}).$$

APPENDIX

D

VARIABLES

The tables in this section present a lot of the variables that are used in the thesis along with a brief explanation. While not all variables in the thesis are in these table, we try to display the most important ones. There are many objects that have general sizes in the thesis (such amount of data in a data set), thus in Table D.2 and Table D.4 we give a list of these variables, along with the indices that correspond to it. For example, if we have N_t time steps corresponding to the time t_n, $n = 1, \ldots, N_t$, then we have Variable: N_t, and indices: n. Note that while there are a lot of unique variables for indices, some variables (such as i and l) are used multiple times. We try our best to make sure that the same index is not used in a situation where there may be some ambiguity in what it may represent. There are two tables for the neural network project and two tables for the tensor project.

Table D.1: A list of variables and functions used in PDEs project

Description	Variable
Multidimensional spatial variable	$\boldsymbol{x} \in \mathbb{R}^D$
Scalar spatial variable for first dimension	$x \in \mathbb{R}$
Scalar spatial variable for Second dimension	$y \in \mathbb{R}$
Time variable	$t \geq 0$

PDE system solution	$\boldsymbol{U}(\boldsymbol{x}, t)$
PDE scalar solution	$u(\boldsymbol{x}, t)$
General time dependent PDE right hand side	$F(\boldsymbol{x}, t, \boldsymbol{U}, \ldots)$
Discretized ODE system right hand side	$F(t, \boldsymbol{U})$
Neural network	$\boldsymbol{N}_\Theta$
Neural networks internal parameter set	Θ
Loss function	L_Θ
Activation function for layer η	$\boldsymbol{\sigma}_\eta$
Loss function regularization Coefficient	μ
Diffusion coefficient	$\boldsymbol{\varepsilon}$
Flux function	$\boldsymbol{F}(\boldsymbol{U})$
Source function	$\boldsymbol{S}(\boldsymbol{U})$
Vector of cell averages at time t_n	$\bar{\boldsymbol{U}}^n$
Cell average at x_j at time t_n	$\bar{\boldsymbol{U}}_j^n$
Neural network approximation to the cell average $\bar{\boldsymbol{U}}^n$	$\hat{\boldsymbol{U}}_j^n$
Spatial discretization	Δx
Temporal discretization	Δt
Numerical flux in x direction at $x_{j+1/2}$ (and y_k for 2D)	$\mathscr{F}_{j+1/2,(k)}$
Numerical flux in y direction at $(x_j, y_{k+1/2})$	$\mathscr{G}_{j,k+1/2}$
CFL number	ν
Parameter set for initial conditions	$\hat{\boldsymbol{p}}$
Uniform distribution	$\mathscr{U}$
Set of all real numbers	$\mathbb{R}$
Set of vectors of size n over the real numbers	$\mathbb{R}^n$
Set of matrices of size $m \times n$ over the real numbers	$\mathbb{R}^{m\times n}$
l_p relative error	E_p
Diffusive term of artificial viscosity	$\boldsymbol{\varepsilon}(\boldsymbol{U})$
Numerical approximation to $\boldsymbol{\varepsilon}(\boldsymbol{U})$	$\boldsymbol{\varepsilon}_{j+1/2}^n$
Artificial viscosity coefficient	C
Total variation of $\boldsymbol{U}$	$TV(\boldsymbol{U})$
Input to neural network (Section 3.3 and Chapter 4)	Φ
Output of neural network (Section 3.3 and Chapter 4)	Ψ
Set of points in the lacuna	Λ_1
Computational domain in Chapter 4	Ω
Subset of Ω where source may be found	Q

Domain of the source	Q_f

Table D.2: Size variables and the indices used to index them in PDEs project

Description	Variable	indices
Spatial dimension	D	d
Number of cells in the x direction	N_x	j
Number of cells in the y direction	N_y	k
Number of cells in the dth dimensions direction	N_d	j_l
Total number of spatial grid points	$N = \Pi_{d=1}^{D} N_d$	$\boldsymbol{j}_\ell$
Number of time steps	N_t	n
Number of equations	N_e	e
Number of point in stencil for neural network flux	N_f	α (β)
Number of simulations run to collect data	N_s	i
Number of hidden layers in neural network	H	η
Number of data pairs for neural network	M	m
Number of training data pairs for neural network	M_{tr}	m_i
Number of validation data pairs for neural network	M_{val}	m_l
Number of epochs	M_e	e
Batch size	B_s	-
Number of batches	β	-
Width of layer η	ω_η	-
Number of discrete C values (see chapter 4)	N_C	l
Number of parameters in $\hat{\boldsymbol{p}}$	N_p	l
Number of points in fine grid discretization	$\tilde{N}_x$	$\tilde{j}$

Table D.3: A list of variables and functions used in convolution project

Description	Variable
Multidimensional spatial variable	$\boldsymbol{x} \in \mathbb{R}^D$
Scalar spatial variable for first dimension	$x \in \mathbb{R}$
Scalar spatial variable for second dimension	$y \in \mathbb{R}$
Spatial discretization	Δx
Convolution input function	f
Convolution kernel	g

Convolution $(f * g)$	I
Discretization of f	$\boldsymbol{f}$
Discretization of g	$\boldsymbol{g}$
Discretization of I	$\boldsymbol{I}$
Set of all real numbers	$\mathbb{R}$
Set of vectors of size n over the real numbers	$\mathbb{R}^n$
Set of matrices of size $m \times n$ over the real numbers	$\mathbb{R}^{m \times n}$
Set of tensors of size $M_1 \times \ldots \times M_K$ over the real numbers	$\mathbb{R}^{M_1 \times \ldots \times M_K}$
Set of all complex numbers	$\mathbb{C}$
Set of vectors of size n over the complex numbers	$\mathbb{C}^n$
Set of matrices of size $m \times n$ over the complex numbers	$\mathbb{C}^{m \times n}$
Set of tensors of size $M_1 \times \ldots \times M_K$ over the complex numbers	$\mathbb{C}^{M_1 \times \ldots \times M_K}$
Vector	$\boldsymbol{v}$
Discrete Fourier transform of $\boldsymbol{v}$	$\hat{\boldsymbol{v}}$
QTT tensor representation of $\boldsymbol{v}$	$\boldsymbol{\mathcal{V}}$
QTT tensor representation of $\hat{\boldsymbol{v}}$	$\hat{\boldsymbol{\mathcal{V}}}$
Added noise	ξ_j
Function f with added noise	f_ξ
Normal distribution	$\mathcal{N}$
l_2 relative error	E_2
Kernel resolution parameter in x dimension	Δ_x
Kernel resolution parameter in y dimension	Δ_y
Tolerance in TT-SVD algorithm	$\boldsymbol{\varepsilon}$

Table D.4: Size variables and the indices used to index them in convolution project

Description	Variable	indices
Spatial dimension	D	d
Number of spatial discretizations along a dimension	N	j_d, j, or j and k
Number of tensor modes	K	k
Size of the kth tensor mode	M_k	i_k
Oversampling parameter	p	-
Max rank in max rank TT-SVD algorithm	R_{max}	-
Max rank in QTT-FFT algorithm	$\hat{R}_{max}$	-